Numéro d'ordre :

THÈSE

présentée à

l'Université de Paris-Sud
U.F.R. Scientifique d'Orsay

par

June ANDRONICK LIÈGE

pour l'obtention du grade de

Docteur en Sciences
de l'université de Paris XI, Orsay

Discipline : informatique

Sujet :

Modélisation et Vérification Formelles de Systèmes Embarqués dans les Cartes à Microprocesseur

Plate-Forme Java Card et Système d'Exploitation

Soutenue le 29 mars 2006, devant la commission d'examen composée de

Président :	Gilles Barthe
Rapporteurs :	Xavier Leroy
	George Necula
Examinateurs :	Boutheina Chetali
	Olivier Ly
	Pierre Paradinas
Directrice :	Christine Paulin-Mohring

Mis en page avec la classe thloria.

Résumé

Les travaux présentés dans ce mémoire ont pour objectif de renforcer le niveau de sûreté et de sécurité des systèmes embarqués dans les cartes à puce, grâce à l'utilisation des Méthodes Formelles. D'une part, nous présentons la vérification formelle de l'isolation des données de différentes applications chargées sur une même carte à puce. Plus précisément, nous décrivons la preuve formelle, dans le système de preuve *Coq*, que le contrôle dynamique d'accès aux données, implémenté par la plate-forme Java Card, assure les propriétés de confidentialité et d'intégrité. D'autre part, nous nous sommes intéressés à la correction et à l'innocuité du code source bas niveau d'un système d'exploitation embarqué. Cette étude est illustrée par un module de gestion de mémoire *Flash* par *journalisation*, assurant la cohérence des données de la mémoire en cas d'*arrachage* de la carte du terminal. La vérification de propriétés fonctionnelles et locales de ce module a été développée à l'aide de l'outil de vérification de programmes *Caduceus*. Cet outil n'acceptant pas certaines constructions de bas niveau du langage C, telles que les unions et les casts, nous proposons une analyse et différentes solutions pour la formalisation de ces constructions. Nous proposons également une extension de Caduceus permettant de spécifier et de vérifier le comportement d'une fonction en cas d'interruption soudaine de son exécution. Puis nous introduisons une méthodologie de vérification de propriétés globales de haut niveau visant l'expression et la preuve de ce type de propriétés sur un modèle formellement lié au code source. Plus précisément, nous décrivons l'extraction automatique d'un système de transitions formel, à partir d'annotations vérifiées par le code source. Ce système de transitions peut alors être plongé dans une logique d'ordre supérieur, apportant toute l'expressivité nécessaire à la définition et la preuve de propriétés complexes.

Mots-clés: Carte à puce, Méthodes Formelles, Sécurité, Isolation d'Applets, Vérification de Programme, Langage C, Modèle Mémoire, Mémoire Flash.

Abstract

The work presented in this thesis aims at strengthening the security and safety level of smart card embedded systems, with the use of Formal Methods. On one hand, we present the formal verification of the isolation of the data belonging to different applets loaded on the same card. More precisely, we describe the formal proof, in the *Coq* proof system, that the run-time access control, performed by the Java Card platform, ensures data confidentiality and integrity. On the other hand, we study the correctness and the safety of low level source code of an embedded operating system. Such source code is illustrated by a case study of a *Flash* memory management module, using a *journalling* mechanism and ensuring the memory consistency in the case of a *card tear*. The verification of functional and local properties has been developed using the *Caduceus* program verification tool. Since this tool does not support some low level constructions of the C language, such as the unions and the casts, we propose an analysis and some solutions for the formalisation of such constructions. We also propose an extension of Caduceus that allows to specify and verify the behaviour of a function in the case of sudden interruption of its execution. Then, we introduce a methodology for the verification of high level and global properties, which is meant for the expression and proof of this kind of properties on a model formally linked to the source code. More precisely, we describe an automatic extraction of a transition system from the annotations that are verified by the source code. This transition system can then be translated into a higher-order logic, with all the expressiveness for the definition of complex properties.

Keywords: Smart Card, Formal Methods, Security, Applet Isolation, Program Verification, C Language, Memory Model, Flash Memory.

Remerciements

Je voudrais en tout premier lieu adresser mes plus chaleureux remerciements à Christine Paulin. Avoir accepté de diriger ma thèse était un honneur pour moi. L'avoir fait avec cette gentillesse, cette attention, cette disponibilité et cette compréhension qui la caractérisent m'a été d'une aide réellement inestimable.

Je dois beaucoup à Boutheina Chetali, qui m'a encadrée au sein d'Axalto tout au long de ma thèse. Je la remercie sincèrement pour m'avoir toujours soutenue dans ma carrière professionnelle et pour avoir fait en sorte que ma thèse se déroule dans les meilleures conditions.

Je suis très reconnaissante à George Necula et Xavier Leroy d'avoir accepté d'être rapporteurs de cette thèse et je les remercie sincèrement pour les remarques précieuses dont ils m'ont fait part. Je remercie également Gilles Barthe et Pierre Paradinas de me faire l'honneur de faire partie de mon jury.

Il est des choses impossibles à formuler... mais je voudrais tout de même profiter de l'occasion qui m'est donnée pour exprimer toute ma reconnaissance pour Olivier Ly. Je l'ai connu il y a près de six ans à Bordeaux, où il essayait de me convaincre de la beauté majestueuse des polytopes simpliciaux... J'ai été plus convaincue par son initiation aux assistants de preuve ! Depuis, il m'a encadrée, guidée et soutenue tout au long de mon parcours, du choix de mon DEA à celui de mon stage et de ma thèse, jusqu'à la lecture minutieuse de ce manuscrit. Je le remercie pour tout cela, pour son amitié et pour m'avoir, malgré lui et malgré moi, laissée m'envoler. Je lui dois infiniment plus que je ne pourrais l'exprimer.

Un immense merci à Jean-Christophe Filliâtre, Claude Marché et Thierry Hubert. Outre le fait que ma thèse n'aurait pu voir le jour sans leurs travaux, je les remercie pour leur collaboration amicale et extrêmement enrichissante pour moi. J'en profite pour remercier toute l'équipe Démons, ainsi que Benjamin Monate pour les discussions que nous avons partagées.

Je voudrais remercier tout particulièrement Mehdi-Laurent Akkar, pour notre amitié qui m'est devenue si précieuse et pour avoir toujours su me mettre de bonne humeur et me redonner le sourire à chaque fois qu'il s'était caché quelque part.

De sincères remerciements à Louis Goubin, avec qui on voudrait que les discussions n'aient jamais de fin. Merci aussi pour être parti moins loin que les autres !

Pour leur amitié unique et irremplaçable, merci infiniment à Johanne et Fabien.

Plus généralement, merci à tous les amis, en particulier à Paula et Fred pour leur joie de vivre et pour les moments de détente qu'ils nous ont offerts, à Patricia pour sa bonne humeur contagieuse et à Julien et Pierre-Yves pour les bons moments partagés au fin fond de la Bavière au milieu des bulles.

Une pensée emprunte de nostalgie pour celle avec qui j'ai découvert l'informatique dans les éclats de rire et avec qui on tente tant bien que mal à garder le contact. Merci Mika.

Je dois mes premiers pas en preuve formelle à Pierre Castéran, qui m'a également appris tous les fondements de la programmation et que je remercie pour sa pédagogie et son enthousiasme. J'en profite pour saluer mes compagnons d'armes sous le commandement de Proof General : Alexis et Séverine.

Ma thèse a été l'occasion d'une collaboration avec Louis de Cabissole, développeur chez Axalto, que je voudrais remercier chaleureusement pour son aide et sa gentillesse.

Un grand merci enfin à Quang Huy Nguyen et Nicolas Rousset, ainsi qu'à tous mes collègues d'Axalto qui n'ont pas encore été cités. Travailler avec vous tous est un réel plaisir. Merci en particulier à Antoinette pour être une vraie mère pour ses équipes, à Arno pour nos discussions de parents ébahis, à Marc pour sa bonne humeur (et sa gourmandise !) et Ludo pour son optimisme permanent face au comportement excentrique de la race des ordinateurs.

Je voudrais dire à mon Papa que je le remercie du fond du cœur pour m'avoir fait aimé les maths, m'avoir appris à regarder chaque problème à résoudre avec une IMMENSE sympathie, m'avoir passé ce coup de fil de l'autre bout de la terre sans lequel cette thèse n'aurait sans doute pas existé et pour être le seul de la tribu à ne pas tourner de l'œil à l'annonce de mon sujet de thèse ! Qu'il soit venu à bout de la lecture de mon mémoire me comble de fierté.

Muchísimas gracias a mi Mamacita para todo lo que representa para mi, para todo lo que hace sin decirlo, para su precensia tan fuerte aunque está lejos y para la misma pasión artística que corre en nuetras venas.

Je les remercie enfin tous les deux pour avoir su me faire grandir dans un cocon de bonheur, de rires et de voyages.

Merci à ma grande sœur Ketut pour avoir toujours su me montrer la voie (et avoir toujours supporté que j'en profite) et merci à son adorable famille.

Merci à mon Than pour tout ce qu'on a partagé et pour me rendre si fière de mon petit frère, de l'avenir qu'il se construit et du jeune homme qu'il est devenu.

Merci enfin à mes grand-parents et à toute ma joyeuse famille pour leur affection, et à toute ma belle-famille pour leur soutien et leur gentillesse.

Mes plus tendres pensées vont bien sûr à l'homme de ma vie, que je remercie profondément pour son soutien inconditionnel, sans lequel cette thèse n'aurait jamais pu aboutir. Je la lui dédie, ainsi qu'à mon petit bout d'Amour, mon rayon de soleil de tous les matins, qui ne m'a pas facilité la rédaction, mais dont les éclats de rire enchantent ma vie.

À Nico et Maël.
À la famille que nous dessinons.

Table des matières

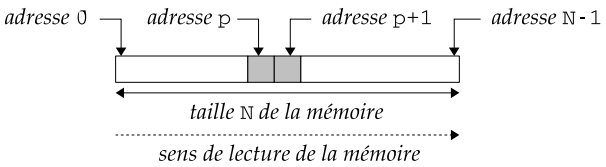
Table des figures

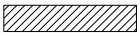
Conventions

1 Représentation de la mémoire

La mémoire d'un ordinateur stocke une suite de chiffres binaires (ou *bit*), dont l'écriture et la lecture se font octet par octet, c'est à dire par paquets de huit bits. Toutefois, pour plus de lisibilité dans le texte et plus particulièrement dans les figures, la mémoire sera représentée de façon séquentielle comme une suite d'octets, dont la lecture des adresses se fera de gauche à droite.

Convention 1 (Représentation de la mémoire)
La mémoire est représentée de façon séquentielle comme une suite d'octets, dont la lecture des adresses se fait de gauche à droite :

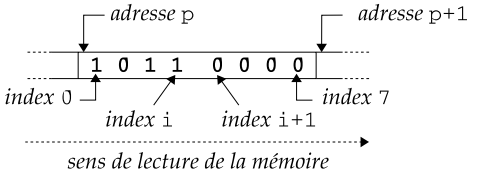


Convention 2 (Représentation d'une partie effacée de la mémoire)
De plus, une partie effacée de la mémoire sera représentée de façon hachurée :  *. Par exemple, en mémoire Flash, où l'état effacé correspond à 1, le bloc de mémoire* `111111111` *sera représenté par*  *.*

2 Représentation binaire

La représentation binaire d'un nombre entier correspond à sa décomposition en puissance de deux. Par exemple, le chiffre 13 correspond à $\mathbf{1} * 2^0 + \mathbf{0} * 2^1 + \mathbf{1} * 2^2 + \mathbf{1} * 2^3$. Il peut donc être noté $(1101)_b$ ou $(1011)_b$ suivant que l'on commence par le bit de poids fort ou celui de poids faible. Par cohérence avec la représentation de la mémoire, nous choisissons ici une écriture du bit de poids faible en premier (correspondant à une lecture de gauche à droite des puissances croissantes, et donc des index).

Convention 3 (Représentation binaire)
L'écriture de la représentation binaire d'un nombre entier se fera avec le bit de poids faible en premier. De plus, l'index, ou offset, d'un bit dans une représentation binaire commencera à être numéroté à partir de zéro. Par exemple, on pourra dire que les index entre 4 et 7 dans le chiffre binaire 10110000 *correspondent à des* 0*.*



3 Little Endian versus Big Endian

Comme déjà mentionné, l'écriture et la lecture d'un nombre entier en mémoire se font octet par octet. Par conséquent, lorsqu'un nombre entier est stocké sur plusieurs octets, il existe deux façons d'ordonner les octets en mémoire, donnant lieu à deux types de processeurs : soit l'octet de poids fort est stocké en premier (à l'adresse la plus petite) soit c'est l'octet de poids faible.

Par exemple, le nombre entier 512 dont la représentation binaire sur deux octets (selon la Convention 3) est $(0000\,0000\ 0100\,0000)_b$, doit être stocké en mémoire par un octet de valeur 0 et un octet de valeur 2. Si l'octet 2 est stocké en premier :

$$\begin{array}{lcl} \text{adresse} \quad \text{i} & \rightarrow & 2 \\ \text{adresse} \quad \text{i+1} & \rightarrow & 0 \end{array}$$

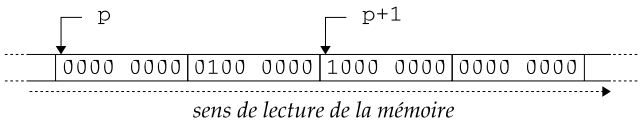
alors le processeur est dit *Big-Endian* (l'octet de poids fort – *big-end* – est stocké en premier). Dans le cas contraire, il est dit *Little-Endian* :

$$\begin{array}{lcl} \text{adresse} \quad \text{i} & \rightarrow & 0 \\ \text{adresse} \quad \text{i+1} & \rightarrow & 2 \end{array}$$

Le type d'architecture d'un processeur est indépendant d'une représentation papier de la mémoire. Toutefois, étant donné que nous représentons ici la mémoire de façon séquentielle comme une suite d'octets, nous devons indiquer comment seront représentés les nombres entiers stockés sur plusieurs octets. De même que nous avons fait le choix d'une représentation binaire avec le bit de poids faible en premier, nous allons choisir une représentation d'un nombre entier sur plusieurs octets avec l'octet de poids faible en premier (correspondant à une architecture Little-Endian). Par exemple, l'état de la mémoire à la fin de l'exécution du programme C suivant :

```
short *p ; *p=512 ; p++ ; *p=1 ;
```

sera représenté de la façon suivante :



Convention 4 (Représentation "Little-Endian" de la mémoire)
Lorsqu'un entier est stocké sur plusieurs octets en mémoire, il sera représenté avec son octet de poids faible en premier dans la mémoire.

4 Tailles des entiers C

Nous allons étudier le langage C, dans lequel sont définis plusieurs types d'entiers (`char`, `short`, `int`, `long`) de tailles différentes. Cette taille n'est pas spécifiée par la norme, seule une taille minimale en octet est spécifiée.

Convention 5 (Taille des entiers C)
Les entiers C seront représentés avec les tailles suivantes :

- `char` *: 1 octet*
- `short` *: 2 octets*
- `int` *: 4 octets*
- `long` *: 4 octets*

Introduction

Motivation

Depuis leur invention il y a près de 30 ans, les cartes à puce sont devenues omniprésentes dans des domaines aussi variés que la santé, les transports, la banque, la téléphonie mobile, l'identité, la fidélisation, etc. Ce marché grandissant doit faire face à des enjeux sécuritaires majeurs, en commençant par la maîtrise de la complexité croissante des programmes embarqués. L'augmentation de la complexité des logiciels en général rend leur sûreté de fonctionnement plus difficile à garantir (comme l'illustrent quelques exemples spectaculaires d'erreurs logicielles, comme l'explosion de la fusée Ariane 5 en 1996). Le domaine des cartes à puce n'y échappe pas. Bien que des vies humaines n'en dépendent pas, les cartes à puce constituent une clé d'accès au monde numérique et doivent garantir la protection des données qu'elles contiennent. Par ailleurs, elles sont dupliquées et émises à des millions d'exemplaires, si bien qu'une erreur logicielle importante aurait un coût tel que l'investissement dans des techniques de renforcement du niveau de sûreté de leurs programmes est justifié.

A cela s'ajoutent les enjeux sécuritaires induits par les nouvelles plate-formes embarquées dans les cartes à puce : les plate-formes *multi-applicatives et ouvertes*. Ces plate-formes comportent une *machine virtuelle* pour une meilleure portabilité et permettent à une même carte d'héberger plusieurs applications différentes, pouvant provenir de différents fournisseurs et pouvant même être chargées sur la carte après sa mise en circulation. L'architecture doit donc assurer une exécution fiable d'applications dont la provenance ne l'est pas nécessairement. En particulier, elle doit garantir, d'une part, la protection de la plate-forme et, d'autre part, la protection des données d'une application vis-à-vis des autres applications. Ces nouvelles architectures complexes répondent aux besoins du marché, mais créent par la même occasion de nouveaux risques potentiels de sécurité, qui doivent être écartés par des analyses sécuritaires minutieuses.

Enfin, les caractéristiques physiques mêmes de la carte doivent être prises en compte. Par exemple, une des particularités de la carte à puce est de devoir assurer un bon fonctionnement même en cas de son retrait prématuré du terminal, appelé un *arrachage*, provoquant sa mise hors tension soudaine. Les programmes embarqués doivent donc prendre en compte, dans leur spécification et dans leur implémentation, la possibilité d'une interruption de leur exécution à tout moment. Un certain nombre de mesures, parfois complexes, sont mises en place pour garantir une cohérence du système lors de sa remise sous tension après une interruption. La *correction*[1] de ces mesures vis-à-vis de ce qu'elles doivent assurer est primordiale. Par ailleurs, les ressources limitées de la carte imposent généralement des algorithmes de gestion de la mémoire qui sont optimisés et de bas niveau. La correction de tels algorithmes peut être difficile à établir. En outre, le caractère limité des ressources restreint l'utilisation de certaines méthodes de renforcement du niveau de sûreté des programmes (essentiellement celles nécessitant une modification du code,

[1] Le mot *correction* est utilisé, dans le contexte de la vérification de programme, dans le sens d'être *correct* et non de *corriger*.

voire sa génération à partir de modèles, et devant faire face à des problèmes d'optimisation).

Dans ce contexte, notre objectif est de développer et d'utiliser des *méthodes formelles* pour, d'une part, la modélisation et la vérification de propriétés *sécuritaires* de haut niveau et, d'autre part, la preuve de la *correction* des programmes embarqués dans les cartes à puce.

Objet des travaux

Nous nous sommes intéressés, dans nos travaux, à deux aspects de la sécurité des cartes à puce :

- prouver des propriétés sécuritaires de la plate-forme *Java Card* concernant l'isolation des applications chargées sur une même carte ;
- définir une méthodologie de vérification, à la fois fonctionnelle et sécuritaire, d'un code source de bas niveau d'un système d'exploitation embarqué, illustré ici par un module de gestion de mémoire *Flash*.

La technologie Java Card (voir [93] ou [34]) est une plate-forme multi-applicative et ouverte, où la sécurité inhérente à Java est renforcée par un mécanisme complexe, appelé *pare-feu* ou *firewall*, permettant d'assurer la protection des données d'une application vis-à-vis des autres applications. Plus précisément, excepté dans des cas spécifiques et bien déterminés, les données d'une application ne peuvent pas être accédées par d'autres applications de la carte. Cette *isolation* des données des différentes applications est assurée, à l'exécution, par un contrôle d'accès effectué par le *firewall*. Ce mécanisme consiste à définir, pour chaque instruction élémentaire du langage Java Card (dite instruction *bytecode*[1]), un ensemble de règles qui doivent être respectées pour que l'instruction puisse être interprétée (sans levée d'exception) par la machine virtuelle.

Nos travaux ont consisté à prouver formellement, dans le système de preuve *Coq* (voir [113]), que cet ensemble de règles assure la *confidentialité* et l'*intégrité* des données d'une application, vis-à-vis des autres applications de la carte. Autrement dit, nous avons prouvé que les autres applications ne peuvent obtenir *aucune information* (ce qui est formalisé par une notion de *non-interférence*) et ne peuvent modifier aucune donnée appartenant à cette application. Cette propriété a été prouvée pour tout le processus d'exécution d'une commande reçue du terminal, comprenant les opérations du *card manager* (qui gère les ressources de la carte, la communication avec le terminal, etc), l'interprétation de la machine virtuelle et les appels potentiels à des méthodes de l'interface de programmation (API) de Java Card. Ces travaux sont décrits plus en détail dans la prochaine section.

Puis nous nous sommes intéressés à étendre la modélisation et la vérification formelles aux programmes du système d'exploitation. L'enjeu est alors de formaliser un code industriel existant, pour une meilleure maîtrise des composants critiques du système. C'est le cas de la mémoire Flash, nouvelle technologie améliorant les capacités de stockage de la carte, mais qui nécessite une gestion complexe et délicate de la mémoire. En effet, l'effacement des données n'est possible, en Flash, que par blocs entiers de mémoire, ce qui implique que les mises à jour de données *en place* (i.e. au même emplacement mémoire) sont impossibles en général. Les données modifiées doivent donc être copiées dans un emplacement fraîchement effacé de la mémoire. De plus, la mémoire Flash ne peut être effacée qu'un nombre limité de fois. Le choix de l'emplacement effacé à utiliser doit donc être optimisé, pour un bon *nivellement de l'usure*. Des algorithmes

[1] Le langage Java Card est un sous-ensemble du langage Java, qui peut être compilé, puis converti en instructions *bytecode*, similaires à des instructions assembleur, qui seront interprétées une par une par la machine virtuelle (voir Section 1.2.1 et par exemple [34]).

sophistiqués de mise à jour des données, comme la *journalisation*, sont donc utilisés pour la gestion de la mémoire Flash.

Dans ce contexte, nos travaux s'intéressent à la vérification de propriétés à la fois fonctionnelles et sécuritaires de ces algorithmes de gestion de la mémoire, et plus généralement des programmes C de bas niveau d'un système d'exploitation de carte à puce.

La vérification fonctionnelle consiste à utiliser un outil de vérification de programme pour prouver la correction du code source. Un tel outil génère automatiquement, à partir d'un programme *annoté* (i.e. un programme contenant, dans son code source, la spécification de chacune de ses fonctions), l'ensemble des conditions à prouver, appelées *obligations de preuves*, pour assurer que le code de chaque fonction vérifie sa spécification annotée. Les obligations de preuves peuvent alors être fournies à un assistant de preuve, pour une preuve interactive, ou à une procédure de décision, pour une preuve automatique. Nous avons utilisé l'outil *Caduceus* (voir [52, 53]) pour spécifier chaque fonction du module de gestion de mémoire Flash et nous avons prouvé, à l'aide de l'assistant de preuve Coq, les obligations générées par l'outil. Cette vérification assure, en particulier, la correction de l'algorithme d'effacement proposé par le module, utilisant la journalisation et assurant la cohérence de la mémoire même après un arrachage de la carte (propriété dite d'*anti-tearing*).

Lors de nos travaux sur ce cas d'étude, nous avons été confrontés aux limitations de l'outil Caduceus vis-à-vis des constructions du langage C qui sont prises en compte. En particulier, les *unions* ou les *casts* ne sont pas gérés par Caduceus, alors qu'ils sont utilisés par le code source embarqué. Nous proposons donc des extensions de l'outil pour la gestion de ces constructions. Nous proposons également une extension de Caduceus qui permet de prendre en compte certaines caractéristiques de la carte à puce. Elle permet, plus précisément, de spécifier et de vérifier le comportement d'une fonction en cas d'arrachage de la carte.

Nous avons également été limités dans le type de propriétés qui pouvaient être exprimées dans les annotations, à savoir des propriétés fonctionnelles et locales aux fonctions spécifiées. Or, la vérification de propriétés globales, combinant plusieurs fonctions, ou de propriétés sécuritaires (que nous appellerons plus généralement des propriétés de *haut niveau*) fait également partie de nos objectifs. La définition de ce genre de propriétés nécessite une modélisation du programme sous la forme d'un *système de transitions*. Notre idée fut alors d'utiliser le modèle formel calculé par l'outil Caduceus, à partir de la spécification annotée, pour en extraire automatiquement un système de transitions. Cette approche permet d'avoir un lien formel entre le code et le modèle, puisque le modèle est extrait de la spécification formelle et que l'on a une preuve formelle que la spécification est vérifiée par le code.

La suite de l'introduction décrit plus en détail les travaux effectués. Premièrement, les travaux de formalisation et de preuve des propriétés de confidentialité et d'intégrité des applications Java Card sont expliqués. Puis, les différents aspects de la vérification de programmes C embarqués sont détaillés. En particulier, les solutions proposées pour la gestion de constructions de bas niveau du langage C sont mises en évidence et la méthodologie de vérification de *haut niveau* est présentée.

Preuve de la confidentialité et de l'intégrité dans Java Card

De nombreux travaux ont pour but d'analyser et de vérifier diverses propriétés de la technologie Java Card, soit du point de vue de la plate-forme elle-même, soit au niveau des applications développées dans le langage Java Card.

L'analyse du code source d'applications Java Card permet de prouver certaines propriétés,

comme la correction du programme vis-à-vis d'une spécification donnée (voir par exemple [115, 116], où les méthodes de l'API Java Card sont spécifiées en *JML* [86] et prouvées avec l'outil LOOP [88]). Elle permet également de vérifier qu'une application respecte une politique de sécurité donnée. En ce qui concerne la propriété d'isolation des applications, une analyse statique d'une application permet de vérifier si elle respecte la politique de sécurité imposée par la plate-forme, comme le propose le système de types défini dans [43], et d'éviter une erreur à l'exécution due à une violation des règles du *firewall*. Par ailleurs, la politique de sécurité de la plate-forme autorise le partage de données entre deux applications, à travers les *interfaces partageables*, et n'interdit pas la retransmission des données à une tierce application. Ces flux d'information "transitifs" sont étudiés dans [19], qui propose une nouvelle politique de sécurité, basée sur des niveaux de sécurité et n'autorisant que certains flux. Citons également [47], qui propose un système de contraintes permettant de calculer l'ensemble des flux d'information entre différentes applications données.

Les travaux cités jusqu'à présent se concentrent sur la vérification d'applications, sous l'hypothèse que la plate-forme est correcte. Nos travaux s'intéressent, au contraire, à la plate-forme elle-même, indépendamment des applications qui seront hébergées sur la carte. Dans cette approche, les études formelles de la plate-forme Java Card se sont essentiellement concentrées sur la preuve de propriétés du *vérificateur de bytecode*, composant essentiel de la technologie Java Card, en charge principalement d'un contrôle de types. En particulier, la propriété de *sûreté de typage* a été prouvée, utilisant soit une modélisation formelle du vérificateur (voir [32, 83, 84, 87]), soit une modélisation de deux machines virtuelles, l'une "typée" et l'autre non, et par la preuve qu'elles coïncident sur les programmes qui vérifient les règles du typage (voir [12, 45]). La plate-forme a également été modélisée dans un but de certification dans [18] et dans [13], à l'aide du système de preuves Coq. De notre côté, nous nous sommes intéressés à la politique de sécurité du *firewall* de la plate-forme Java Card. Cette politique a été modélisée dans [96], avec la méthode B (voir [1]) pour prouver que tout accès est soumis au contrôle des règles du *firewall*.

L'objectif de nos travaux est de vérifier formellement que ces règles assurent la confidentialité et l'intégrité des données d'une application vis-à-vis des autres applications. Cela signifie qu'il faut prouver, d'une part, qu'une application ne peut obtenir *aucune information* d'une autre application de façon illégale (i.e. en n'utilisant pas les mécanismes de partage spécifiques), et d'autre part, que les données d'une application ne sont jamais modifiées illégalement par une autre application chargée sur la même carte.

La vérification formelle de cette *isolation* des données nécessite tout d'abord de modéliser formellement la plate-forme Java Card. Nous avons utilisé dans nos travaux une modélisation existante, développée dans le projet FORMAVIE (voir [18]). Dans cette modélisation en Coq, la plate-forme est représentée par une machine à états, où les transitions représentent toutes les transformations possibles de l'état global de la carte lors de l'exécution de commandes reçues du terminal. Ce modèle a été utilisé pour formaliser les notions de confidentialité et d'intégrité des données des applications. En particulier, l'impossibilité d'obtenir une quelconque information sur une donnée a été formalisée par une propriété de *non-interférence* (voir [56, 57, 110]). Ce principe consiste à considérer que les données d'une application sont confidentielles si la trajectoire du système (en tant que machine à états) lors de l'exécution d'une commande par une autre application n'en dépend pas. Une fois les propriétés de confidentialité et d'intégrité formellement définies, elles ont été prouvées de manière interactive dans le système de preuves Coq et les preuves ont été vérifiées mécaniquement par le moteur du système. Les preuves représentent environ 30 000 lignes de Coq, sachant que la définition du modèle en compte environ 125 000.

Le développement de la preuve a mis en évidence un certain nombre de "conditions d'isola-

tion", à savoir des propriétés sur lesquelles reposent la preuve mais qui n'ont pas pu être prouvées en raison du niveau de détails de la modélisation. Plus précisément, les modèles utilisés sont ceux de la spécification de Sun[1] de la plate-forme. Or, cette spécification ne décrit pas certains détails d'implémentation, qui ne sont donc pas représentés dans les modèles. Les conditions d'isolation concernent également des conditions sécuritaires d'utilisation de méthodes de l'API, qui n'apparaissent pas dans leurs spécifications. La preuve a donc permis de compléter les spécifications de l'API d'un point de vue sécuritaire et d'identifier les conditions supplémentaires à vérifier au niveau de l'implémentation.

Ces travaux mettent en évidence un problème crucial de la modélisation formelle, à savoir le lien qui existe entre le modèle formel et l'implémentation. En effet, pour assurer qu'un code source vérifie certaines propriétés, il faut prouver qu'il implémente correctement le modèle formel sur lequel la vérification de ces propriétés a été faite. Ce point critique est un des critères qui a été pris en compte dans nos travaux sur la vérification du système d'exploitation.

Vérification fonctionnelle et sécuritaire de programmes C embarqués

L'étude formelle d'un système nécessite sa modélisation et la formalisation des propriétés visées. Pour cela, différents outils existent, le choix devant être guidé par les caractéristiques du système (s'il réagit à des événements, s'il possède une infinité de comportements possibles, etc) et par le type de propriétés visées (temporelles, fonctionnelles, sécuritaires, etc). En particulier, une vérification *fonctionnelle de bas niveau* et une vérification dite de *haut niveau* (de propriétés globales, temporelles, sécuritaires, etc) ne suivent pas la même approche.

Vérification fonctionnelle et extensions de l'outil Caduceus.

Une première approche consiste à vérifier le code source du programme, grâce à des outils de vérification de programmes. Ces outils permettent de prouver que le code implémente correctement une spécification donnée, en générant des *obligations de preuves*, qui sont les conditions assurant cette correction. Cela permet une vérification *fonctionnelle de bas niveau* du programme, i.e. la vérification de propriétés locales à chaque fonction. Cette approche a été utilisée pour la vérification de la correction du module de gestion de mémoire Flash, grâce à l'outil Caduceus.

Cette vérification a nécessité des extensions de l'outil, de manière à prendre en compte les spécificités du code embarqué sur une carte à puce, comme la problématique de l'arrachage, ainsi que l'utilisation de constructions de bas niveau. Caduceus a d'une part été étendu de manière à pouvoir spécifier et prouver les propriétés que vérifient certaines fonctions en cas d'interruption de leur exécution (par un arrachage de la carte). D'autre part, un travail majeur a permis de proposer des solutions pour la gestion des constructions de bas niveau du langage C, comme les *unions* et les *casts*, dont la sémantique (voir par exemple [41]) est délicate à formaliser. Ces constructions ne sont d'ailleurs généralement pas traitées dans les études formelles sur le langage C (comme la formalisation du langage en HOL, présentée dans [104], où les unions ne sont pas traitées et où les casts sont modélisés à l'aide d'une fonction de conversion supposée fournie) ou par les outils de vérification de programmes C (comme Caveat [33] ou Caduceus).

Un type union peut être utilisé à la manière d'un type "somme" (où plusieurs types différents sont rassemblés dans un même type), mais également comme un moyen d'avoir plusieurs interprétations d'une même donnée. Dans ce dernier cas, il permet de lire et modifier un même bloc mémoire de plusieurs façons différentes. De nature similaire, l'opération de cast permet de

[1] Sun est l'inventeur du langage Java et l'auteur des différentes spécifications de Java Card.

convertir une variable d'un certain type en un autre type. La particularité de ces constructions est qu'elles sont très dépendantes du compilateur et du processeur. Leur sémantique n'est pas totalement décrite dans la norme du langage (norme *ANSI-C*). En outre, la principale difficulté de leur formalisation provient du fait qu'elle nécessite une vision très bas niveau de la mémoire. Or, les outils de vérification de programmes C choisissent en général un modèle mémoire de suffisamment haut niveau pour que les preuves ne soient pas impraticables.

Par exemple, l'outil Caduceus fait un choix de modèle mémoire qui s'inspire d'une approche de Burstall et Bornat (voir [22, 29]), où la mémoire est représentée, non pas comme un grand tableau d'octets, mais comme l'union de blocs mémoire disjoints (correspondant aux pointeurs alloués et aux champs de structures et d'unions). Cela permet de minimiser la difficulté des preuves à fournir, car, lorsqu'une modification intervient dans un bloc, il n'est pas nécessaire de prouver que tous les autres blocs sont inchangés (alors que si la mémoire est un grand tableau d'octets, on doit indiquer l'effet de la modification sur chaque case du tableau). Le problème est que ce genre de modèle ne permet pas de représenter des constructions de bas niveau, telles que les unions ou les casts, où certains blocs ne peuvent plus être considérés comme disjoints. Nous proposons donc une analyse des limitations d'un modèle à la Burstall et Bornat et un ensemble de solutions possibles pour adapter ce modèle afin de gérer de telles constructions, tout en préservant le plus possible les avantages de la séparation de la mémoire.

Méthodologie de vérification *haut niveau*

Une autre approche de vérification consiste à construire un modèle dit de *haut niveau*, sous la forme d'un système de transitions, représentant les comportements du programme. Des propriétés complexes, comme des propriétés temporelles ou sécuritaires (telle que la confidentialité, dont la formalisation nécessite une quantification sur des exécutions), peuvent alors être exprimées et prouvées sur ce modèle. L'inconvénient de cette approche est que les propriétés sont vérifiées sur un modèle abstrait et non sur le code source du programme. Seul un lien formel entre le code et le modèle permet d'assurer que les propriétés sont réellement vérifiées par l'implémentation. Ce lien formel peut être assuré par la génération du code source à partir du modèle formel (avec la méthode B [1] ou avec Esterel [16]). Seulement, la génération de code n'est actuellement pas assez optimisée pour des programmes de bas niveau de système d'exploitation comme ceux gérant la mémoire. La génération du modèle à partir du code assure également un lien formel, mais peut être fastidieuse.

Nous proposons de combiner l'approche de vérification du code source et l'approche de vérification d'un système de transitions pour pouvoir prouver des propriétés de haut niveau sur du code source. La définition de propriétés de haut niveau sur du code source a également été étudiée dans [67, 11, 15], pour des programmes Java (ou Java Card). Ces travaux proposent une méthode pour exprimer des propriétés temporelles dans le langage de spécification JML, de manière à les prouver sur le code source par des outils classiques de vérification de programmes Java annotés.

Notre approche est différente : au lieu d'exprimer les propriétés dans un langage de bas niveau pour une vérification sur le code, nous proposons d'utiliser les annotations pour décrire le code source (ou une abstraction) et d'extraire de ces annotations un système de transitions sur lequel la vérification sera faite. L'idée majeure est d'utiliser un outil de vérification pour, d'une part, bénéficier du travail de modélisation des états mémoires du programme, pour extraire *automatiquement* (et formellement) le système de transitions. D'autre part, la preuve des obligations générées par l'outil assure que les annotations définissent un modèle correct du programme et garantit donc que le système de transitions est en relation de simulation avec le code. Ce sys-

tème peut alors être plongé dans une logique d'ordre supérieur, apportant toute l'expressivité nécessaire à la définition de propriétés complexes.

Plus précisément, un outil de vérification de programme se sert d'un modèle interne de la mémoire pour construire un modèle formel du programme annoté. Globalement, chaque fonction du programme est représentée par une transformation de l'état global de la mémoire et sa spécification est également traduite en termes de propriétés sur ces états. Les obligations de preuves correspondent alors à des conditions suffisantes pour que ces transformations vérifient les propriétés décrites dans les annotations. Ce modèle est généralement implicite, en ce sens que seules les obligations de preuves sont fournies à l'utilisateur. Notre idée est, au contraire, de bénéficier du calcul des états de la mémoire pour modéliser chaque fonction comme une transition entre ces états.

Cette méthodologie repose donc sur un outil de vérification de programmes C, où le modèle formel implicitement construit est "accessible". D'où notre choix de l'outil Caduceus, qui fournit un modèle formel du programme et une traduction de sa spécification, en fonction des états mémoire du programme.

Cette méthodologie a été utilisée pour la vérification de propriétés globales du module de gestion de mémoire Flash embarqué dans la carte. Nous avons, par exemple, prouvé la propriété d'*anti-tearing* de l'effacement : si la fonction d'effacement est interrompue par un arrachage, alors la fonction de *montage*, appelée à la remise sous tension, assure que l'effacement sera de nouveau effectué, afin que la carte soit de nouveau dans un état cohérent.

Plan de la thèse

Le Chapitre 1 présente le contexte général de nos travaux dans le domaine de la carte à puce. Il propose une synthèse des principales caractéristiques et des principaux enjeux sécuritaires des cartes à puce (pour en savoir plus, voir par exemple [108]). Puis il pose les bases des deux cas d'étude de nos travaux, à savoir d'une part le mécanisme de contrôle d'accès aux données de la plate-forme Java Card (décrit en détail dans le livre [34] ou dans la spécification de Sun [81]) et d'autre part la gestion d'une mémoire Flash embarquée dans un système d'exploitation de carte à puce (inspirée de [117], voir aussi [55]).

Le Chapitre 2 dépeint les principales notions des méthodes formelles, et en particulier de la vérification formelle de programme. Puis les principaux travaux utilisant les méthodes formelles dans le cadre de la carte à puce sont présentés.

Le Chapitre 3 décrit nos travaux sur la plate-forme Java Card (également présentés dans [4, 6, 5]), à savoir la formalisation et la vérification formelle de la confidentialité et de l'intégrité des données d'une application, vis-à-vis des autres applications embarquées sur la même carte.

Le Chapitre 4 présente l'utilisation de l'outil Caduceus (voir [52, 53]) pour la vérification fonctionnelle de programmes C et les extensions de cet outil de manière à prendre en compte les caractéristiques de la carte à puce, comme l'arrachage, et à adapter le modèle mémoire pour gérer des constructions du langage C telles que les unions ou les casts.

Le Chapitre 5 présente notre méthodologie de vérification de propriétés dites de *haut niveau* (propriétés globales, temporelles, sécuritaires, etc) sur un modèle formellement lié au code source, utilisant le calcul du modèle mémoire du programme construit par l'outil Caduceus (méthodologie également présentée dans [7, 3])

Le Chapitre 6 décrit la vérification fonctionnelle et sécuritaire de notre cas d'étude : le module de gestion de la mémoire Flash du système d'exploitation de la carte à puce.

Lectures transversales

Suivant ses connaissances et ses centres d'intérêts, le lecteur pourra lire ce manuscrit de plusieurs façons non séquentielles :

Java Card. Si le lecteur est intéressé par les travaux sur Java Card :

- les principes de Java Card, et en particulier le mécanisme de *firewall*, pourront être trouvés en Section 1.2 (page 32) ;
- les travaux existants dans ce domaine sont exposés en Section 2.2.2.1 (page 70) ;
- nos travaux de formalisation et de vérification formelle de la confidentialité et de l'intégrité des données sont présentés dans le Chapitre 3 (page 77).

Mémoire Flash. Pour une lecture sur les travaux sur la preuve de correction du module de gestion de la mémoire Flash :

- l'architecture générale du module est décrite en Section 1.3 (page 37), et plus particulièrement en Section 1.3.4 (page 50) – le reste étant une introduction aux concepts et aux difficultés d'utilisation de la mémoire Flash.
- la vérification formelle du module fait l'objet du Chapitre 6 (page 161), qui décrit la spécification formelle des principaux composants du module (gestion d'un effacement en cas d'arrachage), la preuve formelle de la correction du code source vis-à-vis de cette spécification et la vérification formelle de propriétés de plus haut niveau.

Vérification de Propriétés de Haut Niveau sur le Code Source. Si le lecteur est intéressé par la méthodologie de preuve permettant de vérifier des propriétés de haut niveau à partir d'un code source en langage C, le Chapitre 5 (page 149) lui est consacrée. Le coeur de la solution est expliqué en Section 5.2.1 (page 154).

Preuve de Programmes C. Enfin, la preuve de programmes C, et en particulier la formalisation de la sémantique des constructions du langage tels que les unions ou les casts, sont présentées dans le Chapitre 4 (page 101) et plus précisément dans la Section 4.5 (page 131).

Coq et Caduceus. Notons enfin que l'outil Coq est brièvement présenté en Section 2.1.4 (page 62), et que l'outil Caduceus et son modèle mémoire sont expliqués dans la Section 4.2.1 (page 103).

Chapitre 1

Contexte de la carte à puce

Résumé

Ce chapitre donne une introduction détaillée à la carte à puce, de sa création (voir les brevets [95, 114, 42, 46, 59]), à ses caractéristiques actuelles (voir [108]), normalisées par de nombreux standards (voir [74, 70, 72]). Ceci permet de mettre en évidence les principaux enjeux sécuritaires qui ont motivé nos travaux, clairement annoncés dans la Section 1.1.6. Enfin nous expliquerons le contexte général de ces travaux, à savoir, d'une part les mécanismes sécuritaires de la plate-forme Java Card (voir [81, 34]), et d'autre part la gestion par *journalisation* (inspirée de [117]) d'une mémoire Flash embarquée.

Sommaire

1.1 Introduction aux cartes à puce

La carte à puce, dont l'idée s'est progressivement dessinée depuis les années cinquante, est en circulation depuis maintenant presque trente ans[1]. L'idée d'utiliser la nouvelle technologie des circuits électroniques, pour faire cohabiter un moyen de stockage d'information et une puissance de calculs permettant une manipulation contrôlée et sécurisée des données, a donné naissance à un nombre insoupçonné d'applications. Les cartes à puce permettent aujourd'hui de remplacer la monnaie, de se connecter à un réseau de téléphonie mobile, d'accéder à des locaux, d'utiliser les transports en commun, de s'identifier auprès des services de douanes, etc. Les nouvelles architectures de cartes à puce, pouvant accueillir des applications aussi variées, sur une même carte, même après la mise en circulation de la carte, doivent faire face à de nouveaux enjeux, en particulier sécuritaires, et ce à différents niveaux : au niveau applicatif, au niveau de la machine virtuelle, au niveau du système d'exploitation et au niveau matériel.

Dans cette section, nous allons retracer les étapes d'innovation qui ont mené à la carte à puce d'aujourd'hui, afin de comprendre les motivations de son utilisation et de sa définition actuelles. Une description détaillée de l'architecture d'une carte à puce permettra alors de comprendre le fonctionnement et les enjeux sécuritaires de ses différents composants. Nos préciserons en Section 1.1.6 l'objet de nos travaux, à savoir l'étude d'un point de vue sécuritaire de deux composants logiciels majeurs d'une carte à puce :

- la machine virtuelle, ici celle de Java Card, et en particulier son mécanisme de *firewall*, dont nous voulons prouver formellement qu'il assure le principe d'isolation des applications ; la plate-forme Java Card et son firewall seront décrits en détail dans la Section 1.2.
- le système d'exploitation, illustré ici par un module de gestion de mémoire Flash, mettant en œuvre un mécanisme de *journalisation* pour faire face aux contraintes d'utilisation de la mémoire Flash ; les caractéristiques de cette mémoire seront présentés dans la Section 1.3, ainsi que le principe de journalisation et le module du système d'exploitation étudié.

1.1.1 Historique de l'invention de la carte à puce

Carte de crédit. L'utilisation de cartes pour remplacer la monnaie ou accéder à des services est très ancienne. Au début du vingtième siècle, des cartes en carton étaient utilisées à Paris pour accéder aux cabines téléphoniques publiques. Dès 1914 aux États-Unis, la compagnie de transfert d'argent Western Union émet une plaque en métal pour fidéliser sa clientèle. L'année suivante, la Compagnie du Télégraphe utilisera des plaques métalliques dans le but d'identifier ses clients et d'authentifier leurs télégrammes.

Les cartes de paiement apparaissent aux États Unis dans les années cinquante. On pourrait attribuer leur invention à l'embarras d'un homme d'affaire new-yorkais, Frank McNamara, qui en 1949 se voit obligé, faut d'argent, de demander à sa femme de régler l'addition dans un grand restaurant. Il déplore qu'une personne pouvant se permettre de faire une dépense soit limitée par le montant qui se trouve être dans son porte-feuille. Il propose alors la possibilité de dîner dans une sélection de restaurants avec une simple signature et un paiement différé. C'est ainsi qu'en 1950 apparaît la première carte de crédit *Diners' Club* (voir Figure 1.1). Il s'agit alors d'une carte en papier, sur laquelle sont imprimées des informations générales, telles que le nom du détenteur de la carte. Utilisée par 200 personnes privilégiées dans 14 restaurants lors de sa création, elle compte 20 000 clients et est acceptée dans 1 000 restaurants une année plus tard.

Le succès de cette carte est suivie par l'apparition en 1958 de la carte American Express, carte de loisirs et de voyage, qui, pour la première fois, est en plastique. Les données personnelles

[1] La carte à puce a en fait mon âge.

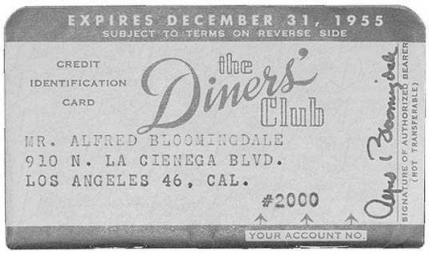


FIG. 1.1 – La carte de crédit *Diners' Club*

du détenteur, telles que son nom et son numéro de compte, sont alors estampées en relief sur la carte, sur laquelle le détenteur a également apposé sa signature. L'utilisation d'un appareil simple et peu coûteux permet alors de transférer ces données sur papier et ce reçu devra être signé par le détenteur de la carte. Bien que primitive, cette technique, par sa simplicité, a connu un très grand succès jusqu'à aujourd'hui.

Le phénomène de la carte de crédit prend alors une ampleur internationale dans les années soixante. Aux États Unis, huit banques américaines vont mettre en circulation la *BankAmericard* (qui s'internationalisera ensuite et deviendra le réseau Visa). A la même époque, six banques françaises créent la première carte de paiement en France : la Carte Bleue.

Bandes magnétiques. Les années soixante-dix voient une grande avancée avec l'introduction d'une bande magnétique sur les cartes de crédit. Déjà utilisées sur des tickets de transport, comme dans le métro londonien, les bandes magnétiques offrent la possibilité de stocker de l'information, sous la forme d'une intensité de champ magnétique. L'identification du détenteur passe alors par un numéro d'identification personnel (*PIN, Personal Identification Number*). En France, la première carte comportant une piste magnétique a été mise en service en 1971 par l'organisation Carte Bleue, en même temps que l'installation des premiers distributeurs automatiques de billets.

Permettant d'éliminer les reçus en papier, les bandes magnétiques n'offrent cependant pas une grande marge de manœuvre ; les informations qu'elles contiennent peuvent être uniquement lues, sans possibilité de modification. Par ailleurs, elles ne sont pas fiables d'un point de vue sécuritaire puisque leur contenu peut être facilement lu, reproduit ou effacé (sous l'effet d'un très fort champ magnétique). Par conséquent, le PIN sera mémorisé dans l'ordinateur central du système plutôt que sur la bande magnétique, impliquant une connexion en ligne avec l'ordinateur central à chaque opération.

Le souci de réduire les coûts élevés de transmission de données, ainsi que d'élargir les possibilités d'échange d'informations et de renforcer le contrôle d'accès aux données, commence à apparaître.

Clé des Gondas. Le concept majeur de la carte "intelligente", qui implique à la fois un moyen de stockage et un moyen de calcul, va se dessiner petit à petit. En 1968 déjà, dans son roman *La Nuit des Temps* [9], René Barjavel avait imaginé une bague multi-fonctionnelle avec des moyens de mémorisation et de communication. Il y décrit une civilisation qui aurait vécu sur Terre il y a 900 000 ans et dont les connaissances scientifiques et technologiques auraient été très avancées. Chaque habitant de Gondawa possédait une bague lui servant, entre autre, de moyen de paiement, d'accès à des services, de localisation, ou encore d'accès à certains lieux. "Chaque fois qu'un Gonda désirait quelque chose de nouveau, des vêtements, un voyage, des objets, il

payait avec sa clé. Il pliait le majeur, enfonçait sa clé dans un emplacement prévu à cet effet et son compte, à l'ordinateur central, était aussitôt diminué de la valeur de la marchandise ou du service demandés."

Carte à mémoire. Le *circuit intégré*, ou *puce électronique*, mis au point en 1958 par Texas Instruments, est un ensemble de composants électroniques reliés entre eux par des connecteurs semi-conducteurs (généralement du *silicium*) permettant de stocker et manipuler des données sous forme binaire, donnant naissance aux mémoires à semi-conducteurs et aux microprocesseurs.

L'idée d'utiliser ce support électronique dans les cartes de crédit est apparue presque simultanément à travers le monde, avec le dépôt de nombreux brevets, pour la plupart nationaux. C'est en 1967 que deux ingénieurs allemands, Jürgen Dethloff and Helmut Grötrupp, annoncent l'intégration d'un circuit intégré sur une carte en plastique. Étonnamment, leur brevet (n°DE1945777C3), déposé en 1969 en Allemagne n'est publié qu'en 1982. Indépendamment, un brevet similaire est déposé au Japon en 1970 par Kunitaka Arimura. La même année, l'américain Jules K. Ellingboe propose, dans son brevet intitulé *"Active Electrical Card Device"* (voir [46]), une carte à contacts permettant le paiement électronique une fois insérée dans un terminal. En 1973, les anglais John W. Halpern et William Ward décrivent un *"composant portatif de données digitales"*, pouvant prendre la forme d'un stylo et pouvant servir de ticket de transport ou de moyen d'identification (voir le brevet [59]).

Mais ce n'est qu'en 1974 que l'idée d'une carte à circuit intégré s'est réellement consolidée avec la réalisation d'un prototype par le français Roland Moreno. Créateur de la société Innovatron (société internationale pour l'innovation), Roland Moreno réalise, en janvier 1974, le premier prototype *"d'objet portatif à mémoire intelligente"*, sous la forme d'une bague (voir Figure 1.2). L'idée est d'utiliser de la mémoire PROM (Programmable Read-Only Memory[1]), qui vient d'être

FIG. 1.2 – Le prototype d'objet portatif à mémoire intelligente de Roland Moreno

mise au point par la firme Texas Instruments. Cette mémoire est constituée de milliers de fusibles pouvant être "grillés" un à un. Utilisant ce type de mémoire, le prototype proposé par Roland Moreno offre la possibilité de gérer un compte en banque et d'enregistrer des transactions. En effet, le prototype contient trois mémoires de ce type : une mémoire d'identification, une mémoire de débit et une mémoire de crédit. Des moyens inhibiteurs de transfert et d'inscription assurent la confidentialité et l'intégrité du contenu de certaines sections de la mémoire. En particulier, ils assurent que la mémoire d'identification n'est accédée qu'en lecture, excepté à l'initialisation de la carte à sa création. Les mémoires de crédit et de débit sont, quant à elles, vierges à l'origine et les transactions sont "brûlées" de manière séquentielle dans la mémoire correspondante. Par exemple,

[1] Voir Section 1.3.1 pour plus d'information sur les différents types de mémoires.

une transaction correspondant à un débit consiste en trois étapes principales : premièrement l'identification, qui consiste à comparer un numéro composé par le porteur de la carte aux données de la mémoire d'identification, puis l'autorisation, qui consiste à comparer le montant du débit demandé au montant du compte (calculé en comparant la somme de tous les crédits et la somme de tous les débits enregistrés) et enfin l'inscription de ce dédit dans la première page vierge de la mémoire de débit en brûlant celle-ci.

En mars 1974, Roland Moreno dépose en France le brevet de base couvrant cette technique (voir [95]), suivi de nombreux autres brevets internationaux. Il est reconnu comme l'inventeur de la *carte à puce*, la *puce* désignant un circuit intégré utilisé ici pour stocker de la mémoire.

Carte intelligente. Le but devient alors de sécuriser les données contenues dans la mémoire de la carte. Ainsi, le même Jürgen Dethloff dépose en 1976 aux États-Unis un brevet décrivant une carte avec un microcircuit contenant de la mémoire, servant de moyen de paiement ou d'identification, incluant un numéro secret d'utilisateur (voir [42]).

Mais la percée majeure vient de l'idée du français Michel Ugon, de Bull, d'utiliser les circuits intégrés de la puce pour ajouter de l'*intelligence*. Il dépose son premier brevet en 1977 (voir [114]). Il s'agit d'une carte comprenant à la fois une mémoire non-volatile programmable et un *microprocesseur*, permettant à la carte de fonctionner comme un micro-ordinateur. Le transfert d'information entre le terminal et la carte se fait alors *via* le microprocesseur, qui peut contrôler l'accès aux données. L'accès à certaines données peut ainsi être soumis à la reconnaissance d'une clé. En 1978, Michel Ugon dépose également le brevet SPOM (Self Programmable One Chip Micro-Computer) qui détermine l'architecture nécessaire au fonctionnement auto-programmable de la puce. Cet aspect d'auto-programmation permet à un microprocesseur de modifier son comportement en fonction de certaines données, comme de s'autodétruire en cas d'alerte.

Finalement, le 21 mars 1979, la première carte à microprocesseur Bull CP8 est opérationnelle. Elle comporte alors deux puces, une pour la mémoire et une pour le microprocesseur dont les composants sont fournis par le fondeur Motorola. Le manque de sécurité lié à la nécessité de transfert de données entre les deux puces est rapidement comblé par la conception d'une puce monolithique donnant le jour, en octobre 1981, au premier micro-ordinateur équipé d'une puce auto-programmable.

La *carte à microprocesseur*, ou *carte intelligente* (*smart card* en anglais) était créée. Elle est toutefois généralement appelée simplement carte à puce.

1.1.2 Différentes utilisations de la carte à puce

Premières utilisations. En 1984, des expériences simultanées sur la carte à mémoire et la carte à microprocesseur sont menées en France par l'administration des services publics des Postes et des Télécommunications (*PTT, Postes, Télégraphes et Téléphones*). D'une part l'apparition de la « télécarte », une carte à mémoire prépayée pour le téléphone, rencontre un succès impressionnant, qui atteindra les 60 millions de cartes dès 1990. D'autre part, une première expérimentation en matière de télépaiement donne l'avantage à l'utilisation de la carte à microprocesseur (plutôt qu'à celle de la carte à logique cablée), lançant la généralisation de son utilisation dans le domaine bancaire. La présence du microprocesseur permet un contrôle des accès aux données, qui est renforcé par l'utilisation de la *cryptographie*. En effet la puissance de calcul apportée par le microprocesseur permet le codage d'algorithmes cryptographiques complexes, mettant un haut niveau de sécurité, jusqu'alors réservé au domaine militaire et des services secrets, à la portée de

tout le monde. En dix ans, toutes les cartes bancaires françaises ont progressivement intégré des microprocesseurs.

Le succès des cartes téléphoniques et bancaires s'est rapidement propagé en Allemagne. C'est également en Allemagne qu'est apparue une autre grande application des cartes à microprocesseur : la téléphonie mobile. En 1988, dans le but de réduire la fraude liée à l'utilisation des bandes magnétiques, le *German Post Office* propose l'utilisation d'un microprocesseur pour une identification sur un réseau de téléphonie mobile. La carte à puce, appelée carte *SIM* (*Subscriber Identity Module*, module d'identité d'abonné), contient les informations sur les droits d'accès au réseau. La réussite de l'expérience dans ce réseau, réseau qui était limité en termes d'utilisateurs pour des raisons techniques, a contribué au choix des cartes à puces comme moyen d'accès au réseau global de communications mobiles *GSM* (*Global System for Mobile Communications*) mis en place en 1991 en Europe et étendu rapidement au reste du monde, pour atteindre aujourd'hui 170 pays et environ 600 millions d'abonnés.

Les domaines d'application de la carte à puce se diversifient, avec l'émission, en 1994, d'une carte allemande d'assurance maladie à tous les assurés. Le porte-feuille électronique fait son apparition à partir de 1992 au Danemark, puis en Autriche, ou encore aux États Unis lors des Jeux Olympiques d'Atlanta en 1996.

Apparition des normes. Une étape majeure dans l'utilisation future des cartes à puces comme moyen de paiement est la publication, en 1994, de la norme *EMV* : les trois organisations principales de cartes de crédit, *E*uropay, *M*asterCard et *V*isa, se mettent d'accord sur une spécification commune pour les cartes de crédit intégrant un microprocesseur, assurant une compatibilité mutuelle.

La compatibilité entre les différents systèmes utilisant des cartes à puce est cruciale pour leur déploiement. En effet, la diversité des domaines d'applications, ainsi que l'omniprésence grandissante des cartes à puce, impliquent une interaction inévitable entre systèmes de différentes sociétés, de différents domaines et de différents pays. Un travail intense de standardisation, commun à *ISO* (*International Organization of Standardization*, organisation internationale de standardisation) et *IEC* (*International Electrotechnical Commission*, commission électrotechnique internationale), a donné lieu à de nombreuses normes spécifiant les caractéristiques physiques et les protocoles de transmission des données des cartes à puce.

Aujourd'hui. Le principal domaine d'utilisation des cartes à puce aujourd'hui est la téléphonie mobile, suivi des applications bancaires, comprenant toutes les applications contenant une transaction financière (carte bancaire, porte-feuille électronique, paiement de parking, de téléphone, de piscine, etc). Mais la carte à puce devient présente dans des domaines de plus en plus variés.

De nombreux pays utilisent une carte à puce dans le domaine de la santé, dans des buts divers. La carte peut contenir l'identification du patient, les informations liés à l'assurance maladie, les données médicales permettant un suivi, les contacts à prévenir en cas d'urgence, etc.

Les secteurs des transports et des contrôles d'accès ont largement bénéficié de l'apparition des cartes sans contacts, permettant de s'identifier rapidement à l'aide d'une carte, sans avoir à l'insérer dans un terminal.

La principale fonction d'une carte à puce est l'identification de son porteur, pour accéder à des services, payants ou non. Cette identification est parfois l'unique but de la carte, lorsqu'elle est utilisée par exemple par les gouvernements comme carte d'identité des citoyens. Un exemple type est celui du passeport électronique, qui renfermera une puce contenant le nom et la

date de naissance du détenteur ainsi que le bureau de délivrance du passeport et un identifiant biométrique (photo et empreintes digitales).

Plus modestement, les commerçants proposent des cartes de fidélité permettant d'accumuler des points. Cette application de fidélisation est souvent associée à un moyen de paiement, illustrant la grande tendance des cartes à puce visant à faire cohabiter, sur une même carte, différents services, pouvant venir de différents fournisseurs et pouvant même être chargés sur la carte après la mise en service de celle-ci. Les nouvelles plates-formes proposant ces possibilités offrent de nouvelles perspectives d'application de la carte à puce, mais également de nouveaux enjeux sécuritaires majeurs, comme nous le discuterons dans la Section 1.1.5.3.

1.1.3 Caractéristiques techniques d'une carte à puce

Une carte à puce est une fine carte en plastique, dans laquelle est inséré un *micro-module*. Le micro-module est constitué de la puce d'un circuit intégré, contenant à la fois la mémoire et le microprocesseur, par dessus laquelle est apposé un ensemble de contacts permettant le transfert de données avec l'extérieur. La carte peut également contenir une piste magnétique, ainsi que des caractères estampés en relief sur la surface.

Il existe également des cartes *sans contact*, constituées de deux cartes en plastiques renfermant la puce et une antenne permettant d'établir une communication. Ce type de carte étant de fonctionnement globalement similaire aux cartes avec contact (excepté l'interface de communication), nous parlerons en général de cartes avec contacts (pour plus de détails sur les cartes sans contact, voir par exemple [108]).

Caractéristiques physiques. Comme mentionné dans la section précédente, de nombreuses normes, issues des organisations ISO et IEC, standardisent toutes les caractéristiques techniques d'une carte à puce. En particulier, sont définis les dimensions de la carte et de ses contacts, l'emplacement et le rôle de chacun des contacts, les protocoles de communication, le format des commandes échangées avec l'extérieur, l'emplacement de la piste magnétique éventuelle, etc. Les cartes sans contact, où les contacts sont remplacés par une antenne assurant la connexion, sont également standardisées.

La figure 1.3 décrit les principales caractéristiques d'une carte avec contacts de format ID − 1 (format "carte bleue" et non format "carte SIM"). Pour plus de détails, voir les normes ISO 7810 pour les différents formats de cartes (cf [70]), ISO 7811 pour les cartes estampées et les cartes magnétiques (cf [71]), ISO/IEC 10 536 pour les cartes sans contact (cf [72]) et ISO/IEC 7816 pour les cartes à puce avec contacts (cf [74]), en particulier les quatre premières parties (cf [76, 77, 78, 79]).

Processeur et mémoire. Les 25 millimètres carrés de surface autorisés par la norme pour la puce constituent un bloc monolithique contenant les différents types de mémoire, ainsi qu'un micro-processeur pour le traitement des données et l'exécution d'instructions. Certaines cartes possèdent également un coprocesseur cryptographique destiné à l'exécution optimisée d'algorithmes cryptographiques tels que RSA ou DES (voir Figure 1.4).

Une carte à puce contient différents types de mémoires, pour des usages différents (voir la Section 1.3.1 pour plus de détails sur les différents types de mémoire). La mémoire *RAM* est une mémoire volatile et modifiable, permettant de stocker de manière temporaire des données lors de l'exécution d'un programme. Elle sera utilisée dans une carte à puce comme espace de travail du processeur, en raison de ses temps d'accès très rapides en lecture et en écriture. La mémoire *ROM*, ou plus précisément *masked-ROM*, est une mémoire persistante mais non modifiable, utilisée pour

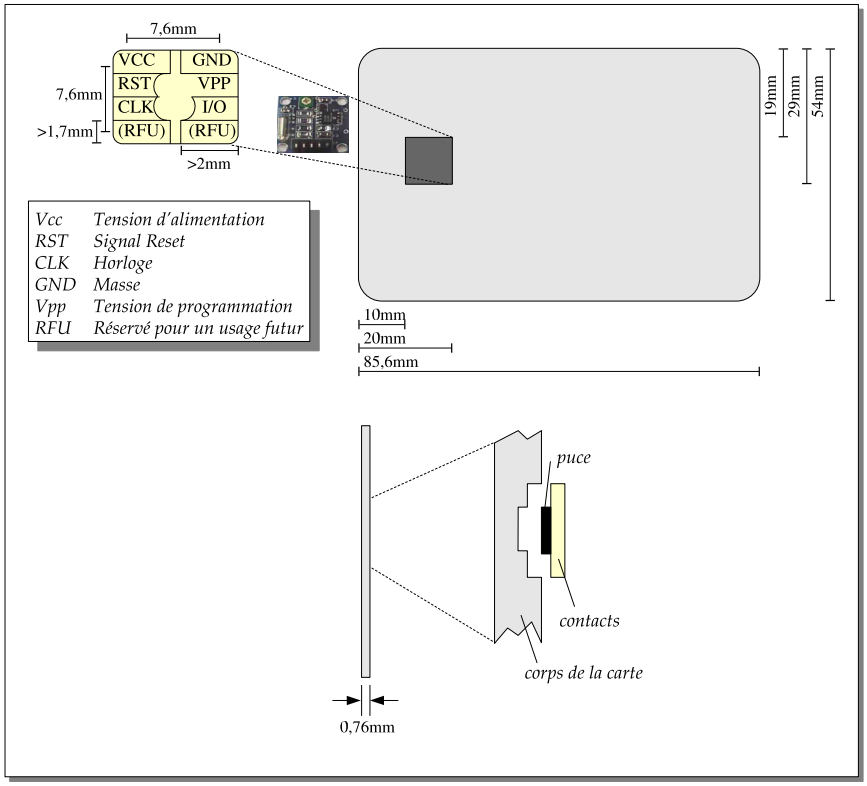


FIG. 1.3 – Caractéristiques d'une carte à puce

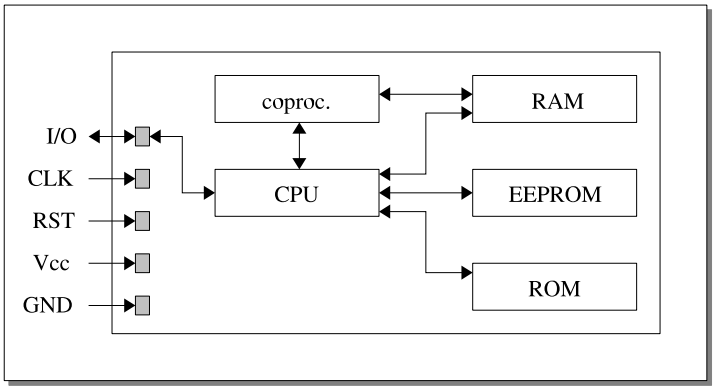


FIG. 1.4 – Architecture de la puce d'une carte à micro-processeur

stocker des données qui n'auront pas besoin d'être modifiées. Pour une carte à puce, cela signifie que les données stockées en ROM sont non seulement "brûlées" chez le fabricant de la puce, mais sont également inchangées durant tout le cycle de vie de la carte. Des mémoires hybrides, à la fois persistantes et modifiables, comme l'*EEPROM*, sont également utilisées dans les cartes à puce, pour stocker les données applicatives, lues et modifiées par l'intermédiaire du système d'exploitation. Mais ces mémoires ont une durée de vie limitée due à une *usure* ne permettant qu'au maximum 100 000 cycles d'écritures et d'effacements.

Initialement tout le système d'exploitation de la carte était stocké en ROM, figeant ainsi les applications possibles de la carte pour tout son cycle de vie. Par la suite, seuls les principaux composants du système d'exploitation ont été stockés en ROM, les parties dépendantes des applications étant transférées en EEPROM afin de pouvoir être modifiées pendant le cycle de vie. L'apparition d'une nouvelle technologie de mémoire hybride, appelée mémoire *Flash*, a contribué à cette flexibilité. Par ses grandes capacités de stockage et ses rapides temps d'effacement, cette mémoire peut contenir la presque totalité du système d'exploitation, laissant à la ROM uniquement les données nécessaires au démarrage de la carte. Toutefois cette nouvelle technologie est soumise à des contraintes en écriture et en effacement particulières. Elle a fait l'objet d'une partie de nos travaux et sera détaillée dans la Section 1.3.

La quantité de mémoire disponible sur une carte à puce évolue rapidement, suivant la loi de Moore[1]. Actuellement, une carte à puce peut contenir jusqu'à 64 kO[2] de ROM, pas plus de 4 à 6 kO de RAM et 64 voire 128 kO d'EEPROM. Lorsque la mémoire persistante et modifiable utilisée est de la mémoire Flash, la carte en contient en général 256 kO, mais peut en contenir jusqu'à 3 MO[3].

Communication. Une carte (avec contact) communique avec le monde extérieur lorsqu'elle est insérée dans un *lecteur*, ou *terminal*, ou encore *CAD* (*Card Acceptance Device*), qui l'alimente en courant électrique. La communication entre la carte et le lecteur se fait à travers un unique contact, celui d'entrée-sortie (I/O), en mode *half-duplex*, à savoir qu'une seule des deux parties peut communiquer à la fois. C'est le lecteur qui fournit l'alimentation à la carte et la communication se fait par l'échange de "paquets" de données, appelés *APDU* (*Application Protocol Data Unit*). Un APDU peut être ou bien une commande du CAD vers la carte, ou bien une réponse de la carte vers le CAD. C'est le CAD qui dirige la communication, la carte ne faisant qu'exécuter les commandes qui lui sont transmises. Une telle communication est dite de type "maître-esclave" ou "client-serveur".

1.1.4 Systèmes d'exploitation d'une carte à puce

Rôle d'un système d'exploitation de carte à puce. Le système d'exploitation d'un ordinateur est la couche intermédiaire entre les applications et le matériel. Lorsqu'une application nécessite le stockage de données en mémoire ou un calcul de la part du processeur, elle doit lui fournir les instructions correspondantes en langage assembleur ou en langage machine. Le système d'exploitation facilite la programmation d'applications en proposant une interface (ou *API* pour *Application Programming Interface*, interface de programmation d'application) entre la couche applicative et la couche matérielle. Le système d'exploitation est également responsable de la gestion des ressources du système (processeur, mémoire, etc).

[1] selon laquelle le nombre de transistors des circuits intégrés double tous les 18 mois.
[2] kilo Octets
[3] Méga Octets

Contrairement aux systèmes d'exploitation des ordinateurs[1], le système d'exploitation d'une carte à puce ne possède pas d'interface utilisateur. Il n'y a pas non plus d'accès à des périphériques externes ou à des moyens de stockage externes. Le système d'exploitation d'une carte à puce est en charge de :

- la gestion des fichiers,
- la gestion de la mémoire,
- le contrôle de l'exécution d'instructions,
- le chargement et la gestion d'applications,
- la communication avec le terminal,
- l'exécution et la gestion d'algorithmes cryptographiques.

Toutefois, on attend d'un système d'exploitation moderne de carte à puce, outre la gestion du matériel et l'abstraction fournie au développeur d'application vis à vis du matériel, qu'il réponde aux nouvelles exigences du marché, telles que :

- la portabilité : les applications doivent pouvoir être exécutées sur des puces et des types de cartes divers et variés ;
- le caractère multi-applicatif : plusieurs applications, provenant de différents fournisseurs, doivent pouvoir cohabiter sur la même carte ;
- l'isolation : les applications doivent être isolées du système d'exploitation, mais également isolées les unes des autres, avec un contrôle strict de l'accès aux ressources assurant un transfert de données sécurisé entre applications ;
- la compatibilité : l'interface de communication avec le terminal doit être standardisée pour qu'un terminal donné puisse communiquer avec n'importe quelle carte ;
- l'ouverture : une application doit pouvoir être chargée ou modifiée après la mise en service de la carte.

Les systèmes d'exploitation de carte à puce actuels répondent à ces exigences, mais cela résulte d'une évolution progressive et combinée des technologies, des capacités mémoire, du matériel et de la demande du marché.

Évolution des systèmes d'exploitation de la carte à puce. Les premières cartes à puces ne contenaient pas de système d'exploitation à proprement parler. Avec seulement 6kO de ROM, 128 Octets de RAM et 3kO d'EEPROM, les cartes à puce du début des années quatre vingt dix ne contenaient que les programmes en langage assembleur d'une unique application, stockés en ROM et s'adressant directement à la couche matérielle. Un *masque*[2], coûteux et long à mettre en œuvre, était donc nécessaire pour toute nouvelle application et aucune ré-utilisation de code n'était possible. Par exemple, le travail difficile de développement d'instructions d'entrée-sortie gérant la communication avec le terminal devait être entièrement recommencé pour chaque nouvelle application.

Une première génération de systèmes d'exploitation est alors apparue, sous la forme d'un ensemble de *bibliothèques* définissant des routines[3] génériques, pour manipuler la couche matérielle (voir Figure 1.5). Ces bibliothèques pouvaient être utilisées par différentes applications, alors stockées en EEPROM, sollicitant une plus grande capacité mémoire. Le principal inconvénient de ce type de système d'exploitation basé sur des bibliothèques, est sa dépendance vis-à-vis du maté-

[1] Pour plus d'informations sur un système d'exploitation en général, voir par exemple [112].

[2] Comme il sera expliqué en Section 1.3.1, le masque consiste à indiquer les emplacements où doivent apparaître des 0 et ceux où doivent apparaître des 1, dans la matrice en silicium qui constitue la mémoire ROM.

[3] Une routine est une petite partie de programme, exécutant une tâche spécifique et souvent spécialement étudié pour sa rapidité (et alors écrit en langage assembleur).

riel. En effet, pour porter[1] les applications sur une autre carte à puce, toutes les instructions des bibliothèques correspondant au matériel d'origine devaient être remplacées par les instructions pour le nouveau matériel. Les applications étaient donc développées pour une carte spécifique.

La deuxième génération de système d'exploitation correspond aux systèmes *monolithiques* (voir Figure 1.5). Le système d'exploitation est alors une véritable interface avec le matériel, permettant aux applications d'être exécutées sur des cartes différentes. Mais ceci implique de porter le système d'exploitation sur les nouvelles cartes, ce qui est une tâche lourde puisque celui-ci n'est pas encore modulaire et n'est formé que d'une seule couche complexe.

Le grand besoin de portabilité a conduit à la troisième génération de systèmes d'exploitation, avec une architecture *en couches* (voir Figure 1.5), où seule la couche la plus basse est dépendante du matériel. Cette architecture permet la définition de plates-formes standards, permettant une meilleure compatibilité entre les différentes cartes à puce. Toutefois le besoin d'uniformisation et normalisation est plus dur à satisfaire pour un système d'exploitation de carte à puce que pour celui d'un ordinateur. En effet les exigences extrêmement fortes en termes de sécurité et qualité de logiciel, la faible capacité mémoire et la demande de confidentialité du logiciel du système d'exploitation, sont autant de freins à l'uniformisation. Ainsi, ce n'est qu'au milieu des années quatre-vingt dix que sont apparues les plates-formes *ouvertes*, autorisant une application à être chargée sur la carte par une tierce personne, sans aucune participation de la part du fournisseur du système d'exploitation, et donc éventuellement après la mise en circulation de la carte à puce.

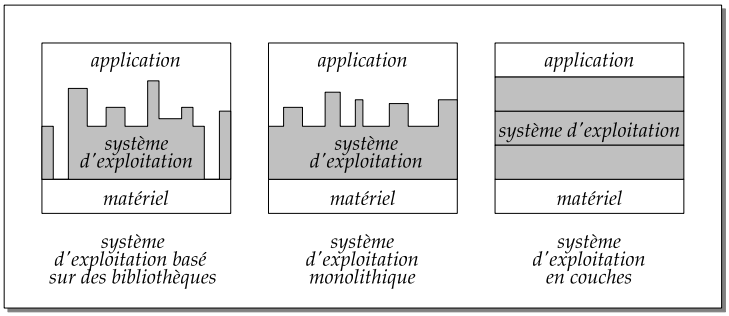


FIG. 1.5 – Évolution des systèmes d'exploitation de cartes à puce

Plates-formes multi-applicatives ouvertes. Au milieu des années quatre vingt dix, face à un marché grandissant, l'attente pour re-brûler une nouvelle carte à chaque modification du service proposé devenait inacceptable pour les fournisseurs de services, qui devenaient de plus en plus nombreux. Les avantages d'une coopération entre fournisseurs de services commençaient également à se dessiner, créant une demande pour des plates-formes pouvant héberger plusieurs applications et gérer un partage de données sécurisé entre ces applications.

C'est ainsi que les plates-formes *multi-applicatives et ouvertes* sont apparues, offrant la possibilité d'héberger plusieurs applications, pouvant provenir de différents fournisseurs et pouvant être chargées à différents moments, y compris après la mise en circulation de la carte. La plus répandue est la plate-forme *Java Card* (voir [81, 34]), dérivée de la technologie Java, qui sera étudiée en détail dans la Section 1.2 et dans le Chapitre 3. Dès sa création en 1996, sa spécification et ses fonctionnalités sont publiques et discutées entre les principaux fabriquants de cartes

[1]Porter une application signifie l'adapter à un système différent de celui sur lequel elle a été originellement développée (et pour lequel elle a été conçue).

à puce et Sun, la compagnie qui développe Java, au sein de ce qui s'appelle maintenant le Java Card Forum. Cette plate-forme a remporté un succès bien plus significatif que les autres systèmes d'exploitation ouverts et multi-applicatifs, tels que :

- *Multos* (voir [89]) développé par le consortium Maosco pour le système de porte-feuille électronique Mondex, mais dont les spécifications sont en grande partie confidentielles ;
- *Basic Card* (voir [118]) développé dès 1996 par une société allemande de cartes à puce, Zeitcontrol Cardsystems, permettant la programmation d'application en langage Basic ;
- ou encore *Windows for Smart Cards*, proposé par Microsoft en 1998, mais qui a dû être abandonné faute de demande après plusieurs versions successives et un effort promotionnel important.

Notons également les développement plus récents tels que la plate-forme *.Net Card* d'Axalto et le prototype académique *Camille* (voir [58]) d'un système d'exploitation ouvert.

Le principe général de ces plates-formes est le même : utiliser une *machine virtuelle*[1] pour interpréter un langage intermédiaire d'instructions, permettant d'abstraire complètement la couche matérielle, et de proposer un ensemble d'interfaces aux applications, qui peuvent alors être exécutées sur différentes cartes (voir Figure 1.6).

Une machine virtuelle est une couche logicielle qui simule la couche matérielle. Le programme d'une application est traduit en instructions exécutables mais indépendantes du matériel, destinées à la machine virtuelle. Celle-ci va alors exécuter ces instructions, ou plus précisément les *interpréter* (d'où son autre nom d'*interpréteur*), c'est à dire les traduire en instructions natives propres au processeur. Ainsi les applications peuvent être programmées indépendamment du système sur lequel elles vont être exécutées et seule la machine virtuelle devra être portée sur un nouveau système.

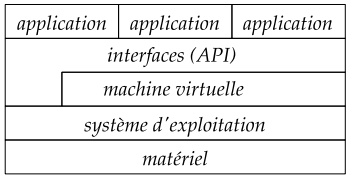


FIG. 1.6 – Architecture des plates-formes multi-applicatives et ouvertes

À la portabilité qu'offre une machine virtuelle s'ajoute la sécurité qu'elle peut assurer par des contrôles lors de l'interprétation du code intermédiaire. Par exemple, la machine virtuelle de Java Card effectue des contrôles d'accès aux données assurant une isolation entres différentes applications chargées sur une même carte. Ce mécanisme d'isolation, qui sera décrit en détail dans la Section 1.2, répond à un enjeu sécuritaire majeur des cartes multi-applicatives, comme nous allons le voir dans la section suivante. Une partie de nos travaux, présentée dans le Chapitre 3, a consisté à prouver formellement la correction de ce mécanisme.

[1] appelée *JCVM* (pour *Java Card Virtual Machine*) dans Java Card, *AAM* (*Application Abstraction Machine*) dans Multos, ou simplement *interpréteur* dans BasicCard et *VM* pour virtual machine dans Windows for Smart Card.

1.1.5 Sécurité et enjeux de la carte à puce

1.1.5.1 La notion de sécurité

Le terme *sécurité* est à prendre avec précaution, car il englobe un grand nombre de notions et est source de confusion, notamment lorsque l'on mélange les termes anglais et français, faisant apparaître le faux ami "sûreté". Précisons donc que la confiance accordée à un système repose sur :

- sa *conformité* ou *correction* (*correctness* en anglais), à savoir l'assurance que le comportement du système est celui attendu, sous-entendu vis-à-vis d'une spécification donnée ;
- sa *sûreté de fonctionnement* (*dependability* en anglais), propriété qui permet aux utilisateurs du système de placer une confiance justifiée dans le service qu'il délivre et qui comprend :
  - la *fiabilité* (*reliability* en anglais), assurant la continuité du service que le système doit accomplir, c'est à dire l'absence de défaillance dans le temps ;
  - la *maintenabilité* (*maintenability* en anglais) à savoir l'aptitude à pouvoir réparer et faire évoluer le système ;
  - la *"sécurité-innocuité"* (*safety* en anglais), qui comme son nom l'indique, assure que le système n'est pas nuisible, c'est-à-dire qu'il ne contient pas de défaillance catastrophique (vis-à-vis du risque encouru par les utilisateurs du système) ;
  - et enfin la *"sécurité-confidentialité"* (*security* en anglais) qui assure une absence d'accès ou de manipulations non autorisées de l'information, ou de façon plus générale, de fautes intentionnelles. Cela comprend la confidentialité, l'intégrité et la disponibilité des données du système.

Dans ce mémoire, le terme de sécurité, employé tout seul, représentera la confiance totale accordée au système, à savoir à la fois la correction et la sûreté de fonctionnement. Dans nos travaux, dont l'objet sera décrit en Section 1.1.6, nous nous sommes intéressés à la sécurité-confidentialité de la machine virtuelle, ainsi qu'à la correction et la sécurité-innocuité du système d'exploitation de la carte à puce.

Il est à noter que la notion de sécurité est souvent associée à celle de *cryptographie*. En effet cette science a pour but de concevoir et mettre au point des mécanismes, souvent algorithmiques, afin d'assurer la confidentialité, l'authenticité et l'intégrité de l'information. Notre approche est en quelque sorte en aval et de plus haut niveau, puisqu'elle consiste à fournir des *preuves* que certaines propriétés sont vérifiées et non à mettre en place des mécanismes pour assurer ces propriétés. La vérification formelle est, de manière plus générale, complémentaire, puisqu'elle permet de prouver des propriétés sur le *comportement* des programmes embarqués.

1.1.5.2 Enjeux

Les cartes à puce ont pour but de fournir un environnement sécurisé pour des données et des programmes. Dans le même temps, la flexibilité, la portabilité, la facilité d'utilisation, sont autant de critères qui qualifient également la carte à puce et qui mettent la sécurité à rude épreuve. Voici quelques spécificités des cartes à puces qui constituent un véritable défi d'un point de vue sécuritaire.

Plates-formes multi-applicatives et ouvertes. La diversification croissante des domaines d'application de la carte à puce, illustrée par les nombreuses utilisations décrites en page 20, ainsi que la tendance à proposer des plates-formes ouvertes et multi-applicatives, décrites en page 25, offrent de nouvelles perspectives de marché, mais imposent également de nouveau modèles de

sécurité. En effet, une même carte peut contenir plusieurs applications, pouvant provenir de différents fournisseurs et pouvant même être chargées après la livraison de la carte au client. Un exemple parmi d'autres vient de la Malaisie, qui teste depuis 2001 le premier passeport électronique, qui fait aussi office de pièce d'identité, de permis de conduire, de carte de santé et de porte-monnaie électronique. Il existe donc un risque important de chargement d'une application hostile qui exploiterait un défaut d'implémentation, soit de la plate-forme, soit des autres applications. Le fabricant de cartes à puce veut protéger sa plate-forme des applications qu'elle héberge, et le fournisseur d'applications veut protéger son application vis à vis des autres applications chargées sur la même carte.

L'arrachage. Une des particularités principales de la carte à puce, par rapport aux autres systèmes informatiques, est que l'interruption de ses programmes doit être considérée comme un comportement normal dans son utilisation et être pris en compte dans la spécification de son cycle de vie. En effet, comme on l'a vu, une carte à puce n'est alimentée en courant électrique que lorsqu'elle est insérée dans un lecteur. Or il est généralement possible de retirer la carte du lecteur de façon prématurée, interrompant toutes les opérations qui étaient en cours au sein de la carte. C'est ce que l'on appelle un *arrachage* de la carte (*tearing* ou *card tear* en anglais).

Les programmes embarqués sur une carte à puce doivent donc tenir compte dans leur spécification de l'éventualité d'être interrompus à tout moment. Un certain nombre de mesures sont mises en place afin de rendre possible un *crash recovery*, ou récupération des données après une interruption. Une carte implémentant de telles mesures possède la propriété dite d'*anti-tearing*, à savoir la protection du contenu de la mémoire en cas d'arrachage.

Les mesures d'*anti-tearing* dépendent de ce qui doit être protégé en cas d'interruption. Il existe essentiellement deux problèmes majeurs en cas d'interruption : les données volatiles sont perdues et le système peut se trouver dans état incohérent. Dans le premier cas, une protection consiste généralement à utiliser des *indicateurs d'état* (ou *flags*), stockés en mémoire persistante, afin de connaître, lors de la remise sous tension de la carte, l'état dans lequel elle était avant l'interruption. Dans le deuxième cas, une méthode générale pour maintenir la cohérence des données du système est d'utiliser un mécanisme de *transaction*.

Regardons tout d'abord le cas de la mémoire volatile. Certaines données, en raison de leur utilisation fréquente, sont stockées en mémoire volatile pour son temps d'accès bien plus rapide. Ces données seront alors définitivement perdues en cas d'arrachage de la carte. Il faut donc veiller à pouvoir retrouver, au moment de la remise sous tension de la carte, la valeur de ces données par un calcul, plus ou moins long, sur des données en mémoire persistante. Par exemple, lors du remplissage d'un tableau (en mémoire persistante), on fera en sorte de "marquer" de façon persistante les entrées du tableau qui sont déjà remplies, et une variable volatile indiquera où en est le remplissage du tableau (voir Figure 1.7). Cette variable sera souvent lue et modifiée, elle sera donc stockée en mémoire volatile, mais un parcours du tableau et une analyse des états de chaque entrée permettront facilement de retrouver la valeur de cette variable en cas d'interruption, et donc de perte de cette valeur.

Le deuxième problème majeur d'un arrachage de la carte est que le système peut se retrouver dans un état incohérent. En effet, certaines opérations doivent être exécutées de façon *atomique*, c'est-à-dire que soit toutes les instructions de l'opération sont exécutées, soit aucune ne l'est. Lorsqu'une partie seulement des instructions est exécutée, le système est dans un état incohérent. Par exemple, une copie de données d'une partie de la mémoire dans une autre partie de la mémoire doit être atomique : le système est incohérent jusqu'à ce que toutes les données soient copiées. Un virement d'une somme d'argent d'un compte à un autre doit également être atomique : lorsque la

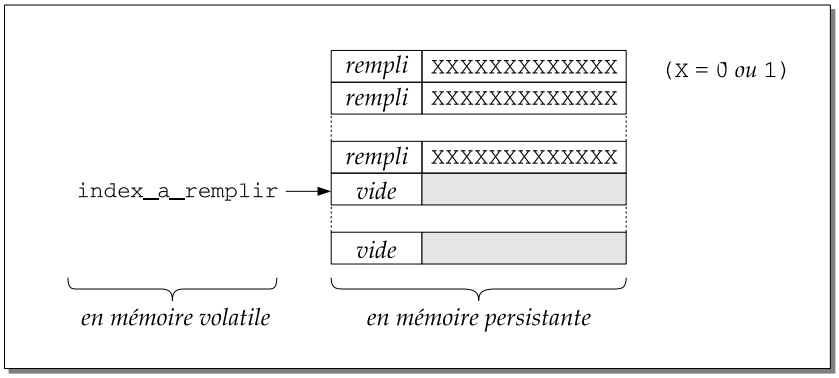


FIG. 1.7 – Exemple d'utilisation d'indicateurs d'état pour le remplissage d'un tableau

somme a été débitée d'un compte, mais pas encore créditée sur l'autre compte (ou inversement), le système est incohérent.

En particulier, si une interruption (ou une défaillance du système) survient pendant l'exécution d'une opération qui doit être atomique, le système est dans un état incohérent lors de sa remise sous tension. Un moyen de maintenir la cohérence du système est d'utiliser un mécanisme de *transaction*. La notion de transaction, commune aux base de données, permet d'assurer l'exécution atomique d'un ensemble d'instructions. Le principe consiste à rendre les instructions de la transaction *conditionnelles*, i.e. effectives uniquement lorsque toutes les instructions sont exécutées. Trois opérations sont utilisées : une opération de début de transaction (*begin*), une opération de validation de la transaction (*commit*) et une opération d'abandon ou d'avortement de la transaction (*abort* ou *rollback*). Un abandon de transaction consiste à annuler toutes les opérations effectuées depuis le début de la transaction. Pour permettre ce retour en arrière, une variable mémorise la valeur initiale de toutes les données modifiées pendant la transaction. Un début de transaction consiste alors, d'une part, à commencer à enregistrer toutes les modifications dans cette variable et, d'autre part, à "marquer" de façon persistante qu'une transaction est en cours. Une validation de transaction consiste, quant à elle, à libérer la variable et indiquer que la transaction en cours a été validée. Enfin un abandon consiste à redonner à toutes les données modifiées pendant la transaction leur valeur initiale, mémorisée dans la variable.

Par conséquent, afin de rendre un ensemble d'instructions atomiques, il suffit de le faire précéder d'un début de transaction et le faire suivre d'une validation de transaction, et également d'abandonner toute transaction signalée comme commencée (et non validée) lors d'une remise sous tension. Dans l'exemple d'un virement d'un compte à un autre, l'opération consistera, par exemple, à commencer une transaction, faire le débit sur le premier compte, faire le crédit sur le deuxième compte et valider la transaction. Si un interruption se produit entre le débit et le crédit, alors, au moment de la remise sous tension, la transaction commencée n'aura pas été validée et sera donc abandonnée : le débit sera annulé et les deux comptes reprendront leurs valeurs initiales.

Notons enfin que les opérations atomiques ne sont pas les seules opérations sensibles aux interruptions et pouvant rendre le système incohérent. En effet, certaines opération, au lieu d'être annulées en cas d'interruption, doivent au contraire être de nouveau exécutées. C'est le cas, par exemple, de l'effacement de données : si une interruption survient pendant l'effacement d'un bloc de mémoire, le bloc peut ne pas être entièrement effacé, donc le système est incohérent. Par conséquent, l'opération doit être de nouveau effectuée lors de la prochaine mise sous tension

de la carte. Le mécanisme de transaction n'est pas approprié dans ce cas, qui sera plutôt géré avec des indicateurs d'état : une variable persistante indique si un effacement a commencé et lors d'une remise sous tension, si un effacement est indiqué comme non terminé, il sera recommencé.

L'arrachage est donc une spécificité des cartes à puces, qui impose une gestion particulière des opérations et des données et qui nécessite une vérification minutieuse.

Système critique. La carte à puce n'est pas par définition un système réellement *critique*, c'est-à-dire dont dépendent des vies humaines. Toutefois, un dysfonctionnement du système peut avoir des conséquences économiques sérieuses. En effet la carte à puce se différencie tout de même d'autres applications où les erreurs peuvent être, certes, gênantes en termes d'image, mais peuvent être corrigées dans une nouvelle version mise à la disposition des utilisateurs. En effet, une telle maintenance dans le monde de la carte à puce, où les systèmes sont produits à des millions d'exemplaires, aurait un coût prohibitif. De plus, on a vu que, mis à part les cartes utilisant de la mémoire Flash, la majorité des programmes de la carte, à savoir ceux qui constituent le système d'exploitation, est stockée en mémoire ROM. Or une erreur logicielle provenant d'un programme mémorisé en ROM nécessite de reproduire entièrement la carte, entraînant un coût et des délais de production une fois encore inacceptables. Ceci démontre l'importance de la nécessité de confiance dans le logiciel embarqué dans une carte à puce, plus encore que dans d'autres systèmes informatiques.

1.1.5.3 Composants impliqués dans la sécurité

La sécurité d'une carte à puce doit être assurée à trois niveaux : au niveau physique (le corps de la carte), au niveau matériel des composants (*hardware*) et au niveau logiciel (*software*) (voir Figure 1.8).

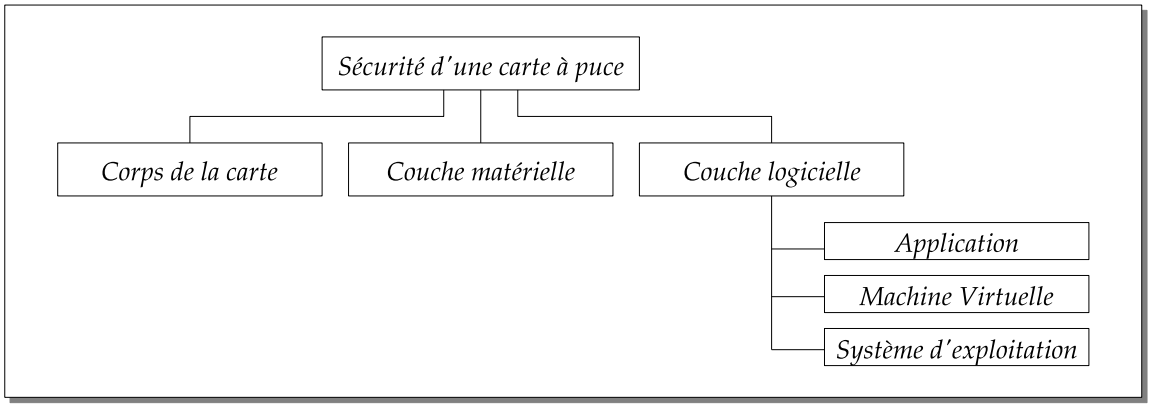


FIG. 1.8 – Composants impliqués dans la sécurité

Au niveau *physique*, le corps de la carte doit répondre à certaines exigences, principalement de résistance (à la température, aux torsions, à certains produits chimiques, aux vibrations, etc). Ces tests sont normalisés dans la norme ISO/IEC 10 373 (voir [73]) qui décrit en détail chaque méthode de test. Par exemple, un test de torsion consiste à tordre la carte de plus ou moins 15 degrés selon son axe longitudinal, à une cadence de 30 torsions par minute et le standard exige 1000 torsions sans défaillance fonctionnelle de la puce ou dégât mécanique visible de la carte.

La qualité de la couche *matérielle* de la puce est quant à elle primordiale pour assurer le bon fonctionnement du processeur et de la mémoire. Elle fait l'objet de nombreux tests et vérifications, avant et après l'encartage de la puce sur la carte, avant et après la personnalisation. Ces tests peuvent être effectués, entre autre, grâce à une partie de la ROM réservée pour les tests, offrant un accès externe au processeur et à la mémoire par l'intermédiaire de commandes qui sont bloquées de façon irréversible après exécution. Ces tests permettent de vérifier, par exemple, que tous les octets en mémoire EEPROM peuvent être écrits et de nouveau effacés, ou que la mémoire RAM est pleinement opérationnelle.

Enfin, la confiance accordée à une carte à puce repose fortement sur sa couche *logicielle*. Comme nous l'avons vu dans la section précédente, une erreur logicielle dans une carte à puce peut avoir des conséquences économiques très sérieuses. La recherche d'un logiciel zéro-défaut est traduite par l'amélioration des méthodes de test et de vérification de programme. La couche logicielle comprend à la fois le système d'exploitation, la machine virtuelle et les applications. Les applications sont des sources potentielles d'attaques. Elles peuvent être chargées sur la carte sans le consentement des programmes déjà existants, qu'il s'agisse de la plate-forme ou des autres applications. Par conséquent, en plus de l'exigence d'une grande qualité logicielle, le système ainsi que les applications doivent se protéger et assurer un partage d'information sécurisé.

1.1.6 Objet de nos travaux

Le but de nos travaux consiste à concevoir et appliquer des méthodes *formelles* de vérification, répondant aux enjeux sécuritaires de la carte à puce, et ce au niveau *logiciel* de la carte à puce, et en particulier au niveau de la machine virtuelle et du système d'exploitation.

Dans une première partie des travaux, nous nous sommes intéressés à l'étude de ***la propriété d'isolation des applications assurée par la machine virtuelle Java Card***. Comme mentionné page 25, la plate-forme Java Card est une plate-forme multi-applicative et ouverte, permettant à plusieurs applications d'être hébergées par une même carte et d'être chargées sur celle-ci après sa mise en service. Pour des raisons de sécurité évidentes, toute interaction entre les applications d'une même carte doit être strictement contrôlée. La machine virtuelle Java Card assure cette isolation entre les applications grâce à un mécanisme complexe appelé le *firewall* (pare-feu). Le but de nos travaux a été de modéliser formellement la propriété d'isolation, sous la forme des propriétés de confidentialité et d'intégrité des données des applications, et ensuite de prouver formellement que le mécanisme de firewall mis en place par la machine virtuelle Java Card assurait effectivement ces propriétés.

La section suivante décrit en détail le principe d'isolation des applications mis en œuvre par la machine virtuelle Java Card et le Chapitre 3 présente nos travaux de modélisation et de vérification des propriétés de confidentialité et d'intégrité, à partir d'un modèle formel existant de la technologie Java Card.

Lors de nos travaux sur la machine virtuelle, certaines propriétés ne pouvaient être prouvées compte tenu du niveau d'abstraction de la modélisation. Ces propriétés constituaient alors un ensemble de conditions suffisantes à l'isolation des applications, qui devaient être vérifiées au niveau de l'implémentation de la plate-forme. Ceci a mis en évidence un problème de lien entre un modèle de haut niveau sur lequel la vérification est effectuée et le code source de l'implémentation. Ce problème a été pris en compte dès le départ dans la deuxième partie de nos travaux, qui a consisté à trouver ***une méthode de modélisation et de vérification formelle du système d'exploitation***. Dans un premier temps, une vérification fonctionnelle de la correction du code

source des programmes du système d'exploitation est proposée, dans le Chapitre 4. Puis une méthode est définie, dans le Chapitre 5, pour la vérification de propriété de *haut niveau* sur un modèle *formellement liée* au code source.

Le code source étudié pour nos travaux est celui d'***un module de gestion de mémoire Flash au sein du système d'exploitation***. Ce module, comprenant des programmes de bas niveau représentatifs du code du système d'exploitation, met en évidence les précautions à prendre lors de l'utilisation de la mémoire Flash, justifiant le recours à la vérification formelle de correction et d'absence de défaillance. Les caractéristiques de la mémoire Flash, ainsi que le fonctionnement de ce module de gestion de mémoire Flash, sont expliqués en détail dans la Section 1.3.

1.2 L'isolation des applications dans Java Card

La technologie Java Card a été développée par Sun (voir [93]), le créateur du langage Java, dans le but de permettre aux systèmes aux ressources limitées, tels que les cartes à puce, d'hériter des avantages de la technologie Java, à savoir la flexibilité, la portabilité, la sécurité, ainsi qu'un langage de programmation fiable et de haut niveau rendant le développement d'applications à la portée des développeurs Java expérimentés. En raison des contraintes et des exigences liées aux cartes à puces, le langage Java Card (version 2.1) n'accepte qu'un sous-ensemble du langage Java (sans *string*, *thread*, etc), mais propose un environnement d'exécution avec des fonctionnalités sécuritaires additionnelles. En particulier, un mécanisme complexe, appelé le *firewall* (pare-feu), un contrôle d'accès aux données assurant l'*isolation* des applications qui cohabitent sur une même carte. Sommairement, cela signifie qu'une application ne peut pas lire ou modifier des données appartenant à une autre application, à moins que cette dernière n'ait explicitement indiqué qu'elle partageait ces données.

Nous décrivons dans cette section les caractéristiques principales de la technologie Java Card et, en particulier, les détails précis du mécanisme de firewall assurant la sécurité des applications embarquées (pour plus d'information, voir [34, 93]).

1.2.1 Les applets Java Card

La technologie Java Card permet d'exécuter sur une carte à puce des applications, appelées *applets*[1], écrites dans le langage de programmation Java, suivant la spécification Java Card (voir [93]). Cela apporte un certain nombre d'avantages aux développeurs d'applications de cartes à puce. La sécurité assurée par la plate-forme en termes de contrôle d'accès est un avantage majeur, qui sera discuté dans la Section 1.2.3. Les autres avantages sont l'indépendance des applications vis à vis du matériel, et la facilité de développement dans un langage haut niveau.

Regardons ici le processus de développement d'une applet Java Card. Une applet est une instance d'une classe utilisateur qui étend la classe `javacard.framework.Applet` de Java Card. La classe utilisateur doit surcharger[2] les méthodes de la classe `Applet` afin d'implémenter la spécification de son applet. En particulier, chaque applet doit fournir les méthodes `install`, `register`, `select`, `deselect` et `process` dont les signatures sont les suivantes :

[1] Il semblerait que le terme exact en français soit *appliquette*, mais nous utiliserons le terme anglais plus répandu d'applet.

[2] Une méthode d'une classe C est surchargée dans une classe C', qui hérite de C, si elle est redéfinie avec une implémentation différente dans la classe C'.

```
public static void install (byte[] bArray, short bOffset, byte bLength)
protected final void register ()
public boolean select ()
public void deselect ()
public abstract void process (APDU apdu)
```

Les différentes étapes nécessaires pour qu'une applet soit prête à être exécutée sur la carte sont les suivantes (voir la Figure 1.9). Premièrement, l'applet est compilée, comme une application Java, en un fichier `.class`. Comme dans le langage Java, tous les composants d'une application sont groupés ensemble en *package*[1]. Le fichier `.class` est ensuite converti en un fichier `.cap` (où `cap` signifie *Converted APplet*, applet convertie), où les structures de données sont optimisées en espace. Les applets compilées (et converties) sont composées d'instructions Java Card primitives, appelées des *instructions bytecodes*, ou simplement *bytecodes*, semblables à des instructions assembleur mais incluant du typage et des traits orientés objet tels que des appels à des méthodes virtuelles (voir [93] pour plus de détails).

Après cette étape de compilation, le fichier `.cap` est vérifié par un *vérificateur de bytecode*, qui effectue plusieurs contrôles statiques. D'un côté, il vérifie que le fichier `.cap` est bien formé, comme par exemple que la relation d'héritage de classe n'a pas de boucle. D'un autre côté, il vérifie que les règles de typage sont respectées durant l'exécution des méthodes contenues dans le fichier `.cap`.

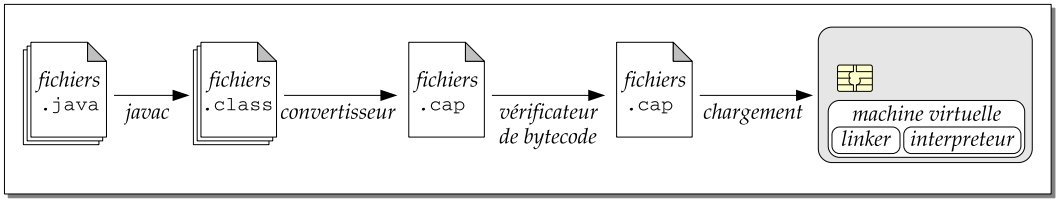


FIG. 1.9 – Chaîne d'exécution d'une applet Java Card

Après avoir été vérifié, le fichier `.cap` est chargé sur la carte à puce en utilisant un protocole de chargement sécurisé (assurant l'intégrité du fichier pendant le chargement). Il est enfin *lié* aux autres programmes sur la carte, par le *linker*. Les applets peuvent enfin être instanciées par l'invocation de leur méthode `install`. La méthode `install` créé une instance de l'applet et l'enregistre auprès du système par un appel à la méthode `register` avec l'AID (Application IDentifier, identifiant de l'applet) de l'applet. Les applets, ou plus précisément leurs méthodes, sont alors prêtes à être exécutées.

1.2.2 Architecture de la plate-forme Java Card

Une fois les applications installées sur la carte, elles peuvent être utilisées pendant le traitement d'une commande entrante. La commande peut, soit utiliser les services proposés par l'applet actuellement sélectionnée, en faisant un appel à sa méthode `process`, soit sélectionner une autre applet. Plus précisément, les différents composant d'une carte à puce Java Card sont les suivants (voir Figure 1.10).

Le *card manager* gère le cycle de vie de la carte et de ses applications embarquées. Il est responsable du chargement de nouvelles applets et de la transmission des commandes APDU aux applets. Lorsqu'une commande APDU est reçue, le *card manager* doit, soit sélectionner une nouvelle applet comme indiqué dans la commande, soit transmettre la commande reçue à l'applet

[1]Ici encore nous utiliserons le terme anglais, plutôt que le terme français correspondant, à savoir *paquetage*.

en train de s'exécuter (l'applet actuellement sélectionnée). La transmission à l'applet actuellement sélectionnée signifie invoquer la méthode `process` de cette applet, avec la commande reçue en argument. Quant à la sélection d'une nouvelle applet, cela signifie, premièrement, invoquer la méthode `deselect` de l'applet actuellement sélectionnée, puis informer la nouvelle applet à sélectionner en invoquant sa méthode `select`. Cette applet peut refuser d'être sélectionnée, en retournant `false` ou en levant une exception. Si elle retourne `true`, sa méthode `process` est finalement invoquée, avec la commande APDU de sélection en argument. Lorsque le traitement de la commande est terminé, le *card manager* transmet la réponse au terminal.

Dans les deux cas, l'invocation d'une méthode d'une applet signifie l'exécution de chacune des instructions *bytecode* par la *machine virtuelle Java Card*. La machine virtuelle interprète les instructions *bytecode* une par une, comme un microprocesseur exécute des instructions assembleur. La machine virtuelle assure également un partage sécurisé des données entre les applets Java Card, par l'intermédiaire de son mécanisme de *firewall*.

Pendant le traitement des commandes, les applets peuvent utiliser l'API Java Card (*Application Programming Interface*, interface de programmation d'application). L'API offre une aide aux applications pour les fonctionnalités spécifiques à Java Card et pour certains services du système. Il est à noter que certains modules de l'API ne sont pas développés en Java mais en langage C ou assembleur.

Finalement, la plate-forme contient le système d'exploitation et des *méthodes natives*, qui sont des fonctions implémentées en langage C ou assembleur, qui gèrent la mémoire, les protocoles de communication bas niveau, le support cryptographique, etc.

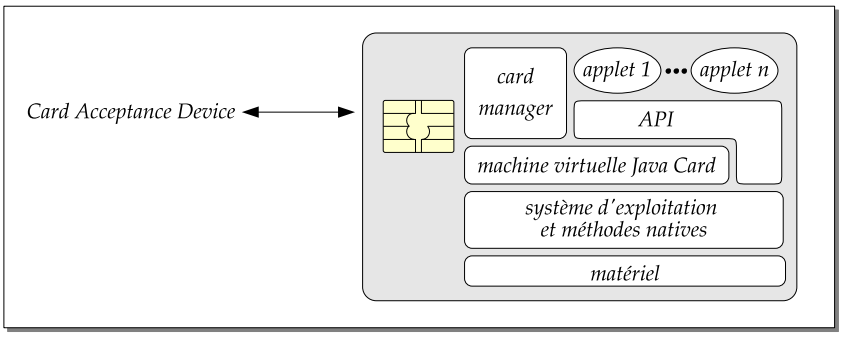


FIG. 1.10 – Architecture d'une carte à puce basée sur la technologie Java Card

1.2.3 Le firewall de Java Card

Comme déjà mentionné, un des avantages majeurs de la technologie Java Card consiste en la sécurité qu'elle assure en termes de partage de données entre les applets embarquées. En particulier, la machine virtuelle assure le principe d'*isolation* des applets, c'est à dire l'isolation en termes d'accès aux données et invocation de méthodes virtuelles entre applets embarquées sur la même carte à puce. Cette propriété est essentielle pour la sécurité des carte à puces multi-applicatives, où plusieurs applications peuvent être embarquées sur la même carte à puce et donc ont besoin d'être protégées les unes contre les autres.

Le modèle du bac à sable (sand box). Le principe d'isolation des applications est assuré dans Java Card par une politique de sécurité dite du "bac à sable" ou *sand-box*. Cela signifie que

les applications sont réparties en espaces protégés distincts, appelés des *contextes* et associés à leurs packages. En d'autres termes, le contexte d'une applet est son package. Il existe de surcroît un contexte privilégié, appelé le *contexte JCRE*[1], associé aux opérations du système.

La politique de sécurité stipule que l'accès à des objets à l'intérieur d'un contexte est autorisé, alors que l'accès à des objets n'appartenant pas au même contexte est interdit, excepté dans des cas particuliers bien précis (que nous allons décrire). Plus précisément, un objet appartient à un unique *propriétaire*, qui peut être ou bien une instance d'applet ou bien le système. Le propriétaire d'un objet est celui qui était *actif* quand l'objet a été créé.

Le propriétaire qui est *actif* pendant l'exécution d'une méthode dépend du fait que cette méthode est statique ou non. Lorsqu'une méthode *non statique* est en train d'être exécutée, le *propriétaire actif* est le propriétaire de l'objet qui contient cette méthode. Lorsque la méthode est *statique*, le *propriétaire actif* est celui qui était actif lors de l'exécution de la méthode *appelante*.

Alors, le *contexte actif courant* peut être défini comme étant le contexte du propriétaire actif (le contexte du système étant le contexte JCRE). Par exemple, pendant l'exécution d'une méthode non statique m d'une applet appartenant à un contexte C, le contexte actif courant est le contexte C.

Finalement, la politique de sécurité peut être énoncée comme suit : un objet ne peut être accédé que si le contexte de son propriétaire est le contexte actif courant. Par exemple, la méthode m appartenant au contexte C ne peut invoquer que des méthodes ou accéder à des objets des applets appartenant également au contexte C.

Accès à des objets de contexte différent. En général, accéder à un objet appartenant à un contexte différent est interdit par la politique de sécurité. Toutefois, la technologie Java Card autorise de tels accès dans des situations particulières où le partage de données est nécessaire. Ces mécanismes bien définis et sécurisés de partage de données sont définis par les quatre règles suivantes :

1. Toutes les méthodes et tous les champs de n'importe quel objet peuvent être accédés par le système, c'est à dire à partir du contexte JCRE.
2. En revanche, les applets ne sont pas autorisées à accéder aux données du système (appartenant au contexte JCRE). Pourtant certains services proposés par le système doivent pouvoir être accessibles aux applets. Ceci est possible dans Java Card au moyen d'*objets de point d'entrée* dans le système (*entry point objects*). Il s'agit d'objets appartenant au système, mais qui ont été marqués (par le système) comme contenant des méthodes de point d'entrée. Les méthodes de ces objets peuvent alors être invoquées à partir de n'importe quel contexte. Par contre les champs de ces objets sont toujours protégés par le firewall. Un partage de données entre le système et une applet n'est donc pas possible.
3. Un partage de données entre le système et une applet n'est possible qu'à l'aide de *tableaux globaux* (*global arrays*). Le système a le droit de marquer certains tableaux de ses objets de point d'entrée comme étant *globaux* et ils peuvent alors être accédés de n'importe quel contexte (mais pour des raisons de sécurité, aucune référence à un de ces tableaux ne peut être mémorisée, pour éviter une utilisation ultérieure dans des situations non spécifiées). C'est le cas par exemple du tableau d'octets en argument de la méthode `install` ou encore le *buffer APDU* permettant de partager les données de la commande entrante.
4. Une interaction entre applications de différents contextes est possible, mais de façon contrôlée grâce aux *interfaces partageables* (*shareable interfaces*) : lorsqu'une applet veut rendre

[1] *JCRE* signifie *Java Card Runtime Environment*, environnement d'exécution Java Card.

certaines données ou certaines méthodes accessibles aux autres applications[1], elle fournit un objet dont la classe implémente une interface partageable, c'est à dire une interface qui étend l'interface `javacard.framework.Shareable`. Une telle interface définit un ensemble de méthodes qui sont les services que l'applet rend accessibles aux autres applications. Ces méthodes peuvent être invoquées *à partir de n'importe quel contexte*.

Lors de ces mécanismes de partage, un changement de contexte (*context switch*) se produit, et il ne peut se produire de changement de contexte que dans ces quatre cas. Par exemple, à la mise sous tension de la carte, le contexte actif courant est le JCRE contexte. Lorsque le *card manager* transmet la commande à l'applet sélectionnée, la méthode `process` de l'applet est invoquée (ceci est possible grâce à la première règle) et son contexte devient le contexte actif courant.

Le firewall (pare-feu). Pour mettre en œuvre la politique de sécurité décrite jusqu'à présent, la technologie Java Card effectue des contrôles dynamiques de ces règles, assurés par son *firewall* (voir [93]). Cela signifie que pour chaque instruction *bytecode* (`getfield`, `putfield` ...), un ensemble de règles spécifiques est défini pour assurer la politique d'isolation. Lorsque la machine virtuelle interprète une instruction *bytecode*, elle contrôle que les conditions d'accès spécifiées par les règles du firewall sont remplies. Si elles ne le sont pas, une exception `SecurityException` est levée. Par exemple, la spécification Java Card précise que l'instruction *bytecode* `getfield` lève une exception `SecurityException` si le contexte courant n'est pas le contexte propriétaire de l'objet référencé à l'adresse `objectref`, où `objectref` est le sommet de la pile d'exécution.

1.2.4 Motivation d'une vérification formelle de l'isolation

La description qui vient d'être expliquée représente la spécification de la technologie Java Card définie par Sun (voir [93]). La plupart des constructeurs de cartes implémentent leur plateforme Java Card à partir de ces spécifications. La preuve que la spécification garantit certaines propriétés sécuritaires est donc primordiale. Nous sommes intéressés dans nos travaux (qui seront présentés dans le Chapitre 3) aux propriétés de *confidentialité* et d'*intégrité*.

Notre objectif était de démontrer formellement que la politique de sécurité définie par Java Card, sous la forme d'*ensembles* de règles de *firewall* pour *chaque* instruction *bytecode*, assure qu'une application ne peut obtenir aucune information et ne peut modifier aucune donnée appartenant à une autre application, de façon illégale (i.e. en n'utilisant pas les mécanismes de partage spécifiques décrits dans la section précédente).

Par la suite nous avons généralisé la vérification de ces propriétés d'*isolation* à tout le processus d'exécution d'une commande. En effet, comme décrit dans la présentation de l'architecture de Java Card, l'exécution d'une commande débute par des opérations du *card manager*. Ensuite une méthode m d'une applet donnée est interprétée par la machine virtuelle. Cette exécution consiste en l'interprétation successive de toutes les instructions *bytecode* de m et peut inclure des appels à des fonctions de l'API. Enfin, le *card manager* reprend le contrôle et envoie une réponse au terminal.

Le mécanisme du firewall décrit dans la section précédente n'intervient que lors de l'interprétation de *bytecode*, c'est à dire uniquement lorsque la machine virtuelle interprète des méthodes écrites en Java. En particulier, les opérations du *card manager* d'une part, ainsi que l'exécution

[1] Ce mécanisme d'interface partageable autorise la retransmission des données reçues à une tierce application. D'où certaines études que nous évoquerons (par exemple [19]), qui proposent une méthode pour contrôler ce flux transitif d'information.

de méthodes de l'API qui ne sont pas écrites en Java d'autre part, ne sont pas sous le contrôle du firewall.

En effet, les méthodes de l'API sont majoritairement des méthodes *natives*[1]. Par exemple, la méthode `javacard.framework.Util.arrayCopy` qui effectue une copie de tableau est native pour des raisons évidentes d'efficacité ; tandis que la méthode `javacard.framework.JCSystem.beginTransaction` est intrinsèquement native puisqu'il s'agit d'une demande directe au système. Ceci implique que les règles du firewall ne peuvent pas être directement appliquées dans ce contexte puisqu'il n'y a pas d'interprétation de *bytecode*. De plus, aucun mécanisme de sécurité n'est spécifié pour l'exécution de méthode API ; les décisions sont laissées au développeur. Pourtant, un tel mécanisme est crucial pour la sécurité de Java Card. Par exemple, considérons la méthode

```
static short arrayCopy(byte[] src,  short srcOff,
                       byte[] dest, short destOff, short length)
```

de la classe `javacard.framework.Util`. Cette méthode copie un tableau de taille `length` à partir du tableau source `src` donné en argument, dans le tableau `dest` donné en argument, et à partir des index `srcOff` et `destOff` dans ces tableaux. Aucune contrainte de sécurité sur les propriétaires de `src` et de `dest` n'est imposée par la spécification de cette fonction. Par conséquent, n'importe quelle applet pourrait voler le contenu de n'importe quel tableau de n'importe quel contexte, ce qui est clairement contraire aux objectifs de sécurité de Java Card. Aussi, des contraintes de sécurité devraient être ajoutées à la spécification. Dans notre cas, une extension évidente des règles du firewall imposerait que `src` and `dest` doivent tous les deux être accessibles à partir du contexte d'exécution (à savoir le contexte de la méthode qui a appelé la méthode `arrayCopy` puisque cette dernière est statique) et qu'une exception `SecurityException` est levée si ce n'est pas le cas.

Enfin, un flux d'information peut aussi provenir des opérations effectuées par le *card manager*. Par exemple, un flux d'information pourrait venir des ressources globales partagées (telles que le buffer APDU) utilisées par le *card manager* pour communiquer avec les applications. Contrairement au cas de l'API, ici certaines règles de sécurité sont spécifiées. Mais une fois encore, le fait que cet ensemble de règles assure les propriétés de confidentialité et d'intégrité reste à établir.

L'objectif d'une partie de nos travaux a donc été de modéliser formellement les propriétés de confidentialité et d'intégrité des données de différentes applets chargées sur une même carte et de prouver formellement qu'elles sont vérifiées tout au long du processus d'exécution d'une commande. Ces travaux sont décrits dans le Chapitre 3.

1.3 La gestion d'une mémoire Flash embarquée

Dans une deuxième partie de nos travaux, nous nous sommes intéressés à étendre la modélisation et la vérification formelle aux couches logicielles de plus bas niveau, i.e. aux programmes du système d'exploitation. Notre objectif était de répondre aux préoccupations du moment par l'utilisation de méthodes formelles pour une meilleure maîtrise des composants critiques. Nous nous sommes donc intéressés à un module responsable de la gestion d'un nouveau type de mémoire : la mémoire *Flash*.

Dans cette section, nous allons tout d'abord décrire les différents types de mémoire existants afin de mieux comprendre les particularités de la Flash. Nous détaillerons ensuite ces particularités, qui engendrent des contraintes d'utilisation spécifiques à la Flash, telles que l'utilisation

[1] Une méthode native est une méthode écrite dans un langage bas niveau tel que le langage C ou assembleur.

d'algorithmes de journalisation. Enfin nous décrirons le module de gestion de mémoire Flash étudié, permettant de gérer de façon générique des mesures d'*anti-tearing*.

1.3.1 Différents types de mémoires

Différents supports. Une *"mémoire"* désigne support de rétention d'information, comme un composant électronique capable d'enregistrer, conserver et restituer des informations. Le support physique utilisé pour fabriquer la mémoire peut être de différentes natures, variant ainsi le codage de l'information. Il existe essentiellement trois grandes catégories de support : le support magnétique, le support optique et le support semi-conducteur.

Le support magnétique se présente sous la forme d'une bande constituée de très fines particules magnétiques collées sur un support en plastique. En présence d'un champ magnétique, induit par un électro-aimant, ces fines particules s'alignent plus ou moins, suivant l'intensité du champ. Telles des objets en fer aimantés, les particules restent magnétisées, ce qui correspond alors à un enregistrement. Ainsi, en variant l'intensité du champ, il est possible de stocker de l'information sur la bande magnétique. Inversement, la lecture consiste à mesurer la tension électrique induite par le champ magnétique provoqué par le passage de l'aimant sur la bande. La bande magnétique est essentiellement utilisée pour le stockage de masse, comme dans un disque dur. On la trouve également sur les cartes de crédit, comme par exemple les cartes prépayées.

Le support optique, utilisé par exemple dans les CD-ROM ou DVD-ROM, est également destiné au stockage en masse de données. Les données sont représentées par des petites alvéoles sur une couche métallique réfléchissante, lues par un laser.

Enfin, le *circuit intégré*, mis au point en 1958 par Texas Instruments, est également un support de mémoire. Un circuit intégré, ou *puce électronique*, est un ensemble de composants électroniques (transistors, résistances, condensateurs, etc.) reliés entre eux dans un boîtier comportant des connecteurs d'entrée-sortie en métal semi-conducteur, généralement du *silicium*. Pour cette raison, ces mémoires sont appelées *mémoires à semi-conducteur*. Le terme de *puce* est dû a l'aspect du circuit intégré possédant des broches d'entrée-sortie ressemblant à des pattes (voir Figure 1.11). Les circuits intégrés les plus simples sont des portes logiques (i.e. représentant les opérations booléennes *et*, *ou*, *non*), les plus compliqués sont des microprocesseurs et les plus denses sont des mémoires. Dans un circuit intégré, l'information est enregistrée sous forme binaire, c'est à dire qu'elle est codée à l'aide de chiffres binaires, ou *bits* (pour BInary digiTs), valant 0 ou 1. Plusieurs composants ou procédés permettent de coder sous forme binaire, donnant naissance à différents types de mémoires. Les caractéristiques techniques des mémoires utilisant des circuits intégrés varient selon le type de composants utilisés. Ainsi certaines mémoires sont *volatiles*, c'est à dire que leur contenu est perdu lors de la mise hors tension de la puce, alors que d'autres sont au contraire *persistantes*, ou *non volatiles*, et leur contenu est préservé. Par ailleurs, certaines mémoires sont modifiables, ou *programmables*, alors que d'autres ont un contenu figé définitivement. La Figure 1.12 donne une classification des différents types de mémoire à semi-

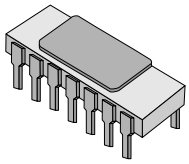


FIG. 1.11 – Une puce électronique

conducteur, que nous allons maintenant détailler. Un tableau récapitulatif sur les caractéristiques techniques de chaque type de mémoire est donné en fin de section, dans la Figure 1.15.

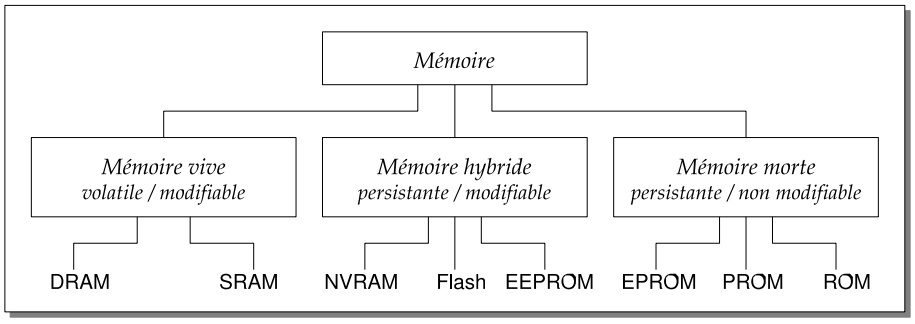


FIG. 1.12 – Les différentes types de mémoire à semi-conducteur

RAM. La mémoire vive ou *RAM* (*Random Access Memory*, mémoire à accès direct[1]) est une mémoire volatile et modifiable, permettant de stocker de manière temporaire des données lors de l'exécution d'un programme. Elle peut être *statique* ou *dynamique*. La mémoire statique (SRAM) est constituée de *bascules* (ou *flip-flop*). Une bascule est un ensemble de six transistors qui peut "basculer" vers un état logique ou l'autre, sous l'effet d'une impulsion électrique. La mémoire dynamique (DRAM), moins coûteuse, est constituée de petits condensateurs emmagasinant des charges formées par le courant qui les traverse. Lorsqu'il est chargé, l'état logique du condensateur est égal à 1, dans le cas contraire il représente le bit 0. Étant donné que la charge contenue dans un condensateur "fuit", les condensateurs doivent être constamment rechargés, ou *rafraîchis* à un intervalle de temps régulier. Par conséquent, chaque condensateur est couplé à un transistor permettant de "récupérer" ou de modifier l'état du condensateur. L'aspect volatile de la mémoire, statique ou dynamique, provient de la nécessité de la présence d'un courant électrique pour coder les informations. Ainsi, lors de la mise hors tension de la puce, toutes les données stockées en RAM sont perdues. La mémoire RAM est par ailleurs très coûteuse, son avantage majeur étant ses temps d'accès très rapides en lecture et en écriture.

Une variante de la RAM, la NVRAM (*Non Volatile Random Access Memory*), a pour principe d'adjoindre une batterie à une SRAM classique afin d'en préserver le contenu lors d'une mise hors tension. Elle devient alors persistante, mais est encore plus coûteuse.

ROM. La mémoire morte ou *ROM* (*Read Only Memory*, mémoire à lecture seule[2]) est une mémoire persistante mais non modifiable, utilisée pour stocker les données qui n'auront pas besoin d'être modifiées, comme celles nécessaires au démarrage du système, gravées une fois pour toute dans la mémoire.

Les premières ROM (souvent appelée *masked ROM*) étaient fabriquées à l'aide d'un procédé inscrivant directement les données binaires sur une plaque de silicium grâce à un masque.

[1] Le terme "random access" se traduirait plutôt par "accès aléatoire", mais dans les faits, cela correspond en fait à un accès *direct*, ou *non séquentiel*, indépendant des autres données, en opposition aux mémoires *SAM* (*Sequential Access Memory*) où il faut parcourir la mémoire de façon séquentielle depuis son origine pour trouver une adresse donnée, comme sur une bande magnétique.

[2] Bien que la ROM soit souvent opposée à la RAM, c'est également une mémoire à "accès aléatoire", ou accès direct.

Plus précisément, cette mémoire est composée d'une grille de conducteurs électriques, à l'origine isolés, où les lignes peuvent être reliées aux colonnes par des diodes ou des transistors. Le masque consiste à indiquer les emplacements où une diode doit apparaître dans la matrice (voir Figure 1.13).

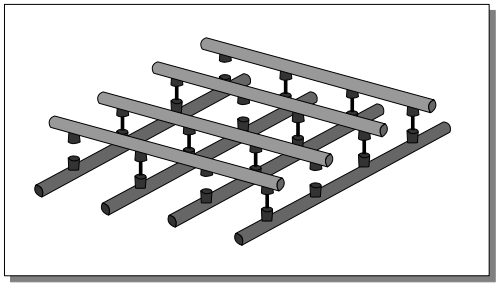


FIG. 1.13 – Principe de la ROM

Par essence même, cette mémoire est non modifiable. De plus, le coût de production du masque, ainsi que l'obligation de donner ce masque au constructeur de semi-conducteur qui créera la puce, font préférer à la ROM de nouvelles mémoires palliant ces problèmes.

PROM. La mémoire PROM (*Programmable Read Only Memory*, ROM programmable), a été mise au point à la fin des années soixante-dix par la firme Texas Instruments. Cette mémoire est une ROM où les liaisons sont assurées par des *fusibles* (ou des diodes où la jonction peut être "claquée"). Un appareil, le *programmateur*, permet de "griller" des fusibles (ou claquer des diodes) en appliquant une forte tension aux cases mémoire devant être marquées. La mémoire est à l'origine constituée uniquement de 1, la programmation permettant d'indiquer où se trouvent les 0.

La programmation peut être effectuée après la fabrication de la puce, par l'utilisateur et non plus le fabricant, mais elle reste irréversible et toute erreur est définitive.

EPROM. L'EPROM (*Erasable Programmable Read-Only Memory*, PROM effaçable) autorise, quant à elle, une ou plusieurs re-programmations ultérieures. Les fusibles aux intersections des lignes et des colonnes sont remplacés par des transistors A l'origine *bloqués*, ces transistors deviennent *passants* dès lors qu'une tension électrique les traversent. Leur état passant se maintient alors, à moins d'une irradiation d'une dizaine de minutes par des rayons ultraviolets qui les rendent de nouveau bloqués. Pour cette raison, les transistors d'EPROM ont la particularité d'avoir une petite fenêtre laissant passer les rayons ultra-violets (voir Figure 1.14).

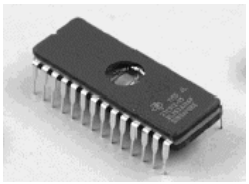


FIG. 1.14 – Une mémoire EPROM

La puce est donc dans un état non programmé à l'origine et peut être programmée par simple

courant électrique. Elle peut également être remise dans un état non programmé en présence de rayons ultra-violets, mais doit pour cela être retirée de son support. Par ailleurs, toute écriture *locale* est encore impossible (la puce entière est effacée puis programmée de nouveau).

EEPROM. L'EEPROM (*Electrically Erasable Programmable Read-Only Memory*, PROM effaçable électriquement) est une variante avantageuse de l'EPROM, puisqu'elle peut être effacée, non pas par des rayons UV, ce qui suppose de la retirer au préalable de son emplacement, mais par un courant électrique. Un autre type de transistors est utilisé, pouvant être bloqué à nouveau par simple impulsion électrique le traversant dans l'autre sens. Chaque transistor peut donc indépendamment passer de l'état bloqué à l'état passant *et inversement*. Cette mémoire accepte donc des *écritures*. Elle est cataloguée comme une mémoire *hybride* puisque persistante *et modifiable*. Elle a le comportement d'une RAM non volatile, mais reste coûteuse et surtout lente en écriture du fait de son effacement adresse par adresse. Par ailleurs, le nombre d'effacement n'est pas illimité étant donné que les transistors contiennent une couche isolante qui finit par s'user. L'EEPROM a donc un nombre limité de *cycles d'effacement*, correspondant à sa *durabilité*.

Flash. La mémoire Flash, également persistante et modifiable, présente de nombreux avantages techniques, mais une forte contrainte d'utilisation. Cette technologie a fait l'objet de nos travaux et est décrite en détail dans la section suivante.

Les caractéristiques techniques des principaux types de mémoire sont représentées dans le tableau de la Figure 1.15.

		RAM	Flash	EEPROM	EPROM	PROM	ROM
Volatile		oui	non	non	non	non	non
Programmable		oui	oui	oui	oui, avec un appareil	une seule fois, avec un appareil	non
Taille d'effacement		octet	page (8-128 kO)	octet	puce entière	-	-
Durabilité		illimitée	~10 000	~100 000	100	-	-
Coût (par octet)		très élevé	moyen	élevé	moyen	moyen	bas
Vitesse	lecture	60 ns	80 - 120 ns	150 ns	150 ns	150 ns	150 ns
	écriture / effacement	70 ns	10 – 17 µs	5-10 ms/octet ou 10ms/page	50ms	100ms	-
			1s/page		-	-	-
Taille d'un point mémoire		~1700 µm²	~200 µm²	~400 µm²	~200 µm²	_	~100 µm²

FIG. 1.15 – Caractéristiques des différents types de mémoire

1.3.2 La mémoire Flash

La technologie Flash. La mémoire Flash (ou EEPROM Flash), apparue vers le milieu des années quatre-vingt, est une variante de l'EEPROM plus rapide à effacer, plus dense, moins coûteuse et avec une plus grande durabilité. Également plus résistante aux chocs, en raison de l'absence d'éléments mécaniques, la mémoire Flash est utilisée dans de nombreuses applications - comme les appareils photos numériques, les téléphones cellulaires, les imprimantes, etc. Elle commence également à être utilisée dans les cartes à puces. Elle permet par exemple de contenir

tout le système d'exploitation de la carte, évitant ainsi son stockage en ROM qui implique un masquage et une transmission du masque au constructeur de semi-conducteur. Ceci ne pouvait être fait avec de l'EEPROM par manque de capacité.

En revanche, comme nous allons le voir plus loin, la mémoire Flash présente de fortes contraintes d'utilisation, liées à l'obligation d'effacer par bloc entier de mémoire, ce qui empêche toute modification "en place" (modification de la valeur d'une donnée sans changement d'adresse).

La densité de la mémoire Flash est due au fait que contrairement aux EEPROM classiques, utilisant 2 à 3 transistors par bit à mémoriser, la Flash mémorise un, voire plusieurs[1] bits à l'aide d'un seul transistor. Par conséquent, la taille d'un *point mémoire*[2] en mémoire Flash ($200\mu m^2$) est environ deux fois plus petite que celle d'un point mémoire en EEPROM (voir Figure 1.16). Sur une surface très restreinte comme les 25 millimètres carrés de la puce d'une carte, la mémoire Flash offre des capacités de stockage bien plus importantes que les autres mémoires. Par ailleurs, le prix de la mémoire étant lié à la quantité de silicium utilisée pour stocker un bit de donnée, la densité de la Flash rend également son coût de fabrication moins élevé.

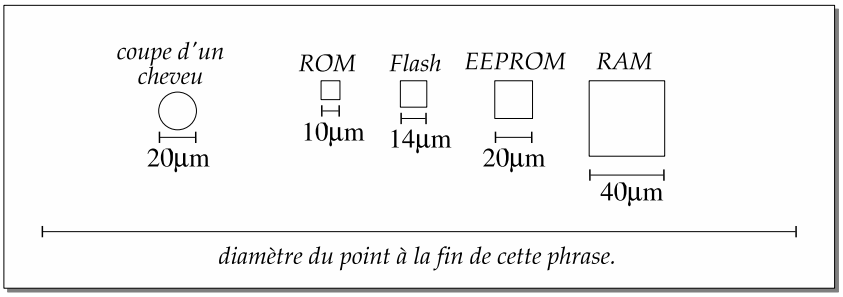


FIG. 1.16 – Comparaison des tailles approximatives des points mémoires des différents types de mémoire (pour une technologie 0,8μm). En comparaison, la limite de résolution d'un œil humain est de 40μm et la taille du point à la fin de cette ligne est de 400μm.

La rapidité d'effacement de la Flash est, quant à elle, due à l'ajout d'un circuit interne dans la puce, permettant d'effacer, non plus adresse par adresse comme dans l'EEPROM, mais par blocs entiers de mémoire, appelés des *pages* ou des *secteurs* (ensemble de plusieurs pages). Pour éviter des confusions entre les termes de *bloc*, également utilisé de façon générique, celui de *page* et celui de *secteur*, nous utiliserons le mot *page d'effacement*, ou simplement *page*, pour parler de l'effacement de la Flash. La puce est donc divisée en pages, chaque page étant parcourue par un circuit permettant de l'effacer. Le choix de la taille des pages doit être un compromis entre la vitesse d'effacement et la facilité de lecture et d'écriture. Par ailleurs, cette approche offre la possibilité de ne pas câbler certaines pages, comme celle utilisée pour stocker les informations nécessaires au démarrage du système, afin de les rendre impossible à effacer.

Cet effacement par page accroît la vitesse d'effacement, mais l'opération reste toutefois longue. De plus, il rend impossible toute modification en place d'une donnée, comme il sera expliqué au paragraphe suivant. Enfin, tout comme l'EEPROM, la mémoire Flash a une durabilité limitée

[1] La technologie appelée *multi-level cell structure* (structure de cellule à multiples niveaux) permet de stocker plusieurs bits d'information dans un seul transistor. Le secret repose sur la précision dans le chargement du transistor, afin d'avoir plus de deux niveaux observables (le niveau "chargé" et le niveau "non chargé"). Pour plus d'information, voir par exemple [69].

[2] quantité de silicium nécessaire au stockage d'un bit de donnée.

et l'usure des pages doit être nivelée afin de ne pas utiliser trop souvent les mêmes pages. Une gestion très particulière de la mémoire est donc nécessaire afin de gérer au mieux les écritures dans une page et son effacement dès lors qu'il devient nécessaire (voir par exemple [55]).

Problématique de l'effacement par page. L'effacement par page est un des avantages de la mémoire Flash puisqu'il est plus rapide qu'un effacement adresse par adresse. Mais il représente également une forte contrainte d'utilisation de la mémoire, puisque, comme nous allons le voir, une écriture "quelconque" n'est plus possible *en place*, c'est à dire qu'une donnée, stockée à une adresse donnée, ne peut pas être mise à jour avec une nouvelle valeur, à cette même adresse, à moins que la nouvelle valeur ait une forme particulière par rapport à la valeur initiale.

Plus précisément, en Flash, l'état *effacé* est représenté par le bit 1. La *programmation* consiste à écrire un 0 à la place d'un bit effacé. L'effacement consiste quant à lui à remettre la mémoire en état effacé, donc à mettre un 1 à la place d'un 0. Par conséquent, ce qui est "interdit", c'est le passage d'un 0 à un 1, à moins que la page de Flash ne soit entièrement effacée (voir Figure 1.17).

$1 \longmapsto 1$:	autorisé par octet	(*identité*)
$0 \longmapsto 0$:	autorisé par octet	(*identité*)
$1 \longmapsto 0$:	autorisé par octet	(*programmation*)
$0 \longmapsto 1$:	"INTERDIT" par octet (autorisé par page)	(*effacement*)

FIG. 1.17 – Principes de la mémoire Flash

Une remarque importante est que le passage de 0 à 1 n'est pas "interdit" dans le sens où une grave erreur se produit ou qu'une exception est levée. En fait l'écriture correspond à un "*et*" logique et si l'on essaie d'écrire un 1 à la place d'un 0, il ne se passe rien (on garde le 0). Ceci a pour conséquence qu'il est impossible de savoir si une écriture "interdite" a été effectuée. Seule la confiance en le code assure que chaque écriture a été effective dans la mémoire. Ceci est une remarque primordiale dans la motivation de vérification formelle du code. En effet, seule une telle vérification peut permettre de donner une assurance que toutes les écritures ont été effectives dans la mémoire Flash.

Les conséquences des principes de la mémoire Flash tels que présentés dans la Figure 1.17 sont les suivantes :

- une écriture est toujours possible sur une partie effacée de la mémoire ;
- une donnée de valeur v ne être mise à jour (en place, i.e. à la même adresse) avec la valeur v' que si tous les bits à 0 de v restent à 0 dans v'. Une telle écriture sera alors appelée *"écriture valide en Flash"*.

Par conséquent, il y a deux façons de mettre à jour une donnée de valeur v avec une valeur v' :

1. **[Écriture valide de petites données : en place]** Une première façon est de vérifier et comparer tous les bits de v et v' deux à deux et vérifier que l'écriture de v' à la place de v est un écriture valide. Si c'est le cas, alors la donnée peut être mise à jour en place. Étant donné que la probabilité que l'écriture soit valide diminue lorsque la taille de la donnée augmente, une écriture en place sera généralement effectuée uniquement pour des données de petite taille, où la vérification est triviale. De plus, ce procédé ne peut être appliqué indéfiniment et la donnée devient inutilisable après un certain nombre d'écritures (en effet, lors d'une écriture valide – excepté l'identité – le nombre de 1 de la donnée diminue strictement ;

il atteint donc zéro dans un temps fini et aucune écriture n'est plus possible). L'écriture en place sera donc généralement réservée aux petites données qui prennent uniquement quelques valeurs, dans un "ordre" bien défini et de façon irréversible. Typiquement, si un indicateur d'état n'a que trois valeurs possibles : "libre", puis "valide", puis "obsolète" de façon définitive, alors la valeur "libre" sera représentée par 11, "valide" par 01 (ou 10) et "obsolète" par 00 (voir Figure 1.18).

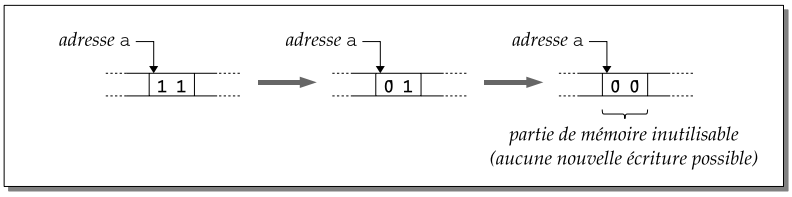


FIG. 1.18 – Écriture en place en mémoire Flash

2. **[Écriture en général : sur une partie effacée]** La deuxième façon d'effectuer une écriture en Flash est de s'arranger pour qu'elle se fasse toujours sur une partie effacée de la mémoire, car elle est alors toujours valide et aucune vérification n'est nécessaire. Éliminons tout de suite l'idée naïve de mémoriser le contenu de la page entière en mémoire vive, puis d'effacer cette page et enfin de réécrire la page en entier avec la modification des données en question. Cette méthode est clairement inefficace et très risquée en cas d'interruption. Une partie déjà effacée de la mémoire sera donc choisie pour écrire la nouvelle valeur v' de la donnée, ce qui implique de mettre à jour les références, i.e. signaler que la donnée a changé d'adresse (voir Figure 1.19). En revanche, on se heurte à la problématique de

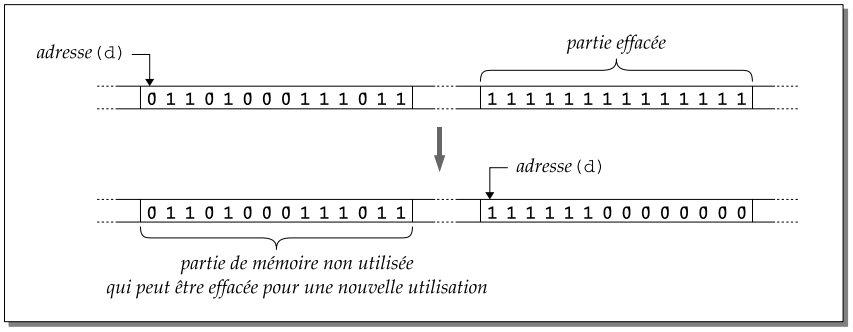


FIG. 1.19 – Écriture sur une partie effacée de la mémoire Flash

l'effacement par page et le nivellement de l'usure. En effet, il faut tout d'abord être en mesure de trouver un espace mémoire effacé suffisamment grand pour accueillir la nouvelle valeur. Ensuite, il faut choisir cet espace mémoire de telle sorte que les pages utilisées ne soient pas toujours les mêmes. Enfin, il faut pouvoir réutiliser l'emplacement de l'ancienne valeur de la donnée, par un procédé de *garbage collector* ("ramasse-miettes"). Ceci implique d'effacer la page entière dans laquelle la donnée se trouve, y compris les autres données de cette page. Une manière simple de faire est de choisir une page entièrement effacée pour la mise à jour de la donnée : la donnée modifiée, ainsi que toutes les autres données de la page initiale, sont recopiés dans la nouvelle page effacée. La page initiale peut alors être effacée pour une utilisation ultérieure (voir Figure 1.20). Mais ce procédé n'est pas

très efficace, puisqu'il nécessite l'écriture de toute une page et l'effacement de toute une page uniquement pour la mise à jour d'une donnée dans la page. De plus, le nivellement de l'usure nécessite un choix judicieux et non trivial de la page effacée à utiliser. C'est pourtant de cette façon que la mémoire Flash a commencé à être utilisée.

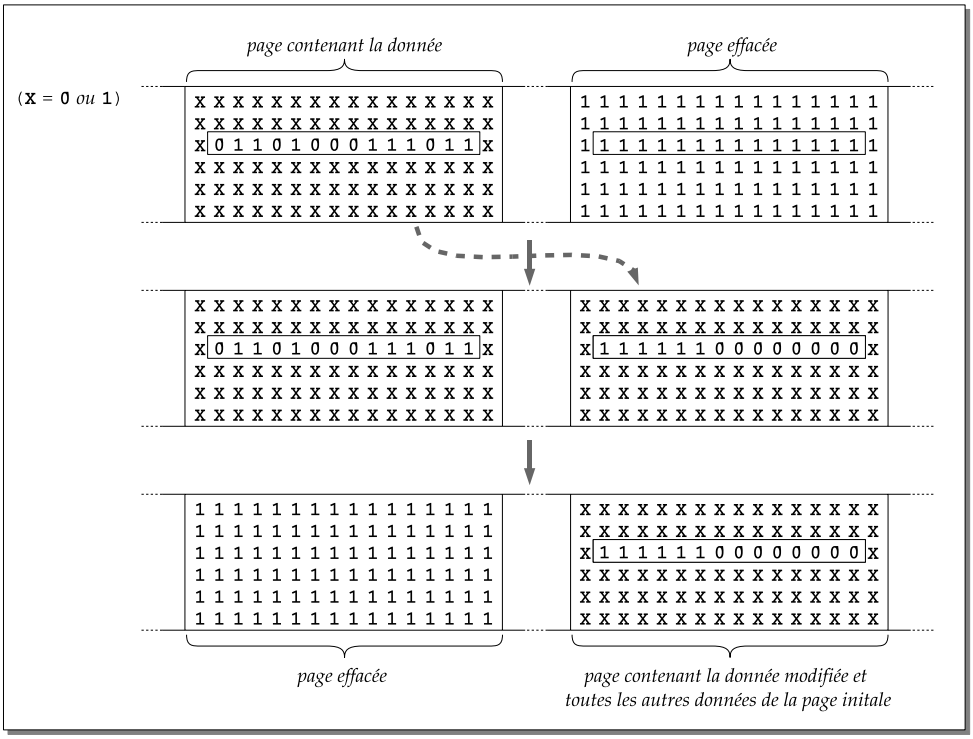


FIG. 1.20 – Écriture naïve en mémoire Flash

FTL. La mémoire Flash a tout d'abord été utilisée en remplacement d'un autre type de mémoire, pour des applications existantes. Une interface permettait alors de la faire passer pour l'ancien support mémoire. Par exemple, le remplacement d'un disque dur, qui est composé de plusieurs *secteurs*, par de la mémoire Flash consiste à décomposer la Flash en secteurs virtuels simulant les secteurs du disque dur. Une "couche de traduction" (*Translation Layer*) est alors utilisée pour rediriger les accès aux secteurs du disque à des accès aux secteurs virtuels en mémoire Flash. C'est le cas du *Flash Translation Layer*, ou "FTL" (voir [68]), défini dans un standard de *PCMCIA* (*Personal Computer Memory Card International Association*, association internationale de standardisation pour les cartes à circuit intégré). FTL permet à un système d'exploitation existant (ou d'autres applications embarquées), à l'origine conçu pour travailler sur un *disque dur*, d'être utilisé avec de la mémoire Flash de façon transparente. Étant donné que, comme nous venons de le voir, une écriture *en place* est impossible en mémoire Flash, une projection directe simulant un secteur de disque dur par un secteur virtuel de la Flash est impossible. Par conséquent, la mise à jour d'une donnée dans un secteur est gérée en choisissant un nouveau secteur virtuel entièrement effacé et en recopiant les données modifiées. L'ancien secteur virtuel est alors invalidé, i.e. "marqué" comme étant *obsolète*. Les secteurs virtuels sont donc dispersés en différents endroits de la mémoire Flash et la "couche de traduction" permet de garder la

trace de l'emplacement réel des secteurs (voir Figure 1.21). Elle donne l'illusion d'une écriture en place alors qu'un nouveau secteur virtuel (fraîchement effacé) est utilisé pour la modification. En particulier, FTL est utilisé pour les systèmes de fichier de type *FAT* (*File Allocation Table*),

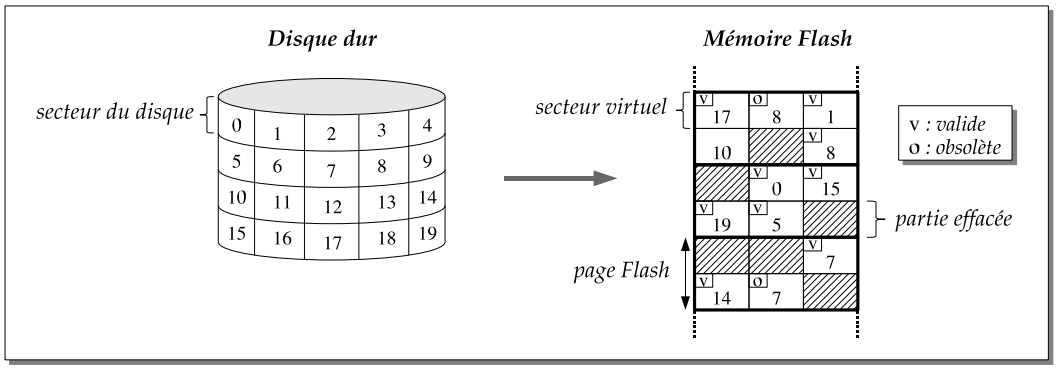


FIG. 1.21 – Principe du *Flash Translation Layer* (FTL)

où une table d'allocation de fichiers sert à retrouver les différents secteurs constitutifs du fichier. FTL redirige les liens de la table vers l'emplacement réel des secteurs simulés dans la mémoire Flash. Le *crash recovery* est assuré par l'utilisation d'un *état* indiquant si le secteur est en train d'être écrit ou si l'écriture est terminée. Si une interruption a lieu, tous les secteurs indiquant une écriture commencée seront invalidés. Une réorganisation de la mémoire, par un *garbage collector*, peut, par la suite, récupérer l'espace mémoire utilisé par des secteurs obsolètes, en recopiant tous les secteurs valides d'une page dans une nouvelle page fraîchement effacée, et en effaçant l'ancienne page.

La méthodologie de FTL permet d'utiliser de la mémoire Flash pour des applications existantes qui n'étaient pas conçues à l'origine pour ce type de mémoire. Mais, comme nous l'avons signalé, le procédé qui consiste à recopier, à chaque mise à jour d'une donnée dans un secteur, toutes les données du secteur dans un nouveau secteur fraîchement effacé, n'est pas efficace et rend le nivellement de l'usure délicat. En effet celui-ci repose essentiellement sur le choix du secteur effacé à utiliser.

Une gestion de la mémoire par *journalisation* permet de répondre aux problèmes posés par l'effacement par page de la mémoire Flash et le nivellement de son usure. L'idée est d'écrire de façon *séquentielle* dans la mémoire, page après page, ce qui assure la même usure à chaque page et permet de garder une partie toujours effacée de la mémoire pour les écritures.

1.3.3 La journalisation

La journalisation a été tout d'abord proposée comme moyen efficace de gestion de la mémoire sur un disque, indépendamment de la problématique de la mémoire Flash. Elle n'a été utilisée que plus tard comme moyen efficace de gestion de mémoire Flash, lorsque des systèmes ont été spécialement conçus pour travailler sur de la mémoire Flash, en remplacement des systèmes adaptés à la mémoire Flash avec une "couche de traduction".

Le principe de journalisation. La journalisation est un procédé qui consiste à noter dans un journal tout ce qui se passe dans un système au fur et à mesure de son fonctionnement.

Lorsqu'elle est utilisée comme un procédé de gestion de la mémoire, la journalisation consiste à ne plus mémoriser uniquement la donnée elle-même, mais tout l'historique de ses modifications. Chaque mise à jour de la donnée n'est plus effectuée en place, mais inscrite en fin de journal. Il y a deux façons de faire (voir Figure 1.22) : soit seule la modification est inscrite au journal, ce qui oblige à garder la valeur initiale de la donnée ainsi que toutes les modifications intermédiaires, soit la donnée modifiée est entièrement inscrite en fin de journal et la donnée devenue obsolète est *marquée* comme telle, afin de ne plus utiliser sa valeur, et également pour permettre une récupération de l'espace occupé, par un *garbage collector*.

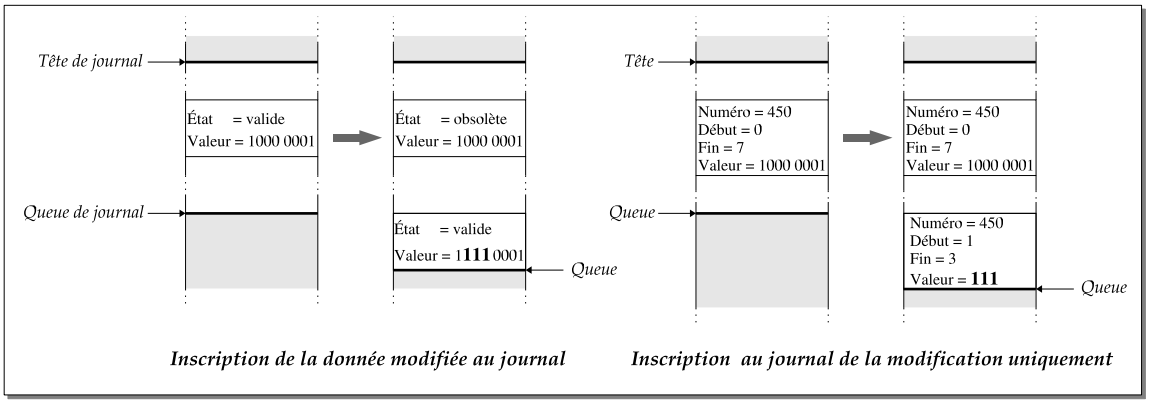


FIG. 1.22 – Principe de la Journalisation

La mémoire est donc organisée en journal, à savoir un ensemble d'*entrées* successives, donnant la valeur de nouvelles données ou indiquant une modification sur une valeur existante dans le journal. Un journal est composé de deux parties : une partie utilisée, située entre la *tête* et la *queue* du journal, et une partie non utilisée. Lorsque cela devient nécessaire, une réorganisation de la mémoire peut permettre de récupérer l'espace occupé inutilement par les entrées obsolètes. Cela consiste en général à recopier en queue de journal des données valides situées en début de journal pour pouvoir déplacer la tête de journal (voir Figure 1.23). Dans le cas

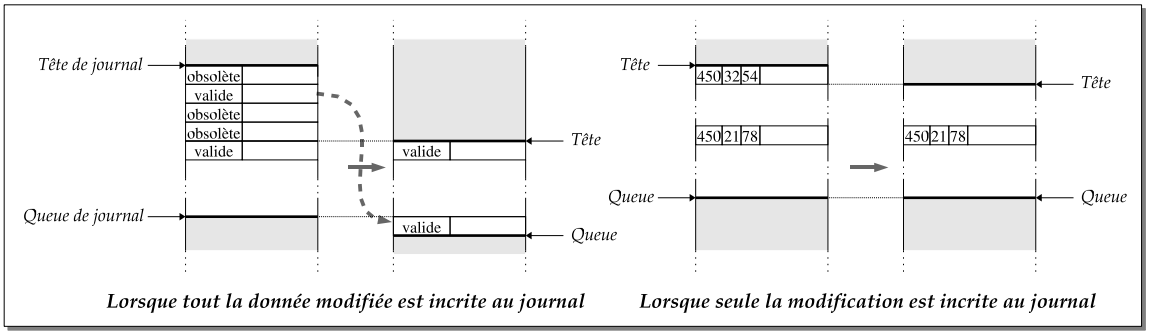


FIG. 1.23 – Récupération d'espace mémoire dans une mémoire journalisée

où la modification d'une donnée est effectuée en inscrivant *toute* la donnée modifiée en fin de journal, la récupération d'espace mémoire est plus facile étant donné que les données obsolètes sont explicites. En revanche, si seule la modification est inscrite dans le journal, la récupération d'espace mémoire est plus délicate puisqu'elle nécessite d'identifier les entrées qui ne sont plus

nécessaires. Par exemple pour une mise à jour de fichier, si une modification concerne les bits compris entre l'index 32 et l'index 54, elle peut devenir obsolète si une modification ultérieure, ou la combinaison de plusieurs modifications ultérieures, concerne les bits compris, par exemple, entre l'index 21 et l'index 78 (voir Figure 1.23).

La journalisation permet une meilleure gestion de la mémoire, mais présente tout de même des inconvénients significatifs, tels que la lenteur de lecture, lorsqu'il faut parcourir le journal pour connaître les modifications successives d'une donnée afin d'en connaître sa valeur actuelle ; ou encore la nécessité d'un "montage" de la mémoire, c'est à dire la mise à jour des variables mémorisées en mémoire vive telles que la tête et la queue de journal ; ou enfin l'occupation plus importante d'espace pour la même quantité d'information. La journalisation doit donc être utilisée lorsque les caractéristiques qu'elle offre améliorent de façon significative une gestion de mémoire classique. C'est le cas lorsqu'elle est utilisée avec de la mémoire Flash, bien que ce n'était pas sa vocation première.

LFS. La journalisation de la mémoire a d'abord été utilisée dans le but d'améliorer les temps d'écriture et le *crash-recovery* sur un disque, indépendamment de la problématique de l'utilisation de la mémoire Flash. Ainsi un nouveau type de systèmes de fichiers utilisant la journalisation est apparu, le plus connu étant le *LFS* (*Log-structured File System*, système de fichiers journalisé, voir [109]).

Dans un système de fichiers, les fichiers sont stockés de façon dispersée sur le disque et les données d'un même fichier peuvent être stockées en différents endroits du disque. Par conséquent, les temps d'écriture et de lecture sur le disque peuvent être très lents si la fragmentation est importante. C'est pourquoi les fichiers en cours d'utilisation sont généralement stockés en mémoire *cache*, mémoire vive très rapide, mais très coûteuse, placée généralement à l'intérieur même du processeur. Plus précisément, lorsqu'un fichier doit être utilisé, il est d'abord transféré en mémoire cache. Puis il est lu et modifié uniquement en mémoire cache, où les temps d'accès sont très rapides. La mise à jour du fichier sur le disque se fait à intervalles réguliers afin d'assurer une sauvegarde des modifications.

La lecture de données devient donc très efficace puisqu'elle n'est faite sur le disque que pour transférer un fichier en mémoire cache. Les temps de modification sont également améliorés puisque la mise à jour sur le disque est moins fréquente, mais ils restent lents à cause de la recherche des données à modifier.

Le but du LFS est d'améliorer le temps des écritures sur le disque, en structurant le disque entier sous forme de journal et en mémorisant uniquement les *modifications* en mémoire cache (voir Figure 1.24). Une écriture se faisant toujours en fin de journal, elle ne nécessite plus de recherche des données à modifier. De plus, toutes les mises à jour des fichiers mis en mémoire cache peuvent être faites avec une unique écriture sur le disque. Une *entrée* du journal est donc composée des données à modifier, ainsi que des informations nécessaires pour accéder rapidement au fichier. Le temps nécessaire à un *crash-recovery* est également amélioré grâce à des points de contrôle (*checkpoint*) dans le journal indiquant que, jusqu'à ce point de contrôle, le journal est consistent et complet. Un parcours complet du disque n'est donc plus nécessaire, seule la partie la plus récente du journal est à inspecter. Enfin, pour plus d'efficacité, le disque est décomposé en *segments*, avec un procédé de nettoyage des segments lorsqu'ils sont trop fragmentés.

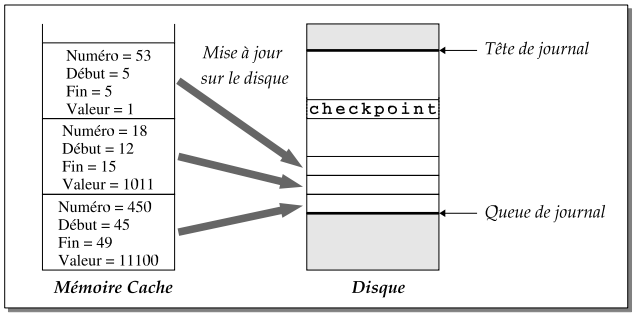


FIG. 1.24 – Principe du *Log-Structured File System* (LFS)

JFFS. Le premier système de fichiers spécifique à la mémoire Flash est le *JFFS* (*Journalling Flash File System*, voir [117]). Il utilise pleinement le principe de la journalisation, très adapté à la gestion de la mémoire Flash. En effet, les écritures se faisant toujours en fin de journal, il est plus facile de faire en sorte que ce soit toujours une partie effacée. De plus l'écriture séquentielle assure le même niveau d'usure pour toutes les pages.

Le principe du JFFS est très proche de celui du LFS (voir Figure 1.25). La mémoire est

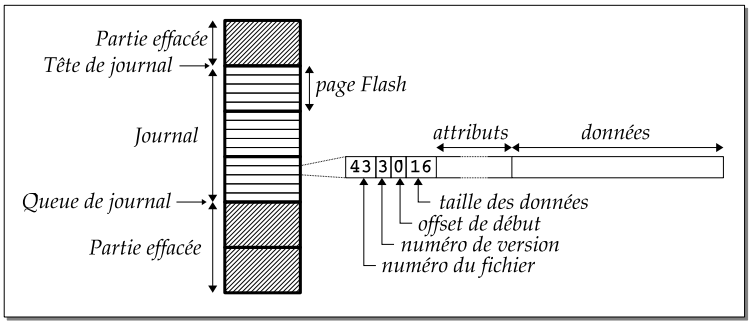


FIG. 1.25 – Principe du *Journalling Flash File System* (JFFS)

à l'origine effacée et la partie "non utilisée" du journal (celle qui ne se situe pas entre la tête et la queue du journal) garde la propriété d'être en état effacé. Quant à la partie utilisée du journal, aucune écriture n'y est effectuée afin d'éviter une écriture non valide en Flash (exceptées des "écritures valides de petites données" qui peuvent être effectuées en place, comme expliqué page 43).

Chaque entrée du journal est associée à un fichier. Elle contient, soit les données du fichier à sa création, soit une modification du fichier. Elle est donc composée d'un en-tête et des données en question. Chaque fichier est identifié par un numéro unique, contenu dans l'en-tête de l'entrée. Afin de tracer les modifications successives d'un fichier, chaque en-tête contient également un numéro de version. Enfin, il contient l'index indiquant où commence la modification dans le fichier, ainsi que la taille des données. En plus des données nécessaires à la journalisation, l'en-tête contient les attributs (*metadata*) du fichier, tels que son nom, ses droits d'accès, son répertoire parent, etc.

De même que pour le LFS, certaines entrées peuvent être rendues obsolètes par une ou plusieurs modifications ultérieures. Dans l'exemple donné dans la Figure 1.26, la version 3 du fichier n°45 rend ses versions 1 et 2 obsolètes. En effet, si les versions 1 et 2 sont "oubliées" et

que la version 3 succède à la version 0, la modification du fichier est la même. Un système de

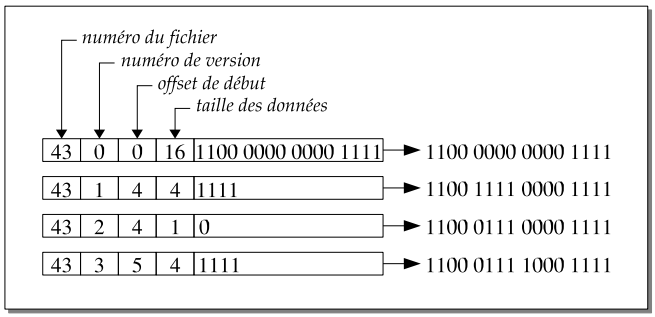


FIG. 1.26 – Modification d'un fichier dans le JFFS

récupération d'espace mémoire, ou *garbage collector*, permet de libérer l'espace utilisé inutilement par les données obsolètes. Le procédé est le même que celui expliqué dans le principe de la journalisation, à savoir recopier les données valides de début de journal en fin de journal. Étant donné que l'effacement se fait par page, le but est plus précisément d'effacer la première page du journal. Chaque entrée de cette page est donc examinée pour savoir si elle est obsolète, compte tenue des entrées ultérieures. Si elle l'est, l'entrée suivante est examinée. Si elle est valide, elle est recopiée en fin de journal et l'entrée suivante est examinée. Une éventuelle optimisation permet de fusionner plusieurs entrées au moment de cette démarche. Lorsque la fin de la page est atteinte, celle-ci peut être effacée, puisque toutes les données valides ont été recopiées en queue de journal. La tête de journal se trouve alors en début de la page suivante (voir Figure 1.27).

1.3.4 Description du module de gestion de mémoire Flash étudié

Mesures d'*anti-tearing*. Comme expliqué en Section 1.1.5.2, la possibilité d'arrachage de la carte doit être prise en compte dans les programmes embarqués. Un certain nombre de mesures, comme les transactions, sont donc mises en place pour assurer la propriété d'*anti-tearing*, à savoir la cohérence de la carte lors de sa remise sous tension. Le module que nous avons étudié permet de gérer, de façon générique, de telles mesures d'*anti-tearing*, en présence de mémoire Flash. Le module est générique dans le sens qu'il peut être appliqué à tout type de mesure et peut être étendu à un nombre quelconque de mesures. En général, les mesures d'*anti-tearing* sont associées à une fonction d'*abandon* et ont besoin de mémoriser un certain nombre de données, qui seront utilisées par la fonction d'abandon pour remettre la carte dans un état cohérent si un arrachage a eu lieu. Ces données sont donc stockées en mémoire persistante, ici en mémoire Flash, et sont journalisées pour une meilleure utilisation de la mémoire.

Par exemple, le mécanisme de transaction permet d'assurer qu'un ensemble d'instructions est exécuté de manière atomique, en encapsulant les instructions entre un appel à une fonction de début de transaction et un appel à une fonction de fin de transaction. À partir de l'appel à la fonction de début de transaction, un état indique qu'une transaction est en cours et toutes les opérations effectuées jusqu'à l'appel à la fonction de fin de transaction sont mémorisées. Si un arrachage a lieu avant la fin de la transaction, alors, lors de la remise sous tension, l'état indique qu'une transaction était en cours et toutes les opérations mémorisées sont annulées. Le problème est que la prochaine transaction ne pourra pas utiliser le même espace mémoire que la précédente pour mémoriser les opérations effectuées, puisque la mémoire Flash n'autorise pas d'écriture en place. D'où l'utilisation d'un journal, où chaque entrée correspond à une nouvelle

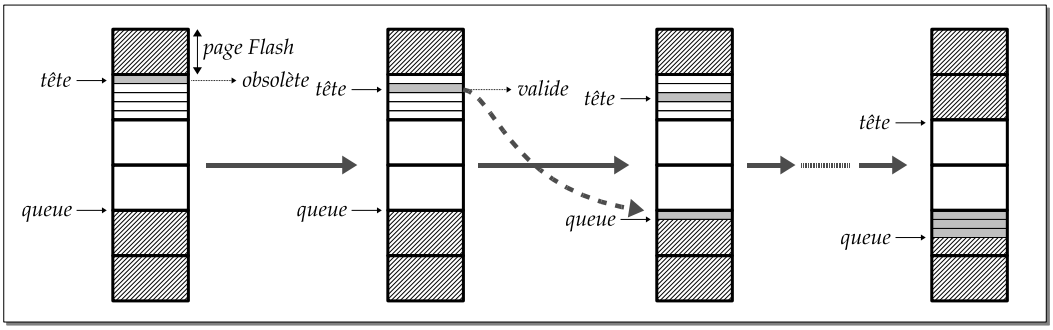


FIG. 1.27 – Récupération d'espace mémoire dans le JFFS

transaction et contient les données nécessaires pour gérer cette transaction (comme indiqué dans la Figure 1.28). Les anciennes entrées sont donc toutes obsolètes et l'entrée courante est soit utilisée, si une transaction est en cours, soit libre si aucune transaction n'est en cours.

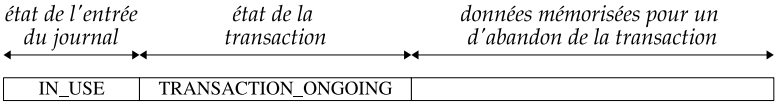


FIG. 1.28 – Une entrée du journal gérant les transactions

Un autre exemple de mesure d'*anti-tearing* permet d'effacer une partie d'une page de mémoire Flash, en assurant que les parties non effacées de la page seront conservées, même en cas d'arrachage. Cela consiste à mémoriser les données à conserver, ainsi que le numéro de la page d'origine. Puis un état indique que l'effacement a commencé et la page entière est effacée. Enfin, les données mémorisées sont recopiées sur la page nouvellement effacée et l'état indique que l'effacement est terminé. De même, cette opération peut être gérée à l'aide d'un journal, où chaque entrée contient l'état de l'effacement, le numéro de la page et les données mémorisées (comme indiqué en Figure 1.29). En cas d'arrachage, si l'état indique qu'un effacement était en cours, la page dont le numéro est indiqué est effacée et les données mémorisées dans le journal sont recopiée dans la page. Cela assure bien que les données à conserver ne seront pas perdues, même en cas d'arrachage.

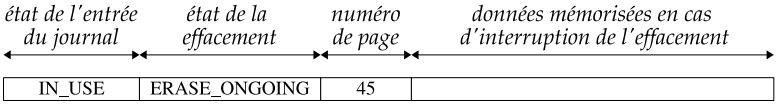


FIG. 1.29 – Une entrée du journal gérant les effacements

Le module générique de notre cas d'étude permet de gérer ce genre de journaux, pour tout type de mesure d'*anti-tearing*. Une entrée contient un état indiquant si elle est utilisée ou libre, un entête spécifique à chaque journal et un ensemble de données, dont la nature est également spécifique à chaque journal. Nous allons décrire dans la suite les différents composants de ce module, qui à fait l'objet de notre étude formelle décrite dans le Chapitre 6 et utilisant les résul-

tats de nos travaux présentés dans les Chapitres 4 et 5. Notons que la description qui va suivre représente une tâche importante des travaux effectués puisqu'elle a été réalisée essentiellement à partir du code source et constitue donc une première étape de formalisation du comportement du module.

Gestions des journaux. Le module contient un ensemble de variables globales, contenant les données des différents journaux, et de fonctions de manipulation des journaux. Les journaux sont utilisés de manière suivante (voir Figure 1.30) : lorsqu'aucune opération (comme une transaction ou un effacement) n'est en cours, la queue du journal pointe sur une entrée libre. Lorsque l'opération commence, l'entrée est marquée comme utilisée et sert pour stocker l'entête et les données nécessaires à un abandon éventuel de l'opération en cas d'arrachage. Lorsque l'opération est terminée, la queue du journal est simplement déplacée à l'entrée suivante (qui est obligatoirement libre). Il n'est pas nécessaire d'indiquer que l'entrée qui vient d'être utilisée est obsolète puisque la particularité de ces journaux est de n'avoir qu'une seule entrée valide, toutes les autres étant obsolètes.

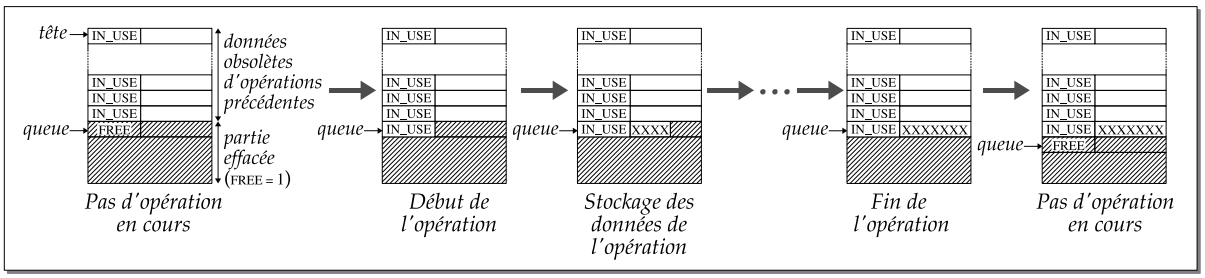


FIG. 1.30 – Écriture dans un journal

Du fait qu'au plus une entrée est valide dans un journal, la récupération d'espace mémoire n'implique pas de sauvegarde ou de transfert de données, mais consiste tout simplement à effacer le journal dès que l'on en atteint la fin. Plus précisément, lorsqu'une opération est terminée et que la queue de journal doit pointer sur l'entrée suivante, si cette entrée suivante est en dehors de l'espace alloué pour le journal, alors le journal est entièrement effacé et la queue pointe sur la tête du journal. La tête de journal se situe donc toujours à l'origine de l'espace alloué, et la queue se situe entre l'origine et la fin de cet espace alloué. (voir Figure 1.31).

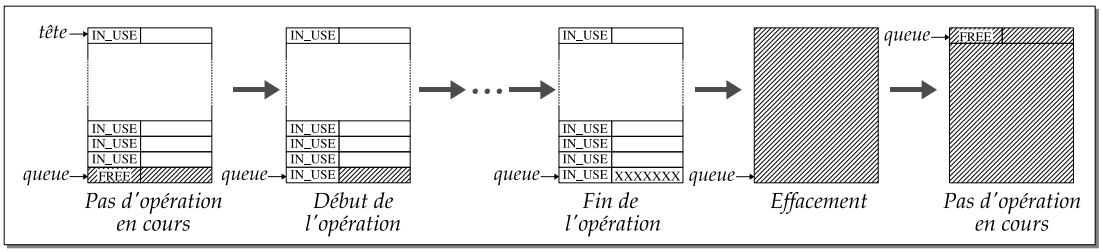


FIG. 1.31 – Récupération d'espace mémoire : lorsque la fin du journal est atteinte

Lorsqu'un arrachage a lieu, l'opération associée à chaque journal doit être *abandonnée*. Par conséquent, lorsque la carte est remise sous tension après une interruption, le journal est parcouru depuis son origine pour trouver la dernière entrée utilisée du journal. L'entête spécifique de

l'entrée est analysé pour savoir si l'opération a été interrompue, et si c'est le cas, la fonction d'abandon spécifique à l'opération est appelée. Par exemple, dans le cas des transactions, l'entête contient l'état de la transaction. S'il indique qu'une transaction était en cours, alors la fonction d'abandon de la transaction va annuler toutes les opérations mémorisées dans les données de l'entrée. De même, dans le cas de l'effacement d'une page de Flash, si l'entête indique qu'un effacement était en cours, alors la fonction d'abandon de l'effacement consiste à effacer la page dont le numéro est également indiqué dans l'entête et de recopier, dans la page effacée, les données mémorisées dans l'entrée du journal.

Notons que la dernière entrée utilisée du journal pourrait être trouvée directement grâce à un pointeur sur la queue du journal. Mais étant donné que la valeur de ce pointeur est mise à jour très souvent (et qu'elle peut être retrouvée par un parcours du journal), elle est uniquement stockée en mémoire RAM, dans une des variables globales définies dans le module (sa valeur est donc perdue en cas d'arrachage).

Variables globales. La gestion de chaque journal nécessite de connaître l'emplacement où il a été alloué (l'adresse et la taille de l'espace alloué), la taille des données qu'il manipule, la fonction d'abandon qu'il faut appliquer si un arrachage a eu lieu et l'entrée courante du journal. Les variables globales suivantes sont donc définies :

- *JournalRegistry* contient les informations relatives à chaque journal, à savoir l'adresse où il est stocké en mémoire, le nombre de *secteurs*[1] de Flash qu'il occupe, la taille des entrées du journal et enfin un pointeur sur la fonction d'abandon correspondant à l'opération gérée par le journal. Ces informations intrinsèques à chaque journal ne sont jamais modifiées.
- *JournalContext* est une variable stockée en mémoire RAM contenant les objets souvent mis à jour et dont il n'est pas indispensable de conserver la valeur après une mise hors tension. Cette variable contient par exemple le pointeur sur la queue de chaque journal (l'entrée courante du journal). Elle contient également le pointeur sur l'*état d'effacement courant* de chaque journal, à savoir l'index de l'état courant dans la variable *JournalStateRegistry* suivante.
- *JournalStateRegistry* est une variable permettant de gérer un effacement des journaux possédant la propriété d'*anti-tearing*. Elle est composée des *états d'effacements* de chaque journal, comme décrits dans le prochain paragraphe.

Ces trois variables globales contiennent des informations indépendantes pour chaque tableau. Ce sont donc des tableaux à N éléments, où N correspond au nombre de journaux gérés. La vue globale du module générique d'utilisation de journaux est donnée en Figure 1.32.

États d'effacement. Lorsque la fin d'un journal a été atteinte, le journal doit être entièrement effacé pour que de nouvelles entrées puissent être utilisées. Si un arrachage survient pendant l'effacement du journal, les données qu'il contient sont incohérentes et ne doivent pas être interprétées comme des entrées normales du journal. Il est donc indispensable de détecter qu'un arrachage a eu lieu pendant l'effacement, pour effacer de nouveau le journal.

L'*état d'effacement* d'un journal est donc mémorisé. Cet état peut être *en cours* (*ongoing* en anglais) si l'effacement du journal a commencé, ou alors *terminé* (*committed* en anglais). Ainsi, lorsque la carte est mise sous tension, l'état de chaque journal est consulté et si un état indique que l'effacement d'un journal était en cours, alors le journal est de nouveau effacé.

[1] Rappelons qu'il existe deux granularités pour l'effacement en Flash : un effacement par page et un effacement par secteur, où un secteur est un ensemble de plusieurs pages.

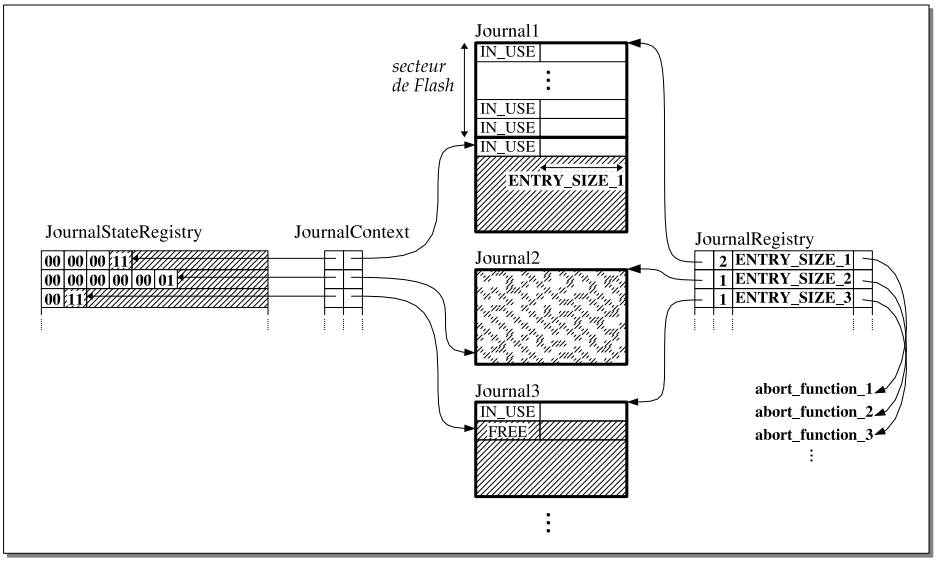


FIG. 1.32 – Vue générale du module générique d'utilisation de journaux

On remarque que les valeurs des états ne doivent pas être perdues lors d'une mise hors tension. Elles sont donc stockées en mémoire Flash. Cependant, elles sont également souvent mises à jour. Elles sont donc à leur tour *journalisées*. En d'autres termes, un tableau organisé sous forme de journal va contenir les valeurs successives de l'état d'effacement d'un journal (voir Figure 1.33). A l'origine effacé, il indique qu'aucun effacement n'a encore eu lieu. Lorsqu'un effacement commence, la queue du tableau prend la valeur correspondant à l'état `ongoing`. Et lorsque l'effacement est terminé, la queue prend la valeur correspondant à l'état `committed` et la queue est "incrémentée" pour pointer sur l'entrée suivante du tableau, indiquant qu'aucun effacement n'a lieu.

Il y a donc trois états possibles : `ongoing`, `committed` et `unused` qui indique qu'aucun effacement n'a lieu. Étant données les contraintes en espace mémoire sur une carte à puce, on cherche à utiliser le moins d'espace possible pour mémoriser les informations. Ici, seuls deux bits sont nécessaires pour stocker trois valeurs. Par ailleurs, les valeurs sont choisies en fonction des règles d'écritures autorisées en mémoire Flash. Ainsi, l'état `unused` prendra la valeur 3, soit $(11)_2$ en chiffres binaires, puisqu'il correspond à un état effacé de la mémoire. Puis l'état `ongoing` prendra la valeur 2, soit $(01)_2$ et l'état `committed` prendra la valeur 0, soit $(00)_2$.

L'état courant d'effacement, i.e. l'index de l'état courant du tableau, est mémorisé en mémoire RAM, dans la variable *JournalContext*. Sa valeur est donc perdue lors de l'arrachage. Par conséquent, lors de la remise sous tension après un arrachage, le tableau d'états d'effacement sera inspecté. S'il n'est pas *cohérent*, alors le journal correspondant est de nouveau effacé, puis le tableau lui même est également entièrement effacé. Le tableau d'états est dit *cohérent* s'il est organisé en journal et que tous les états qu'il contient dans sa partie non effacée ont la valeur `ongoing`. Autrement dit, le début du tableau ne doit être composé que d'états `ongoing` et la fin ne doit être composée que d'états `unused` (voir Figure 1.34).

Les fonctions. Les principales opérations possibles sur un journal sont : le marquage d'une entrée comme étant utilisée avant de commencer à écrire des données, l'"incrémentation" de la queue du journal vers une entrée libre, lorsque l'opération de l'entrée courante est terminée, et

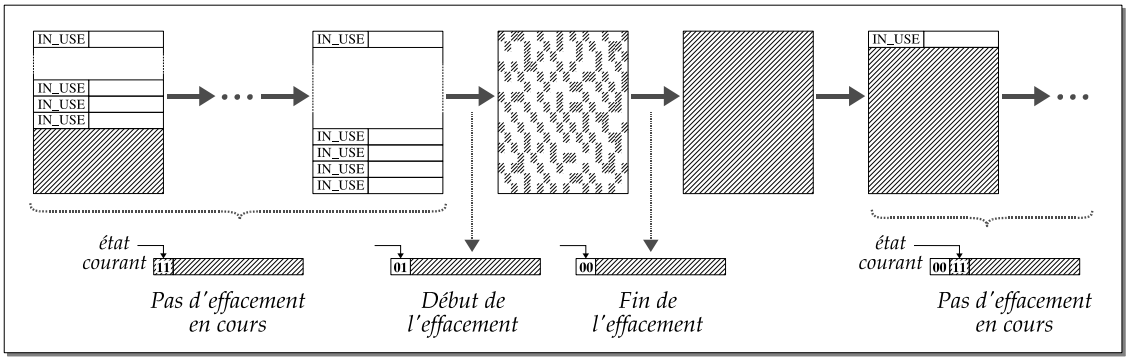


FIG. 1.33 – Tableau des états d'effacement

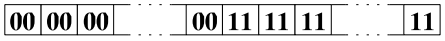


FIG. 1.34 – Tableau d'états d'effacement cohérent

enfin l'effacement du journal en entier lorsque la fin est atteinte. L'écriture des données-mêmes d'une entrée est spécifique à chaque opération (transaction, effacement de page, etc) et ne fait donc pas partie de ce module générique.

Par ailleurs, une fonction de *montage* (*mount* en anglais) est appelée à chaque mise sous tension de la carte. Elle contient les vérifications permettant de déterminer s'il y a eu un arrachage, et, si c'est le cas, les opérations nécessaires à la récupération d'un état cohérent. Ces opérations comprennent, d'une part, l'analyse de l'état d'effacement des journaux, pour un éventuel nouvel effacement du journal, et, d'autre part, un appel à la fonction d'abandon de chaque journal, qui effectuera les opérations nécessaires, suivant la valeur de l'entête de l'entrée courante. La fonction de montage contient également la mise à jour des données qui étaient stockées en mémoire volatile et qui ont été perdues.

Enfin, une fonction d'initialisation est utilisée, une seule fois, à la création de la carte. Les journaux sont effacés ; le tableau des états d'effacement des journaux, *JournalStateRegistry*, est également effacé ; les informations intrinsèques aux journaux sont stockés définitivement dans la variable globale *JournalRegistry* ; enfin les pointeurs de *JournalContext* sont initialisés : tous les états d'effacements sont *inutilisés* et la queue de chaque journal pointe à sa tête.

1.4 Conclusion

Omniprésentes dans des domaines de plus en plus variés, les cartes à puce doivent répondre aux exigences du marché, tout en faisant face à des enjeux sécuritaires fondamentaux. Ce chapitre a présenté deux composants majeurs de la carte à puce, qui ont fait l'objet de nos travaux : la machine virtuelle, ici celle de Java Card, et le système d'exploitation, illustré ici par un module de gestion de mémoire Flash.

Dans la technologie Java Card, plate-forme multi-applicative et ouverte, la protection des données de chaque application est assurée par une réglementation stricte de toute interaction entre applications, par un ensemble de règles d'isolation appelé le *firewall*. Étant donné que ces règles sont définies de façon indépendante pour chaque instruction possible de l'application, le

fait que l'ensemble de ces règles assure la confidentialité et l'intégrité des données des applications mérite une vérification. C'est pourquoi une partie de nos travaux s'est concentrée sur la preuve *formelle* que les propriétés de confidentialité et d'intégrité sont respectées durant toute exécution de commande par une application quelconque dans la plate-forme Java Card.

Par ailleurs, les cartes à puces sont livrées à des millions d'exemplaires, rendant impossible toute modification du code embarqué et par conséquent très coûteuse toute erreur logicielle. D'où un besoin, ici encore, de vérification *formelle*, motivé par la volonté de produire du code sûr. Nous avons donc cherché à trouver une méthode de vérification de propriétés sécuritaires de code source embarqué, et en particulier celui d'un système d'exploitation, composant central de la carte à puce. Pour cela nous avons étudié un module du système d'exploitation : celui de la gestion d'une mémoire Flash embarquée.

La mémoire Flash est un nouveau type de mémoire présentant de nombreux avantages. À la fois persistante et modifiable, elle possède également des temps d'effacement plus rapides et offre de plus grandes capacités que la mémoire EEPROM. Mais elle possède également de fortes contraintes d'utilisation, puisque l'effacement n'est possible que par page entière de mémoire. Les algorithmes de gestion, comme la journalisation, sont donc plus compliqués et leur correction est plus difficile à établir.

Les méthodes formelles représentent un moyen de vérifier que les différents composants d'une carte à puces vérifient certaines propriétés de sécurité et permettent également de renforcer la confiance accordée au code source, même bas niveau, de la carte à puce.

Chapitre 2

Vérification formelle de programmes

Résumé

Dans ce chapitre, nous présentons les principes généraux des méthodes formelles et les caractéristiques des principales techniques (voir par exemple [37, 111]). Nous évoquerons en particulier le système de preuve Coq (voir [113]) et l'outil de vérification de programme Caduceus (voir [53] et Section 4.2.1). Puis, l'application des méthodes formelles aux cartes à puces et les nombreux travaux s'y rattachant seront présentés.

Sommaire

2.1 Introduction aux méthodes formelles

2.1.1 Motivation

L'ubiquité grandissante des programmes, à des positions de plus en plus critiques (aérospatial, régulation de trafic, assistance au pilotage de véhicules[1], etc.) ou moins critiques mais ayant un enjeu financier important (transactions bancaires, ou logiciels dupliqués à des millions d'exemplaires engendrant des coûts de mise à jour prohibitifs) renforce les exigences demandées à un logiciel. Quelques exemples spectaculaires d'erreurs logicielles, engendrant des pertes financières colossales et parfois même des pertes humaines, confirment la nécessité absolue d'augmenter le niveau de sûreté des programmes.

L'exemple le plus connu et le plus cité est sans doute celui de l'explosion du premier lanceur Ariane 5, le 4 juin 1996, après 40 secondes de vol. L'erreur était logicielle : elle provenait d'un composant (un gyroscope), développé pour Ariane 4 et transféré à Ariane 5 sans reprise des spécifications, ni tests complémentaires. Le code contenait une conversion d'un nombre flottant codé sur 24 bits (correspondant à la vitesse horizontale de la fusée) en un nombre entier codé sur 16 bits. Cette conversion était correcte dans le cas d'Ariane 4, où le nombre flottant ne dépassait jamais 32768 (plus grand entier *signé* pouvant être stocké sur 16 bits) compte tenu des trajectoires de vol possibles pour Ariane 4. Mais les trajectoires de vol avec Ariane 5, sensiblement différentes, notamment en phase de décollage, ont provoqué ce dépassement, rendant la valeur du nombre entier incohérente. Cette valeur a été transmise au calculateur de pilotage, provoquant un comportement anormal de la fusée et obligeant son auto-destruction. La fusée avait coûté un demi milliard d'euros, sur un projet de sept milliards d'euros dont la réputation a été sérieusement atteinte.

Le 25 février 1991, pendant la guerre du Golfe, un missile anti-missile Patriot ne parvient pas à détecter et intercepter un missile Scud irakien, qui tue 28 soldats américains. La commission d'enquête a conclu à une erreur *logicielle* dans le calcul du temps de parcours. Ce temps était calculé à partir de l'horloge interne du système, qui comptait en dixième de seconde. Or, le nombre 1/10 n'a pas d'écriture finie dans le système binaire (1/10 s'écrit 0,1 en décimal et 0,0001100110011001100110011... en binaire). Par conséquent, son stockage dans le système, sur un registre de 24 bits, introduisait une erreur de 0,00000000000000000000000011001100... en binaire, soit environ 0,000000095 en décimal, et ce à chaque top de l'horloge. Le système n'était donc pas destiné à fonctionner "longtemps". Or, ce 25 février, il était en fonctionnement depuis plus de 100 heures, entraînant une erreur de 0,34 secondes ($0{,}000000095 \times 100 \times 60 \times 60 \times 10$), laissant le temps au missile Scud de parcourir plus de 500 mètres et être hors du champ de recherche du Patriot.

Le 23 septembre 1999, la sonde Mars Climate Orbiter s'écrase sur le sol martien. Ses moteurs de freinage ne se sont mis en route qu'à une altitude de 57 kilomètres, au lieu des 140 kilomètres prévus. L'erreur était là encore *logicielle* : une partie du programme avait été écrit par une équipe utilisant une unité de mesure anglo-saxonne pour la force du moteur de freinage (*pound-force*), l'autre équipe travaillant dans le système métrique international (Newton). La perte financière s'élève à 160 millions d'euros.

Ces exemples, parmi une multitude d'autres moins spectaculaires, ont poussé les industries à mettre en œuvre de plus en plus de moyens pour le test et la validation de programme, voire même pour la vérification de programme. La *validation* consiste à vérifier la *correction*

[1] Il y a autant de logiciels dans une automobile de moyenne gamme actuelle que dans les fusées des années soixante-dix.

du programme vis-à-vis de sa *spécification*, c'est-à-dire prouver ou tester que le système fait ce qu'il est censé faire (comme, par exemple, "les moteurs de freinage se mettent en route à une altitude de 140km"). Il s'agit de propriétés *fonctionnelles*, indiquant le comportement attendu du système. La *vérification*, quant à elle, s'intéresse à prouver certaines propriétés de *sûreté de fonctionnement* telles que la sécurité-innocuité (*safety*) ou la sécurité-confidentialité (*security*) du système (voir Section 1.1.5.1 pour les différentes notions de sécurité). Elle indique généralement ce que le programme ne doit *pas* faire (comme "la conversion ne doit jamais échouer").

Le test, ou la simulation, constitue la méthode la plus répandue de validation de programme. Cela consiste généralement en une étude *dynamique* du programme : un ensemble d'exécutions du programme sont effectuées, pour un certain nombre de valeurs d'entrée, et les réponses sont comparées aux résultats attendus. La principale difficulté tient dans le choix des valeurs à tester. En effet, l'ensemble des valeurs à tester est généralement démesuré, voire infini, compte tenu de la complexité et la taille des systèmes actuels. *Le test exhaustif est donc impossible.* Le testeur doit donc faire un choix de valeurs "judicieuses", censé représenter les différents comportements possibles du système. Or, les erreurs surviennent en général lors de comportements auxquels personne n'avait pensé. De plus, certaines erreurs proviennent également de la compréhension de la spécification du système. En effet, la définition des "résultats attendus" du système peut être ambiguë (comme l'unité de mesure de la force des moteurs de freinage), incomplète (comme ne pas contenir l'indication du temps maximal d'utilisation du Patriot avant un redémarrage obligatoire, ou les conditions sur la vitesse de la fusée pour que la conversion soit correcte) ou incohérente (lorsque deux comportements attendus sont contradictoires). Enfin certaines propriétés, en particulier de sécurité, ne sont pas vérifiables par les tests. Par exemple, la propriété de *confidentialité*, étudiée dans le Chapitre 3, s'exprime sous la forme d'une *quantification universelle sur des trajectoires* : on montrera que la propriété n'est vérifiée que si toutes les exécutions possibles de commandes reçues ne changent pas de comportement lorsque les valeurs des données confidentielles changent.

Les *méthodes formelles* commencent donc à s'imposer, de façon alternative, ou plus souvent complémentaire[1], pour la validation et la vérification de programme.

2.1.2 Les méthodes formelles

Idée générale. Une *méthode* désigne un ensemble de procédés permettant d'atteindre un but donné. L'adjectif *formel* précise que la *forme* doit être strictement structurée et rigoureusement définie. Les méthodes formelles désignent donc *un ensemble de techniques mathématiquement bien définies et non ambiguës* permettant d'exprimer, d'analyser et de résoudre un problème.

Dans notre cas, le problème consiste à représenter un système et à raisonner sur ses propriétés. Une méthode formelle comporte donc un langage formel, dans lequel le système et ses propriétés pourront être représentés, i.e. *formalisés* ou *modélisés formellement*, et un ensemble d'outils pour raisonner sur des éléments de ce langage. La *syntaxe* du langage définit la forme des éléments qui appartiennent au langage et la *sémantique* du langage permet d'interpréter les éléments en leur donnant une signification, et donc d'associer un élément à un système ou une propriété. Le langage est *formel* s'il est non ambigu et ne peut être interprété que d'une seule manière. Plus précisément sa syntaxe et sa sémantique doivent être mathématiquement bien définies et l'appartenance d'un élément au langage doit pouvoir être vérifiée par une machine.

[1] La question classique est en effet : "monteriez-vous dans un avion dont on vous assure que le comportement a été prouvé formellement même s'il n'a jamais été testé ?"

Vérifier formellement une propriété d'un système comporte donc trois étapes. La première étape consiste à modéliser le système dans un environnement formel. Cela signifie qu'un langage formel doit être choisi, dans lequel le système étudié pourra être représenté. Le choix de ce langage formel dépend généralement des spécificités du système (s'il dépend du temps, s'il possède un infinité de comportements possibles, etc). Dans une deuxième étape, les propriétés visées sont formalisées sur ce modèle. Enfin, la dernière étape consiste à prouver formellement que les propriétés formelles sont vérifiées par le modèle formel.

Utilisations. Les méthodes formelles peuvent être utilisées à différentes étapes du processus de développement d'un système.

Spécification. Une spécification formelle, indépendamment d'une vérification ultérieure, représente à elle seule un avantage majeur de l'utilisation de techniques formelles. Elle permet, en effet, de définir de façon rigoureuse les différents comportements attendus du système, sans incohérence, sans incomplétude, et sans ambiguïté d'interprétation lors de l'implémentation, la ré-utilisation ultérieure du code ou le partage du code entre plusieurs programmeurs. Les catastrophes citées précédemment auraient peut-être pu être évitées simplement par une spécification formelle.

Validation par le test. Les méthodes formelles peuvent être utilisés pour automatiser la production des jeux de tests, afin de les rendre plus représentatifs et plus complets (voir par exemple [44, 35, 25, 94]).

Preuve formelle de correction. Prouver formellement que le code implémente sa spécification donne une garantie absolue d'un code sans erreur. Cette vérification, très lourde pour des systèmes importants et complexes, n'est généralement appliquée qu'à certaines parties sensibles du code ou pour certaines propriétés critiques. Notons que la première étape est indispensable à cette vérification puisque prouver formellement qu'un programme implémente sa spécification nécessite que cette spécification soit formelle.

Vérification. En plus des propriétés fonctionnelles correspondant à la correction du programme, certaines propriétés globales et sécuritaires peuvent être vérifiée formellement sur un modèle du système. Cette vérification formelle comporte trois étapes : la modélisation du système, la modélisation des propriétés à vérifier, et enfin la preuve que les propriétés formelles sont vérifiées par le modèle formel.

Certification. Différents standards définissent, pour divers systèmes informatiques, des conditions à remplir pour l'obtention de certifications sécuritaires, avec plusieurs niveaux de certification. Les méthodes formelles sont souvent imposées comme moyen de modélisation et de vérification du produit à évaluer, pour l'obtention des plus hauts niveaux de certification. C'est le cas des niveaux *EAL5* à *EAL7* des *Critères Communs* ([38]), standard utilisé dans le monde de la carte à puce.

La diversité des objectifs à atteindre, des types de systèmes à étudier, des propriétés à montrer, a conduit à la définition et au développement d'une grande variété de méthodes formelles. Notre but ici n'est pas d'en donner une description exhaustive, de plus amples informations pouvant être trouvées, par exemple, dans [54, 37, 111, 31, 40]. Nous donnons ici un simple aperçu des principales techniques et des différentes approches des méthodes formelles, en détaillant celles qui ont été utilisées lors de nos travaux ou dans des travaux que nous citerons. Nous distinguons ici les deux facettes des méthodes formelles : la modélisation d'une part et la preuve de propriété d'autre part.

2.1.3 Modélisation d'un système

Un système peut être modélisé formellement de plusieurs façons différentes. Tout d'abord, le système peut être représenté sous la forme d'un *système de transitions*, comprenant d'une part un ensemble d'états, où chaque état représente l'état global du système à un instant donné, et d'autre part un ensemble de transitions entre ces états, où chaque transition représente une façon possible pour le système de passer d'un état à un autre. Cette approche est utilisée dans les *machines à états abstraits* (*ASM* pour *Abstract State Machine*, voir par exemple [30]), où le système est défini par un ensemble de règles de transitions, pouvant être appliquées aux états, sous certaines conditions, pour faire évoluer le système en "exécutant" la machine. Cette approche est également utilisée dans les *automates*, où les transitions représentent des réactions d'un système à des stimuli extérieurs. Ce type de modèle est utilisé pour représenter des systèmes réactifs ou des protocoles de communications. Lorsqu'il s'agit de modéliser un programme, la représentation sous la forme d'un système de transitions correspond à la *sémantique opérationnelle* du programme, c'est-à-dire que le programme est représenté par la suite des différents états successifs de son exécution.

Un système peut également être modélisé par une *fonction* (au sens mathématique) d'un état global du système vers un nouvel état global. Pour un programme, cela correspond à sa *sémantique dénotationnelle* : le programme est représenté par une fonction prenant en argument l'état global de la mémoire avant l'exécution du programme (ou simplement l'ensemble des valeurs des variables manipulées par le programme) et dont le résultat correspond à l'état global du programme après son exécution.

Enfin, un système peut être représenté par les propriétés qu'il vérifie. Pour un programme, cela correspond à sa *sémantique axiomatique* : le programme est représenté par les propriétés que vérifient, après exécution, les variables qu'il manipule, en fonction des propriétés qu'elles vérifiaient avant l'exécution. Mais rien n'est dit sur comment les calculer. En d'autres termes, le programme est simplement vu comme un transformateur de prédicats sur les variables qu'il manipule.

2.1.4 Preuve de propriétés d'un système

La modélisation d'un système est parfois un but en soi, permettant de définir rigoureusement les comportements du système. Toutefois, certaines propriétés sur ces comportements ont souvent besoin d'être prouvées formellement. Il s'agit alors de traduire la propriété à prouver en termes du modèle formel du système qui a été construit. Il existe alors plusieurs techniques de preuves.

Le *model checking*. La vérification de modèle, ou *model checking*, consiste à vérifier des propriétés sur des systèmes de transitions. Les propriétés sont exprimées dans une logique temporelle et la vérification de la propriété sur le modèle consiste à prouver la véracité de la formule logique pour tous les états du système. Si le nombre d'états du système est fini, la vérification peut être *automatique* : un algorithme, globalement basé sur une analyse par cas, permet de décider si la formule logique est vérifiée par la machine à états, et si elle ne l'est pas, de donner un contre-exemple. Ce contre-exemple peut malheureusement être sous la forme d'une succession d'une centaine d'états, difficile à interpréter pour corriger le programme. Une réponse positive quant à elle ne donne aucune indication sur *pourquoi* la propriété est vérifiée. Mais la principale contrainte de la vérification par modèle repose sur l'explosion du nombre d'états à analyser, rendant la vérification impossible faute de ressources suffisantes. Cela a donné naissance à des techniques telles que la vérification symbolique ou l'abstraction de modèles, réduisant le nombre

d'états à analyser, mais dépendantes des propriétés à prouver.

Les systèmes de preuves. Le système et ses propriétés peuvent tous deux être modélisés dans un *système logique*, permettant alors de raisonner sur ensemble infini d'états, grâce à des techniques telles que l'induction. Un système logique est associé à une *théorie* composée d'*axiomes*, qui sont des formules supposées vraies sans démonstration, et de *règles d'inférences*, qui permettent de déduire une formule (la *conclusion*) à partir d'un ensemble de formules vraies (les *prémisses*). La *preuve* d'une formule consiste alors en une succession d'applications de règles du système, à partir des axiomes du système, permettant d'aboutir à la formule visée. Une telle preuve s'appelle un "arbre de preuve", dont la racine est la propriété prouvée et les feuilles sont les axiomes.

Le choix du système logique doit faire un compromis entre deux objectifs contradictoires : la *décidabilité* et l'*expressivité*. L'expressivité augmente les applications possibles de la méthode ainsi que son niveau de confiance. Plus un langage est expressif, plus on pourra exprimer de propriétés dans ce langage et plus le modèle formel pourra être proche du système étudié et ainsi réduire les risques d'erreur pendant la traduction du système en formule logique. Seulement, une logique devient en général *indécidable* dès lors que son langage est suffisamment expressif, c'est-à-dire qu'il est impossible décider de façon *automatique* de la véracité de toute formule.

Lorsque la logique utilisée est une logique du premier ordre, le système est modélisée sous la forme d'axiomes et de règles d'inférence. La théorie est donc dépendante du système étudié et la modélisation est axiomatique, ce qui augmente les risques d'incohérence. En contre partie, certaines propriétés décidables peuvent être prouvées *automatiquement* par des *procédures de décision*.

La logique d'ordre supérieur est, quant à elle, composée d'un petit nombre de règles logiques, définissant la théorie mathématique, indépendamment du système à analyser, et offre une grande expressivité pour la *définition* de structures de données permettant de modéliser le système et ses propriétés. Les règles élémentaires qui définissent la théorie peuvent être combinées pour créer de nouvelles règles, appelée *tactiques*, sorte de "sucre syntaxique" offrant une plus grande facilité d'utilisation. Les propriétés ne pouvant pas, en général, être prouvées automatiquement, l'utilisateur doit guider la preuve de manière interactive, en indiquant quelle règle ou quelle tactique utiliser. La problématique de ces *assistants de preuves* n'est plus de pouvoir décider de la véracité de toute formule arbitraire, mais de fournir les moyens à l'utilisateur de décrire comment prouver une formule donnée et ensuite de simplement vérifier que c'est une preuve correcte du théorème (d'où leur autre nom de *vérifieurs de théorèmes*). La tâche du vérifieur est donc fortement simplifiée (il est plus facile de dire qu'un joueur a gagné en appliquant les règles du jeu que de trouver une stratégie permettant de gagner). La confiance accordée au système de preuves repose sur son noyau (les axiomes et les règles élémentaires). Elle peut être renforcée par la notion d'*objet de preuve* présente dans certains systèmes (comme Coq, voir ci-après) permettant de vérifier, par d'autres moyens, que l'objet construit est bien une preuve de la propriété cherchée. Parmi les assistants de preuves existants, citons ceux que nous évoquerons dans ce mémoire, à savoir HOL [62], Isabelle [75], PVS [107] et COQ [113]. Nous allons décrire succinctement l'assistant de preuve Coq, utilisé pour nos travaux, le lecteur pourra se reporter aux références pour de plus amples informations sur Coq ou sur les autres systèmes.

COQ. Développé dans le projet LogiCal, Coq est un assistant de preuve basé sur une logique d'ordre supérieur typée, appelée le *Calcul des Constructions Inductives*. Cette logique est basée sur la *Théorie des Types*, dont le principe repose sur le fait qu'il est possible de définir un

système de types où les types peuvent être vus comme des propositions logiques et où les règles de typage n'autorisent que des déductions correctes. En d'autres termes, dans la théorie des types, un terme T de type τ peut être vu comme une preuve T de la proposition τ (c'est ce que l'on appelle l'*isomorphisme de Curry-Howard*). La théorie peut donc être vue à la fois comme un langage de programmation et un système logique. La preuve d'une propriété dans le système Coq consiste à appliquer une succession de tactiques permettant de construire un *terme de preuve*. Ce terme est ensuite vérifié mécaniquement par le noyau, ce qui signfie que le moteur de preuve vérifie, par typage, que le type du terme de preuve est bien la propriété à prouver.

2.1.5 Preuve formelle de programmes

Regardons à présent le cas particulier de la preuve formelle de propriétés d'un *programme*. Lorsque l'on veut prouver formellement des propriétés sur un programme, il faut choisir un modèle du programme, un modèle de la propriété et un outil de preuve. Les techniques de preuves dépendent du type de propriétés que l'on veut vérifier.

Lorsque l'on veut exprimer des propriétés *globales et de haut niveau* du programme (telles que des propriétés sur une succession de fonctions, ou l'absence de *dead-lock*, i.e. de blocage du système, etc), les fonctions du programme sont en général vues comme des transitions entre états : la propriété "si la fonction g est appelée après la fonction f, alors la propriété P est vérifiée" peut être représentée sous la forme : $s \xrightarrow{f} s' \wedge s' \xrightarrow{g} s'' \Rightarrow P(s'')$. Par conséquent, pour définir ce genre de propriétés, le programme est modélisé par la représentation de chacune de ses fonctions sous la forme de transitions agissant sur l'état global du programme, et la preuve des propriétés se fait sur ce système de transitions.

Une autre approche consiste vouloir prouver des propriétés *fonctionnelles* du code source du programme. Ces propriétés constituent alors la *spécification* du programme et leur vérification correspond à une preuve de la *correction* du programme vis-à-vis de cette spécification. Le programme et sa spécification sont généralement représentés sous la forme d'un *triplet de Hoare* (voir [61]), noté $\{P\}e\{Q\}$, où P et Q sont des propositions logiques portant sur les variables du programme e. Cette formule signifie que si la propriété P, appelée la *précondition*, est vérifiée avant l'exécution du programme e, alors la propriété Q, appelée la *postcondition*, doit être vérifiée après l'exécution de e. Pour prouver formellement la validité d'une telle formule, il faut avoir un modèle formel du programme e, et donc une formalisation du langage de programmation utilisé. Il y a deux approches pour la formalisation d'un langage de programmation : une approche par *plongement profond* (*deep embedding* en anglais), où la syntaxe du langage est formalisée et sa sémantique est définie à partir de la syntaxe, et une approche par *plongement superficiel* (*shallow embedding* en anglais), où seule la sémantique est représentée. Cette terminologie a été introduite dans [23], papier concernant la formalisation de langages de description de composants matériels, et est maintenant utilisée pour distinguer deux approches possibles lors de la formalisation de programmes (voir par exemple [8] ou [103] pour une comparaison des deux approches).

Plongement profond. Dans un plongement profond, la *syntaxe* des programmes est modélisée dans le langage formel. Un type est utilisé pour représenter l'ensemble des programmes et un programme donné est un terme de ce type. La construction de la syntaxe formelle devient inutilement lourde lorsqu'il s'agit de prouver des propriétés d'un programme *donné*, (où seule la sémantique suffirait), mais est indispensable pour des études méta-théoriques sur le langage de programmation lui-même. Le plongement profond permet en effet de quantifier sur des structures syntaxiques et donc de raisonner sur des sous-ensembles de

programmes, en définissant différentes sémantiques pour le langage. Une sémantique dans cette approche est définie à partir de la syntaxe abstraite, sous la forme d'une fonction qui associe à chaque terme une *signification*. On pourra par exemple parler de typage et prouver des propriétés de ce typage, comme la *sûreté de typage* (absence d'erreur de typage à l'exécution : "si un terme T est bien typé et qu'il s'exécute en un terme T', alors T' sera également bien typé"). On remarque que formaliser cette propriété nécessite de formaliser les termes du langage, i.e. sa syntaxe, ainsi que la sémantique opérationnelle de l'exécution d'un terme. Il s'agit bien là d'un plongement profond, permettant de prouver une propriété de typage du langage lui-même.

D'importants travaux ont été menés pour une formalisation du langage Java : divers sous-ensembles de Java, nommés Bali[1] (dans [101]), Java^{light} (dans [105]) ou encore μJava (dans [102]), ont été formalisés dans le système de preuve Isabelle/HOL et la sûreté de typage a été prouvée pour ces langages.

Le langage C a également été formalisé, en plongement profond, en HOL (voir [104]). Toutefois certaines constructions du langage C, telles que les types unions, ne sont pas considérées dans cette formalisation. Par ailleurs, la sémantique de l'opération de *cast* (coercion) suppose qu'une fonction de conversion, capable d'évaluer une expression "castée", est fournie. Ces constructions du langage C constituent des points critiques de la formalisation du langage et ont fait l'objet d'une partie de nos travaux sur la vérification du code C du système d'exploitation, comme il sera expliqué dans le Chapitre 4.

Plongement superficiel. Une variante plus souple consiste à ne formaliser que la *sémantique* du programme : le programme est traduit dans le langage formel en une structure logique sémantiquement équivalente. Par exemple, le programme peut être vu comme une fonction entre deux états de la mémoire. Le fait de ne pas traduire la syntaxe allège considérablement le procédé de formalisation, rendant plus efficace la vérification de programmes concrets donnés. En revanche, l'adequation entre le modèle formel et le code source repose sur la correction de la traduction, i.e. sur la formalisation de la sémantique du langage.

Le plongement superficiel est donc généralement utilisés par les outils de vérification de programmes. Ces outils prennent un programme spécifié $\{P\}\ e\ \{Q\}$ en entrée, modélisent formellement le programme par une sémantique donnée du langage, et construisent une preuve formelle que le modèle du programme vérifie la spécification donnée. Plus exactement, étant donné que la construction automatique d'une telle preuve est impossible en général, l'outil construit une preuve *partielle*, où les fragments de preuve manquants, appelés *conditions de vérification*, doivent être prouvés par d'autres moyens (interactivement dans un assistant de preuve ou à l'aide de procédures de décision).

Il existe plusieurs outils de vérification formelle au niveau du code source d'un programme donné. L'idée générale est de proposer un langage de spécification pour l'écriture de la spécification, de formaliser le langage de programmation par un plongement superficiel (pour faciliter la preuve de programmes concrets donnés) et de générer, à partir de la spécification et de la modélisation du programme, des conditions de vérification assurant la correction du programme vis-à-vis de sa spécification. La spécification est généralement insérée dans le programme sous la forme d'*annotations* (commentaires spéciaux, ignorés par le compilateur du langage et interprétés par l'outil de vérification).

Suivant cette idée, plusieurs outils ont été développés, en particulier pour les programmes écrits en Java. Le langage JML (*Java Modeling Language*, [86]) est un langage formel d'annotation

[1] bien que l'île de Bali ne soit pas un sous-ensemble de celle de Java...

qui peut être analysé par différents outils dans le but de produire de la documentation, d'effectuer des tests dynamiques et de faire de la vérification de propriétés. Ce langage est utilisé pour la vérification de programmes Java dans les outils ESC/Java [48, 49], LOOP [88], Jive [92], Jack [28, 27] ou Krakatoa [90]. D'autre part, l'outil Key [82, 2] propose une spécification basée sur UML pour la vérification d'application Java Card, et l'outil Jass [80, 10] est une extension "Design by Contract" pour le langage Java, permettant des contrôles de violation de spécification à l'exécution, avec la possibilité de spécifier des propriétés globales en utilisant des traces. Enfin, l'outil Bandera [39] permet de générer automatiquement, à partir d'un programme Java et des propriétés attendues pour ce programme (exprimées par exemple en logique temporelle), un modèle vérifiable par des *model checkers* tels que SPIN (voir [63]).

Concernant les programmes C, il existe des travaux proposant une analyse *dynamique* du programme, comme par exemple dans le projet *CCured* (voir [100, 98]), qui présente une transformation de programme consistant à insérer des contrôles dynamiques qui assurent l'arrêt du programme plutôt que l'exécution d'une instruction qui violerait la sûreté de la mémoire. Il existe également un grand nombre d'outils permettant de faire de l'analyse *statique* de code C (énumérés, par exemple, à l'adresse `http://www.spinroot.com/static/`), afin de détecter un certain nombre d'erreurs données, mais peu d'entre eux gèrent des préconditions et postconditions explicites. Toutefois, l'outil Caveat [33] propose de la vérification semi-automatique de programmes C, où les annotations sont construites séparément du code. Enfin, l'outil Caduceus [53] est une adaptation directe de la technologie Java/JML pour les programmes C. Comme nous le verrons dans le Chapitre 4, nous avons utilisé et étendu l'outil Caduceus pour nos travaux de vérification de système d'exploitation embarqué. L'outil a également été utilisé pour la définition d'une méthodologie de vérification de propriétés de haut niveau à partir du code source, permettant de préserver un lien formel entre le modèle vérifié et l'implémentation (voir le Chapitre 5).

Ce lien entre le modèle vérifié et l'implémentation est crucial dans la vérification de programme, comme nous allons l'expliquer dans la section suivante.

2.1.6 Lien entre le modèle et l'implémentation

La vérification formelle d'un programme donné nécessite d'avoir un modèle de ce programme dans une logique formelle, afin de raisonner sur ce modèle pour vérifier certaines propriétés. Nous avons donc trois éléments : le code, le modèle formel et les propriétés formelles. Étant donné que les propriétés seront vérifiées formellement sur le modèle, et que le but est de prouver que le code vérifie ces propriétés, il est indispensable d'avoir un lien formel entre le code et le modèle. Pour cela, plusieurs approches existent.

Raffinements. Une façon classique de renforcer ce lien entre le modèle et le code est de raffiner le modèle formel de haut niveau sur lequel la vérification a été faite, en une succession de modèles de plus proches de l'implémentation. Une preuve d'abstraction permet de garder un lien formel entre deux niveaux successifs et le modèle de plus bas niveau se rapproche le plus possible de l'implémentation, en termes de structures de données et de fonctions.

Un exemple, parmi d'autres, d'utilisation de cette approche dans le monde de la carte à puce est celui de la modélisation formelle de la technologie Java Card, développée dans le projet FORMAVIE ([18]). Cette modélisation a été utilisée dans le cadre de nos travaux de preuve d'isolation d'applications Java Card et sera décrite en détail dans le Chapitre 3. Toute une chaîne de raffinements successifs a été construite à partir du modèle haut niveau jusqu'à l'implémentation réelle de la machine virtuelle sur une carte. Cette méthodologie a été présentée dans [24] et évaluée

avec succès comme une interprétation des exigences de niveau EAL7 de certification pour les Critères Commun (voir [38]).

Les inconvénients et les faiblesses de cette approche sont multiples. Tout d'abord, un inconvénient majeur concerne l'optimisation, la maintenance et la réutilisation. En effet, toute modification au niveau du code source nécessite une mise à jour du modèle, ou des modèles, développé(s) dans la chaîne de raffinement ainsi que des liens formels entre ces modèles. Une autre faiblesse majeure réside dans la perte de l'aspect formel dans le lien entre entre le modèle de plus bas niveau et l'implémentation. Une manière de rendre ce dernier lien formel est d'utiliser la génération automatique du code source à partir des modèles formels, comme le propose la méthode B (voir [1]) ou Esterel (voir [16]). Ceci permet de dériver un code à partir de la spécification, après avoir prouvé que cette spécification vérifie les propriétés de sécurité voulues.

La méthode B est par essence une méthode de développement de modèles par raffinements, depuis un modèle abstrait du comportement d'un programme, sous la forme d'une machine d'états abstraits, jusqu'à un modèle concret. Ce modèle concrêt peut à son tour être traduit en langage C, c'est pourquoi nous présentons cette méthode dans l'approche par génération de code.

Exécution. Notons une autre modélisation de la technologie Java Card, toujours en Coq, proposée dans [13, 45]. La particularité de cette modélisation est qu'elle est *exécutable*. En d'autres termes, le modèle formel de la plate-forme prend en entrée le modèle formel d'un *cap file*[1], simule l'exécution de ce *cap file* et retourne le résultat de cette exécution. Par ailleurs, une boîte à outils, nommée *JCVM Tools*, permet de transformer automatiquement un ensemble de *cap files* en sa formalisation en Coq. Notons que ceci constitue un plongement profond, non pas de la machine virtuelle (écrite en C), mais du langage de *bytecode*. Il est donc possible, pour toute application Java Card, d'observer son exécution à la fois par une implémentation de la machine virtuelle et par sa modélisation en Coq. Cette démarche se rapproche alors des techniques de test puisqu'elle consiste à choisir des applications pertinentes afin de comparer les résultats dans la modélisation et dans l'implémentation.

Génération de code. La méthode B (voir [1]), que nous avons déjà évoquée, est une méthode formelle permettant de définir des modèles de haut niveau et de les raffiner successivement jusqu'à un modèle bas niveau, nommé *implémentation*. Ce modèle d'implémentation est écrit dans le langage B0, sous-ensemble du langage B. De nombreux travaux ont étudié la possibilité de générer un code source en langage C ou en langage Java, à partir de ce modèle d'implémentation. C'est l'objectif du projet BOM (voir [21]), qui propose également des optimisations de la génération de code source, afin de pouvoir embarquer le code généré sur des cartes à puces (voir [17]). Les optimisations consistent essentiellement en un "aplatissement" des programmes, comme par exemple le remplacement d'appel de fonction par le code de la fonction lui-même, ainsi qu'à un meilleur choix du type entier (`int`, `char`, `short`) suivant l'intervalle des valeurs possibles pour une variable donnée. Cette génération de code a été utilisée dans les travaux de vérification formelle du *vérificateur de bytecode* de la technologie Java Card, dans [32], où le modèle formel du vérificateur de bytecode a été traduit en un programme C pouvant être embarqué sur une carte à puce.

Notons toutefois que dans ces travaux, les composants du système d'exploitation, et en particulier la gestion de la mémoire, ne sont pas générés à partir de modèles formels. Ils sont développés de manière "traditionnelle" en langage C par des développeurs expérimentés en matière d'optimisations spécifiques à la carte à puce. La vérification utilise donc une abstraction de la couche

[1] i.e. un fichier Java compilé et converti, voir Section 1.2.1.

proche du matériel, pour laquelle il n'y a pas de lien formel avec le code.

De plus, cette approche correspond à une démarche de *conception* de programme, ce qui est différent d'une démarche de *vérification* de propriétés sur des programmes *existants*. C'est à cette dernière approche que nous nous sommes intéressés dans nos travaux, pour vérifier les programmes du système d'exploitation, tels qu'ils sont écrits par les développeurs expérimentés. Le code source est donc considéré comme le point de départ de la vérification. Une approche symétrique est alors de générer un modèle formel à partir du code source.

Génération de modèle. Générer un modèle à partir d'un programme a été principalement étudié avec une approche de *model checking*. Une méthode est proposée pour transformer le programme en une machine à états, définie dans le langage d'un *model checker*. Les propriétés du programme, traduites en formules logiques, peuvent être automatiquement vérifiées sur le modèle du programme par le *model checker*.

Nous avons déjà cité l'outil Bandera [39], permettant d'extraire automatiquement un modèle d'un programme Java et de ses spécifications en logique temporelle. Dans ce cas, le modèle pourra être étudié par divers *model checkers*, comme par exemple SPIN (voir [63]). D'autres outils permettent la traduction de programmes C ou Java en *Promela*, le langage d'entrée du *model checker* SPIN. Par exemple, [64] présente l'extraction d'un modèle SPIN à partir d'un programme C, à l'aide du système de vérification *FeaVer* (voir [65]). De son côté, [60] présente le *Java PathFinder*, traducteur d'un programme Java en Promela. Dans les deux cas, l'absence de *dead-locks* (blocage du sytème), ainsi que la preuve de propriétés annotées dans le code, peuvent être prouvées automatiquement. Toutefois, dans les deux cas, le modèle généré possède le même nombre d'états que le programme. Le programme doit donc générer un nombre *fini* d'états, i.e. de configurations possibles de la mémoire. Cette méthode ne peut donc être appliquée qu'aux programmes à *espace d'états fini*.

Il est à noter également que l'extracteur de modèle *FeaVer* nécessite la définition d'un *contexte de vérification* par l'utilisateur. Le contexte de vérification permet de réduire la complexité du code et faciliter la définition du modèle, en indiquant l'interprétation des instructions *pertinentes*. Ainsi certaines affectations ou appels de fonctions peuvent être ignorés, les tests sur des valeurs qui en dépendent devenant alors tout simplement non-déterministes.

L'impossibilité de prouver certaines propriétés, telle que la terminaison de programme (propriété indécidable), ainsi que la restriction aux programmes engendrant un ensemble fini d'états mémoire, font préférer les systèmes de preuves aux *model checkers* pour certaines applications. Pouvoir traduire automatiquement n'importe quel programme écrit dans un langage de programmation donné dans le langage formel d'un système de preuve, nécessite de formaliser la sémantique, et éventuellement la syntaxe, du langage de programmation visé. C'est le cas du *JCVM Tools* déjà mentionné, permettant de générer un modèle Coq de tout programme écrit en Java Card (plus précisément tout *cap file*, construit à partir d'un programme Java, avec le compilateur et le converter Java). Un tel plongement profond, utilisé pour la vérification de la correction de la modélisation de la JCVM, n'est pas adapté à la preuve de programme à proprement dit, à savoir prouver des propriétés sur une application Java Card donnée. Par ailleurs, il n'existe pas à notre connaissance de travaux similaires concernant la génération de modèle formel en plongement profond, dans un système de preuve, pour le langage C. La formalisation de C en HOL, déjà évoquée page 64, ne propose pas d'outil permettant une conversion automatique d'un programme C en formule en HOL.

En revanche, un plongement superficiel est utilisé dans les outils de vérification de pro-

grammes à partir du code source. Ces outils s'appuient sur un modèle généré automatiquement à partir du code source, mais qui n'est pas manipulable directement par l'utilisateur, comme nous l'expliquons dans le paragraphe suivant.

Vérification à partir du code source. Considérer le code source comme le point de départ de la vérification permet, dans le milieu industriel en particulier, de prouver des propriétés sur le code source même des systèmes, sans avoir à le modifier ou à le traduire en un modèle plus abstrait. Il existe diverses méthodes de vérification de code source, par analyse statique, analyse automatique de *menaces* (comme un accès en dehors des bornes d'un tableau), etc. Mais aujourd'hui, des propriétés plus complexes sont visées, et sont donc spécifiées par l'utilisateur, via les annotations. Des outils de vérification permettent alors de vérifier ces propriétés locales du programme, en générant l'ensemble des conditions nécéssaires pour que la spécification soit vérifiée par le code. Lors de cette vérification, la sémantique du langage de programmation est formalisée et est utilisée pour construire un *modèle formel du programme*, sur lequel sera effectuée la vérification. Ce modèle est en général *implicite*, dans le sens que l'utilisateur ne voit que le résultat de la vérification, à savoir soit les conditions à prouver pour établir la correction, soit un contre-exemple.

Cette approche offre la garantie que les propriétés visées sont vérifiées par le code source même, sous réserve de la correction de l'outil et sous réserve également que la sémantique du langage utilisée par l'outil corresponde à la sémantique implémentée dans le compilateur qui sera utilisé lors de la conversion du programme en langage machine. Cela nécessite de formaliser la sémantique du compilateur, comme proposé dans de récentes études sur le développement d'un compilateur C certifié en Coq (voir [20]).

Par ailleurs, le fait que ces propriétés doivent être exprimées dans le langage d'annotation implique plusieurs limitations :

- le langage d'annotation doit faire un compromis entre l'expressivité, pour la définition de propriétés logiques sur certaines constructions du programme, et la facilité d'utilisation, pour un écriture intuitive des spécifications, proche du langage de programmation. La logique du premier ordre est donc souvent utilisée, mais est parfois insuffisante ou lourde à manipuler pour la définition de certaines propriétés (comme l'atteignabilité dans une structure de donnée), qui serait immédiate dans un langage d'ordre supérieur. C'est pourquoi certains outils, comme Caduceus, permettent de déclarer des prédicats qui ne seront définis qu'*a posteriori* dans le système de preuve d'ordre supérieur utilisé.
- si on a besoin de prouver plusieurs propriétés, l'annotation de chaque fonction contiendra la conjonction de toutes ces propriétés. Le code se verra plus "pollué" et le procédé de vérification peut être plus lourd.
- les propriétés exprimées à l'aide des annotations sont des propriétés *locales* à la fonction considérée. Ceci convient pour la vérification de propriétés *fonctionnelles*, comme le fait que le résultat d'une fonction ne doit pas être nul. Mais il est souvent nécessaire de prouver des propriétés *globales* sur une combinaison de plusieurs fonctions ou des propriétés temporelles d'ordre supérieur, comme l'absence de *dead-lock* (blocage du système). Toutefois, il existe des méthodes qui proposent une manière d'exprimer des propriétés globales dans le langage d'annotation, utilisant des variables pour représenter l'état global du programme (voir [11, 67]).

En conclusion, chacune des méthodes proposées pour la vérification formelle de programme possède des inconvénients, plus ou moins contraignants suivant les utilisations. C'est pourquoi

nous proposons, dans le Chapitre 5, une méthodologie combinant plusieurs approches présentées ici, pour la vérification de propriétés globales de haut niveau sur un modèle formellement lié au code. Plus précisément, nous choisissons une approche à partir du code source, mais où le modèle implicite construit par l'outil de vérification de programme, qui est généralement utilisé uniquement pour la preuve des propriétés décrites dans les annotations, sera utilisé, dans notre méthode, pour la preuve des propriétés de plus haut niveau.

2.2 Les méthodes formelles et les cartes à puce

2.2.1 Carte à puce, domaine privilégié pour la vérification formelle

Les cartes à puce ont pour vocation d'offrir un niveau élevé de sécurité pour les données embarquées. Or, elles doivent faire face aux enjeux induits par leurs caractéristiques techniques et leurs utilisations, comme détaillés dans la Section 1.1.5.3. En effet, les nouvelles plate-formes multi-applicatives ouvertes, la possibilité d'arrachage de la carte ou encore la duplication des cartes à des millions d'exemplaires, sont autant d'enjeux qui justifient l'utilisation de techniques, en particulier formelles, de vérification de la correction, de l'innocuité et de la sécurité des programmes embarqués dans les cartes à puce.

Par ailleurs, les ressources limitées des cartes à puce restreignent la taille des programmes, les rendant plus accessibles à l'application de vérifications formelles que d'autres systèmes plus complexes, tels que les programmes régissant le fonctionnement d'un avion. Toutefois, cette contrainte en termes de ressources restreint l'utilisation de certaines méthodes formelles pour des raisons d'optimisation. En effet, les méthodes qui impliquent une modification du code (comme l'insertion de contrôles dynamiques, voir [100, 98]) ou une génération automatique du code à partir de modèle, ne sont pas assez optimisées à l'heure actuelle pour que le code soit embarqué sur une carte (du moins du code de bas niveau tel que celui de la gestion de la mémoire).

Comme déjà mentionné en Section 2.1.2, les méthodes formelles peuvent être utilisées à chaque étape du processus de développement des logiciels, et c'est le cas également pour les logiciels embarqués : la spécification, la validation par le test, la preuve formelle de correction, la vérification de propriété de sécurité, et la certification, en particulier la certification Critères Communs [38].

Malheureusement, la mise en œuvre des méthodes formelles nécessite un investissement dont les retombées économiques peuvent ne survenir qu'à long terme. Dans un marché dirigé par la rentabilité, seules les méthodes formelles pouvant apporter un avantage financier à court terme sont considérées. Comme expliqué dans la reflexion présentée dans [85] sur l'utilisation des méthodes formelles dans le monde de la carte à puce, cette utilisation intervient essentiellement dans trois domaines :

- la maîtrise des systèmes de plus en plus complexes, pour augmenter la sécurité des produits et leur valeur ajoutée ;
- la certification de haut niveau, pour des raisons marketing de différenciation des produits concurrents ;
- la réduction des coûts des tests.

Citons les principales études formelles effectuées dans le monde de la carte à puce, en distinguant ces trois objectifs.

2.2.2 Études formelles dans le monde de la carte à puce

2.2.2.1 Étude formelle de systèmes complexes

L'arrivée des nouvelles plate-formes multi-applicatives ouvertes a fortement complexifié les systèmes embarqués sur les cartes à puce, justifiant le recours à la modélisation formelle pour une meilleure maîtrise et à la vérification formelle de propriétés sécuritaires pour une plus grande confiance. En particulier, la technologie Java Card a fait l'objet de nombreuses études formelles.

D'une part, la possibilité de charger des applications sur la carte après sa mise en service est à l'origine de la sécurisation du système vis-à-vis des applications qu'il héberge et de vérification sur le comportement des applications. D'autre part, la cohabitation, sur une même carte, de diverses applications (bancaires, médicales, de fidélisation, d'identification, etc) nécessite une isolation des données des applications au sein de la carte, qui a également fait l'objet d'études formelles. La vérification formelle de ces deux aspects (comportement des applications et isolation des données) se fait soit au niveau de la plate-forme, soit au niveau du code source des applications.

Au niveau de la plate-forme, la vérification des applications est assurée par le vérificateur de *bytecode*. Ce composant essentiel de la plate-forme Java Card a été modélisé formellement dans divers travaux. Nous avons déjà cité sa modélisation à l'aide de la méthode B (voir [32]), utilisée pour vérifier la sûreté de typage et la génération du code C du vérificateur à partir du modèle formel. Le vérificateur de *bytecode* a également été formalisé en Isabelle dans [83, 84] et dans de nombreux autres travaux décrits dans [87]. Par ailleurs, la modélisation exécutable de la plate-forme Java Card en Coq, déjà mentionnée dans la Section 2.1.6 (présentée dans [13, 45]) est utilisée pour prouver la correction du vérificateur de *bytecode*. Cela consiste à modéliser une machine virtuelle *défensive*, i.e. qui effectue un contrôle des types à l'exécution, et une machine *offensive* qui n'en effectue pas, et à prouver que les deux machines coïncident sur les programmes qui sont acceptés par le vérificateur de *bytecode*.

Quant à l'isolation des applications, elle est assurée au sein de la plate-forme Java Card par le mécanisme de *firewall*. Les travaux que nous allons présenter dans le Chapitre 3 proposent de prouver formellement que les règles qui constituent le *firewall* assurent la confidentialité et l'intégrité des données (dans le cas où il n'y a pas d'*interface partageable*). Ces travaux s'appuient sur la modélisation de la plate-forme Java Card en Coq, développée dans le projet FORMAVIE, qui sera également décrite dans ce chapitre. Notons que la politique de sécurité du *firewall* a également été formalisée dans le système B dans [96], dans le but de prouver que tout accès est soumis au contrôle des règles du *firewall*.

Le fait que la plate-forme assure une vérification et une isolation des applications implique que le chargement ou l'exécution d'une application ne respectant pas les politiques de sécurité de la plate-forme provoquera une erreur. Pour éviter ce genre d'erreur à l'exécution, des analyses sont effectuées au niveau du code source des applications.

Une première approche possible est d'associer à chaque application la preuve qu'elle respecte une certaine politique de sécurité. C'est le principe du *Proof-Carrying Code* (*PCC*, voir [97]), où chaque programme contient à la fois le code source et une preuve facilement vérifiable par la carte. Les propriétés pouvant être vérifiées ne doivent cependant pas être trop complexes pour pouvoir être embarquées sur une carte.

Le code source des applications Java Card peut également être vérifié par des outils de vérification de programme. Par exemple, les applications de l'API Java Card ont été modélisée, en JML [86], en particulier les classes AID (dans [115]) et APDU (dans [116]) et vérifiées en

utilisant l'outil *Loop*.

En ce qui concerne plus particulièrement l'isolation des applications, une vérification au niveau du code source permet d'éviter le déploiement d'une application ne respectant pas les règles imposées par la politique de sécurité de la plate-forme. Par exemple, [43] propose un système de type pour le langage Java Card, permettant de détecter la plupart des violations des règles de *firewall* par une analyse statique du code source de l'application. D'autres études s'intéressent à certains flux d'information non détectés par le *firewall*, en particulier en présence d'interface partageable. Une interface partageable permet de partager des données d'une application avec toutes les autres applications de la carte, ce qui autorise des flux d'information *transitifs* : lorsqu'une application partage des données avec une deuxième application, rien n'empêche cette deuxième de re-transmettre l'information. Ce genre de flux d'information peut être ou ne pas être autorisé, suivant les applications considérées. Une méthode est proposée, par exemple dans [19], pour modéliser les flux d'information autorisés entre applications, par la définition de niveaux de sécurité pour chaque champ et chaque méthode d'applications. Une politique de sécurité définit ensuite les flux autorisés entre les différents niveaux et une technique de *model checking* est utilisée pour vérifier qu'un ensemble d'applications implémente bien la politique de sécurité. Mentionnons également [47] qui présente une façon de calculer tous les flux potentiels entre applications par un système de contraintes modélisant les effets du *firewall*.

D'autres composants de la carte à puce ont été formellement modélisés, notamment au niveau de la couche matérielle, plus adaptée à l'application de méthodes formelles. Citons, par exemple, la modélisation formelle de la gestion de la mémoire du processeur Infineon SLE88, par des Machine d'États Interactives, en Isabelle, dans [106].

Nous remarquons que toutes ces différents études portent sur la couche matérielle, la machine virtuelle et la couche applicative de la carte à puce, le système d'exploitation étant généralement supposé correct. Or les systèmes d'exploitation, de plus en plus complexes, occupent une position centrale dans l'architecture de la carte et devraient, à ce titre, faire l'objet de vérifications rigoureuses. C'est pourquoi, comme nous l'expliquerons plus loin, nous nous sommes intéressés dans nos travaux à la *correction* d'un système d'exploitation embarqué dans une carte à puce.

2.2.2.2 Certification

La principale différenciation entre les divers fabricants de cartes à puce se fait en termes de niveau de sécurité qu'offrent leurs produits. Un moyen de mesurer et de confirmer ce niveau est de soumettre le produit à une certification par un organisme indépendant. Les Critères Communs, standard utilisé dans le monde de la carte à puce, définissent un procédé standard pour l'obtention d'une certification sécuritaire, à différents niveaux. Les méthodes formelles commencent à être utilisées à partir du niveau cinq (*EAL5*) et sont obligatoires au niveau le plus élevé (*EAL7*) pour la formalisation de la politique de sécurité, la modélisation de toute l'architecture du produit et la vérification que la politique de sécurité est implémentée par le produit. La certification au niveau EAL7 de la plate-forme Java Card est l'objectif du projet FORMAVIE, déjà mentionné. Une méthodologie de modélisation formelle de la politique de sécurité et d'une chaîne de raffinement de modèles formels a été acceptée comme procédé de certification de niveau EAL7 (voir [24]).

2.2.2.3 Utilisation des méthodes formelles pour le test

Comme mentionné précédemment, les méthodes formelles sont également utilisées pour renforcer et faciliter la phase de test en rendant plus automatique et plus complète la production des "jeux de tests" (valeurs à choisir pour l'exécution du programme). La principale méthode est de modéliser le programme par une machine à états représentant le flot de contrôle du programme et d'analyser les différents parcours possible de cette machine. Le site [94] présente une liste des différents travaux sur la génération de tests à partir de modèles. Dans le monde de la carte à puce, citons par exemple [44] qui propose la génération de tests à partir d'UML, [35] qui utilise des spécifications JML, ou encore [25] où les tests sont générés à partir d'une modélisation en Esterel.

2.2.2.4 Correction de programme

Une vérification formelle au niveau du code source d'un programme, comme nous l'avons vu, permet de garantir que le programme a le comportement attendu. La preuve formelle de *correction* est une tâche fastidieuse, qui n'est appliquée que dans le cas de petits programmes ou de sous-programmes cruciaux. C'est pourquoi la principale utilisation de la preuve formelle de correction dans le monde de la carte à puce concerne la vérification d'applets Java Card. Cela consiste à définir le comportement de l'application dans un langage de spécification pour Java Card, essentiellement JML, et à prouver que le code source de l'application vérifie ces propriétés annotées, à l'aide d'outils tels que Jack, Krakatoa, ESC/JAVA, Loop. Des applications concrètes ont été prouvées correctes vis-à-vis de leur spécification à l'aide de ces outils (voir par exemple [36, 26]).

D'autres études se sont penchées sur une des caractéristiques principales des cartes à puces, citée dans les enjeux de la Section 1.1.5.3, à savoir l'*arrachage*. Le langage de programmation Java Card, spécialement conçu pour la programmation d'applications embarquées sur la carte à puce, contient un mécanisme, dit de *transaction*, permettant de gérer des interruptions. Ce mécanisme a été modélisé dans divers travaux, dans le but de prouver sa correction ou de démontrer certaines propriétés en cas d'arrachage de la carte. Plus précisément, Java Card distingue deux types de mémoire : la mémoire persistante et la mémoire volatile ("transiente"). En particulier, la méthode `isTransient` permet de savoir dans quel type de mémoire une donnée est stockée. De plus, trois méthodes sont fournies pour gérer l'atomicité d'un ensemble d'instructions : `beginTransaction`, `abortTransaction` et `commitTransaction`. Lorsque la méthode `beginTransaction` est invoquée, toutes les modifications de données persistantes, jusqu'au prochain appel à la méthode `commitTransaction` ou `abortTransaction`, sont stockées dans le *commit buffer*. Si la transaction se termine par un `commitTransaction`, les modifications deviennent effectives sur les données. En revanche, si elle se termine par un `abortTransaction`, signifiant qu'une erreur est survenue impliquant l'abandon de la transaction, alors les modifications ne sont pas effectuées (voir [34] pour plus de détails).

Ce mécanisme de transaction a été modélisé, par exemple, dans [66] et [14]. [66] propose une méthode permettant de définir, en JML, le comportement d'un programme Java Card en cas d'arrachage et de le vérifier en utilisant l'outil LOOP. Plus précisément, le programme Java Card est traduit dans le langage Java. Rappelons que le langage Java Card est un sous-ensemble de Java, mais avec des caractéristiques supplémentaires : principalement le *firewall*, le mécanisme de transaction et la distinction des types de mémoire. Il s'agit donc, d'une part, de modéliser une interruption de programme en termes des valeurs des données volatiles et, d'autre part, traduire le mécanisme de transaction, rendant conditionnelles les opérations sur les données persistantes.

L'interruption de programme est traduite par une *levée d'exception* et les opérations conditionnelles sont représentées par une modification du code source : les valeurs données persistantes sont mémorisées avant chaque transaction et mises à jour à la fin de la transaction. Cette approche permet de modéliser le comportement attendu en cas d'arrachage, en spécifiant, dans les annotations, la propriété vérifiée en cas de levée de l'exception d'arrachage.

Les travaux présentés dans [14] sont basés sur l'outil Key déjà mentionné. Cet outil permet de traduire une spécification UML en une logique adaptée à la modélisation de programmes Java Card (*Dynamic Logic*). Dans cette logique, les invariants exprimés sont *faibles*, c'est-à-dire qu'ils sont vrais avant et après les appels de méthodes, mais ils ne sont pas nécessairement vérifiés à chaque pas de l'exécution de la méthode. Cela signifie qu'un arrachage, pouvant intervenir à chaque instant, peut violer cet invariant. La logique est donc étendue pour pouvoir exprimer des *invariants forts* du programme, permettant de définir des propriétés qui doivent être vérifiées à tout moment, *y compris en cas d'arrachage*. Cette approche, en revanche, ne permet pas de définir le comportement, en cas d'arrachage, d'une méthode *donnée*.

Enfin, les travaux de traduction de propriétés de la logique temporelle en JML (voir [11, 67]) ont été utilisés pour prouver la correction du mécanisme de transaction, comme l'impossibilité d'avoir une imbrication de transactions.

Parmi les phases de développement où les méthodes formelles peuvent intervenir (décrites dans la Section 2.1.2), nous nous sommes intéressés à la spécification et à la preuve de correction au niveau du système d'exploitation. Bien plus fastidieuse étant donné la complexité et la taille du système, cette vérification doit également gérer un langage de programmation de plus bas niveau qu'est le langage C. Trouver une méthodologie pour la preuve formelle d'un code C de système d'exploitation embarqué constitue une partie majeure de nos travaux.

2.2.3 Objectifs de nos travaux

2.2.3.1 Vérification formelle de propriétés sécuritaires

La vérification de propriétés sécuritaires de haut niveau est une de nos priorités, et ce pour les deux composants qui ont été étudiés :

Preuve d'isolation des applications Java Card. Comme déjà expliqué à plusieurs reprises, les propriétés de confidentialité et d'intégrité des données de différentes applications embarquées sur une même carte à puce sont primordiales pour la sécurité de la carte. Dans le Chapitre 3, nous allons décrire la vérification formelle de ces propriétés, impliquant une modélisation de la technologie Java Card et une formalisation des concepts de confidentialité et d'intégrité. Cette formalisation nécessite que le modèle utilisé soit de haut-niveau pour pouvoir exprimer une exécution en termes de séquences de transitions entre états de la machine virtuelle.

Preuve de propriétés haut niveau du système d'exploitation. Le système d'exploitation d'une carte à puce, peu analysé d'un point de vue sécuritaire dans les diverses études formelles existantes, est pourtant central dans l'architecture de la carte. Notre but est de pouvoir prouver des propriétés de haut niveau, telles que des propriétés globales de combinaison de plusieurs fonctions, tout en conservant un lien formel avec le code. En effet, notre vérification formelle des propriétés sécuritaires de la plate-forme Java Card ne permet pas d'affirmer que les propriétés sont vérifiée par une implémentation donnée des modèles. Nous proposons donc une méthodologie pour la preuve propriétés globales sur un modèle formellement lié au code. Cette méthodologie est présentée dans le Chapitre 5 et a été

utilisée pour la vérification de notre module de gestion de mémoire Flash présentée dans le Chapitre 6.

2.2.3.2 Spécification formelle et preuve formelle de correction d'un code source de bas niveau

Le but de nos travaux qui seront présentés dans le Chapitre 4 est de trouver une méthodologie de preuve formelle de correction d'un code de *bas niveau* de système d'exploitation embarqué, en prenant pour illustration la gestion de la mémoire Flash. Pour cela, nous devons faire face à diverses particularités d'un tel code de bas niveau, qui devront être prises en compte dans le choix de la méthode de vérification formelle, discutée dans le chapitre suivant. Le langage de programmation utilisé pour le développement de système d'exploitation est le langage C.

Le langage C. Créé au début des années soixante-dix par Dennis Ritchie dans les laboratoires Bell, ce langage a été conçu pour l'implémentation d'un nouveau un système d'exploitation UNIX portable. En 1978, Brian Kernighan et Dennis Ritchie publient *The C programming langage*, première définition rigoureuse du langage. Mélange savant de fonctionnalités de haut niveau et d'aspects proches de la mémoire, le langage connut un succès impressionnant au sein de la communauté des programmeurs. "C is a quirky, flawed and an enormous success" (C est excentrique, imparfait et un succès immense) - Dennis Ritchie. Ses principaux avantages sont sa rapidité, équivalente à celle du langage assembleur, et sa *portabilité* : à l'époque de la création du langage, le transfert d'un système vers une autre machine obligeait une réécriture du système en entier dans le nouveau langage machine. Avec le langage C, un compilateur de programmes C en langage machine est défini pour chaque langage machine, et un programme C peut être utilisé sur différents ordinateurs simplement en étant recompilé. De nombreux compilateurs ont vu le jour, ainsi que de nouvelles implémentations du langage. Les incompatibilités naissantes ont eues pour conséquence la normalisation du langage, en 1983, par l'American National Standard Institute (ANSI), sous le nom de norme *ANSI-C* [1]. La norme ANSI-C a pour but d'améliorer et sécuriser le langage. Mais elle permet également aux programmes d'avant la norme d'être acceptés par la norme. Cet objectif contradictoire entraîne la conservation de certaines formes désuètes ou redondantes, rendant la sémantique du langage plus lourde et plus délicate à *formaliser* (pour plus d'information sur le langage, voir par exemple [41]).

Caractéristiques dépendantes du compilateur. Un certain nombre de caractéristiques du langage n'est pas spécifié par la norme, laissant le choix d'implémentation au compilateur, suivant le langage machine pour lequel il est conçu. Ceci est un point très important lors de la formalisation du langage. En effet, un choix s'impose entre une formalisation de la norme ANSI uniquement et une formalisation "complète" avec un choix de compilation pour les caractéristiques non spécifiées. Dans le premier cas, certaines propriétés, touchant aux caractéristiques non spécifiées, ne pourront être prouvées. Le deuxième cas permet quant à lui de représenter toutes les caractéristiques du langage, mais la vérification devient alors dépendante d'un compilateur donné. Nous avons, à ce sujet, adapté l'outil de vérification de programme Caduceus, de manière à ce que les constructions dépendantes du compilateur soient gérées, grâce à un ensemble de paramètres du compilateur, fournis par l'utilisateur.

[1] Cette norme a également été adoptée en 1990 par l'International Standardization Organisation (ISO).

Optimisations et mémoire de bas niveau. Les ressources limitées de la carte à puce imposent une programmation optimisée en terme d'utilisation d'espace mémoire, en particulier dans l'utilisation des structures de données. Ceci a pour conséquence une manipulation fréquente d'une mémoire de bas niveau, à savoir une écriture et une lecture au niveau du *bit* de mémoire. La plus petite entité utilisée dans le langage C étant l'octet, ceci n'est possible qu'en utilisant les opérations de bits du langage C pour l'extraction de certains bits d'octets donnés. Les structures de données, ainsi que les fonctions d'accès et de modification de ces structures, deviennent donc moins intuitives et plus difficiles à formaliser.

2.3 Conclusion

Dans un monde où les logiciels deviennent omniprésents dans tous les domaines, les méthodes formelles commencent à s'imposer comme un moyen de renforcer le niveau de sécurité et de correction des systèmes informatiques et en particulier des programmes. La diversité des systèmes à analyser, des propriétés visées et des moyens mis en oeuvre ont donné naissance à une grande variété de méthodes formelles et d'outils, répondant à des objectifs différents, comme il a été décrit dans ce chapitre.

Le monde de la carte à puce n'échappe pas à cette tendance. Sa vocation sécuritaire motive la vérification formelle de propriétés de sécurité. Par ailleurs, le coût d'une erreur importante sur une carte dupliquée à des millions d'exemplaire justifie un besoin de renforcer la vérification de la correction des programmes embarqués. Les nombreuses études présentées dans ce chapitre illustrent la possibilité et l'intérêt d'appliquer des méthodes formelles dans le monde de la carte à puce.

Les contraintes du marché limitent toutefois l'utilisation, longue et coûteuse, des méthodes formelles dans le monde industriel. Ceci justifie la recherche incessante de nouvelles méthodes, plus efficaces et plus intuitives, pour l'intégration des méthodes formelles au sein même de la chaîne de développement de logiciels.

C'est dans cette optique que nous avons développé des extensions d'outils existants, qui seront présentées dans le Chapitre 4, et une méthodologie de vérification, qui sera présentée dans le Chapitre 5. Forts de notre expérience de vérification de propriétés sécuritaires de la plate-forme Java Card, qui sera présentée dans le Chapitre suivant, nous avons cherché à utiliser un outil de vérification de programme, offrant un langage de spécification proche du langage de programmation et pouvant être utilisé par un programmeur pour spécifier les programmes qu'il développe. Cet outil sera utilisé pour prouver de la correction du programme. Il permettra également d'extraire un modèle pour la vérification de propriétés de haut niveau.

Chapitre 3

Vérification formelle de l'isolation d'applications Java Card

Résumé

Ce chapitre présente une partie de nos travaux (présentée dans [4, 6, 5]), qui consiste à prouver formellement que le mécanisme de *firewall* de la plate-forme Java Card (présentée dans la Section 1.2 et, par exemple, dans [34]) assure les propriétés de confidentialité et d'intégrité des données des applications. Nous avons utilisé un modèle formel existant de la plate-forme Java Card, développé dans le projet FORMAVIE (voir [18]) pour formaliser ces propriétés (la confidentialité étant modélisée sous la forme d'une propriété de non-interférence, voir [56, 57, 110]) et les prouver à l'aide du système de preuve Coq (voir [113]).

Sommaire

3.1 La modélisation de la plate-forme Java Card

Comme nous l'avons mis en évidence dans le Chapitre 1, les nouvelles plate-formes multi-applicatives et ouvertes constituent un enjeu sécuritaire majeur (voir Section 1.1.5.3). Dans ce cadre, la plate-forme Java Card implémente un mécanisme complexe de contrôle d'accès aux données, le *firewall*, sous la forme de règles définies pour chaque instruction *bytecode* du langage (voir Section 1.2.3). Notre but ici est de prouver formellement que l'ensemble de ces règles assure la protection des données d'une application, vis-à-vis des autres applications embarquées sur la même carte. Plus précisément, nous prouvons qu'*aucune information* sur les données d'une application ne peut être obtenue ou modifiée par les autres applications.

Prouver formellement que la plate-forme Java Card assure les propriétés d'isolation d'applets, à savoir la confidentialité et l'intégrité, nécessite trois étapes. Premièrement, un modèle formel de la plate-forme Java Card doit être construit. Ici, nous avons utilisé le modèle déjà développé dans le projet FORMAVIE (voir [18]), utilisant le système de preuve Coq. Ce modèle est décrit en détail dans cette Section. Ensuite, les propriétés de confidentialité et d'intégrité doivent être formalisées dans ce modèle, puis prouvées. La Section 3.2 décrit les étapes successives qui ont mené à un énoncé formel et à des preuves formelles des propriétés d'isolation, et ce pour toute la chaîne d'exécution d'une commande, incluant les opérations du *card manager*, l'interprétation des méthodes par la machine virtuelle et les appels éventuels à des méthodes de l'API. Enfin, la Section 3.3 présente un des bénéfices majeurs du travail, à savoir les conditions sur lesquelles reposent l'isolation et qui doivent être vérifiées au niveau de l'implémentation.

Modéliser la technologie Java Card consiste à définir une représentation formelle de chaque pas d'exécution d'une commande reçue du terminal. Chacun de ces pas peut être vu comme une transformation de l'état global de la carte. Par conséquent, la plate-forme entière est modélisée comme une *machine à états*, où une transition entre deux états correspond à un pas d'exécution de commande. Comme expliqué dans la Section 1.2.2, l'exécution d'une commande consiste en plusieurs étapes successives. Tout d'abord le *card manager* analyse la commande reçue et prépare l'état de la machine à l'interprétation de la méthode appropriée. La méthode est alors interprétée par la machine virtuelle, avec des appels possibles à des méthodes de l'API. Enfin, le résultat de l'exécution de la méthode est analysé par le *card manager* en vue de répondre au terminal et de préparer l'état à la réception d'une nouvelle commande. Par conséquent, trois types de transitions sont définies : les transitions du *card manager*, les transitions de la machine virtuelle et les transitions de l'API.

Il est important de noter que l'exécution des commandes reçues par la carte se fait dans un *environnement* donné. Par exemple, une erreur fatale due à un dépassement de la capacité mémoire de la carte peut survenir à tout moment. De même, la carte peut être retirée du terminal de façon prématurée, provoquant un arrachage. Par conséquent, la carte est modélisée par une machine qui réagit *de façon déterministe* à des évènements extérieurs. Il y a trois types d'évènements possibles, qui sont mutuellement exclusifs :

- une erreur fatale due à un manque de ressources (représentée par `FE`, pour *fatal error*),
- un arrachage de la carte, provoquant sa mise hors tension (représenté par `PO`, pour *power off*),
- ni erreur fatale ni arrachage, ce qui correspond à l'absence des deux évènements précédents (représenté par $\neg$`FE`$\wedge\neg$`PO`).

Notons que les évènements sont effectivement mutuellement exclusifs puisqu' une erreur ne peut pas se produire lorsque la carte est hors tension. Il existe donc une et une seule transition à partir d'un état donné et d'un évènement donné.

Cette section donne les définitions et les codages en Coq des différents composants de la machine à états : les états, les trois types de transitions, et enfin l'exécution totale d'une commande.

3.1.1 Les états

Définition 1 (États) L'*état* global de la carte est représenté par une structure qui rassemble toutes les données nécessaires à l'exécution, par la plate-forme, des commandes reçues du terminal. Il est composé :

- du *statut de commande* qui indique soit que la carte est en attente d'une commande, soit qu'elle est en train d'exécuter une commande ;
- de l'ensemble des *packages* déjà chargés sur la carte ;
- de l'*applet actuellement sélectionnée* qui va exécuter la commande (sauf si la commande requiert la sélection d'une autre applet) ;
- du *buffer APDU* qui est un tableau d'octets utilisé pour stocker les composants de la commande APDU (tels qu'ils sont spécifiés dans la norme ISO/IEC 7816−4, voir [79]), à savoir l'en-tête de la commande, qui décrit l'instruction, et le corps (optionnel) de la commande qui contient les données nécessaires à l'exécution de l'instruction ;
- de l'*état de jcvm*, qui contient les données nécessaires à la machine virtuelle pour exécuter une méthode, c'est-à-dire interpréter chaque instruction *bytecode* de la méthode. Un état de jcvm est composé :
  - du *statut d'exécution*, qui peut être de quatre formes différentes :
    - soit une pile de *frames* (si la machine est en train d'interpréter une méthode), où une *frame* est une structure mémorisant les données d'exécution de la méthode, à savoir :
      - le contexte courant dans lequel est exécutée la méthode,
      - la pile des opérandes (pour les données temporaires),
      - les variables locales utilisées par la méthode (comprenant les arguments de la méthode),
      - le compteur de programme indiquant le bytecode de la méthode qui est en train d'être exécuté ;
    - soit une valeur de retour, si l'exécution de la méthode est terminée ;
    - soit une exception qui aurait été levée durant l'exécution de la méthode, et non rattrapée ;
    - soit une erreur fatale, si l'exécution de la méthode à mené à une défaillance irréversible (telle qu'un dépassement de capacité de la mémoire) ;
  - du *tas* (*heap*) de la machine virtuelle, qui contient tous les objets créés, à savoir les instances de classes et les tableaux ; plus précisément, il s'agit d'une table d'association qui, à une référence (adresse) donnée, associe la structure de donnée représentant l'objet ;
  - de l'*image des champs statiques* (*static field image*) qui contient tous les champs statiques de toutes les classes définies dans tous les packages chargés ;
  - du *journal des transactions* (*transaction log*) qui enregistre les opérations durant une transaction, afin de pouvoir annuler la transaction si une erreur s'est produite (comme un arrachage de la carte).

Formalisation 1 (États) L'ensemble des états de la machine est représenté par un type inductif en Coq :

```
Record state : Set :=
  State { state_command_status : command_status;
          state_loaded_pckgs   : pckg_tbl;
          state_sel_app        : applet_ident;
          state_apdu_buffer    : apdu_buffer;
          state_jcvm_state     : jcvm_state }.
```

A leur tour, tous les composants de l'état (`command_status`, `pckg_tbl`, etc) sont également modélisés par des types inductifs. Par exemple, un statut de commande est défini par :

```
Inductive command_status : Set :=
 | Exec_Comm    : command_status
 | Waiting_Comm : command_status.
```

Regardons également la modélisation des états de jcvm :

```
Record jcvm_state : Set :=
  JCVMState { jcvm_state_execution_status : execution_status;
              jcvm_state_heap             : heap;
              jcvm_state_log              : transaction_log_status;
              jcvm_state_field_images     : static_field_images }.
```

où, par exemple, un statut d'exécution est défini par :

```
Inductive execution_status : Set :=
 | Frame_Stack        : frame_stack -> execution_status
 | Return_Value       : (option data) -> execution_status
 | Uncaught_Exception : reference -> execution_status
 | Fatal_Error        : execution_status.
```

où `frame_stack` est défini par :

```
Definition frame_stack : Set := (list frame).

Record frame : Set :=
  Frame { frame_ctxt   : frame_context;
          frame_stack  : stack;
          frame_locals : locals;
          frame_pc     : program_counter }.
```

Il est à noter que certaines caractéristiques n'ont pas été définies de façon explicite, mais de façon axiomatique. Par exemple, les références (les adresses) sont représentées par un type laissé en paramètre de la spécification :

```
Parameter reference : Set.
```

En effet, la forme exacte des références n'est pas précisée par la spécification de Sun (voir [93]) et dépend de l'implémentation. Or, l'idée générale de la modélisation est d'être le plus proche possible de la spécification de Sun, et, en particulier, de n'ajouter aucune information dépendante de l'implémentation.

Définition 2 (Contexte actif courant) Comme expliqué dans la Section 1.2.3, le *contexte actif courant* est le contexte de la méthode qui est en train d'être exécutée, ou celui de la méthode appelante si la méthode exécutée est statique. Dans notre modélisation, le *contexte actif courant* d'un état est celui de la *frame* au sommet de la pile de *frames* de son état de jcvm.

3.1.2 Les transitions

Le système de transitions de la machine à états représente tous les pas effectués pendant l'exécution d'une commande reçue. Cela comprend les opérations spécifiques effectuées par le *card*

manager et l'interprétation de *bytecode* par la machine virtuelle, qui peut inclure des appels à des méthodes de l'API. Par conséquent, il y a trois types de transitions différents : les opérations du *card manager*, l'interprétation d'une instruction *bytecode* d'une méthode donnée par la machine virtuelle et enfin l'exécution d'une méthode de l'API. Dans la suite, nous décrivons la définition formelle de ses transitions, ainsi que leur modélisation en Coq.

En ce qui concerne la modélisation en Coq, précisons que les transitions ont été modélisées par des *relations* entre deux états (*(transition s s')*), c'est à dire à l'aide de *prédicats inductifs* en Coq. Cette définition relationnelle, plutôt que fonctionnelle (*s'=(transition s)*), permet à un résultat (ici l'état final *s'*) d'être *non spécifié* pour certaines entrées. En d'autre termes, l'opération peut ne pas être *totale*. Cela permet de suivre exactement la spécification informelle de Sun, lorsque certains résultats ne sont pas spécifiés. De plus, une modélisation relationnelle permet également de définir plusieurs résultats différents pour une même entrée, ce qui est utile lorsque l'on modélise des cas où une erreur peut se produire quelle que soit l'entrée, comme lorsque la mémoire est pleine ou qu'un arrachage se produit[1].

3.1.2.1 Transitions du *card manager*

Définition 3 (Transitions du *card manager*) Une *transition du card manager*, entre deux états s et s' (notée $s \xrightarrow{\mathsf{CM}} s'$) représente une opération spécifique du *card manager*. Une telle opération peut être soit une opération qui ne nécessite pas l'intervention de la machine virtuelle (comme l'installation d'une applet ou le chargement d'un *package*), soit une gestion de l'état de la machine avant et après l'intervention de la machine virtuelle. En effet, lorsqu'une commande reçue nécessite l'interprétation d'une méthode par la machine virtuelle, le *card manager* prépare l'état de jcvm pour l'exécution de la méthode (par exemple, il construit la pile de frame avec une frame contenant les arguments de la méthode). Il effectue également des opérations finales afin de construire, à partir du résultat de l'exécution, la réponse qui va être envoyée au terminal, et pour préparer l'état à recevoir la commande suivante, comme par exemple en mettant à zéro son buffer APDU.

Plus précisément, $s \xrightarrow{\mathsf{CM}} s'$ représente en fait une transition du *card manager* "normale", c'est-à-dire où aucun arrachage ou dépassement de capacité de la mémoire ne s'est produit :

$$s \xrightarrow{\mathsf{CM}\ ,\ \neg\ \mathtt{FE}\ \wedge\ \neg\ \mathtt{PO}} s'$$

alors que $s \xrightarrow{\mathsf{CM}\ ,\ \mathtt{FE}} s'$ représente une transition du *card manager* où une erreur fatale s'est produite et $s \xrightarrow{\mathsf{CM}\ ,\ \mathtt{PO}} s'$ une transition du *card manager* où un arrachage s'est produit.

Formalisation 2 (Transition du *card manager*) Chaque opération du *card manager* est modélisée de façon indépendante par une relation entre deux états de la machine. Par exemple, la mise à zéro du buffer APDU par le *card manager* est modélisé par le prédicat inductif :

```
| make_zeroed_apdu_buffer : state -> state -> Prop
```

où le buffer APDU de l'état final a été mis à zéro. En ce qui concerne l'initialisation de l'état de jcvm, le prédicat suivant est défini :

```
| initialise_jcvm_state : state ->  state -> Prop
```

[1] La formalisation exécutable de Java Card citée précédemment (voir [13]) est une modélisation *fonctionnelle*. Dans cette modélisation, également développée en Coq, les situations non spécifiées et les terminaisons soudaines sont gérées par des exceptions.

où l'état final est obtenu en ajoutant une *frame* à la pile de frames de l'état de jcvm de l'état initial. Cette pile de frame est vide dans l'état initial étant donné qu'aucune méthode est en train d'être exécutée. Par conséquent l'état final ne contiendra qu'une seule *frame*, construite de la façon suivante :

- le contexte est celui de l'applet actuellement sélectionnée (cette initialisation implique donc un changement de contexte),
- la pile d'opérandes est vide,
- les variables locales contiennent les arguments de la méthode (obtenus à partir des composants du buffer APDU),
- et le compteur de programme pointe sur le premier *bytecode* de la méthode.

3.1.2.2 Transitions de la machine virtuelle

Définition 4 (Transition de la machine virtuelle) Une *transition de la machine virtuelle* représente l'exécution d'une unique instruction *bytecode* d'une méthode par la machine virtuelle. Plus précisément, une transition de la machine virtuelle d'un état s à un état s' (notée $s \xrightarrow{\mathsf{VM}} s'$) correspond à l'exécution, à partir de s, du *bytecode* courant de s, c'est-à-dire celui pointé par le compteur de programme de la *frame* en haut de la pile de *frames*.

Une fois encore, $s \xrightarrow{\mathsf{VM}} s'$ représente en fait $s \xrightarrow{\mathsf{VM},\ \neg\,\mathtt{FE} \wedge \neg\,\mathtt{P0}} s'$, alors que $s \xrightarrow{\mathsf{VM},\ \mathtt{FE}} s'$, et $s \xrightarrow{\mathsf{VM},\ \mathtt{P0}} s'$ respectivement, représente une transition de la machine virtuelle où une erreur fatale, ou un arrachage respectivement, a eu lieu.

Formalisation 3 (Transition de machine virtuelle) Une transition de machine virtuelle est modélisée de façon relationnelle par le prédicat `vm_transition`. Ce prédicat extrait tout d'abord le *bytecode* courant de l'état initial, puis utilise un prédicat auxiliaire `bytecode_transition`, qui prend en argument le *bytecode* qui doit être exécuté :

```
Inductive vm_transition (currstate newstate : state) : Prop:=
| VMTransition : forall currfrm:frame, forall bytcd:bytecode,
     (current_frame currstate currfrm) ->
     (current_bytecode (frame_program_counter currfrm) bytcd) ->
     (bytecode_transition currstate bytcd newstate) ->
     (vm_transition currstate newstate).
```

Un prédicat inductif exprime une disjonction de clauses. Chaque clause correspond à une façon de construire le résultat et est définie par un constructeur du prédicat. Dans chaque clause, les hypothèses représentent les préconditions et les postconditions de l'opération. Ces hypothèses sont les arguments du constructeur correspondant. Ici, il n'y a qu'une seule façon de construire l'état final `newstate` à partir de `currstate` par la relation `vm_transition`. Il n'y a donc qu'un seul constructeur `VMTransition`, dont les arguments définissent la façon de construire `newstate` : `newstate` doit être en relation avec `bytcd` et `currstate` par la relation `bytecode_transition`, où `bytcd` est le *bytecode* courant de `currfrm`, et où `currfrm` est la frame courante de `newstate`.

Le prédicat `bytecode_transition` est, quant à lui, défini par une analyse par cas sur le genre de *bytecode* à exécuter :

```
Inductive bytecode_transition: state -> bytecode -> state -> Prop:=
| ControlTransition:
    forall bytcd:control_bytecode,
    <other for all quantifications >
    let currfrm   := (Frame ctxt stck locs pc) in
    let currstate := (State comm_status pckgs sel_app apdu_buff
                      (JCVMState (Frame_Stack (cons currfrm frmstck))
                                 hp log sfis)) in
    (control_transition locs stck pc bytcd newstck newpc) ->
    let newfrm    := (Frame ctxt newstck locs newpc) in
    let newstate  := (State comm_status pckgs sel_sep apdu_buff
                      (JCVMState (Frame_Stack (cons newfrm frmstck))
                                 hp log sfis)) in
    (bytecode_transition currstate bytcd newstate)
| SafeFrameDataTransition:   ...
| SafeHeapTransition:        ...
| ...
```

Chaque clause de `bytecode_transition` suit le même schéma, utilisant un prédicat auxiliaire spécifique au genre de *bytecode* exécuté. Par exemple, le *bytecode* peut modifier le flot de contrôle de la méthode courante, comme le bytecode `goto`. Dans ce cas, le compteur de programme et les variables locales de la *frame* en haut de la pile sont modifiés. Cette transition est appelée une "transition de flot de contrôle" et est définie par un prédicat auxiliaire `control_transition` :

```
Inductive control_transition
    (locs:locals)(stck:stack)(pc:program_counter) :
     control_bytecode -> stack -> program_counter -> Prop :=
 | BYGotoTransition:
       forall bytcd:goto_bytecode,
       forall newpc:program_counter,
       (goto_transition pc bytcd newpc) ->
       (control_transition locs stck pc bytcd stck newpc)
 | BYIfTransition: ...
 | BYSwitchTransition: ...
 | BYSubroutineTransition: ...  .
```

A leurs tours, ces prédicats spécifiques sont définis à l'aide d'autres prédicats, jusqu'à ce que les prédicats "primitifs" soient utilisés, i.e. ceux correspondant aux instructions *bytecode* réelles de Java Card, telles que `goto`, `getfield`, etc.

3.1.2.3 Transitions de l'API

Définition 5 (Transition de l'API) Une *transition de l'API* d'une méthode m, d'un état s à un état s' (notée $s \xrightarrow{\mathsf{API(m)}} s'$) correspond à l'exécution de la méthode m à partir de l'état s, résultant dans l'état s'. L'exécution est spécifiée formellement pour chaque méthode m de l'API. La spécification formelle de chaque méthode est sous la forme de préconditions et postconditions sur les états de la machine : $s \xrightarrow{\mathsf{API(m)}} s'$ signifie que s satisfait la précondition de m et que s' satisfait la postcondition de m à partir de s.

La postcondition spécifie la valeur de retour de la méthode, ainsi que l'état résultant de l'exécution de la méthode à partir d'un état initial donné. Le résultat de la méthode peut être la levée d'une exception.

La précondition spécifie les conditions nécessaires pour que la méthode puisse être correctement invoquée. En d'autres termes, elle définit les conditions sur l'état initial pour que la postcondition soit satisfaite. Principalement, la précondition définit la

forme de la pile d'opérandes, qui contient les arguments de la méthode. En particulier elle définit les types de ces arguments.
Une fois encore, $s \xrightarrow{\mathsf{API(m)}} s'$ représente en fait $s \xrightarrow{\mathsf{API(m)}\ ,\ \neg\ \mathsf{FE}\ \wedge\ \neg\ \mathsf{PO}} s'$, alors que $s \xrightarrow{\mathsf{API(m)}\ ,\ \mathsf{FE}} s'$, et $s \xrightarrow{\mathsf{API(m)}\ ,\ \mathsf{PO}} s'$ respectivement, représente une transition de l'API où une erreur fatale, ou un arrachage respectivement, a eu lieu.

Notons que si les postconditions des méthodes de l'API doivent spécifier également l'état de retour, en plus du résultat de la méthode, c'est dû au fait que certaines méthodes de l'API sont utilisées comme point d'entrée dans le système, c'est-à-dire qu'elles sont utilisées pour présenter des requêtes au système. Par exemple, c'est le cas de certaines méthodes de la classe `JCSystem`. Des méthodes comme `beginTransaction()` ou `commitTransaction()` permettent à des applets d'utiliser le mécanisme de transaction, qui est géré par le système. Par conséquent, les spécifications de ces méthodes doivent décrire les effets sur l'état du système lui-même, et en particulier avec la partie qui traite des transactions.

Formalisation 4 (Transition de l'API) Une transition de l'API est modélisée en Coq par un prédicat générique `api_transition` :

```
Definition api_transition
    (precond  : state -> Prop)
    (postcond : state -> state -> method_result -> Prop)
    (initst   : state)
    (newstate : state)(res : method_result) : Prop :=
  (precond  initst) ->
  (postcond initst newstate res).
```

Ce prédicat générique est ensuite instancié pour chaque méthode de l'API.

Exemple. Regardons l'exemple de la méthode

```
static short arrayCopy(byte[] src,  short srcOff,
                       byte[] dest, short destOff, short length)
```

déjà mentionné page 37. Cette méthode copie un tableau de taille `length` à partir d'une source `src`, en commençant à la position `srcOff`, vers la position `destOff` du tableau de destination `dest`. La précondition de cette méthode spécifie que, au moment de l'invocation, la pile des opérandes doit avoir la forme suivante :

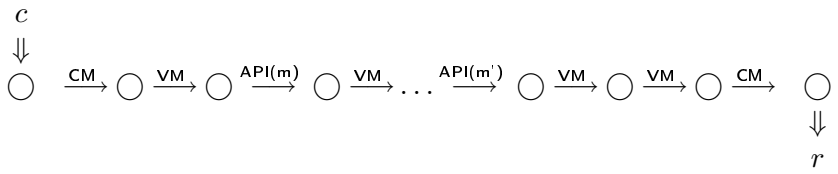
`length`	`: byte`
`destoffset`	`: short`
`dest`	`: reference`
`srcoffset`	`: short`
`src`	`: reference`
...	

La précondition ne spécifie pas que `dest` ou `src` doivent être différents de `null`. En effet, dans ce cas, la spécification de la méthode est que l'exception `NullPointerException` doit être levée. Par conséquent, ceci concerne le comportement de la fonction et doit être compris dans la postcondition.

La postcondition est définie par un type inductif où chaque constructeur correspond à un résultat possible de l'exécution de la méthode : il y a un constructeur correspondant à un comportement "normal" de la méthode et un constructeur pour chaque cas d'exception pouvant être levée par cette méthode. Ici, trois types d'exceptions peuvent être levées : `NullPointerException` si `src` ou `dest` sont la référence `null`, `ArrayIndexOutOfBoundsException` si la copie provoque un accès à des données en dehors des bornes des tableaux et `TransactionException` si la copie (qui est une opération atomique) provoque un dépassement de la capacité de mémorisation des opérations dans une transaction.

3.1.3 Exécution d'une commande

Rappelons une fois de plus que lorsqu'une commande est reçue, le *card manager* commence par préparer l'état initial, de manière à ce que la méthode (`process`, `select`, etc.) puisse être interprétée par la machine virtuelle[1]. Puis les instructions *bytecode* de la méthode sont interprétées une par une par la machine virtuelle, avec d'éventuels appels à des méthodes de l'API. Enfin, le *card manager* analyse le résultat de la méthode pour construire la réponse à rendre au CAD et prépare l'état pour la commande suivante. L'exécution d'une commande peut être représentée de la façon suivante :



Cette exécution correspond en fait à un cas où aucune erreur fatale et aucun arrachage n'a eu lieu pendant l'exécution. Toutefois, chacune de ces transitions dépend en fait de l'environnement de la carte, c'est à dire de l'un des trois évènements possibles (`PO`, `FE`, or ¬`FE`∧¬`PO`). Par conséquent, l'exécution d'une commande dépend également de l'environnement de la carte et peut être formellement définie comme suit.

Définition 6 (Exécution d'une commande) Soit E un *environnement*, c'est-à-dire une séquence e_0, e_1, ..., e_k, ... d'évènements (chaque e_i est soit `PO`, soit `FE`, soit ¬`FE`∧¬`PO`). L'*exécution* d'une commande c, à partir d'un état initial s_0, dans l'environnement E, est une paire (s_0 s_1 ... s_n, r), où r est la réponse à envoyer au terminal et s_0 s_1 ... s_n est l'unique séquence finie d'états telle que :

- dans l'état s_0, la carte est en attente d'une commande : son *statut de commande* est `Waiting_Comm` ;
- n est le nombre minimal (non nul) tel que le *statut de commande* de s_n est de nouveau `Waiting_Comm` ;
- le *statut de commande* de tous les s_i, pour i allant de 1 à $n-1$, est `Exec_Comm` ;
- les états sont reliés par des transitions dans l'environnement donné :

  $$s_i \xrightarrow{\mathsf{R}\,,\, e_i} s_{i+1} \quad \text{pour } i \text{ allant de } 0 \text{ à } n-1$$

  où R est soit CM, soit VM, soit API(m) pour une méthode m donnée.

Le nombre n est appelé la *longueur* de l'exécution et la séquence d'états s_0 s_1 ... s_n est appelée la *trajectoire* de l'exécution.

Formalisation 5 (Dispatcher) L'exécution d'une commande est modélisée en Coq par un type inductif `dispatcher`, dont la signature est la suivante :

```
| dispatcher: state -> command -> state_trajectory -> response -> Prop.
```

Le *dispatcher* associe, à un état initial et une commande reçue, la trajectoire de l'exécution de cette commande et la réponse à renvoyer au CAD. Le type `state_trajectory` est défini comme une liste d'états et une commande est définie par le type inductif :

[1] Sauf pour les commandes ne nécessitant pas l'intervention de la machine virtuelle, comme le chargement d'un nouveau *package*, géré uniquement par le *card manager*.

```
Inductive command : Set :=
 | Select       : apdu_comm -> command
 | Command      : apdu_comm -> command
 | Load_File    : cap_format -> command
 | Install      : aid -> package_info -> install_params -> command
 | Load_Install : cap_format -> aid -> package_info ->
                  install_params -> command
 | Reset        : command.
```

où le type `apdu_comm` contient les composants de la commande APDU reçue (tels que spécifiés dans ISO 7816−4). Le prédicat `dispatcher` est défini par une analyse par cas sur le genre de la commande reçue. Par conséquent, le prédicat `dispatcher` possède six constructeurs, un pour chaque type de commande. Par exemple, si la commande reçue est `Command(apducomm)`, le *dispatcher* utilise d'abord `apducomm` pour initialiser l'état de la machine et appelle ensuite inductivement le prédicat `vm_transition` (défini dans la Formalisation 3) pour construire l'état final après exécution de la commande.

3.2 Formalisation de l'isolation d'applet

Maintenant que la modélisation de la plate-forme Java Card a été définie, les propriétés d'isolation d'applets peuvent être formalisées à l'aide du modèle. Le travail présenté dans ce chapitre consiste à modéliser formellement les propriétés de confidentialité et d'intégrité et de les prouver formellement sur le modèle. Dans cette section, nous allons tout d'abord expliquer la notion informelle d'isolation d'applets. Ensuite, nous allons progressivement formaliser les différents concepts, pour aboutir à un énoncé formel des propriétés de confidentialité et d'intégrité. Enfin, nous parlerons de la preuve de ces propriétés sur le modèle.

3.2.1 Définition de l'isolation d'applets

Les propriétés de confidentialité et d'intégrité expriment le fait qu'il est impossible d'accéder illégalement à toute information appartenant à un autre contexte, où un accès illégal entre contextes est un accès entre contextes qui ne rentre pas dans les cas autorisés décrit dans la Section 1.2.3. Par conséquent, la propriété d'isolation peut être énoncée comme suit :

> *toute applet α ne peut obtenir illégalement aucune information et ne modifier illégalement aucune donnée appartenant à un autre contexte.*

En utilisant la terminologie sécuritaire classique des *sujets*, *objets* et *actions* (comme, par exemple, dans les Critères Communs [38]), les *sujets* sont les applets et le système (c'est-à-dire les *propriétaires* tels que définis en Section 1.2.3) ; les *objets* sont les informations appartenant à un contexte donné, à savoir le contenu de tous les champs de tous les objets appartenant à ce contexte ; enfin, les *actions* d'une applet correspondent à l'exécution de commandes, lorsque l'applet est actuellement sélectionnée. La propriété d'isolation peut donc s'exprimer de la façon suivante :

> *pour toute applet actuellement sélectionnée α, pour tout contexte C qui ne contient pas α, pour toute commande c reçue du terminal, l'exécution de c (par α) ne peut obtenir illégalement aucune information et ne modifier illégalement aucune donnée appartenant au contexte C.*

Nous devons maintenant formaliser la notion d'accès *illégal*. Pour cela, nous devons ajouter les hypothèses suivantes.

Interfaces partageables. Nous supposons que le contexte C ne fournit pas d'interface partageable, puisque cela correspond à un des quatre cas où un accès entre contexte est autorisé (voir page 35). En effet, si un objet appartenant à C fournit une interface partageable, alors, suivant les règles du *firewall*, il peut être accédé depuis n'importe quel contexte et donc en particulier par α.

La commande Select. De plus, la propriété d'isolation d'applet ne tient pas compte des commandes qui demandent la sélection d'une applet β du contexte C. En effet, dans ce cas β devient le propriétaire actif et C devient le contexte actif courant[1]. A partir de ce moment, β est responsable de la protection des données de son contexte. En particulier, β doit s'assurer qu'aucune divulgation ou modification d'information n'a lieu pendant l'exécution de sa méthode `process` ou lors de la transmission de la réponse au terminal. Étant donné que cette exécution dépend de l'argument `apdu`, dont β n'est pas responsable, β peut refuser d'être sélectionné, suivant sa propre politique de sécurité (au niveau de l'application, et non du système). En effet, comme expliqué page 34, il y a un appel préliminaire à la méthode `select` de β, qui peut répondre `false` pour refuser la sélection.

Le contexte JCRE. Enfin, C ne doit pas être le contexte JCRE, puisque, d'après les règles du *firewall*, les objets appartenant au contexte JCRE peuvent être accédés par toutes les applets, et donc en particulier par α.

En incluant ces hypothèses à l'énoncé de la propriété d'isolation, celle-ci devient :

> *Soit α l'applet actuellement sélectionnée. Soit C un contexte différent du contexte JCRE, ne contenant pas α et ne fournissant aucune interface partageable. Soit c la commande reçue du terminal, ne demandant pas la sélection d'une applet du contexte C. La propriété d'isolation d'applet est assurée si l'exécution de la commande c (par α) ne peut obtenir aucune information et ne modifier aucune donnée appartenant au contexte C.*

Les notions qui restent à formaliser sont l'obtention d'information et la modification de donnée pendant l'exécution d'une commande. La formalisation de l'obtention d'information est décrite dans la section suivante, permettant d'énoncer formellement la propriété de confidentialité. La formalisation de la notion, plus simple, de modification de donnée est présentée dans la Section 3.2.3 avec l'énoncé formel de la propriété d'intégrité.

3.2.2 Énoncé formel de la propriété de confidentialité

3.2.2.1 Non-interférence

Notre formalisation de la confidentialité est basée sur le concept classique de *non-interférence* (voir [56, 57, 110]), qui stipule que la confidentialité de données est assurée si les valeurs de ces données confidentielles n'ont aucun effet sur le comportement d'entités externes. Une manière classique de prouver cette propriété est de considérer deux états qui ne diffèrent que par les valeurs des données confidentielles, et de montrer que le *comportement* des entités externes est le même dans les deux états. En effet, si une entité externe a exactement le même comportement quelques soient les valeurs des données confidentielles, cela signifie qu'aucune information sur ces valeurs n'a été obtenue.

Dans notre contexte, étant donnée l'applet actuellement sélectionnée α de l'état s de la machine, les données confidentielles, à savoir les données qui ne doivent pas être lues par α, sont

[1] Voir Section 1.2.3 pour la définition de propriétaire actif et de contexte actif courant.

le contenu de tous les objets de tous les contextes ne contenant pas α. Si l'on considère deux états s_1 et s'_1, de la même carte à puce, qui ne diffèrent que par les valeurs des données confidentielles, alors l'applet actuellement sélectionnée α de s_1 est la même que celle s'_1. Donc, étant donné un contexte C ne contenant pas α (et qui est différent du contexte JCRE et ne fournit pas d'interface partageable), nous voulons prouver que l'exécution d'une commande c (par α) à partir de s_1, dans un environnement E donné, et l'exécution de la même commande à partir de s'_1, dans le même environnement E, ont le même *comportement*.

A présent, le concept de *"comportement de l'exécution de c"* doit être formalisé. Cela pourrait être défini par *"le résultat de l'exécution de c"*, mais ce serait trop restrictif. En effet, il pourrait y avoir une commande c, dont le *résultat* de l'exécution ne dépendrait pas du contexte C, mais dont la partie calculatoire de l'exécution en dépendrait. Par exemple, le temps d'exécution pourrait dépendre d'une donnée confidentielle et pourrait donc être utilisé par un utilisateur hostile pour obtenir de l'information sur le contexte C. De tels flux d'information doivent être pris en compte : un utilisateur hostile peut utiliser non seulement le résultat d'une exécution pour obtenir de l'information, mais également tout ce qui peut être observable concernant cette exécution. Nous devons donc considérer le *comportement* de l'exécution, à savoir sa *trajectoire*.

Au vu de toutes ces remarques, la confidentialité est maintenant définie par :

> *si s_1 et s'_1 ne diffèrent que par les objets appartenant au contexte C, alors l'exécution à partir de s_1 et l'exécution à partir de s'_1 de la même commande c, par la même applet actuellement sélectionnée α, dans le même environnement E, ont les mêmes trajectoires.*

Un point important reste à formaliser : le fait que deux états *ne diffèrent que par les objets appartenant au contexte C*. Cela amène aux définitions d'*équivalence d'état* et d'*équivalence d'état à un contexte près*, qui sont décrites dans les sections suivantes.

3.2.2.2 Équivalence d'états

Définir le fait que deux états s et s' puissent différer seulement sur les objets de C par une simple égalité de leur composants n'appartenant pas à C est trop restrictif. Une notion d'*équivalence* en dehors du contexte C est nécessaire.

Concentrons-nous tout d'abord sur une équivalence simple, c'est-à-dire sans prendre en compte le contexte C pour le moment. Étant donné que les états contiennent des références à des objets, définir l'équivalence d'états comme une simple égalité des composants n'est pas suffisant. En effet, lorsque la machine virtuelle interprète le *bytecode* `new`, une nouvelle référence est demandée au système d'exploitation, dont le comportement n'est pas spécifié. Du point de vue de la machine virtuelle, cette opération n'est pas déterministe. Or, l'exécution du *bytecode* `new` sur deux états qui sont équivalents devrait mener à deux états équivalents, même si les références utilisées sont différentes.

C'est pourquoi l'équivalence d'états est définie comme une égalité des composants *à une permutation des références près*, c'est-à-dire qu'il existe une bijection entre les références. Cette bijection n'est définie que pour les références qui apparaissent dans le *tas* de l'état (voir la définition d'un état en Section 3.1.1). En conséquence, un notion d'état *cohérent* est introduite :

> **Définition 7 (Cohérence)** Un état est dit *cohérent vis à vis des références* (ou simplement cohérent) s'il ne contient dans ses structures de données que des références nulles ou des références apparaissant dans le domaine du tas.

L'équivalence d'états peut alors être définie, pour des états cohérents, par :

Définition 8 (Équivalence d'états) Deux états cohérents s et s' sont *équivalents*, noté $s \sim s'$, s'il existe une bijection $\varphi : \mathrm{dom}(\mathcal{H}_s) \rightarrow \mathrm{dom}(\mathcal{H}_{s'})$ telle que s' est obtenu à partir de s en remplaçant toute référence ρ apparaissant dans une structure de donnée de s par son image $\varphi(\rho)$.

Ici, $\mathrm{dom}(\mathcal{H}_s)$ représente le domaine du tas de s. Plus précisément, le tas de l'état s est modélisé par une relation $\mathcal{H}_s$, qui associe des références aux objets du tas : $(\rho, \Omega) \in \mathcal{H}_s$ si et seulement si le tas de s contient l'élément Ω à la référence ρ. Le domaine de $\mathcal{H}_s$, noté $\mathrm{dom}(\mathcal{H}_s)$, est l'ensemble des références apparaissant dans le tas de s.

3.2.2.3 Équivalence d'états à un contexte près

A présent, dire que *"s et s' diffèrent seulement sur les objets du contexte C"* est formalisé par *"s et s' sont équivalents, sauf éventuellement pour les objets du contexte C"*. Cela signifie qu'il existe une bijection φ comme précédemment, telle que s' est obtenu à partir de s comme précédemment, sauf pour les objets appartenant à C pour lesquels aucune hypothèse n'est faite.

Définition 9 (Équivalence d'états à un contexte près) Deux états cohérents s et s' sont *équivalents à un contexte C près*, noté $s \sim_C s'$, s'il existe une bijection $\varphi : \mathrm{dom}(\mathcal{H}_s) \rightarrow \mathrm{dom}(\mathcal{H}_{s'})$ telle que :

- Chaque composant de s', *excepté le tas*, est obtenu à partir de son vis-à-vis dans s en remplaçant toute référence ρ apparaissant dans les structures de données de ce composant par $\varphi(\rho)$.
- Concernant le tas :
  - Pour chaque $(\rho, \Omega) \in \mathcal{H}_s$ tel que $\Omega \notin C$, nous avons $(\varphi(\rho), \varphi(\Omega)) \in \mathcal{H}_{s'}$, où $\varphi(\Omega)$ représente les objets obtenus en remplaçant toutes les références par leurs images par φ. Remarquons que Ω et $\varphi(\Omega)$ appartiennent au même contexte.
  - Inversement, pour chaque $(\rho', \Omega') \in \mathcal{H}_{s'}$ tel que $\Omega' \notin C$, nous avons $(\varphi^{-1}(\rho'), \varphi^{-1}(\Omega')) \in \mathcal{H}_s$

Aucune supposition n'est faite pour les objets appartenant à C. Par exemple, le tas de s peut contenir un objet Ω appartenant à C à la référence ρ, alors que le tas de s' contient, à la référence $\varphi(\rho)$, un objet Ω', de la même classe que Ω, mais avec des valeurs arithmétiques différentes dans ses champs. Cela correspond exactement au fait que les états peuvent différer sur les objets du contexte C.

3.2.2.4 Énoncé formel de la confidentialité

Nous pouvons à présent établir un énoncé formel de la propriété de confidentialité, utilisant les différents définitions formelles introduites :

Définition 10 (Confidentialité) Soit C un contexte différent du contexte JCRE et qui ne fournit pas d'interface partageable. La confidentialité des données du contexte C est assurée si :

- pour tout environnement E de la carte,
- pour toute commande c reçue du terminal, qui ne demande pas la sélection d'une applet du contexte C,
- pour toute paire d'états cohérents s_1 et s_1' tels que $s_1 \sim_C s_1'$ et tels que l'applet sélectionnée α dans s_1 (et donc dans s_1' également) n'appartient pas au contexte C

l'exécution (s_1 ... s_n,r) de c à partir de s_1 (par α) dans E et l'exécution (s'_1 ... s'_m,r') de c à partir de s'_1 (par α) dans E sont telles que :

- n=m
- $s_i \sim_C s'_i$ pour i allant de 2 à n
- $r = r'$.

Dans cette définition, nous avons en fait l'*équivalence des trajectoires* (exprimée par l'équivalence des états deux à deux) plutôt que l'égalité des trajectoires, pour les mêmes raisons qui ont justifié l'utilisation de l'équivalence d'états plutôt que l'égalité.

3.2.3 Énoncé formel de la propriété d'intégrité

La notion d'intégrité est plus facile à formaliser que celle de confidentialité, puisqu'une violation de la propriété est "observable". En effet, l'intégrité est assurée pendant l'exécution d'une commande si les valeurs des données sensibles à la fin de l'exécution sont les mêmes que celles au début de l'exécution. Il n'est plus nécessaire de considérer deux trajectoires. La propriété d'intégrité peut être formellement énoncée comme suit.

Définition 11 (Integrité) Soit C un contexte différent du contexte JCRE et qui ne fournit pas d'interface partageable. L'intégrité des données du contexte C est assurée si :

- pour tout environnement E de la carte,
- pour toute commande c reçue du terminal, qui ne demande pas la sélection d'une applet du contexte C,
- pour tout état s_1 tel que l'applet sélectionnée α dans s_1 n'appartient pas au contexte C

l'exécution (s_1 ... s_n,r) de c à partir de s_1 (par α) dans E est telle que $\mathcal{F}_C(s_n)$=$\mathcal{F}_C(s_1)$, où $\mathcal{F}_C(s)$ représente le contenu, dans l'état s, de tous les champs de tous les objets appartenant à C, i.e. les données à protéger.

3.2.4 Architecture des preuves

3.2.4.1 Preuve de la confidentialité

À présent que la propriété de confidentialité a été formellement énoncée, sous la forme de la Définition 10, notre but est de prouver sa validité, à savoir prouver le théorème suivant :

Théorème 1 (Confidentialité) *La confidentialité est assurée pour les données de tous les contextes de la carte.*

La vérification est faite à chaque pas d'exécution d'une commande reçue :

$$
\begin{array}{ccccccccccccc}
c & & & & & & & & & & & & r \\
\Downarrow & & & & & & & & & & & & \Uparrow \\
s_1 & \xrightarrow{\text{CM}} & s_2 & \xrightarrow{\text{VM}} & s_3 & \xrightarrow{\text{API(m)}} & \cdots & \xrightarrow{\text{API(m')}} & s_{n-2} & \xrightarrow{\text{VM}} & s_{n-1} & \xrightarrow{\text{CM}} & s_n \\
\wr C & & \wr C & & \wr C & & & & \wr C & & \wr C & & \wr C \\
s'_1 & \xrightarrow{\text{CM}} & s'_2 & \xrightarrow{\text{VM}} & s'_3 & \xrightarrow{\text{API(m)}} & \cdots & \xrightarrow{\text{API(m')}} & s'_{n-2} & \xrightarrow{\text{VM}} & s'_{n-1} & \xrightarrow{\text{CM}} & s'_n \\
\Uparrow & & & & & & & & & & & & \Downarrow \\
c & & & & & & & & & & & & r
\end{array}
$$

Comme le démontre ce diagramme, trois lemmes de confidentialité intermédiaires sont nécessaires : un pour les opérations spécifiques du *card manager*, un pour les transitions de la machine

virtuelle et un pour les transitions de l'API. Cela signifie que, pour tout contexte C qui est différent du contexte JCRE et qui ne fournit pas d'interface partageable, les lemmes suivants doivent être prouvés.

Lemme 1 (Confidentialité pour les opérations du *card manager*)
La confidentialité des données du contexte C est assurée lors des opérations spécifiques du card manager. *Cela signifie que :*

pour toute paire d'états cohérents s_1 et s'_1, tels que $s_1 \sim_C s'_1$ et tels que l'applet sélectionnée α dans s_1 (et donc dans s'_1) n'appartient pas au contexte C, si $s_1 \xrightarrow{\mathsf{CM}} s_2$ et $s'_1 \xrightarrow{\mathsf{CM}} s'_2$, alors $s'_2 \sim_C s'_2$.

Lemme 2 (Confidentialité pour les transitions de la machine virtuelle)
La confidentialité des données du contexte C est assurée par les transitions de la machine virtuelle. Cela signifie que :

pour toute paire d'états cohérents s_1 et s'_1, tels que $s_1 \sim_C s'_1$ et tels que l'applet sélectionnée α dans s_1 (et donc dans s'_1) n'appartient pas au contexte C, si $s_1 \xrightarrow{\mathsf{VM}} s_2$ et $s'_1 \xrightarrow{\mathsf{VM}} s'_2$, alors $s'_2 \sim_C s'_2$.

Lemme 3 (Confidentialité pour les transitions de l'API)
La confidentialité des données du contexte C est assurée par les transitions de toutes les méthodes m de l'API. Cela signifie que :

pour toute paire d'états cohérents s_1 et s'_1, tels que $s_1 \sim_C s'_1$ et tels que l'applet sélectionnée α dans s_1 (et donc dans s'_1) n'appartient pas au contexte C, si $s_1 \xrightarrow{\mathsf{API(m)}} s_2$ et $s'_1 \xrightarrow{\mathsf{API(m)}} s'_2$, alors $s'_2 \sim_C s'_2$.

Méthode générale de preuve pour la confidentialité. Les preuves de chaque lemme de confidentialité suivent le même schéma. Tout d'abord, nous considérons deux transitions de s_1 à s_2 et de s'_1 à s'_2 et nous considérons tous les cas possibles de construction de ces transitions. Chaque cas donne une spécification des états résultants s_2 et s'_2, à partir des états initiaux. Puis, nous prouvons que s_2 et s'_2 sont équivalents excepté sur le contexte C. Soulignons que cette démarche produit un grand nombre de cas à traiter : le carré du nombre de constructeurs qui définissent la transition en question (par exemple 8^2 cas pour le prédicat `bytecode_transition`, voir page 83, qui est composé de huit constructeurs). La plupart des cas ne sont pas cohérents, mais doivent être éliminés.

Invariants. Les hypothèses vérifiées par les états initiaux sont les mêmes pour tous les types de transitions. Cependant, lors de la preuve au niveau d'une exécution complète de commande, seules les hypothèses sur les états initiaux s_1 et s'_1 (du diagramme ci-dessus) sont supposées. Donc, pour pouvoir appliquer les lemmes intermédiaires de confidentialité à chaque étape de l'exécution, nous devons prouver des invariants sur les transitions, c'est-à-dire des propriétés qui sont stables par ces transitions.

Le principal invariant nécessaire à la preuve de confidentialité concerne la cohérence d'état[1]. En effet, chaque lemme de confidentialité est établi pour des états cohérents. Par conséquent, pour appliquer un ou l'autre de ces lemmes à chaque étape d'exécution de la commande, il faut prouver que s'il y a une transition d'un état s_1 à un état s_2 et si s_1 est cohérent, alors s_2 est également cohérent. Ceci doit être fait pour les trois types de transitions possibles : les opérations du *card manager*, les transitions de la machine virtuelle et les transitions de l'API.

[1]Voir Définition 7 pour la définition de cohérence d'état.

Preuve du Lemme 1. La preuve est faite par une analyse par cas des six sortes de commandes possibles (voir page 86). Un grand nombre de lemmes intermédiaires sont nécessaires, un pour chaque opération spécifique du *card manager*. Ces opérations spécifiques sont celles déjà mentionnées, comme le chargement de packages, la sélection et désélection d'applets, la transmission des commandes aux applets concernées, la gestion du buffer APDU, etc ($\sim$ 14 000 lignes de preuves Coq).

Preuve du Lemme 2. La preuve procède par analyse par cas sur le type de *bytecode*. Comme déjà vu (voir page 83), les transitions de *bytecode* sont divisées en familles imbriquées. Par exemple, il y a la famille des *bytecodes* de *contrôle*, dont l'exécution modifie le flot de contrôle, la famille des *bytecodes* de *données de frame*, dont l'exécution modifie les données de la *frame* du haut de la pile, etc. Chaque famille est à son tour divisée en sous-famille. Par exemple, dans la famille des *bytecodes* de *données de frame*, il y a la sous-famille des bytecodes *const*, à savoir `sconst`, `iconst` et `aconst`.

Comme vu en Section 3.1.2.2, pour chaque type de *bytecode*, un prédicat inductif Coq a été défini pour modéliser la transition correspondante à ce *bytecode* (voir l'exemple de `control_transition`, page 83). Chaque transition de *bytecode* fait l'objet d'un lemme de confidentialité, similaire au Lemme 2, mais où les transitions sont celles correspondant au *bytecode* donné. La preuve de chaque lemme ainsi obtenue suit la méthode générale que l'on vient de décrire ($\sim$ 15 000 lignes de preuves Coq ; 90 lemmes).

Preuve du Lemme 3. Ici, une preuve pour chaque méthode de l'API doit être fournie. Par conséquent, une propriété générique de confidentialité est définie pour l'API :

```
Definition confidentiality_api
     (precond  : state -> Prop)
     (postcond : state -> state -> method_result -> Prop)
     (confidentiality_condition : state -> Prop):=
< for all quantifications >
  (state_equivalence_up_to_one_context s1 s1' C phi) ->
  (confidentiality_condition s1) ->
  <other hypotheses of confidentiality) >
  (precond  s1 ) -> (postcond s1  s2  res)  ->
  (precond  s1') -> (postcond s1' s2' res') ->
  (state_equivalence_up_to_one_context s2 s2' C phi)
  /\ (res'=(method_result_isom res phi)).
```

Étant donné que le résultat de la méthode peut contenir des références, nous devons prouver que les résultats sont équivalents. Ici aucune hypothèse n'est faite sur les interfaces partageables puisque l'API n'en contient, ni n'en appelle, aucune. Comme pour la définition générique des transitions de l'API (voir Définition 5), les arguments de la définition de la confidentialité pour l'API sont utilisés pour instancier la propriété pour chaque méthode de l'API :

- `precond` et `postcond` représentent les spécifications de la précondition et de la postcondition de la méthode.
- `confidentiality_condition` contient des hypothèses supplémentaires, si besoin, pour assurer la confidentialité. Nous avons vu page 37 que les mécanismes de sécurité ne sont pas énoncés dans la spécification des méthodes de l'API. Par conséquent, il peut être nécessaire d'ajouter des conditions qui doivent être remplies pour que la confidentialité soit assurée. Ces conditions sont contenues dans cette variable.

Cette propriété générique est instanciée et prouvée pour chaque méthode de l'API ($\sim$ 3 300 lignes de preuves Coq).

Par exemple, pour vérifier la confidentialité pour la méthode

```
static short arrayCopy(byte[] src,  short srcOff,
                       byte[] dest, short destOff, short length)
```

(mentionnée page 37 et page 84), le lemme suivant sera prouvé :

```
Theorem confidentiality_arrayCopy :
   (confidentiality_api arrayCopy_precond
                        arrayCopy_postcond
                        arrayCopy_confidentiality).
```

où la précondition et la postcondition de la méthode sont décrites page 84. Comme expliqué page 37, cette méthode devrait imposer que `src` et `dest` soient accessibles depuis le contexte appelant. En effet, si `src` appartient à un autre contexte, cela correspond à une lecture illégale. D'autre part, si `dest` appartient à un autre contexte, cela correspond à une modification illégale. Étant donné que ces conditions ne sont pas spécifiées par Sun, elles ne sont pas contenues dans la postcondition. Par conséquent, la preuve de confidentialité nécessite une hypothèse supplémentaire sur le fait que `src` est accessible à partir du contexte appelant (et la preuve d'intégrité va, de même, nécessiter une hypothèse supplémentaire sur le fait que `dest` est accessible à partir du contexte appelant). Cette hypothèse supplémentaire est donnée par la variable `arrayCopy_confidentiality`, qui prend en argument l'état initial, où les arguments de la méthode sont contenus dans la pile d'opérandes et le contexte appelant dans le statut d'exécution.

Remarque : L'instanciation de la variable `confidentiality_condition` pour chaque méthode de l'API complète la spécification de la méthode d'un point de vue sécuritaire. Par conséquent, l'ensemble de ces conditions pour toutes les méthodes de l'API spécifie un mécanisme de sécurité pour l'API, suffisant pour assurer la propriété de confidentialité. Ceci est une application importante de notre travail, qui sera discutée en Section 3.3.

3.2.4.2 Preuve de l'integrité

La preuve de la propriété d'intégrité suit le même schéma, sauf qu'une seule exécution est nécessaire :

$$\begin{array}{ccccccccccc} c_i & & & & & & & & & & r_i \\ \Downarrow & & & & & & & & & & \Uparrow \\ s_1 & \xrightarrow{\mathsf{CM}} & s_2 & \xrightarrow{\mathsf{VM}} & s_3 & \xrightarrow{\mathsf{API(m)}} & \ldots & \xrightarrow{\mathsf{VM}} & s_{n-1} & \xrightarrow{\mathsf{CM}} & s_n \\ \wr & & \wr & & \wr & & & & \wr & & \wr \\ \mathcal{F}_C(s_1) & = & \mathcal{F}_C(s_2) & = & \mathcal{F}_C(s_3) & = \ldots = & & & \mathcal{F}_C(s_{n-1}) & = & \mathcal{F}_C(s_n) \end{array}$$

L'architecture est vraiment similaire à celle de la preuve de confidentialité , utilisant des lemmes intermédiaires pour chaque type de transition. Étant donné qu'une seule exécution est considérée, la preuve totale est beaucoup plus petite ($\sim$ 2 500 lignes de preuves Coq).

3.3 Les conditions d'isolation

Une valeur ajoutée majeure du travail présenté dans ce chapitre est l'ensemble des hypothèses mises en évidence pendant le développement de la preuve. En plus des hypothèses apparaissant dans les énoncés formels de confidentialité et d'intégrité (Définition 10 et Définition 11), le développement de la preuve a nécessité des propriétés qui n'étaient pas prouvables au vu du niveau de détail de la modélisation.

En effet, la modélisation de la technologie Java Card du projet FORMAVIE a pour but de suivre exactement les spécifications de Sun. Or, certaines propriétés de la plate-forme ne sont pas entièrement, ou pas du tout, spécifiées par Sun, soit pour laisser le choix d'implémentation, soit par manque de précision (la propriété est supposée implicite), soit par oubli réel, représentant alors un flux potentiel d'information. Dans tous les cas, les aspects non spécifiés par Sun ne sont pas instanciés dans le modèle. Par conséquent, toutes les propriétés sur des aspects non instanciés dans le modèle, et qui sont nécessaires à la preuve du théorème d'isolation, doivent être supposées.

Plus précisément, un grand nombre d'hypothèses concerne des précisions de la spécification informelle de Sun et des conditions triviales sur le choix d'implémentation des aspects non spécifiés par Sun. Ces hypothèses sont donc indépendantes de la propriété à vérifier et complètent le modèle formel. Elles seront présentées dans la Section 3.3.1. Puis quelques hypothèses représentent réellement les *conditions d'isolation* des applets et seront présentées dans les Sections 3.3.2, 3.3.3 et 3.3.4.

Toutes ces propriétés doivent être considérées comme des hypothèses additionnelles des théorèmes, et forment un ensemble de conditions qui sont *suffisantes* pour assurer les propriétés d'isolation. Chacune de ces conditions représente un flux d'information potentiel, puisque si elle n'est pas vérifiée, l'isolation n'est plus respectée. De plus, étant donné que cette liste est suffisante pour établir une preuve complète, il ne peut y avoir de flux d'information qui ne repose pas sur l'une d'elles. Par conséquent, cette méthode fournit une *liste exhaustive des flux d'informations potentiels*. Cette liste permet de compléter la spécification informelle de Sun par les propriétés nécessaires à l'isolation des applets. Elle constitue un bénéfice majeur pour la vérification *au niveau de l'implémentation*, comme il sera expliqué dans la Section 3.3.5, car elle contient les conditions qu'il suffit de vérifier sur le code pour assurer l'isolation des applets (et qui ne peuvent pas être prouvées au niveau du modèle, du fait de son niveau d'abstraction).

3.3.1 Hypothèses qui complètent le modèle et les spécifications de Sun

Les opérations qui ne sont pas entièrement spécifiées par Sun sont modélisées en Coq par des *relations* (plutôt que par des fonctions, qui doivent être totales en Coq). Par exemple, une relation R qui définit un terme y à partir d'un terme x peut ne spécifier que certains composants de y (ceux spécifiés par Sun). Par conséquent, les propriétés nécessaires à la preuve qui portent sur les autres composants du résultat doivent être supposées.

Par exemple, une particularité de Java Card est d'autoriser le chargement de *package* et d'applet après la mise en circulation de la carte. Le chargement d'un nouveau package est modélisé par une relation entre deux états s et s', où s est l'état avant le chargement et s' l'état après le chargement. Dans cette relation, seuls le tas, l'image des champs statiques et l'ensemble des *packages* déjà chargés de s' sont spécifiés. En pratique, les autres composants sont inchangés, mais cela n'est pas précisé dans la spécification et n'apparaît donc pas dans la modélisation. Nous devons donc le supposer. Nous aurons, par exemple, l'hypothèse suivante :

Hypothèse 1 *Le chargement d'un nouveau package ne modifie pas le journal des transitions de l'état de la machine virtuelle.*

Si une opération n'est pas du tout spécifiée par Sun, dans le but de laisser le choix d'implémentation, elle est laissée en paramètre du modèle Coq et toute propriété sur le résultat de cette opération doit être supposée. La majorité de ces propriétés est triviale dès lors que l'implémenta-

tion est donnée, mais doit être supposée lorsqu'il s'agit d'un paramètre non instancié du modèle formel.

Par exemple, un grand nombre de propriétés nécessaires aux lemmes concerne la *fonctionnalité* de relations. Une relation R est fonctionnelle si ces valeurs de sortie sont définies de façon unique à partir des valeurs d'entrée : si $R(x, y_1)$ et $R(x, y_2)$ alors $y_1 = y_2$. Cette propriété est triviale lorsque la relation est effectivement fonctionnelle, mais ne peut être prouvée si la relation est un paramètre Coq `Parameter R : type_x -> type_y -> Prop`. Dans ce cas, la fonctionnalité de R doit être *supposée*.

Par exemple, le codage d'une valeur entière i, stockée à une adresse *loc*, par une paire de mots w_1 et w_2 est modélisé par une relation non instanciée, car le codage n'est pas spécifié par Sun. Au niveau de l'implémentation, le codage peut être de type "Little Endian" ou "Big Endian" (voir les Conventions, page 1). Pour notre preuve de confidentialité, nous avons juste besoin que ce codage soit fonctionnel. Nous avons donc l'hypothèse suivante :

Hypothèse 2 *Si l'entier* i*, stocké à l'adresse* loc*, peut être représenté soit par les mots* w_1 *et* w_2 *soit par les mots* w_1' *et* w_2'*, alors* $w_1' = w_1$ *et* $w_2' = w_2$.

Les hypothèses de fonctionnalité de relation, qui représentent en fait une spécification de la relation, et les hypothèses qui précisent des aspects triviaux mais non spécifiés par Sun, sont nombreuses et permettent de compléter le modèle formel, mais sont indépendantes de la propriété d'isolation.

3.3.2 Conditions d'isolation pour l'API

Toutes les méthodes de l'API (celles qui sont natives et celles qui sont écrites en Java) sont modélisées par leur précondition et leur postcondition, dérivées de la spécification de l'API définie par Sun. Lors de notre preuve de la propriété d'isolation, certaines conditions ont dû être ajoutées car elles n'apparaissaient pas dans la postcondition spécifiée par Sun. Nous avons déjà cité la méthode native `arrayCopy`, dont la spécification a été complétée de manière à imposer que les tableaux source et destination de la copie soient tous les deux accessibles depuis le contexte appelant. Un autre exemple est celui de la méthode

```
boolean equals(byte[] bArray, short offset, byte length)
```

de la classe `AID`, qui vérifie si les octets donnés dans `bArray` (à partir de l'index `offset` de `bArray` et de taille `length`) sont les mêmes que ceux de l'instance d'objet `this` de l'AID. Dans la spécification de l'API de la plate-forme Java Card 2.1, il n'est pas imposé que `bArray` soit accessible depuis le contexte appelant, ce qui viole la propriété de confidentialité. Par conséquent, cette hypothèse a dû être ajoutée pour prouver la confidentialité sur les modèles de la plate-forme 2.1. De façon indépendante à notre travail, cette condition a été ajoutée dans la spécification de Java Card 2.2. Prouver la même propriété pour les modèles de cette dernière plate-forme ne nécessite donc plus cette hypothèse sécuritaire supplémentaire (elle est comprise dans la spécification).

Nos travaux permettent donc de compléter la spécification Sun de l'API du point de vue sécuritaire, de manière à assurer la propriété d'isolation.

3.3.3 Méthodes natives

Les méthodes natives étant des fonctions de bas niveau qui ne sont pas écrites en Java, elles ne sont pas décrites dans le modèle. Par conséquent, les propriétés liées à des telles méthodes

doivent être supposées. Dans le cas de la preuve de la propriété d'isolation, trois hypothèses ont été nécessaires : la non-interférence, l'intégrité et la stabilité de la cohérence vis-à-vis des références (rappelons que c'est un invariant nécessaire pour toute transition dans le théorème de confidentialité, voir Section 3.2.4.1) :

Hypothèse 3 *L'exécution d'une méthode native à partir de deux états équivalents à un contexte près mène à deux états équivalents au même contexte près.*

Hypothèse 4 *Pour tout contexte C différent du contexte JCRE, l'exécution d'une méthode native à partir d'un état s, tel que l'applet sélectionnée dans s n'appartient pas à C, mène à un état s' tel que $\mathcal{F}_C(s')=\mathcal{F}_C(s)$ (où $\mathcal{F}_C(s)$ représente le contenu, dans l'état s, de tous les champs de tous les objets appartenant à C).*

Hypothèse 5 *L'exécution d'une méthode native à partir d'un états cohérent vis-à-vis des références mène à un état cohérent vis-à-vis des références.*

3.3.4 Initialisation de la table des variables locales

Finalement, certaines hypothèses représentent des faiblesses techniques de la spécification Sun vis-à-vis de la confidentialité ou de l'intégrité. Elles révèlent de potentiels flux d'information non autorisés. Dans le cadre de notre preuve, deux hypothèses ont été nécessaires, concernant respectivement la non-interférence et la stabilité de la cohérence, mais cette fois-ci lors de l'initialisation de la table des variables locales.

En effet, lorsqu'une méthode `m` est invoquée, l'état `s` de la machine virtuelle doit être modifié. En particulier, une nouvelle *frame* doit être ajoutée au sommet de la pile de *frame* de `s`. Cette nouvelle *frame* est construite de la façon suivante : le contexte est celui dans lequel la méthode est exécutée, la pile d'opérande est vide, le compteur de programme pointe sur le premier *bytecode* de la méthode et la table des variables locales contient les arguments de la méthode. Le problème vient du fait que la spécification n'impose pas aux autres valeurs de la table des variables locales d'être initialisées. Leurs valeurs ne sont donc pas connues. Ces variables locales peuvent donc être la source d'un flux d'information non spécifié. La preuve a donc nécessité une hypothèse interdisant un tel flux d'information.

Décrivons la construction de la nouvelle table de variables locales `newlocals`. Notons `stck` la pile d'opérandes de l'état avant l'invocation de `m` (c'est-à-dire la pile contenant les arguments de `m`), `nbargs` le nombre d'arguments de `m` et enfin `nblocs` le nombre de variables locales, hormis les arguments, nécessaires à l'exécution de `m`. Pour construire la table `newlocals`, une table de taille (`nblocs+nbargs`) est allouée, avec les arguments de `m` (trouvés dans `stck`) au début de la table. *Mais rien n'est spécifié pour le reste de la table.* En d'autres termes, les `nblocs` autres cases sont allouées, mais leur contenu n'est pas spécifié.

Or, pour prouver que la cohérence vis-à-vis des références est stable par transition, nous devons prouver, en particulier, que l'opération de chargement des arguments d'une nouvelle méthode préserve également la cohérence. Cela signifie que la nouvelle table de variables locales doit être cohérente. Mais cette propriété ne peut pas être prouvée puisque certaines nouvelles variables locales ne sont pas spécifiées. Nous avons donc besoin de l'hypothèse suivante :

Hypothèse 6 *Si la pile* `stck`*, qui contient les arguments de la méthode* `m`*, est cohérente vis-à-vis des références, alors la nouvelle table des variables locales* `newlocals`*, construite à partir de* `stck`*, est aussi cohérente.*

De même, la preuve de la non-interférence ne pourra être prouvée sans l'hypothèse suivante :

Hypothèse 7 *Si* `newlocals` *est construite à partir de la pile* `stck` *et* `newlocals'` *à partir de la pile* `stck'`*, où les piles* `stck` *et* `stck'` *sont équivalentes à un contexte près, alors* `newlocals` *et* `newlocals'` *sont équivalentes au même contexte près.*

Cette dernière hypothèse assure qu'aucun flux d'information ne pourra transiter par des variables locales non initialisées.

3.3.5 Contributions et vérification au niveau de l'implémentation

Ce travail met en évidence certaines faiblesses de la spécification de Java Card vis-à-vis de le confidentialité et de l'intégrité, et propose un moyen de la compléter. En particulier, hormis les hypothèses permettant de préciser le modèle, la preuve a révélé les propriétés à ajouter à la spécification de l'API pour assurer l'isolation, ainsi que cinq *conditions d'isolation* concernant :

- la non-interférence, l'intégrité et la stabilité de la cohérence pour les méthodes natives ;
- la non-interférence et la stabilité de la cohérence pour l'initialisation des variables locales.

Ces propriétés n'étant pas vérifiables au niveau du modèle, du fait de son niveau d'abstraction, l'étape suivante est de procéder à une vérification de ces faiblesses éventuelles au niveau de l'implémentation. Pour cela, nous suivons la chaîne développement en cascade – spécification, design et implémentation – avec des étapes de raffinement entre les niveaux. De cette façon, nous identifions les parties de l'implémentation qui est impactée par les hypothèses, et nous vérifions que ces hypothèses sont effectivement satisfaites par l'implémentation.

Notons ici que dans le modèle de Java Card utilisé, la machine virtuelle est modélisée comme une machine *défensive*, c'est-à-dire où tous les contrôles ont été effectués, en particulier concernant le typage. Au niveau de l'implémentation, l'équivalent de la machine virtuelle défensive est composé, d'une part, du vérificateur de *bytecode* et, d'autre part, de l'interpréteur embarqué. Le vérificateur de *bytecode* est en charge de contrôles statiques sur les applets à exécuter, tels que la vérification du typage. L'interpréteur, quant à lui, effectue des contrôles dynamiques lors de l'interprétation des applets (comme le contrôle d'accès entre différents contextes), mais pas de vérification de type puisque cela a déjà été fait. Dans [12] il a été prouvé qu'une machine virtuelle défensive coïncide avec un interpréteur où les packages ont passé la vérification de *bytecode*. Cela prouve, en particulier, que le système composé du vérificateur de *bytecode* et de la machine virtuelle, respecte les spécifications de Sun.

Nous avons vu précédemment que certaines hypothèses de nos travaux reflètent des faiblesses de la spécification de Sun. Corriger ces faiblesses au niveau de l'implémentation signifie de vérifier que le système formé du vérificateur de *bytecode* et de l'interpréteur vérifie ces hypothèses.

A titre d'exemple, regardons le mécanisme implémenté en pratique dans le cas de l'initialisation de la table de variables locales. En fait, la pile de *frames* est stockée dans un tableau, où les frames sont stockées de façon successive : pour chaque *frame*, la table des variables locales est d'abord stockée sous la forme d'une tableau, puis vient un en-tête contenant le contexte et le compteur de programme, et enfin vient la pile des opérandes (voir la Figure 3.1). Lorsqu'une nouvelle méthode est invoquée, une nouvelle *frame* doit être ajoutée, à la suite de l'ancienne, dans le tableau. Or, la nouvelle table des variables locales doit contenir les arguments de la méthode, stockés dans la pile d'opérandes de l'ancienne frame. Au lieu de recopier ces données, la table des variables locales de la nouvelle *frame* commence au début des arguments de la méthode dans la pile d'opérandes de l'ancienne *frame*.

La spécification de Sun n'impose pas que les variables locales restantes (celles qui ne sont pas utilisées par les arguments de la méthode) soient effacées. Par conséquent, si la *frame* d'une

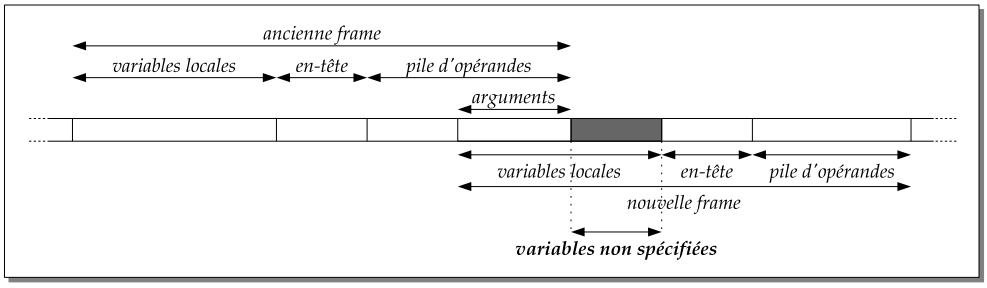


FIG. 3.1 – Le stockage d'une Pile de *Frames* en Pratique

autre méthode avait utilisé ces emplacements précédemment, certaines informations sont restées à ces index et sont visibles par la nouvelle méthode. Cela constitue un flux d'information non autorisé.

Une implémentation stricte de la spécification de Sun n'assure donc pas l'isolation des applets, puisque les Hypothèses 6 et 7 ne pourront pas être prouvées. Il a deux méthodes pour résoudre ce problème.

- Premièrement, on peut modifier l'implémentation de l'interpréteur embarqué de manière à ce que la table des variable locales soit effacée à chaque fois qu'une nouvelle *frame* est ajoutée. Cette modification assurerait les Hypothèses 6 et 7.
- Une autre façon de faire est d'interdire toute lecture d'une variable locale non initialisée, ce qui correspond à une spécification "défensive". Au niveau de l'implémentation, cela revient à dire que ce problème est du ressort du vérificateur de *bytecode*, qui doit assurer qu'aucun accès en lecture d'une variable locale n'a lieu avant que cette variable n'ait été initialisée.

3.4 Conclusion

Le travail présenté dans ce chapitre s'inscrit dans un contexte général visant à prouver que la technologie Java Card propose un environnement sécurisé pour le déploiement d'applications embarquées dans les cartes à puce. En particulier, nous nous sommes intéressés à prouver formellement que cette plate-forme multi-applicative et ouverte, pouvant héberger plusieurs applications différentes, assure qu'une application ne peut obtenir illégalement aucune information et ne modifier illégalement aucune donnée d'une autre application.

Ces propriétés de confidentialité et d'intégrité des données d'une applet ont été formalisées, puis prouvées, sur un modèle formel de la plate-forme, à l'aide de l'assistant de preuve Coq. Les étapes successives de formalisation, à partir d'une notion informelle, jusqu'à un énoncé formel de théorème en Coq, soulignent la difficulté d'énoncer une propriété, apparemment simple, de façon correcte et sans ambiguïté possible, tout en mettant en évidence les avantages d'une telle définition formelle. Quant à la preuve formelle, elle a permis d'identifier un certain nombre de conditions sur lesquelles reposent la confidentialité et l'intégrité, qui permettent, d'une part, de compléter les spécifications de l'API d'un point de vue sécuritaire, et d'autre part d'obtenir les conditions à vérifier au niveau de l'implémentation.

Ce dernier point confirme un problème évoqué dans la Section 2.1.6 concernant le lien entre le modèle et l'implémentation. Lorsqu'une propriété a été vérifiée formellement sur un modèle, il reste à prouver qu'elle est également vérifiée au niveau de l'implémentation, ce qui revient à prouver que le code source implémente correctement le modèle. Ces considérations ont motivé

nos travaux de recherche d'une méthodologie permettant de prouver des propriétés à partir du code source d'un système. Le chapitre suivant présente la vérification de propriétés fonctionnelle à partir du code source et une méthodologie de vérification de propriétés de haut niveau sera définie dans le Chapitre 5.

Chapitre 4

Vérification fonctionnelle de programmes C embarqués, avec Caduceus

Résumé

Ce chapitre présente l'utilisation de l'outil Caduceus (voir [53, 52]) pour la vérification fonctionnelle de code C de bas niveau embarqué sur une carte à puce. Nous décrirons tout d'abord l'outil et nous proposerons une méthode générale d'utilisation. Les limitations de l'outil seront ensuite mise en évidence. Une analyse de la formalisation de la sémantique du langage C permettra alors de proposer des améliorations du modèle mémoire de Caduceus, où la mémoire est divisée en emplacements disjoints (inspirée d'une approche de Burstall et Bornat), de manière à gérer les constructions de bas niveau du langage telles que les *unions* et les *casts*. Une extension de l'outil sera également proposée pour permettre de spécifier et de vérifier les propriétés d'une fonction en cas d'arrachage de la carte.

Sommaire

4.1 Introduction

À l'issue de nos travaux de modélisation et de preuve formelle de propriétés sécuritaires de la plate-forme Java Card, nous nous sommes intéressés à la vérification de propriétés du système d'exploitation. Nous avons donc cherché une méthodologie de vérification de programmes de bas niveau, écrits en langage C et optimisés. Le choix de la méthode dépend du type de propriétés que l'on veut prouver. En effet, comme nous l'avons vu dans la Section 2.1.5, un outil de vérification de programme au niveau du code source est préconisé pour la vérification de propriétés fonctionnelles et locales du programme, alors que la vérification de propriétés globales, de combinaison de fonctions, nécessite un modèle sous la forme d'un *système de transitions*, où chaque fonction est vue comme une transition entre l'état global du programme avant son exécution et l'état global après l'exécution.

Le cas d'étude qui a été choisi pour nos travaux sur le système d'exploitation est celui d'un module de gestion de mémoire Flash (qui a été décrit dans la Section 1.3.4). Comme nous l'avons vu, cette nouvelle technologie de mémoire nécessite le développement de nouveaux algorithmes de gestion. L'objectif est donc d'identifier les propriétés vérifiées par ces nouveaux algorithmes. Dans un premier temps, le but est de s'assurer que chacune des fonctions de gestion de la mémoire implémente correctement ce pour quoi elle a été conçue. Puis, dans un deuxième temps, on peut également chercher à établir des propriétés plus globales ou des propriétés sécuritaires pour le module.

Le présent chapitre se concentre sur la vérification fonctionnelle de programmes C de bas niveau embarqués dans une carte à puce. La modélisation d'un système de transitions pour une vérification sécuritaire sera décrite dans le chapitre suivant.

La vérification fonctionnelle consiste, d'une part, à formaliser la spécification de chaque fonction, i.e. à exprimer, dans un langage de spécification formel, les propriétés attendues pour chaque fonction, et, d'autre part, à prouver la correction du code source de chaque fonction vis-à-vis de sa spécification formelle, grâce à un outil de vérification de programme. Cet outil doit permettre la vérification de programmes C à partir du code source et doit pouvoir être modifié pour être adapté à du code embarqué sur une carte à puce. Notre choix s'est donc porté sur l'outil *Caduceus* (voir [52, 53]). Comme nous le verrons dans le chapitre suivant, cet outil permet également une extraction automatique du système de transitions pour la vérification sécuritaire, répondant ainsi au problème majeur de notre étude sur la plate-forme Java Card, puisque le lien entre le modèle et le code est à présent formel.

Ce chapitre commence, dans la Section 4.2, par une description détaillée de l'outil Caduceus, ainsi qu'une méthode générale d'utilisation de l'outil pour la vérification fonctionnelle de programme. Dans un deuxième temps, les limitations de l'outil en termes de constructions C gérées seront mises en évidence, dans la Section 4.3, et des solutions seront proposées pour la gestion

de ces constructions. En particulier, une interprétation correcte des structures est présentée dans la Section 4.4. Puis l'adaptation du modèle mémoire de Caduceus (qui s'inspire d'une approche de Burstall et Bornat) pour la gestion des *unions* et des *casts* est décrite dans la Section 4.5. Enfin, nous proposerons une extension de l'outil de manière à pouvoir spécifier et prouver le comportement du programme en cas d'interruption, de manière à représenter l'*arrachage* de la carte à puce.

4.2 Spécification et preuve de correction avec Caduceus

4.2.1 L'outil Caduceus

Caduceus est un outil de vérification de programmes C, et plus précisément de programmes répondant à la norme ANSI-C (voir Section 2.2.3.2). Il permet à la fois de vérifier l'absence de *menaces* (telles que l'accès à l'adresse d'un pointeur nul, appelé *déréferencement*, ou l'accès en dehors des bornes d'un tableau) et de prouver des propriétés fonctionnelles d'un programme, spécifiées sous la forme d'annotations insérées dans le code source. A partir d'un programme annoté, Caduceus génère des *conditions de vérification*, à savoir des propriétés logiques du premier ordre dont la validité implique la correction du programme vis-à-vis de l'absence de menaces et de sa spécification annotée. Ces conditions de vérification peuvent être prouvées dans divers systèmes de preuves.

4.2.1.1 Annotations

D'un point de vue pratique, une annotation est insérée dans le code source sous la forme d'un commentaire du langage de programmation (de manière à être ignorée par le compilateur du langage), mais avec un signe distinctif (ici `/*@...*/`) permettant à l'outil de vérification de les identifier. Les annotations sont utilisées pour spécifier les propriétés fonctionnelles de chaque fonction du programme, dans un langage de spécification donné. Le langage de spécification de l'outil Caduceus est fortement inspiré de *JML* (*Java Modeling Language*, langage de spécification pour programmes Java, voir [86]). Une annotation dans ce langage est une formule du premier ordre, construite à partir d'expressions C sans effet de bord, de variables de prédicats et de mots clé spécifiques. Notons qu'il y a deux sortes d'annotations : celles qui constituent les spécifications des fonctions du programme et celles qui sont nécessaires à la preuve des conditions de vérification.

Commençons par décrire les annotations de spécification. Suivant un style à la Hoare, la spécification d'une fonction est constituée de sa précondition (correspondant à la clause `requires` de l'annotation) et de sa postcondition (correspondant à la clause `ensures`). Dans la postcondition, le mot clé `\result` représente le résultat de la fonction et `\old(x)` la valeur d'une variable `x` avant l'exécution de la fonction. La spécification d'une fonction peut également décrire les effets de bords sous la forme de la liste des variables potentiellement modifiées par la fonction (dans la clause `assigns`). Le mot clé `\nothing` peut être utilisé pour indiquer que la fonction n'a aucun effet de bord. Des invariants globaux peuvent également être décrits à l'aide d'annotations, afin d'alléger les préconditions de fonctions, lorsqu'une propriété est vraie dans tout le programme. De la même façon, des fonctions logiques (`logic`) ou des prédicats logiques (`predicate`) peuvent être utilisés dans la spécification des fonctions. Ils sont soit définis dans le langage de spécification, soit simplement déclarés et instanciés uniquement dans la logique formelle de l'outil. L'exemple suivant résume les principaux éléments qui peuvent composer la spécification d'une fonction :

```
/*@ predicate is_positive(int p) { p>=0 } */
int n;
/*@ invariant n_is_always_positive : is_positive(n) */
/*@ requires is_positive(i)
    assigns n
    ensures is_positive(\result) && n==\old(n)+1 */
int f(int i){n++; return i+n;}
```

La Section 4.2.2 décrit plus en détail chaque composant d'une spécification de fonction et propose une méthode générale d'utilisation.

Par ailleurs, lorsqu'il s'agit de prouver la correction du programme vis-à-vis de la spécification annotée, certaines preuves nécessitent des annotations supplémentaires de la part de l'utilisateur, qui sont essentiellement les annotations de boucles. Ces annotations sont nécessaires au calcul de plus faible précondition, comme nous allons l'expliquer dans la Section 4.2.1.3. Les annotations de boucle comprennent les variants de boucle (assurant la terminaison), les invariants de boucle et les effets de bord de boucle. Leurs définitions et leurs utilisations en pratique seront décrites dans la Section 4.2.3.

4.2.1.2 Traduction

Modèle mémoire. Caduceus est en fait un outil construit sur la base de l'outil de vérification *Why* (voir [51, 50]). Why est un outil de vérification de programmes purement fonctionnels avec références, utilisé comme moteur principal de vérification pour les outils Caduceus et Krakatoa (voir [91, 90]), comme le montre la Figure 4.1. En d'autres termes, les outils Caduceus et Krakatoa

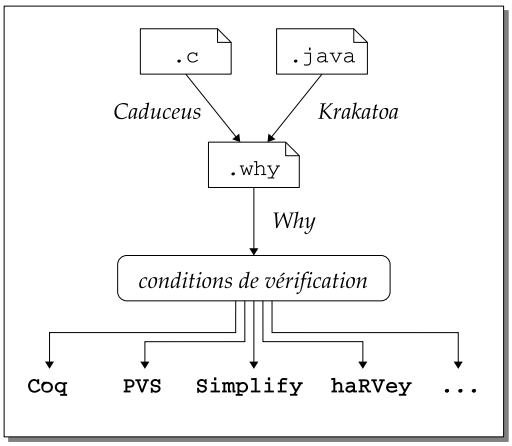


FIG. 4.1 – Architecture des outils de vérification Why, Caduceus et Krakatoa

traduisent les programmes C et Java respectivement, dans le langage d'entrée de Why, suivant un modèle mémoire choisi pour le langage. Le langage d'entrée de Why est un langage dédié à la vérification, où les types sont uniquement des types fonctionnels purs et des références sur ces types, sans aucune notion de tableaux, structures ou pointeurs. Par conséquent, la traduction effectuée par Caduceus nécessite un modèle mémoire du langage C, de façon à représenter les structures de données du langage C et l'état courant d'un programme C donné.

Au lieu de définir la mémoire comme un grand tableau d'octets, Caduceus suit une approche de Burstall et Bornat (voir [22, 29]), où la mémoire est divisée en emplacements disjoints, ou *adresses de base*, correspondant à aux champs de structure et aux blocs alloués par des pointeurs dans le programme. Autrement dit, les champs d'une structure sont considérés comme alloués

dans des emplacements disjoints. De même, deux pointeurs appartiennent au même emplacement s'ils pointent dans le même bloc mémoire alloué, ce qui signifie qu'il y a eu une affectation entre les deux pointeurs (ils ont été *aliasés*). Une arithmétique de pointeur est possible à l'intérieur de l'emplacement, mais le pointeur ne pourra pointer vers un autre emplacement mémoire que par une affectation à un autre pointeur (voir Figure 4.2).

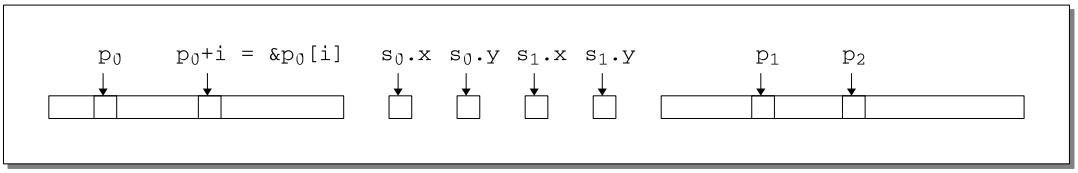


FIG. 4.2 – Idée du modèle mémoire de Caduceus pour les pointeurs et les structures

Cette séparation assure "gratuitement" qu'une modification dans un emplacement mémoire n'affecte pas les autres emplacements. Par exemple, lorsqu'un champ d'une structure est modifié, il ne sera pas nécessaire de *prouver* que les autres champs ne sont pas touchés (contrairement au cas où la mémoire est modélisée par un grand tableau d'octets, où une preuve est nécessaire). De la même manière, si deux pointeurs appartiennent à deux emplacements disjoints, il ne sera pas nécessaire de prouver que la modification de l'un ne modifie pas l'autre.

Dans ce modèle mémoire, il n'y a que deux types de base : les types numériques (les entiers et les flottants) et un nouveau type `pointer`. Une valeur `p` de type `pointer` est soit le pointeur `null` soit une paire `(base_addr(p),offset(p))` composée de l'adresse de base de l'emplacement mémoire où le pointeur a été alloué et l'index où pointe `p` à l'intérieur de ce bloc (voir Figure 4.3). Les notions de tableaux C et de pointeurs C sont confondues dans le modèle et sont toutes deux

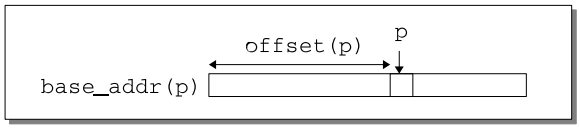


FIG. 4.3 – Représentation du type `pointer` du modèle mémoire de Caduceus

associées au type `pointer`. Quant au type structure de C, il est également vu comme un pointeur, dont l'adresse de base peut être associée aux différents segments de mémoire correspondant à ces champs.

A partir de là, l'état mémoire d'un programme C est représenté par un ensemble de variables globales Why, englobant, de façon structurée, tous les emplacements mémoires du programme. En particulier, une variable `intP` sera utilisée pour représenter l'ensemble des segments mémoires contenant un pointeur d'entiers (ou un tableau d'entiers, puisque les deux notions sont confondues dans le modèle) à un état donné du programme. Cette variable peut donc être vue comme associant à chaque adresse de base d'un pointeur, le segment mémoire où a été alloué le pointeur. Une autre variable `intPP` représentera les pointeurs sur pointeurs d'entiers, et ainsi de suite. De même, une variable sera utilisée pour chaque champ de structure, associant à toute variable du programme dont le type correspond à cette structure, le segment mémoire contenant la valeur du champ en question. A titre d'illustration, les pointeurs et les structures représentés dans la Figure 4.2 seront modélisés grâce à trois variables Why `intP`, `x` et `y`, comme représenté dans la Figure 4.4.

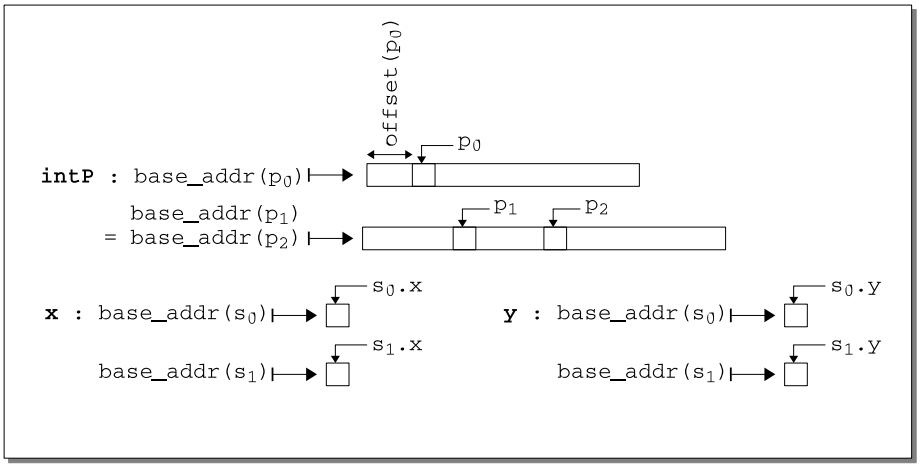


FIG. 4.4 – Exemples de variables Why pour représenter les segments mémoires à un instant donné du programme

Une variable supplémentaire, notée `alloc`, représente la *table d'allocation* à un instant donné. Cette variable indique quelles sont les adresses qui sont allouées et quelle est la taille du bloc sur lequel elles pointent. Par conséquent, un état donné du programme est généralement représenté dans le modèle mémoire de Caduceus comme indiqué dans la Figure 4.5. Notons qu'un tableau de structure est interprété par une structure où chaque champ est un tableau.

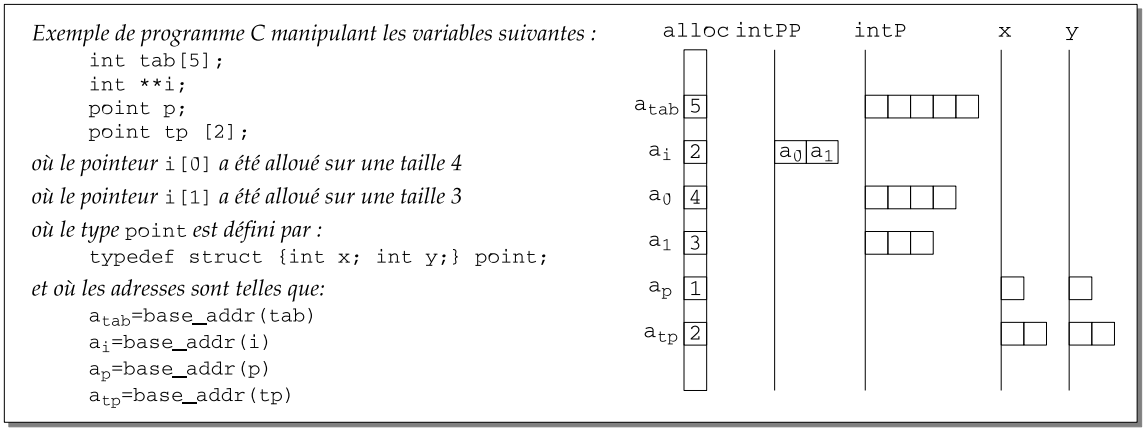


FIG. 4.5 – Représentation d'un état mémoire dans le modèle de Caduceus

Prédicats de validité. Le modèle mémoire défini par Caduceus est associé à une théorie, permettant de manipuler et de raisonner sur les composants de ce modèle (voir [52, 53]). Tout d'abord, un ensemble de prédicats permettent de définir la *validité* des pointeurs, à savoir la condition pour qu'ils puissent être déréférencés (i.e. pour que leur adresse soit accédée). Un pointeur peut être déréférencé s'il n'est pas le pointeur `null` et s'il pointe dans un segment alloué :

```
valid(alloc,p)  ≡  p≠null ∧ 0≤offset(p)<block_length(alloc,p)
```

A partir de ce prédicat, la validité du pointeur `p+i` et la validité de tout un intervalle `p[i...j]` peuvent être définies :

```
valid_index(alloc,p,i)    ≡  p≠null ∧ 0≤offset(p)+i<block_length(alloc,p)
valid_range(alloc,p,i,j)  ≡  p≠null ∧ 0≤offset(p)+i ∧
                             i≤j ∧ offset(p)+j<block_length(alloc,p)
```

Notons que la taille du bloc ne dépend en fait que de l'adresse de base du pointeur, puisque plusieurs pointeurs peuvent appartenir au même bloc.

Fonctions d'accès et de modification. Par ailleurs, un ensemble de fonctions permettent la gestion des variables Why et la modélisation des opérations du langage C. Tout d'abord, une fonction d'accès `acc(m,p)` retourne la valeur contenue à l'index `offset(p)` dans le tableau associé à `base_addr(p)` par l'état mémoire `m`. Cette fonction est utilisée pour modéliser à la fois l'accès à la valeur pointée par un pointeur (`*p` est représenté par `acc(intP,p)` si `p` est un pointeur sur entier) et l'accès à un champ d'une structure (`s.x` est représenté par `acc(x,s)`). Une fonction de mise à jour `upd(m,p,v)` retourne, quant à elle, un nouvel état `m'`, où la valeur "pointée" par `p` devient `v`. Enfin, une fonction d'arithmétique de pointeur `shift(p,i)` permet de représenter l'expression `p+i` ou `p[i]` du langage C. Plus précisément, `shift(p,i)` est de type `pointer` et vaut `(base_addr(p),offset(p)+i)`.

Ces fonctions sont en réalité définies de façon *axiomatique* dans Why. Par exemple, les axiomes suivants sont définis :

```
acc(upd(t,i,v),i) = v
i <> j -> acc(upd(t,i,v),j) = acc(t,j)
shift(p,0) = p
shift(shift(p,i),j) = shift(p,i+j)
...
```

La clause `assigns`. Pour finir, notons que la clause `assigns` est modélisée par le prédicat Why suivant :

```
assigns(alloc,m1,m2,loc)  ≡
      ∀ p:pointer, valid(alloc,p) ∧ unchanged(p,loc) ⇒ acc(m2,p) = acc(m1,p)
```

qui indique que seules les variables contenues dans la *location* `loc` ont été modifiées entre les états mémoires `m1` et `m2`, où une location représente un ensemble de pointeurs (par exemple un seul pointeur `pointer_loc(p)`, un sous-ensemble d'un tableau `range_loc(p,i,j)`, etc). Le prédicat `unchanged` est défini de manière axiomatique, par exemple par :

```
∀ p1:pointer, ∀ p2:pointer, p1≠p2 ⇒ unchanged(p1,(pointer_loc p2)).
```

Puis un ensemble de lemmes sont fournis (et seront complétés lors de nos travaux, comme il sera indiqué à la fin de la Section 4.2.3) pour une meilleure efficacité lors des preuves des obligations, comme par exemple la transitivité du prédicat `assigns` :

```
∀ alloc, ∀ m1, ∀ m2, ∀ m3, ∀ loc,
assigns(alloc,m1,m2,loc) ∧ assigns(alloc,m2,m3,loc) ⇒ assigns(alloc,m1,m3,loc)
```

Calcul d'effets. Un dernier point qui doit être pris en compte dans la traduction effectuée par Caduceus est que les spécifications des programmes dans le langage d'entrée de Why doivent expliciter les effets de bords et les variables lues par les fonctions. En effet, la spécification d'une fonction Why annotée a la forme suivante :

```
parameter f_parameter :
 a1:t1 -> ... -> an:tn -> { Pre } T reads v1,...,vm writes w1,...,wp { Post }
```

Caduceus calcule donc, par une analyse statique, l'ensemble des variables Why lues et modifiées par chacune des fonctions du programme C.

En conclusion, la traduction de Caduceus :

- définit les variables globales Why qui modélisent les états mémoires C,
- calcule statiquement les variables globales Why lues ou modifiées,
- et enfin traduit la sémantique des constructions C en instructions Why, par des affectations de ces variables globales et l'ajout d'annotations (comme l'ajout automatique de prédicats de validité).

4.2.1.3 Vérification

Une fois que le programme C annoté a été traduit dans le langage d'entrée de Why, le procédé de vérification est assuré par l'outil Why. Ce procédé consiste à traduire une formule, qui est sous la forme d'un triplet de Hoare $\{P\}e\{Q\}$, en une preuve que si les valeurs des variables du programme e vérifient la précondition P alors les valeurs de ces variables après l'exécution de e vérifient la postcondition Q. Pour cela, l'outil Why fait deux choses : il effectue un *calcul de plus faible précondition* (*weakest precondition* ou *wp* en anglais) pour générer les obligations de preuve et il fournit une *validation* assurant la correction.

Calcul de plus faible précondition. Ce calcul consiste à modéliser le programme sous la forme d'un transformateur de prédicats, c'est-à-dire une fonction WP qui à toute proposition $\mathcal{Q}$ associe la plus faible précondition que doivent vérifier les variables avant l'exécution du programme pour que $\mathcal{Q}$ soit vérifiée après l'exécution. La précondition est la *plus faible* dans le sens qu'elle est impliquée par toute autre précondition. Pour prouver la correction du programme e, i.e. que P est une précondition pour Q, il suffit alors de prouver que $P \Rightarrow WP(Q)$.

Le calcul de plus faible précondition se fait en remontant le corps de la fonction, instruction par instruction, comme indiqué par la Figure 4.6. Cependant, dans l'outil Why, certaines instructions, comme l'appel de fonction et les boucles, sont considérées comme des "boîtes noires" dont on ne connait que la spécification. Ces "coupures" permettent un calcul immédiat de la plus faible précondition, sans dérouler le corps de la fonction ou de la boucle. Lors d'un appel de fonction, seule sa spécification est utilisée, sans regarder les instructions du corps de la fonction. La précondition doit donc être vérifiée au moment de l'appel et la postcondition doit impliquer la propriété recherchée. La plus faible précondition est donc calculée à partir de la spécification de la fonction, sous réserve que la précondition est vérifiée au moment de l'appel. De même, une boucle est spécifiée par son invariant, utilisé pour calculer la plus faible précondition, sous réserve que la boucle vérifie effectivement la propriété exprimée dans l'invariant (voir Figure 4.7).

Un certain nombre de *contraintes* sont donc ajoutées au programme, engendrant des obligations de preuves supplémentaires (preuves que les invariants de boucles sont bien des invariants, preuve que l'invariant implique la postcondition, preuve que les préconditions des fonctions appelées sont vérifiées, etc). Notons que les spécifications (des fonctions et des boucles) doivent être suffisantes pour impliquer la postcondition voulue, sinon les obligations de preuves engendrées automatiquement ne pourront être prouvées.

Les *conditions de vérification* engendrées par l'outil sont donc composées de la preuve que $P \Rightarrow WP(Q)$ et de l'ensemble des obligations de preuves nécessaires au calcul de la plus faible

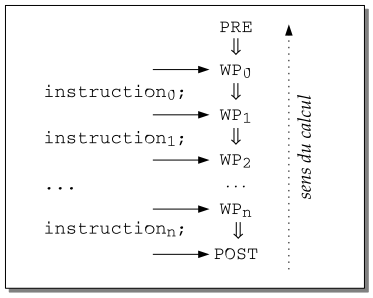


FIG. 4.6 – Schéma du calcul de plus faible précondition

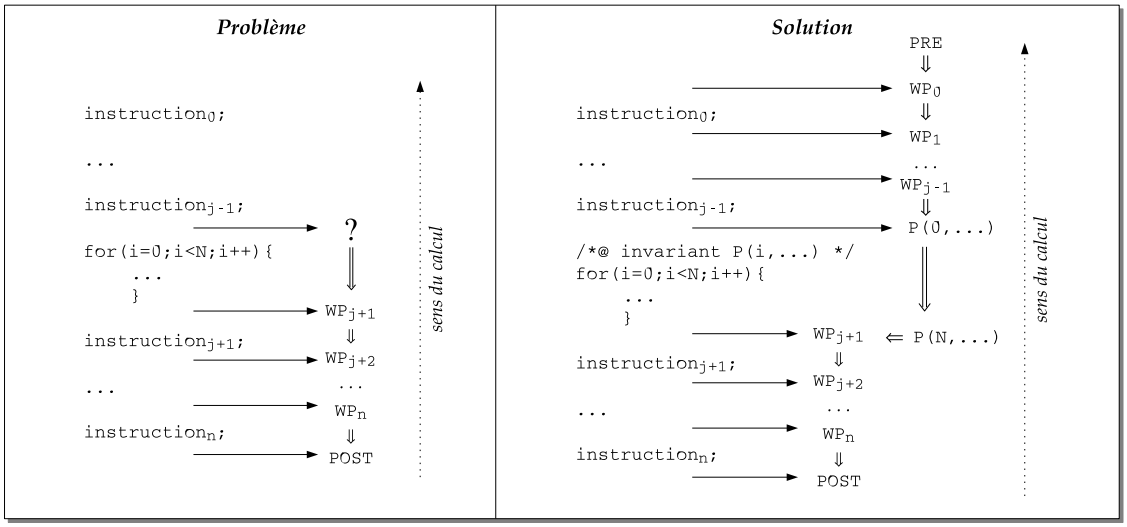


FIG. 4.7 – Schéma du calcul de plus faible précondition dans le cas d'une boucle

précondition. Ces conditions peuvent être vérifiées dans divers systèmes. En effet, une des particularité de Why est sont indépendance vis-à-vis du système de preuve utilisé pour la vérification des obligations. Elles peuvent donc être prouvées dans divers assistants de preuves (Coq, PVS) ou par diverses procédures de décision (Simplify, HaRVey, etc).

Validation. D'autre part, Why fournit un *terme de preuve*, ou *terme de validation*, lorsqu'il est utilisé avec l'assistant de preuve Coq. Comme nous l'avons vu, Coq est basé sur la *théorie des types*, et une preuve de la correction du programme e vis-à-vis de sa précondition P et de sa postcondition Q est un terme de type :

$$\forall x.\ P(x) \ \rightarrow\ \exists x'.\ Q(x, x')$$

Dans Coq, où les preuves sont constructives, un terme de ce type est une fonction qui à tout x et à toute preuve que x vérifie la précondition, associe une paire composée du témoin x' de l'existence et de la preuve de $Q(x, x')$, construite à partir des obligations de preuves. Ce terme de preuve est donc à la fois :

- une représentation dénotationnelle du programme (on a une fonction qui a toute entrée x associe la sortie x') ;
- une *preuve* de la correction, dont il suffit de vérifier le type pour se convaincre de la correction du programme.

Ce terme de validation a été utilisé dans l'étape de modélisation de haut niveau, qui sera décrite en Section 5.2.1

Notons que dans la suite, nous ne distinguerons pas, en général, le travail effectué par Caduceus et celui effectué par Why, et nous parlerons de Caduceus pour désigner tout le processus de vérification.

4.2.2 Spécification

Cette section propose une méthode générale pour la définition d'annotations dans le langage de spécification de Caduceus. Comme nous l'avons expliqué, il y a deux sortes d'annotations : celles qui constituent la spécification des fonctions, et celles qui sont nécessaires à la preuve des conditions de vérification, et plus précisément au calcul de plus faible précondition. Nous allons donc décrire ici l'utilisation du langage d'annotation pour la définition des spécifications de fonctions et les annotations de boucles seront décrites dans l'étape de validation fonctionnelle dans la section suivante.

Précondition et postcondition. La spécification d'un programme consiste à décrire, sous la forme d'annotations, le comportement attendu de chaque fonction du programme, à savoir les propriétés qui doivent être vérifiées par les variables intervenant dans l'exécution de la fonction. Plus précisément, la spécification d'une fonction comporte sa *précondition* (clause `requires` de l'annotation) et sa *postcondition* (clause `ensures` de l'annotation). La postcondition décrit les propriétés vérifiées par les variables après l'exécution de la fonction, sous réserve que les propriétés décrites dans la précondition soient vérifiées. Par exemple, la fonction triviale suivante :

```
/*@ requires x>0
  @ ensures \result>=0 */
int f(int x) { return (x-1); }
```

est spécifiée par le fait qu'elle rend un résultat positif ou nul, sous réserve que l'argument reçu est strictement positif.

La clause `assigns`. La spécification d'une fonction peut également contenir une clause `assigns` qui spécifie quelles sont toutes les variables, et plus généralement toutes les expressions pouvant se trouver à gauche d'une affectation (appelées *valeurs gauches*) qui sont susceptibles d'être modifiées par la fonction. Plus précisément, cette clause signifie que les variables qui n'y apparaissent pas ne sont pas modifiées par la fonction. Par exemple, la spécification de la fonction suivante :

```
int counter;
/*@ requires counter>=0 && \valid(x)
  @ assigns *x
  @ ensures *x>=\old(*x) */
void g(int *x) { *x=*x+counter; }
```

indique implicitement que la variable *counter* n'est pas modifiée par la fonction (ni le pointeur `x` d'ailleurs).

Invariant global. Lorsqu'une précondition contient une propriété, portant sur une ou plusieurs variables globales, qui est vraie dans l'ensemble du programme, cette propriété peut être décrite sous la forme d'un *invariant global*. Cette propriété sera alors ajoutée, d'une part, aux préconditions de toutes les fonctions manipulant ces variables et, d'autre part, aux postconditions modifiant les dites variables. Il s'agit donc d'invariants faibles, supposés vérifiés en début de fonction et devant être prouvés en fin de fonction. Dans l'exemple précédent, si la variable globale `counter` est toujours positive ou nulle, l'invariant suivant pourra être défini :

```
/*@ invariant counter_is_positive: counter>=0 */
```

Alors la fonction `g` n'aura plus besoin de sa précondition. En revanche, pour toute fonction qui modifie `counter`, une obligation de preuve sera générée pour assurer que la variable reste positive ou nulle après l'exécution de la fonction.

Méthodologie de spécification. Nous proposons la méthode générale suivante pour l'écriture de spécifications de fonctions.

1. Identifier tous les effets de bord que l'on attend pour la fonction, sous la forme d'expressions sur les arguments de la fonction et les variables globales du programme. Construire la clause `assigns` de la fonction à partir de cet ensemble d'expressions.
2. Construire la postcondition de la fonction de manière à ce qu'elle décrive les valeurs attendues, après exécution, de toutes les variables mentionnées dans la clause `assigns`. Notons que la postcondition peut, en général, être moins précise et se concentrer sur certaines propriétés des variables, mais ne décrit alors pas tout le comportement de la fonction.
3. Construire la précondition de la fonction en analysant les conditions que doivent respecter les variables globales et les paramètres de la fonction pour que la postcondition soit vérifiée (notons que cela implique que les préconditions de toutes les fonctions appelées dans le corps de la fonction soient satisfaites) et pour qu'il n'y ait aucune *menace*, tel qu'un accès invalide à un tableau ou à la valeur d'un pointeur. En particulier, si un argument de la fonction est de type pointeur, il doit avoir été correctement alloué et la précondition de la fonction doit donc exiger la validité du pointeur. De même, si un argument est un index dans un tableau, alors la précondition doit exiger que l'index soit compris dans les bornes d'allocation du tableau.

4.2.3 Validation fonctionnelle

Une fois la spécification de chacune des fonctions du programme définie dans le langage d'annotation, le but est de prouver la correction du programme vis-à-vis de ces specifications, ce qui revient à une preuve des propriétés fonctionnelles du programme et de l'absence de *menaces* (accès en dehors des bornes d'un tableau, etc).

La preuve de correction du programme vis-à-vis des spécifications annotées consiste à prouver les obligations de preuves générées par Caduceus. Ces obligations de preuves dépendent évidemment de la postcondition attendue pour le programme et il n'y a donc pas de méthode générale de preuve de ces obligations. En revanche, comme nous l'avons expliqué lors de la description du calcul de plus faible précondition implémenté par l'outil, des annotations doivent être ajoutées à tout programme pour spécifier les boucles qu'il contient. Ces annotations doivent être suffisantes pour assurer la postcondition du programme, sans quoi les obligations générées par l'outil ne seront pas prouvables. La spécification des boucles fait donc plutôt partie de la phase de vérification du programme et est décrite dans cette section. Notons que la preuve des obligations fait généralement intervenir les prédicats de la théorie définie par l'outil (comme `assigns`) et que nous avons, à ce propos, développé une "boîte à outils" pour une plus grande facilité d'utilisation de ces prédicats, comme nous l'evoquerons à la fin de la section.

Annotation de boucle. En plus des annotations qui décrivent la spécification des fonctions, certaines annotations, essentiellement les *annotations de boucle*, sont nécessaires pour pouvoir prouver les obligations générées par l'outil. Ces annotations contiennent le *variant* de boucle, l'*invariant* de boucle et éventuellement une clause `loop_assigns`. Le variant de boucle permet d'assurer la terminaison de la boucle, en indiquant quelle est l'expression dont la valeur décroit strictement à chaque tour de boucle. La clause `loop_assigns`, à l'image de la clause `assigns` d'une fonction, indique l'ensemble des expressions qui sont modifiées par la boucle. L'invariant de la boucle exprime la propriété qui est vérifiée à chaque tour de boucle par les variables intervenant dans le corps de la boucle. Il est donc vérifié à l'entrée de la boucle, et s'il est vérifié à une itération de la boucle, il l'est également à l'itération suivante.

La clause `loop_assigns` et l'invariant de boucle sont nécessaires au calcul de plus faible précondition. En effet, une boucle est une exécution d'un ensemble d'instructions un nombre *paramétré* de fois. La plus faible précondition qui doit être vérifiée en début de boucle, pour qu'une postcondition donnée soit vérifiée à la sortie de la boucle, ne peut être calculée qu'à l'aide de la preuve de la préservation d'une certaine propriété à chaque itération.

La définition de cette propriété peut être fastidieuse et nécessiter plusieurs essais, mais elle doit être guidée par ce qui est attendu du programme, à savoir ce qui a été mentionné dans sa spécification. En effet, lorsqu'un programme est implémenté de manière à avoir un certain comportement, les boucles qu'il contient sont censées contribuer elles aussi à assurer ce comportement. Elles ont été développées dans ce but et l'invariant ne fait que traduire leur effet sur le comportement du programme.

Méthodologie de définition d'invariants de boucle. Nous proposons la méthode générale suivante pour l'écriture d'annotations de boucles.

1. Le variant de la boucle doit simplement indiquer quelle est la valeur qui décroit dans la boucle et qui assure sa terminaison. Pour une boucle `for`, il s'agit en général simplement de la borne supérieure de la boucle moins le compteur de boucle (pour `for(i=0;i<N;i++)` le variant de boucle sera `N-i`). Pour une boucle `while`, il faut déterminer l'expression sur les

variables modifiées dans la boucle qui diminue strictement après toute exécution possible de la boucle.

2. La clause `loop_assigns`, tout comme la clause `assigns` d'une fonction, doit être construite à partir d'une analyse des expressions susceptibles d'être modifiées lors de l'exécution de la boucle.
3. Comme nous venons de l'expliquer, la définition de l'invariant est plus délicate. Intuitivement, elle doit être guidée par la postcondition et doit indiquer les valeurs des variables (de la clause `loop_assign`) à la sortie d'un tour de boucle en fonction de leurs valeurs à l'entrée du tour de boucle. Cela permet, par induction, de connaître leurs valeurs après l'exécution de toute la boucle en fonction de leurs valeurs avant l'exécution de la boucle. La définition d'un invariant peut être relativement intuitive dans les exemples simples. Mais dès lors que la postcondition est complexe, une bonne compréhension du schéma de calcul de l'outil est utile pour trouver la formulation qui permettra de prouver les obligations.

Boîte à outil pour la validation dans Coq. Lors de la preuve des obligations générées par Caduceus, les lemmes de la théorie du modèle mémoire (comme la transitivité du prédicat `assigns`) sont utilisés. Nous avons complété cet ensemble de lemmes, par une vingtaine de nouveaux lemmes, permettant une utilisation plus facile de la théorie lors du développement de la preuve. Ces lemmes concernent essentiellement les prédicats `unchanged` et `assigns`, comme par exemple le lemme suivant :

```
∀ alloc, ∀ m1, ∀ m2, ∀ p, ∀ i1, ∀ i2,
i2≤i1 ⇒ j1≤j2 ⇒
(assigns alloc m1 m2 (range_loc p i1 j1)) ⇒
(assigns alloc m1 m2 (range_loc p i2 j2))
```

4.3 Limitations du modèle mémoire de Caduceus

Comme nous l'avons évoqué dans la Section 2.2.3.2 (page 74), le langage C a une sémantique compliquée, dont un certain nombre de cas ne sont pas spécifiés, ce qui rend sa formalisation délicate. C'est pourquoi, en dehors des analyseurs statiques permettant la détection, à la compilation, de certains types d'erreurs (division par zéro, etc), peu de travaux ont tenté la formalisation du langage de programmation C (comme en HOL dans la thèse [104]) et encore moins le développement d'un outil (comme Caveat [33] ou Caduceus [53]). Les limitations communes de ces travaux concernent les constructions C de bas niveau et dépendantes de l'implémentation, telles que les *unions* ou les *casts*. La formalisation de la sémantique de telles constructions nécessite un modèle mémoire de suffisamment bas niveau, ce qui est contraire à l'objectif de ces outils de rendre les preuves praticables par le choix d'un modèle mémoire de haut niveau. En effet, un modèle mémoire *à la Burstall-Bornat*, où la mémoire est séparée en segments disjoints, permet de réduire considérablement le nombre d'obligations de preuves générées pour assurer la correction du programme.

C'est ce type de modèle mémoire qui est utilisé dans l'outil Caduceus, que nous avons choisi pour nos travaux de vérification de programmes C embarqués. Caduceus, décrit en détail dans la Section 4.2.1, a été développé progressivement depuis mars 2004 et est toujours en cours de développement. Il est à noter que lorsque les travaux de cette thèse ont débuté, cet outil n'existait pas et l'interprétation de programmes impératifs était intégrée à l'outil Why (voir Section 4.2.1.2). Elle était alors limitée à un langage très simple avec références et quelques

instructions impératives telles que les boucles. Puis, un travail considérable a été fourni pour pouvoir gérer les structures de données du langage de programmation C, *via* un modèle mémoire défini en dehors de l'outil Why, au sein du nouvel outil Caduceus. Le sous-ensemble du langage C accepté par Caduceus s'est enrichi au fur et à mesure du développement de l'outil et des besoins des utilisateurs. En tant qu'utilisateurs pour la vérification de code C embarqué, nous avons pu mettre en évidence des limitations ou des erreurs de l'outil et contribuer à son évolution. Aujourd'hui, une très large partie du langage C est gérée par l'outil, qui connaît également d'importants progrès en termes d'utilisation, d'automatisation des preuves, etc. Le fait que les preuves ne soient pas insurmontables en pratique est grandement dû au choix du modèle mémoire. Mais ce modèle, utilisant la séparation en mémoires disjointes, impose également des contraintes quant au code accepté et restreint en particulier la gestion des constructions de bas niveau.

Cette section a pour but de mettre en évidence les limitations du modèle mémoire utilisé par l'outil Caduceus et les imperfections de l'interprétation de la sémantique de C. Commençons par décrire les constructions C dont la formalisation est délicate.

4.3.1 Constructions critiques du langage C

4.3.1.1 Structures, unions et champs de bits

Structures. Le langage C permet de définir des *structures* et des *unions*. Une structure est un ensemble de valeurs de types différents, qui sont repérées à l'intérieur de la structure par un identificateur, appelé *nom de champ*. Par exemple, la structure suivante :

```
struct ma_structure {short x; char y;};
```

possède deux champs `x` et `y` qui peuvent être accédés ou modifiés par la notation `s.x` ou `s.y`, où `s` est une variable de type `struct ma_structure`. Le nom de type `struct ma_structure` peut être abrégé en utilisant la construction `typedef` τ_1 τ_2 qui indique que τ_2 est un autre nom pour τ_1 :

```
typedef struct ma_structure {short x; char y;} nouveau_nom;
```

(où `nouveau_nom` peut-être de nouveau `ma_structure`), ou plus simplement :

```
typedef struct {short x; char y;} nom;
```

En mémoire, les champs d'une structure sont globalement juxtaposés (voir Figure 4.8[1]). Plus

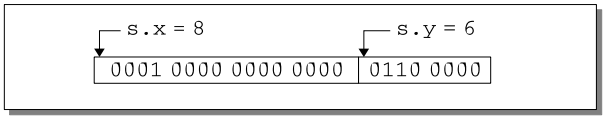


FIG. 4.8 – Représentation en mémoire d'une variable de type `struct ma_structure`

exactement, la représentation mémoire d'une structure dépend beaucoup du compilateur. La norme impose uniquement que les champs doivent être alloués suivant l'ordre de leur déclaration et que l'adresse de la structure est la même que celle du premier champ. Ainsi il peut y avoir des *octets de remplissage* entre deux champs (mais jamais en début de structure). En effet, la norme autorise les compilateurs à imposer des contraintes d'alignement pour des raisons d'efficacité. Par exemple, les entiers de deux octets peuvent être alignés sur des adresses paires[2]. Par conséquent, la structure :

[1] Voir page 1 pour les conventions prises dans ce mémoire pour les tailles en octets des différents types entiers de C.

[2] ce qui permet, sur des machines à 16 bits, d'accéder en une fois à l'entier correspondant.

```
struct exemple { short a; char b; short c; };
```

peut contenir un octet de remplissage, comme le montre la Figure 4.9.

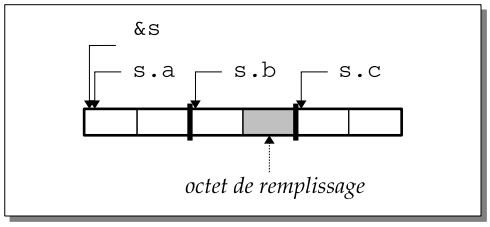


FIG. 4.9 – Exemple de structure avec un octet de remplissage

Unions. Une union permet, quant à elle, de considérer un même objet avec différents types. Syntaxiquement semblable à une structure, elle possède des champs, qui sont accédés de la même façon. Mais sémantiquement, les champs d'une union sont globalement *superposés*. Par exemple, l'union suivante :

```
union mon_union {short s; struct{char x; char y;} p;};
```

permet de considérer une variable de deux octets soit par la valeur de l'entier contenu dans les deux octets, soit par les deux valeurs distinctes, contenues dans chacun des deux octets.

Cette union peut être utilisée, par exemple, pour la représentation d'une adresse mémoire dans une mémoire séparée en blocs de 256 octets. Si `a` est une variable de type `mon_union`, `a.s` peut représenter l'adresse complète, `a.p.y` le numéro de bloc et `a.p.x` l'index de l'adresse dans le bloc (voir Figure 4.10). Les champs `x` et `y` sont utilisés pour accéder directement à l'index dans

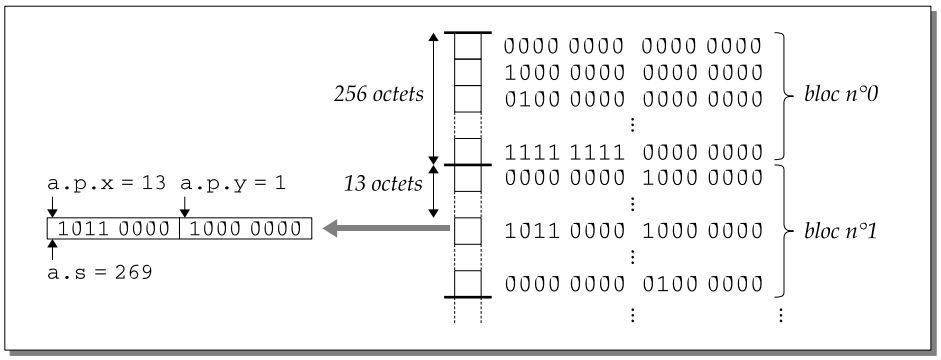


FIG. 4.10 – Utilisation d'une union pour décomposer une adresse en numéro de bloc et index

le bloc et au numéro de bloc, sans avoir à faire de calcul sur l'adresse. Le champ `s`, quant à lui, permet de parcourir la mémoire de façon séquentielle, sans avoir à calculer si c'est le champ `x` ou le champ `y` qui doit être incrémenté, suivant que la fin du bloc a été atteinte ou non (voir Figure 4.11).

La définition d'une union dans la norme indique que l'adresse d'un objet de type union est la même que l'adresse de chacun de ses champs. La notion d'octet de remplissage entre deux champs n'a alors plus de signification, puisque tous les champs ont la même adresse. En revanche, il peut

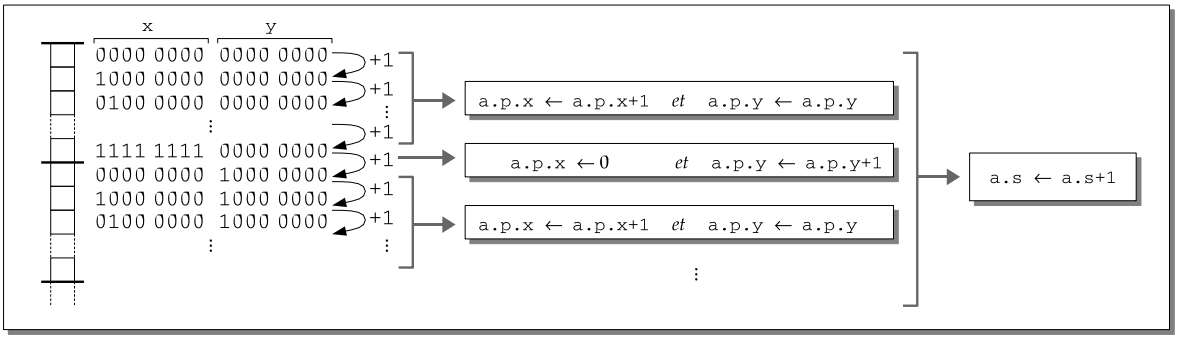


FIG. 4.11 – Utilisation du champ `s` pour incrémenter l'adresse

y avoir des octets de remplissage non utilisés par un champ mais utilisés par un autre, notamment à l'intérieur d'un champ de type structure. Par exemple, l'union suivante :

```
union exemple {struct{short x; short y;} vision1;
               struct{char c;  short s;} vision2;}
```

sera représentée avec un octet de remplissage dans le champ `vision2`, comme montré dans la Figure 4.12.

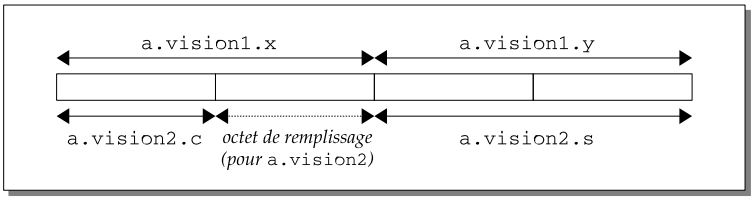


FIG. 4.12 – Octet de remplissage dans une union

Comme pour les structures, les emplacements des octets de remplissage dans une union ne sont pas spécifiés par la norme. Ils servent en particulier à faciliter la conversion d'un champ à un autre, ce qui est très dépendant du compilateur.

Notons que l'utilisation des unions telle que présentée dans la Figure 4.11, où l'incrémentation du champ `s` permet d'incrémenter le champ `x`, ou le champ `y` suivant les cas, n'est possible que parce que la taille du champ `x` est exactement de huit bits. Si la taille des blocs de mémoire n'est pas de 256 octets, mais par exemple de 64 octets, le champ `x` devrait avoir une taille de 6 bits. Pour garder l'avantage de l'union dans ce cas, il est possible d'utiliser les *champs de bits*.

Champs de bits. Un des champs d'une structure peut être un *champ de bits*, c'est-à-dire une suite de bits, comme dans l'exemple donné en Figure 4.13. Le chiffre qui suit les deux-points indique le nombre de bits qu'occupe le champ de la structure. Par exemple, `e.libre`, où `e` est de type `etat`, peut prendre deux valeurs (0 ou 1).

Une telle possibilité peut s'avérer intéressante pour deux raisons, et en général lorsque plusieurs champs consécutifs sont des champs de bits. Premièrement, les champs de bits peuvent être utilisés pour compacter une information, lorsque, par exemple, plusieurs champs peuvent être stockés dans un seul octet (comme dans l'exemple de la Figure 4.13). Dans les cartes à puces, où les ressources sont limitées, ce mécanisme permet d'utiliser le minimum d'espace possible.

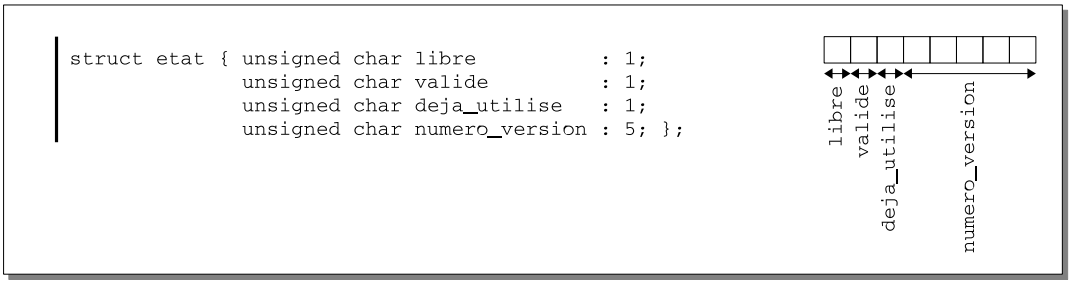


FIG. 4.13 – Exemple d'utilisation des champs de bits pour compacter de l'information

Les champs de bits sont également utilisés lorsqu'un champ doit être stocké sur un nombre de bits donné. Dans notre exemple d'utilisation des unions pour représenter un parcours séquentiel de la mémoire (Figure 4.11), si les blocs de la mémoire ont une taille de 64 octets, alors il faudra définir :

```
union mon_union {short s; struct{short x:6; short y:10;} p;};
```

pour garder la propriété que les champs `x` et `y` valent respectivement l'index et le numéro de bloc de l'adresse et que l'incrémentation du champ `s` permet l'incrémentation de `x` ou de `y` suivant si la fin du bloc est atteinte ou non (voir Figure 4.14).

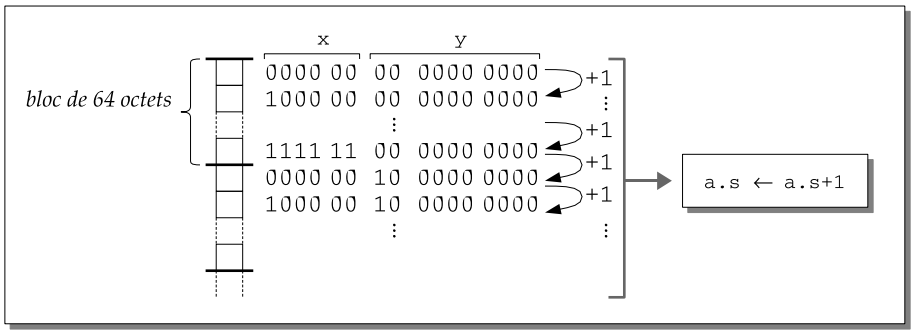


FIG. 4.14 – Utilisation des champs de bits pour représenter une adresse dans une mémoire séparée en bloc de 64 octets

La norme ne précise pas l'ordre dans lequel les champs de bits sont stockés en mémoire. En général, sur une machine d'architecture Little-Endian[1], les premiers champs décrivent les bits de poids faibles, alors qu'ils décrivent les bits de poids forts si l'architecture est Big-Endian. Dans l'exemple de l'union `etat` de la Figure 4.13, cela signifie que si le champ `libre` vaut 1, le champ `valide` vaut 0, le champ `deja_utilise` vaut 1, et le champ `numero_version` vaut 9, alors la valeur de l'octet contenant les champs est de 77 en architecture Little-Endian et de 329 en architecture Big-Endian (voir Figure 4.15).

Exemple de JournalStateRegistry. Dans notre cas d'étude de gestion de la Flash, nous avons vu (Section 1.3.4) qu'un tableau, le `JournalStateRegistry`, contenait l'ensemble des états d'effacement de chaque journal, sous forme journalisée. Étant donné qu'il n'y a que trois états d'effacement possibles pour un journal, seuls deux bits suffisent pour représenter un état. Le

[1]Voir Conventions page 1.

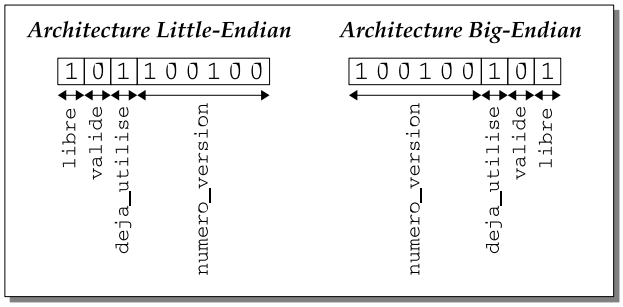


FIG. 4.15 – Influence de l'architecture sur l'ordre des champs de bits dans une structure

tableau d'états `JournalStateRegistry[jid]`, pour un journal donné, est donc un tableau d'éléments de deux bits. Cependant, la plus petite granularité pour un tableau en C est l'octet, donc `JournalStateRegistry[jid]` est un tableau d'octets (de `char`, ou plus précisément de `unsigned char`) :

```
typedef unsigned char   u1;     /* (1 byte) */
typedef u1 journal_state[FLASH_PAGE_SIZE];
journal_state JournalStateRegistry[JOURNAL_REGISTRY_DIM];
```

et chaque élément du tableau (`JournalStateRegistry[jid][i]`) contient quatre états. Pour accéder à un état, il faut d'abord accéder à l'octet, puis à l'ensemble de deux bits représentant l'état à l'intérieur de cet octet, comme indiqué dans la Figure 4.16. L'index d'un état est donc constitué de l'index `byteNb` de l'octet et du numéro `slotNb` de l'état dans l'octet. Comme décrit

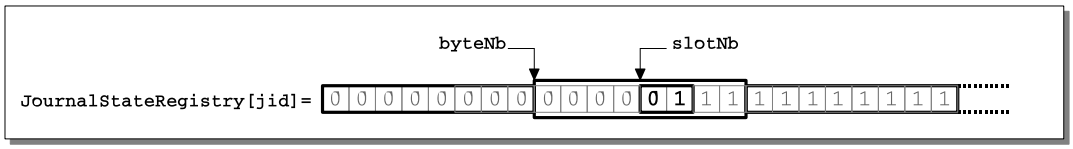


FIG. 4.16 – Accès à un état d'effacement d'un journal dans `JournalStateRegistry`

dans le paragraphe précédent, l'utilisation d'une union permet de parcourir facilement le tableau en incrémentant l'index, sans avoir à penser si c'est `slotNb` ou `byteNb` qui doit être incrémenté, suivant si on a atteint ou non la fin de l'octet. Mais pour cela, la taille de `slotNb` doit être exactement de deux bits, puisque `slotNb` varie de zéro à trois. L'index d'un état peut donc être représenté par la structure suivante :

```
#define JOURNAL_STATE_NB_BITS           2
typedef unsigned short  u2;     /* (2 bytes)*/
typedef union {
     u2  w;
     struct { u2 slotNb   :JOURNAL_STATE_NB_BITS;
              u2 byteNb   :8*sizeof(u2)-JOURNAL_STATE_NB_BITS; } s;
} journal_state_index;
```

Cette définition permet un parcours séquentiel du tableau. En effet, lorsque le champ `w` est incrémenté, si `slotNb` était strictement inférieur à trois, alors il est incrémenté et `byteNb` est inchangé, et si `slotNb` valait exactement trois, alors il devient nul et c'est `byteNb` qui est incrémenté (voir Figure 4.17). De plus, le champ `w` peut être utilisé pour une mise à zéro des champs `slotNb` et `byteNb` en une seule opération, ramenant l'index au premier état du tableau.

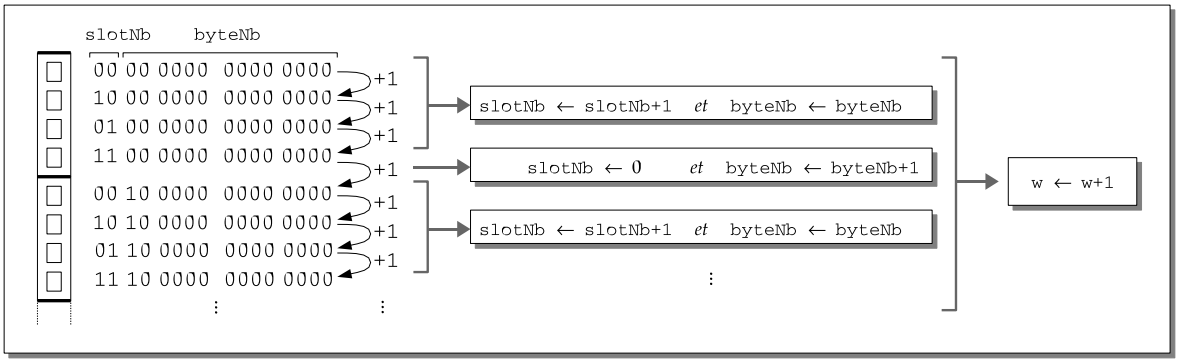


FIG. 4.17 – Parcours du tableau `JournalStateRegistry`

En conclusion, le langage C est un langage de programmation de bas niveau permettant une manipulation manuelle de la mémoire, illustrée par les structures de données telles que les unions ou les champs de bits, qui permettent de lire la mémoire de plusieurs façons différentes et avec une précision de l'ordre du bit. Cette caractéristique du langage le rend adapté au développement de programmes très optimisés, comme ceux embarqués sur les cartes à puces, où les ressources sont limitées. Mais lorsqu'il s'agit de modéliser formellement ces structures de données dans un but de vérification, le choix du modèle mémoire est très délicat, comme nous le verrons dans la Section 4.3. De plus, la dépendance vis-à-vis du compilateur utilisé accentue la difficulté de la définition d'un modèle général utilisable dans un outil de vérification de programme.

4.3.1.2 Casts de pointeurs

L'opérateur de *cast* du langage C permet une conversion forcée de type. C'est un opérateur unaire, qui se note sous la forme du nom du type entre parenthèse, et dont le rôle est d'effectuer la conversion, dans le type donné, du résultat de l'expression qui le suit. Ainsi, l'expression `(`τ`)e` correspond à l'expression `e` convertie dans le type τ. Les conversions autorisées sont les conversions :

- entre types scalaires ; par exemple, si `n` et `p` sont des variables entières, alors l'expression `(short)(n+p)` aura comme valeur le résultat de l'expression `n+p` convertie en `short` ;
- entre pointeurs ; par exemple, si `p` est un pointeur sur un entier de type `int`, alors l'expression `(char*)p` correspond à considérer l'objet pointé avec un type différent et arithmétique différente (l'expression `((char*)p)++` ; augmentera le pointeur d'un seul octet, alors que `p++` augmente le pointeur de quatre octets).
- entre pointeur et entier ou entre entier et pointeur ; cette conversion est relativement désuète, mais autorisée par la norme, qui précise très peu de chose sur cette conversion à part qu'elle dépend très fortement du compilateur.

Nous nous intéressons essentiellement aux casts de pointeurs, qui permettent une manipulation bas-niveau de la mémoire, comme dans l'exemple suivant :

```
typedef struct point{int x; int y;} point;
void f(point* p){
  int *i;
  i=(int*) p;
  *i=0;
}
```

où le champ `x` de la structure pointée par `p` est mis à zéro à la fin d'un appel à la fonction `f`.

Là encore, la manipulation de la mémoire bas niveau et dépendante de l'implémentation pose des problèmes pour le choix d'un modèle mémoire. En effet, un modèle mémoire permettant une vérification de programme C praticable est en général incompatible avec une vision bas niveau de la mémoire, comme nous allons le voir dans la section suivante.

4.3.2 Structures : incorrectes dans Caduceus

Le problème lié à l'interprétation des structures est indépendant du choix d'un modèle à la Burstall-Bornat. Il provient du fait qu'une structure est interprétée, dans Caduceus, par un *pointeur*, c'est-à-dire par son adresse (sa valeur gauche) et non pas par la valeur de ses champs (sa valeur droite) comme le veut la sémantique du langage C (voir par exemple [41]). Ceci entraîne une interprétation incorrecte de l'affectation globale d'une structure et de la transmission d'une structure en argument d'une fonction.

Affectation. La sémantique de C indique que l'affectation globale d'une variable de type structure à une autre variable du même type est une abréviation pour l'affectation de chacun de ses champs. Par exemple, si `p` et `q` sont deux variables du type `point` vu précédemment, alors l'affectation `p=q ;` correspond à la séquence d'instructions : `p.x=q.x ;p.y=q.y ;`.

Étant donné que les structures sont interprétées par des pointeurs dans le modèle de Caduceus, l'affectation est vue comme une affectation de pointeur, entraînant un *alias* (i.e. deux adresses désignant le même emplacement mémoire), ce qui est incorrect, comme le montre l'exemple suivant :

```
1  typedef struct {int x; int y;} point;
2  point p = {0,0};
3  point q = {1,1};
4  p=q;
5  p.x=1;
```

dont l'exécution réelle est représentée dans la Figure 4.18 et dont l'interprétation dans Caduceus est représentée dans la Figure 4.19. On remarque que l'outil Caduceus est incorrect puisqu'il permet de prouver, par exemple, que le champ `x` de `q` vaut 1 à la fin de l'exécution de cet ensemble d'instructions, alors qu'en réalité il vaut toujours 0.

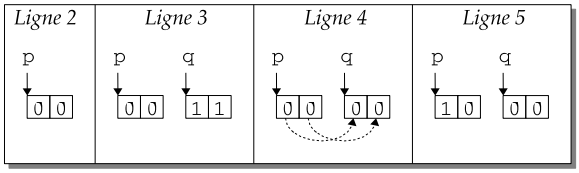


FIG. 4.18 – Exécution réelle d'une affectation globale de structure

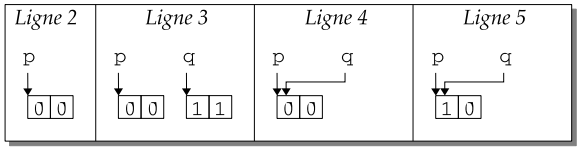


FIG. 4.19 – Interprétation de l'exécution d'une affectation globale de structure dans Caduceus

Passage de paramètres. Il existe plusieurs façons de transmettre des arguments à une fonction lors d'un appel. Les arguments peuvent principalement être passés *par valeur*, *par adresse* (ou *référence*) ou par *résultat* (voir Figure 4.20). Une transmission *par valeur* signifie que les

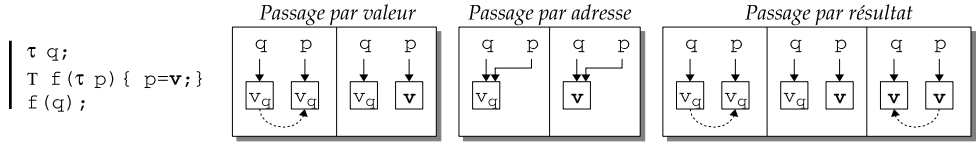


FIG. 4.20 – Les différents modes de transmission de paramètres

valeurs des arguments sont recopiées dans des emplacements locaux à la fonction, qui travaille sur des copies. Toute modification de la copie n'a aucune incidence sur la valeur de l'original. Lors d'une transmission *par adresse*, ce sont les adresses des arguments qui sont transmises. La fonction travaille donc directement sur les originaux et peut éventuellement en modifier la valeur. Une transmission *par résultat* consiste, quant à elle, à recopier les valeurs des arguments dans des emplacements locaux à la fonction et à recopier les valeurs en fin d'exécution de fonction dans les arguments fournis à l'appel. Ce dernier mode de transmission est équivalent à une transmission par adresse, excepté en présence d'*aliasing*, i.e. lorsque la même variable est donnée pour deux arguments différents d'une même fonction ou lorsqu'une variable globale est donnée en argument alors qu'elle est modifiée elle-même par la fonction (voir la Figure 4.21).

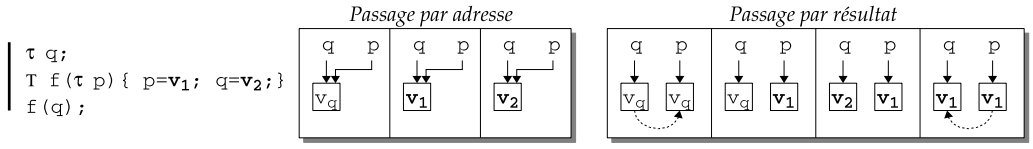


FIG. 4.21 – Exemple où le passage par résultat diffère du passage par adresse

Dans le langage C, la transmission des arguments se fait toujours par valeur. Toutefois, une fonction peut modifier la valeur de l'argument grâce aux pointeurs. En effet, si une fonction prend en argument un pointeur, alors la valeur de ce pointeur, à savoir l'adresse d'un objet, lui sera transmise. La valeur du pointeur sera recopiée localement, ce qui signifie que la fonction travaille sur une copie du pointeur, mais dont la valeur est toujours l'adresse de l'objet. Par conséquent, l'objet peut être modifié, si la valeur pointée est modifiée, mais la valeur du pointeur lui-même ne peut être modifiée. Dans l'exemple donné dans la Figure 4.22, la valeur du *pointeur* `q` ne sera pas modifiée par la fonction `f`, puisque seule sa valeur a été transmise. Donc le pointeur `q` ne vaudra pas `NULL` après l'appel. En revanche, la valeur pointée par `q` sera modifiée et vaudra 5

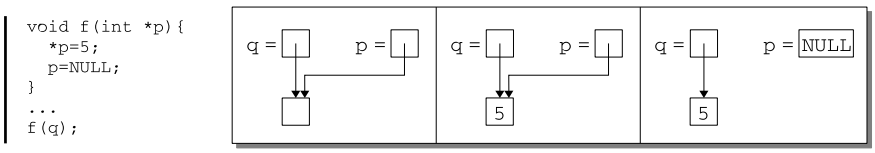


FIG. 4.22 – Exemple de passage d'un paramètre de type pointeur dans le langage C

après l'appel. En conclusion, le passage par adresse n'existe pas à proprement dit en C, mais peut être programmé explicitement, par le biais des pointeurs.

Une structure, comme tout autre objet, est passée par valeur à une fonction, ce qui signifie que si une fonction prend une variable de type structure en argument, alors les champs de cette variable ne pourront pas être modifiés par un appel à la fonction. Or, étant donné qu'une structure est interprétée par sa valeur gauche par Caduceus, c'est son adresse qui est transmise à la fonction lors d'un appel, permettant à la fonction de modifier les champs de la structure. Ceci est incorrect, comme le montre l'exemple suivant :

```
typedef struct {int x; int y;} point;
point q={1,1};
void f (point p) {p.x=0;}
f(q);
```

dont l'exécution réelle est représentée dans la Figure 4.23 et dont l'interprétation dans Caduceus est représentée dans la Figure 4.24. On remarque que l'outil Caduceus est incorrect puisqu'il permet de prouver que le champ `q.x` vaut zéro après l'appel de la fonction, alors qu'il vaut toujours 1 (puisque la fonction ne modifie rien).

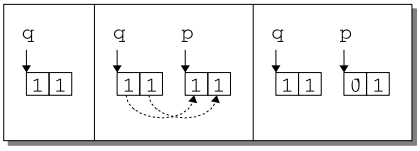


FIG. 4.23 – Exécution réelle d'un passage de structure en paramètre

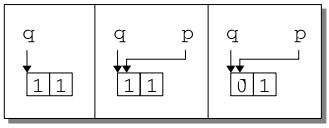


FIG. 4.24 – Interprétation d'un passage de structure en paramètre dans Caduceus

L'interprétation des structures par des pointeurs dans Caduceus doit donc être corrigée de manière à rendre correcte l'affectation globale de variable de type structure et le passage d'un paramètre de type structure à une fonction. C'est ce que nous proposons dans la Section 4.4.

Valeur de retour. Notons qu'en C, le résultat d'une fonction est toujours transmis à la fonction appelante par recopie de sa valeur. Lorsqu'il s'agit d'un pointeur, il n'y a pas de nouvelle allocation et l'objet pointé n'est pas recopié (ce qui engendre un risque potentiel d'erreur puisque

si l'objet est local a la fonction appelée, il est détruit à la sortie). Concernant les structures, étant donné qu'elles sont interprétées dans Caduceus par des pointeurs, les valeurs des différents champs ne sont pas recopiés au sein de la fonction appelante, ce qui ne correspond pas à ce qu'il se passe en C. Ceci n'est toutefois pas traité dans la solution que nous proposons, qui refuse donc les structures en retour de fonction.

4.3.3 Unions : non supportées dans Caduceus

L'interprétation des unions pose un problème lorsque l'on a fait un choix de modèle mémoire à la Burstall-Bornat. Comme il a été expliqué à la Section 4.2.1, les unions, tout comme les structures, sont représentées dans ce modèle par un ensemble de mémoires disjointes, correspondant aux différents champs. Dans le cas des structures, cela permet d'assurer "gratuitement" qu'une modification sur un champ de la structure laisse les autres champs inchangés. Mais cette séparation s'avère incorrecte dans le cas des unions. En effet, la sémantique de C indique que les champs d'une union sont superposés, alors que le modèle à la Burstall-Bornat les considère dans des emplacements mémoire disjoints. Autrement dit, lorsqu'un champ d'une union est modifié, les autres champs sont également modifiés en pratique, alors qu'ils sont considérés comme inchangés dans le modèle.

Plus précisément, cela pose un problème dans le cas d'une utilisation particulière du type union. En effet, un type union peut être utilisé de deux façons différentes. D'une part, il peut être utilisé comme un *type somme*, pour rassembler plusieurs types différents dans un même type. Par exemple, si un registre temporaire est utilisé pour stocker des objets de différentes nature, on pourra définir l'union suivante :

```
typedef union {
  char barray [TEMP_BARRAY_SIZE];
  long larray [TEMP_LARRAY_SIZE];
  struct{char x:3; char y:5; long id;} header;
  struct{short class; short ins; char p1; chap p2; int body[BODY_SIZE];} apdu;
  ...
} temp_buffer;
```

Si une variable `tmp` est de type `temp_buffer`, alors elle occupe un emplacement de taille suffisante pour y stocker chacun de ses champs. Ceci permet de l'utiliser pour stocker tantôt un tableau d'octets, tantôt les données d'une commande APDU, etc. Lorsqu'une union est utilisée de cette façon, la valeur d'un champ n'est généralement lue que si c'est par ce champ que l'union a été mise à jour pour la dernière fois. En effet, lorsque l'on se sert de la variable temporaire pour stocker un tableau d'octets, on n'a, en général, pas besoin de pouvoir connaître sa valeur en tant que commande APDU.

Dans ce cas le modèle à la Burstall-Bornat convient, puisqu'il n'est pas nécessaire de prouver qu'une mise à jour d'un champ entraîne une modification des autres champs. Toutefois, il faut tout de même s'assurer que *les unions ne sont utilisées que de cette façon*, ce qui est indécidable en général. Notons qu'une façon de s'en assurer serait de considérer qu'une modification d'un champ d'union engendre la modification des autres champs, mais avec des valeurs qui sont quantifiées existentiellement : la mise à jour `temp.apdu=e ;` mène à un état mémoire où il existe un `x` tel que `temp.header=x` (par exemple). Cela signifie que si ce n'est pas le champ `apdu` qui est lu ensuite, mais le champ `header`, alors on ne pourra rien déduire de faux puisqu'on saura que sa valeur a été modifiée, même si l'on ne connaît pas la valeur de `x`.

En revanche, une union peut également être utilisée comme un moyen d'avoir plusieurs interprétations d'un même emplacement mémoire. Dans ce cas, elle permet de lire et de modifier un bloc mémoire de plusieurs façons différentes. Il est alors nécéssaire de connaître la valeur (notée `x` précédemment) de tous les champs de l'union lors de la modification d'un des champs. C'est le cas de notre exemple de `journal_state_index` (donné page 118 et redéfini ci-dessous) qui peut être vu soit comme un entier de deux octets, soit comme deux valeurs distinctes indiquant respectivement le numéro d'octet dans le tableau d'états et le numéro d'état dans cet octet. Dans ce cas, le modèle de Burstall-Bornat pour les unions est incorrect. Par exemple, considérons le programme suivant :

```
1  typedef union { unsigned short  w;
2                  struct { unsigned short slotNb:2;
3                           unsigned short byteNb:14; } s;} journal_state_index;
4  journal_state_index jsi;
5  jsi.s.slotNb=3;
6  jsi.s.byteNb=5;
7  jsi.w=0;
```

dont l'exécution réelle est représentée dans la Figure 4.25 et dont l'interprétation dans un modèle mémoire de Burstall-Bornat est représentée dans la Figure 4.26. Dans cet exemple, on pourra prouver, avec un modèle à la Burstall-Bornat, que le champ `jsi.s.slotNb` vaut toujours 3 après l'exécution, alors que sa valeur a été mise à zéro par l'affectation du champ `jsi.w`.

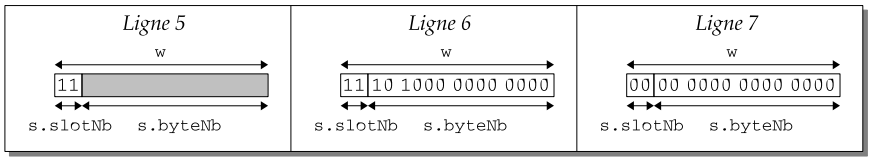


FIG. 4.25 – Exécution réelle d'affectations de champs d'union

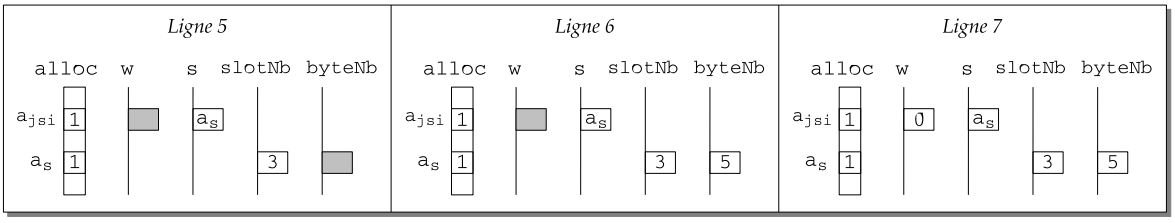


FIG. 4.26 – Interprétation d'affectations de champs d'union dans un modèle à la Burstall-Bornat

Notons que cette interprétation des unions pose également des problèmes dans les annotations. En effet, si une fonction a la postcondition `/*@ ensures pjsi->w==0 */`, alors il faut que les champs `pjsi->s.slotNb` et `pjsi->s.byteNb` valent aussi zéro après l'exécution de la fonction. En effet, un modèle à la Burstall-Bornat serait incorrect dans un exemple comme le suivant :

```
1  /*@ ensures jsi->w==0 */
2  void f(journal_state_index *jsi);
3  journal_state_index *pjsi;
4  pjsi->s.slotNb=3;
5  f(pjsi);
```

où on arrive à prouver qu'à la fin, `pjsi->s.slotNb` vaut toujours 3.

Une solution doit donc être trouvée pour pouvoir interpréter correctement la sémantique des unions, soit dans un modèle mémoire de plus bas niveau, soit en adaptant le modèle mémoire à la Burstall-Bornat de manière à représenter le lien qui existe entre les emplacements mémoires correspondant à des champs d'une même union. C'est ce qui sera étudié dans la Section 4.5.

4.3.4 Casts : non supportés dans Caduceus

Comme nous l'avons expliqué dans la Section 4.3.1.2, il existe plusieurs sortes d'opérations de cast. Commençons par exclure tout de suite les casts entre entiers et pointeurs, qui, comme nous l'avons dit, sont peu utilisés et très peu spécifiés. Nous ne les traiterons pas du tout dans ce mémoire. Nous allons donc nous intéresser aux casts entre pointeurs et aux casts entre entiers. Mais nous ferons plutôt la distinction entre les casts faisant intervenir des structures et les casts ne faisant intervenir que des entiers.

Le problème lié aux casts ne faisant intervenir que des entiers provient de l'interprétation identique, par Caduceus, de tous les types entiers, sans prendre en compte le nombre d'octets qu'ils occupent en mémoire. Ainsi, si la variable `i` est de type `int`, alors elle a la même valeur que `(char)i` dans Caduceus, alors qu'en réalité `(char)i` ne correspond qu'au premier des quatre octets de `i`. Concernant les casts de *pointeurs* d'entiers, le problème de la conversion est toujours présent puisque `*((char*)i)` et `*i` ont la même valeur dans Caduceus, si `i` est de type `int*`. Mais à cela s'ajoute un problème d'interprétation de l'arithmétique correspondante. En effet, un tableau d'`int` est interprété de la même manière qu'un tableau de `char` dans Caduceus, alors que l'arithmétique est différente dans les deux tableaux. Par exemple, si `i` est de type `int*`, alors `i++` incrémente le pointeur de quatre octets, alors que `((char*)i)++` ne l'augmente que d'un octet (voir la Figure 4.27).

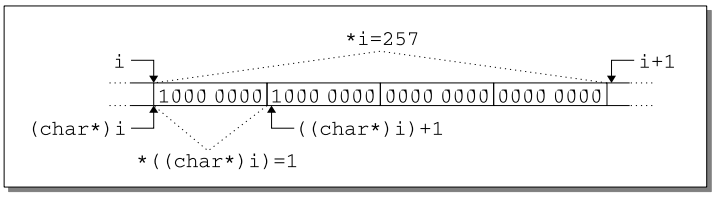


FIG. 4.27 – Représentation mémoire d'un cast d'entier

Or, les casts d'entiers et de pointeurs d'entiers, qui sont les seuls casts autorisés par l'outil, sont interprétés sans conversion. Il est donc possible de prouver, par exemple, que si `i` est de type `int*` et si `*i` vaut 257, alors `*((char*)i)` vaut également 257, alors qu'en réalité il vaut 1. De même, à la suite des instructions suivantes :

```
char tab[5]={1,2,3,4,5};
int *i=(int *)tab;
i++;
```

il est possible de prouver que `*((char*)i)` vaut 2 alors qu'il vaut 5 puisque l'index a été incrémenté de quatre octets. Caduceus doit donc être modifié de sorte que la lecture de la valeur pointée par un pointeur d'entier et l'incrémentation d'un pointeur d'entier dépendent du type C de l'entier considéré.

La deuxième catégorie de casts concerne ceux faisant intervenir un type structure, à savoir un cast entre deux pointeurs de structure ou entre un pointeur de structure et un pointeur d'entier. Ce genre de cast n'est pas accepté par Caduceus et pose en effet un problème plus sérieux car, à l'image des unions, il "viole" le modèle à la Burstall-Bornat en considérant que des segments mémoires, supposés disjoints dans le modèle, sont en fait confondus. Par exemple, considérons les instructions suivantes :

```
1  typedef struct {char x; char y;} point;
2  char  *i = malloc(sizeof(char)*5);
3  point *p = malloc(sizeof(point));
4  *i=1;
5  p->x=2; p->y=3;
```

suivie de :

```
6  i=(char*)p;
7  *i=0;
```

ou de :

```
6  p=(point*)i;
7  p->x=0;
```

Dans cet exemple, les variables `i` et `p` pointent sur le même emplacement mémoire après le cast, comme le montre la Figure 4.28, ce qui est difficilement représentable dans un modèle à la Burstall-Bornat où les emplacements mémoire des champs de la structure sont disjoints de celui où sont alloués les entiers, comme l'illustre la Figure 4.29. L'*alias* entre le pointeur `i` et

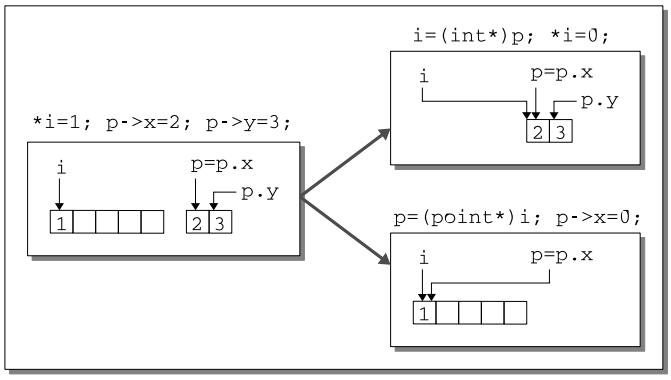


FIG. 4.28 – Représentation mémoire d'un cast entre un pointeur d'entier et un pointeur sur structure

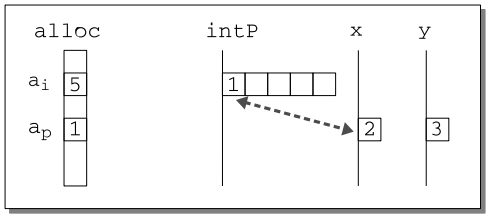


FIG. 4.29 – Illustration du problème d'un cast entre pointeur sur structure et pointeur sur entier dans le modèle à la Burstall-Bornat

le pointeur `p` permet de modifier le même emplacement mémoire tantôt par la variable `i`, i.e. en accédant au bloc mémoire en tant qu'entier, tantôt par la variable `p`, i.e. en y accédant en tant

que structure. Les casts sont, en ce sens, très proches des unions et, seront d'ailleurs interprétés par l'intermédiaire d'unions dans la solution que nous proposons en Section 4.5.4.

Notons qu'en plus de créer des alias entre des segments qui sont disjoints dans le modèle, les casts rendent la validité d'un accès à un pointeur plus difficile à établir. En effet, dans l'exemple ci-dessus, un accès à `i[2]` après l'affectation `i=(char*)p ;` ne devrait pas être valide. L'interprétation des casts devra prendre en compte ce problème de validité.

Enfin, l'arithmétique de pointeur sur une variable qui a été castée doit également être prise en compte. Par exemple, les casts peuvent permettre de modifier une variable structurée en modifiant chacun de ces bits un à un. Cette arithmétique de pointeur à l'intérieur d'une structure implique qu'il faut savoir à quel segment mémoire est liée l'adresse après sa modification. Par exemple, les instructions suivantes :

```
typedef struct {int a; point p; long l; short tab[10];} my_struct;
my_struct s;
int k;
char *c = (char*) my_struct;
for (k=0; k<sizeof(my_struct); k++){ *c=0; c++; }
```

permettent de mettre à zéro tous les champs de la structure sans se soucier de comment elle est formée. Ce problème vient de la vision *arborescente* d'une structure dans le modèle de Caduceus, alors que les champs sont stockés de façon séquentielle en mémoire, comme l'illustre la Figure 4.30.

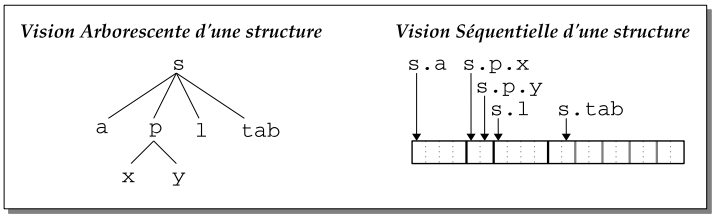


FIG. 4.30 – Différence de représentation d'une structure

Notons que ce problème est en fait indépendant des casts, comme le montre l'exemple suivant :

```
struct {char x; char y;} p;
char *i;
i=&p.x;
*i=1;
i++;
*i=2;
```

En effet, le déréférencement du champ `x` sera interprété dans Caduceus par une indirection (comme le montre la Figure 4.31) et l'accès au pointeur après son incrémentation ne sera pas valide dans le modèle. L'outil n'est pas incorrect dans ce cas, on ne pourra simplement pas prouver la validité du pointeur `i` après son incrémentation. Par conséquent, la précondition de l'accès à `*i` ne pourra pas être prouvée, identifiant ainsi une "menace".

4.4 Solution pour les structures

Rappelons que nous voulons ici corriger l'interprétation de Caduceus de l'affectation globale de structures et du passage de paramètres de type structure. Nous allons décrire la façon dont l'outil a été modifié pour cette correction. Plus précisément, l'outil Why ne sera pas touché,

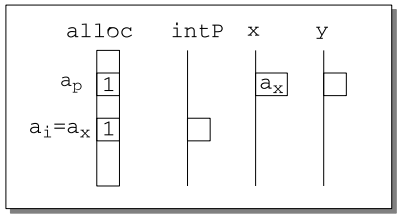


FIG. 4.31 – Interprétation d'un déréférencement de champ par une indirection

seule la traduction d'un programme C dans le langage d'entrée de Why sera corrigée (voir Section 4.2.1.2). La Figure 4.32 décrit les étapes de cette traduction et des exemples d'arbres de syntaxe abstraite (*ASA*) construits pendant la traduction. Ces exemples donnent une idée de la syntaxe des *ASA*, que nous utiliserons pour décrire les modifications de l'outil.

Les modifications que nous proposons se situent en général après l'étape de typage, pour bénéficier du calcul des types, et avant la phase de calcul des effets, pour que ceux-ci soient déterminés sur une version correcte de l'ASA.

Rappelons que les retours de fonction de type structure sont refusés, comme nous l'avons expliqué à la fin de la Section 4.3.2. L'utilisateur devra donc modifier son programme pour renvoyer le résultat dans une variable.

Notons enfin que l'interprétation de l'affectation et du passage de paramètre doit aussi être modifiée pour les types unions. Cependant, la modification est légèrement différente du fait de la particularité de la gestion des unions. Nous ferons une remarque sur ce point à la fin de l'explication de la gestion des unions.

4.4.1 Affectation

Pour une interprétation correcte des structures, il faut que l'affectation d'une expression d'un type structure soit traduite par une affectation de chaque champ de la structure. Par exemple, si `p` est une variable du type `point` déjà cité, alors l'interprétation de l'instruction `q=p;` devrait être remplacée par l'interprétation de la séquence d'instructions : `q.x=p.x; q.y=p.y;`. Plus généralement, l'instruction `e1=e2;` où `e1` et `e2` sont du même type structure devra être interprétée par l'affectation de chaque champ de cette structure. En termes d'arbre de syntaxe abstraite, cela signifie que le typage de `CEassign(e1,e2)` doit être remplacé par le typage de `CEassign(CEdot(e1,c),CEdot(e2,c))` pour chaque champ `c` de la structure.

Modification 1 (Affectation correcte des structures) L'étape de typage de l'outil Caduceus est modifiée de la façon suivante (résumée dans la figure 4.33) :

Lors du typage de l'expression `CEassign(e1,e2)` :

1. calculer le type `t1` de `e1` et le type `t2` de `e2` ;
2. si `t1` et `t2` sont le même type structure, alors :
   (a) calculer la liste `[c1 ; ... ; cn]` des champs de la structure ;
   (b) construire la liste `[a1 ; ... ; an]` des affectations de ces champs, où `ai` vaut :
   ```
   | CSexpr (CEassign(CEdot(e1,ci),CEdot(e2,ci))
   ```
   (c) lancer le typage sur le bloc d'instruction ainsi obtenu ;

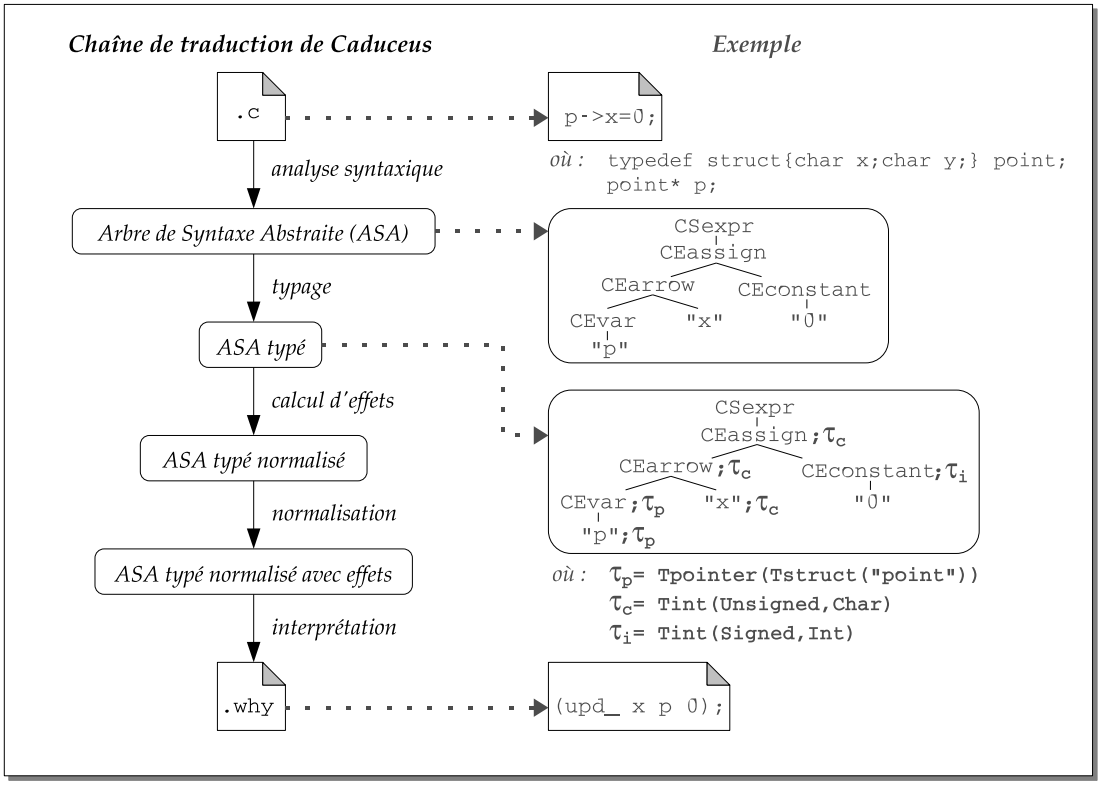


FIG. 4.32 – Représentation de la chaîne de traduction effectuée par Caduceus

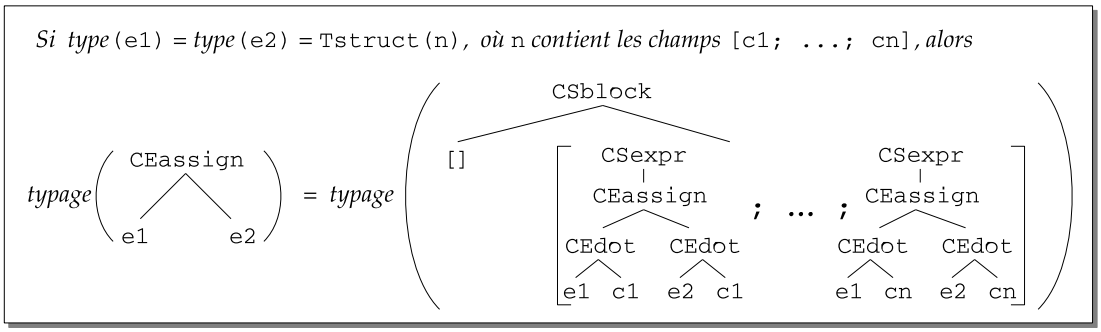


FIG. 4.33 – Typage de l'affectation d'une expression de type structure

3. sinon, conserver le typage initial des affectations de Caduceus (`TEassign(te1,te2)`, où `te1` et `te2` sont les expressions typées).

Il est à noter que cette implémentation gère également les structures imbriquées. En effet, si un champ `ci` est de nouveau de type structure, alors l'affectation `CEassign(CEdot(e1,ci), CEdot(e2,ci))` sera traduite par l'affectation de chaque champ de la structure en question, puisque l'expression `CEdot(e1,ci)` sera de type structure. Par ailleurs, il n'y a pas de problème de terminaison, puisque les structures récursives ne sont autorisées qu'à l'aide d'un pointeur, comme dans :

```
struct liste_chainee{
  int val;
  struct liste_chainee* suivant;
};
```

Une affectation `l1=l2 ;` sera traduite par l'interprétation des instructions `l1.val=l2.val ;` et `l1.suivant=l2.suivant ;`. Cette deuxième instruction correspond à une égalité de pointeur, et non pas une affectation d'une variable de type `struct liste_chainee`, qui aurait entraîné une nouvelle décomposition en affectations des champs `val` et `suivant`, et donc un problème de terminaison.

Restriction. Notons que cette modification n'est pas correcte si l'expression `e2` a des effets de bords. Par exemple, l'affectation `e1=e[i++] ;` où `e` est un tableau de structures, sera remplacé par un ensemble d'affectations de chacun des champs du type structure, où chaque affectation incrémentera l'index `i`. *les affectations à des valeurs qui ont des effets de bords ne sont donc pas acceptées.* Néanmoins, une simple transformation de programme (par exemple `j=i++ ;e1=e[j] ;`) permettrait de résoudre ce problème. C'est d'ailleurs ce qui est effectué par des outils de transformation de programmes C, comme *CIL* (voir [99]). Il est donc possible de s'appuyer sur des outils existants pour lever cette restriction.

Remarques. L'interprétation de la clause `assigns` doit également être modifiée. En effet, si cette clause contient une expressions `e` de type structure, elle doit être interprétée par le fait que chaque champ "final" (qui ne soit pas de type structure) est potentiellement modifié par la fonction.

4.4.2 Passage de paramètre

Le problème du passage de paramètre provient du fait que les structures sont interprétées par des pointeurs dans Caduceus. Par conséquent ce sont leurs adresses qui sont transmises aux fonctions, qui peuvent alors modifier la valeur des champs de la structure. Une manière de corriger cela est d'utiliser des variables locales pour chaque paramètre de type structure, de recopier la valeur de la structure dans la variable locale et de travailler sur la variable locale. Par exemple, l'interprétation de la fonction `void f(point p){p.x=0 ;}` sera remplacée par l'interprétation de `void f(point p){point local_p=p ; local_p.x=0 ;}`. Notons que l'initialisation `local_p=p ;` consiste bien en une recopie des arguments, grâce à la Modification 1.

Modification 2 (Passage par valeur des paramètres de type structure) L'étape de typage de l'outil Caduceus est modifiée de la façon suivante (résumée dans la figure 4.34) :

Lors du typage de `Cfundef(s,ty,f,pl,bl)` où `s` est la spécification de la fonction, `ty` le type de retour, `f` le nom de la fonction, `pl` la liste des arguments et `bl` le corps de la fonction :

1. calculer les types de chaque argument de la fonction et construire la sous-liste `[t1 a1 ;...;tn an]` des arguments de type structure ;
2. construire la liste `decls` des déclarations des variables locales :

   ```
   decls = [Cdecl(t1,"local_1") ;... ; Cdecl(tn,"local_n")]
   ```

3. construire la liste `init` des initialisations des variables locales :

   ```
   init = [CSexpr(CEassign(CEvar("local_1"),CEvar(a1))) ;... ;
           CSexpr(CEassign(CEvar("local_n"),CEvar(an)))]
   ```

4. effectuer la substitution dans le corps de la fonction :

   ```
   bl' = bl [ CEvar(ai) ← CEvar("local_i")]
   ```

5. procéder au typage de la fonction tel qu'il était défini dans la version courante de Caduceus, mais avec le corps de la fonction valant `CSblock(decls,init@bl')`.

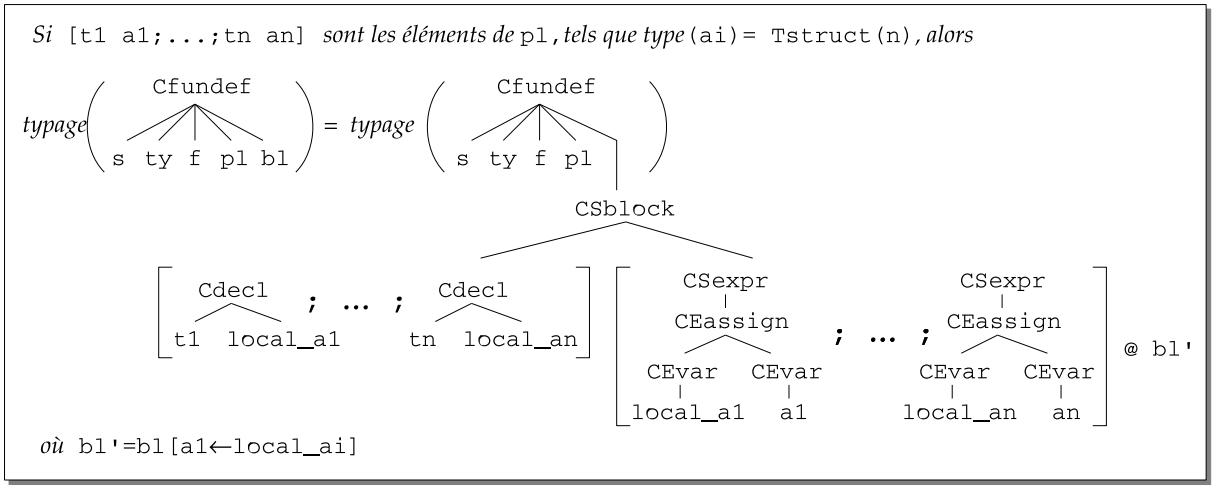


FIG. 4.34 – Typage de l'appel d'une fonction avec arguments de type structure

Remarque. Il est à noter, d'une part, que l'ordre d'évaluation est préservé, puisque l'évaluation des arguments a lieu au moment de l'appel de la fonction. Les variables `ai` sont donc déjà évaluées lors de l'affectation des variables `local_i`. D'autre part, la substitution dans le corps de la fonction peut être évitée, en subsitutant plutôt les noms des paramètres. Par exemple, l'interprétation de la fonction `void f(point p){p.x=0 ;}` serait plutôt remplacée par l'interprétation de la fonction `void f(point local_p){point p=local_p ; p.x=0 ;}`.

4.5 Solutions pour adapter le modèle à la Burstall-Bornat

Notre but est de proposer une modification d'un modèle à la Burstall-Bornat de manière à pouvoir représenter les constructions bas niveau du langage C, tout en préservant le plus possible les avantages de la séparation de la mémoire.

4.5.1 Idée générale

Le modèle mémoire à la Burstall-Bornat utilisé dans Caduceus (décrit dans la Section 4.2.1) ne permet pas de représenter les constructions de bas niveau du langage C telles que les unions ou les casts, comme nous l'avons expliqué aux Sections 4.3.3 et 4.3.4. Ces constructions représentent une manipulation bas niveau de la mémoire qui sont difficiles à formaliser, comme le montre la rareté des travaux sur le sujet. Nous proposons ici une manière d'adapter le modèle mémoire afin de pouvoir interpréter ces notions. Nous nous concentrons d'abord sur le cas des unions et décrirons ensuite comment adapter la solution au cas des casts.

Une première solution pour formaliser la représentation mémoire d'un type union est de considérer un modèle de bas niveau, qui correspond justement à ce qu'il se passe en mémoire. Autrement dit, les objets de type union sont considérés comme des tableaux d'entiers, et tous les accès à des champs de l'union sont remplacés par un calcul d'index dans le tableau, suivant la taille des différents champs. Le problème de cette approche est qu'elle n'est pas modulaire et que le modèle de bas niveau se "propage" de sorte que le modèle de la mémoire peut rapidement devenir un grand tableau de bits. En effet, prenons l'exemple suivant :

```
typedef struct {short s1; short s2;} spoint;
void f(spoint* p){p->s1==0;}
typedef struct {char c1; char c2; char c3; char c4;} cpoint;
typedef union  {spoint s; cpoint c;} une_union;
une_union u;
f(&u.s);
```

Avec l'approche d'un modèle bas niveau, l'union de cet exemple sera considérée comme un tableau de `char`. Mais alors la fonction `f` doit prendre en argument un tableau de `char` et non une structure, alors qu'elle a été définie complètement indépendamment de l'union. La modularité est donc perdue. Les avantages de la séparation du modèle à la Burstall-Bornat sont également perdus, puisque, par exemple, il faudra prouver que la fonction `f` ne modifie pas le champ `s2` de `p` qui est devenu un tableau.

Cette solution comporte les inconvénients en termes de lourdeur de la vérification que l'outil Caduceus voulait éviter en choisissant un modèle à la Burstall-Bornat.

Notre idée est donc d'adapter un modèle à la Burstall-Bornat pour la gestion des unions, tout en préservant au maximum les avantages de la séparation. Étant donné que les champs d'une union sont considérés comme dans des segments mémoires disjoints dans le modèle, alors qu'ils sont confondus dans la réalité, il faut définir artificiellement des *liens* entre ces segments, signifiant que les modifications d'un des segments doivent être répercutées sur les segments qui lui sont liés (qui correspondent aux autres champs de l'union). Comme nous l'avons vu dans les exemples de la Section 4.3.3, ce lien doit être établi non seulement dans le code source au moment de la modification d'un champ d'une union, mais également dans les annotations comportant une propriété sur un champ d'union. Nous voulons donc :

1. connaître, dans les annotations, le lien qui existe entre les différents champs d'une union ;
2. faire en sorte qu'une modification d'un champ d'une union entraîne la modification des autres champs de cette union.

Le premier objectif peut être atteint à l'aide de la définition d'un invariant global. Si l'on considère l'union `journal_state_registry` déjà présentée, cela revient à définir :

```
/*@ predicate journal_state_index_invariant (journal_state_index jsi){
      jsi.s.slotNb==f(jsi.w) &&
      jsi.s.byteNb==g(jsi.w)  &&
      jsi.w==h(jsi.s.slotNb,jsi.s.byteNb) } */

/*@ invariant always_journal_state_index_invariant :
      \forall journal_state_index jsi;
      journal_state_index_invariant(jsi) */
```

où les fonctions `f`, `g` et `h` définissent justement les liens entre les valeurs des champs `slotNb`, `byteNb` et `w`. La définition de cet invariant a pour conséquence d'ajouter les propriétés qui y sont définies à toutes les clauses d'annotations manipulant des variables de type `journal_state_registry`. Par exemple, si une fonction a pour précondition `/*@ requires jsi.w==0 */`, alors la propriété `always_journal_state_index_invariant(jsi)` sera ajoutée à cette précondition, ce qui permettra d'en déduire que `jsi.s.slotNb` est également nul à l'appel de la fonction (en supposant que `f(0)=0`). Si c'est la postcondition qui contient la propriété `/*@ ensures jsi.w==0 */`, comme dans l'exemple donné dans la Section 4.3.3 où le modèle à la Burstall-Bornat est incorrect, alors, de même, la propriété `always_journal_state_index_invariant(jsi)` sera ajoutée à la postcondition. Par conséquent, `jsi.s.slotNb` vaudra zéro à la sortie de la fonction et il sera donc impossible de prouver que sa valeur est restée à 3. Notons que si la fonction est implémentée, par exemple par l'unique instruction `jsi.w=0 ;` alors la postcondition devra être prouvée, ce qui ne pourra être fait qu'en répondant au deuxième objectif, plus délicat, qui consiste à répercuter les modifications d'un champ d'une union sur ses autres champs.

En d'autres termes, ce deuxième objectif revient à rétablir l'invariant à chaque modification d'un champ d'union (de manière, en fait, à avoir un invariant fort). En effet, pour que la modification d'un champ d'une union soit correcte, il faut la faire suivre d'une "re-synchronisation" de l'union, à savoir la mise à jour des autres champs de l'union. Par exemple, l'affectation `jsi->w=0 ;` doit être suivie de `jsi->s.slotNb=f(0) ;` et de `jsi->s.slotNb=g(0) ;` où les fonctions `f` et `g` sont les mêmes que dans l'invariant. Plus précisément, l'affectation du champ `w` sera suivie d'un appel à une fonction de synchronisation `(sync_w jsi)` qui suppose que le champ `w` est à jour et qui modifie les autres champs en fonction de la valeur de `w` (voir la Figure 4.35). Avec la syntaxe d'un programme C annoté, cette fonction aurait la forme suivante :

```
/*@ assigns jsi->s.slotNb, jsi->s.byteNb
  @ ensures journal_state_index_invariant(*jsi)  */
void sync_w (journal_state_index* jsi);
```

L'affectation est à présent correcte puisque la clause `assigns` assure que tous les champs de l'union ont été modifiés et la postcondition indique que l'invariant a été rétabli. La fonction de synchronisation permet de représenter le fait que les champs de l'union qui sont disjoints dans le modèle à la Burstall-Bornat sont en réalité confondus en mémoire et que leurs valeurs sont liées.

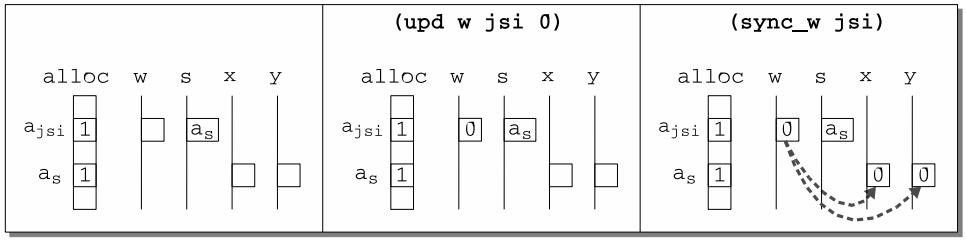


FIG. 4.35 – Rôle de la fonction de synchronisation dans le modèle à la Burstall-Bornat

Un premier problème, commun aux deux objectifs, provient du fait que la valeur d'un champ d'une union en fonction de la valeur d'un autre champ est très dépendante de l'implémentation. En effet, aussi bien la définition de l'invariant que celle de la fonction de synchronisation dépendent des fonctions `f` et `g` qui définissent la façon dont l'union est stockée en mémoire, ce qui est dépendant du compilateur et du processeur utilisés (comme nous l'avons expliqué dans la Section 4.3.1.1). Nous avons donc développé une bibliothèque permettant la définition de ces fonctions, présentée dans la Section 4.5.2.

Le deuxième problème, bien plus important et qui concerne la synchronisation, provient du fait qu'un champ d'une union peut être modifié autrement que par une affectation directe de ce champ, par l'intermédiaire de ses sous-structures lorsqu'elles sont utilisées ailleurs dans le programme. C'était le cas dans l'exemple présenté au début de cette section, ou dans l'exemple plus simple suivant :

```
typedef struct {char x; char y;} point;
void f(point* p){p->x=0;}
typedef union  {short s; point p;} mon_union;
mon_union u;
f(&u.p);
```

Dans cet exemple, l'appel à la fonction `f` sur `&u.p` doit également modifier le champ `s` de l'union `u`, or la fonction `f` a été définie complétement indépendamment de l'union. Donnons également cet autre exemple :

```
typedef union {int i; short j;} union1;
typedef union {int a; struct{char c1; char c2;} c; } union2;
union1 u1;
union2 u2;
int *n;
if (...) n=&u1.i; else n=&u2.a;
*n=0;
```

où la modification de `n` doit se répercuter soit sur le champ `j` de `u1`, soit sur les champs `c1` et `c2` de `u2`, et il n'est pas possible de savoir *statiquement* quelle est la mise à jour à faire (et donc de savoir quelle fonction de synchronisation doit être appelée). C'est pourquoi nous proposons l'utilisation d'une table *dynamique* pour se souvenir des liens à mettre à jour, comme nous l'expliquons dans la Section 4.5.3.

4.5.2 Dépendance vis-à-vis de l'implémentation

La façon dont la valeur d'un champ d'union dépend des autres champs est très dépendante du compilateur et du processeur. En effet, la taille en octets des types entiers de C, ou encore le type d'architecture (Little Endian ou Big Endian[1]), sont autant de critères qui rentrent en compte dans la façon dont une union est réellement stockée en mémoire.

Nous avons vu dans la Section 4.3.1.1 que, par exemple, l'ordre de stockage des champs de bits dans une structure dépend de l'architecture. Par conséquent, dans la définition du type `journal_state_index`, pour que l'incrémentation du champ `w` permette un parcours correct des index dans le tableau `JournalStateRegistry` (comme montré dans la Figure 4.17, page 119), le champ `slotNb` devra apparaître en premier si l'architecture est Little Endian, mais c'est `byteNb` qui doit être en premier si elle est Big Endian (voir Figure 4.36).

[1] Voir Conventions page 1.

```
typedef union {
     u2  w;
     struct {
       #ifdef _LITTLE_ENDIAN
         u2 slotNb   :JOURNAL_STATE_NB_BITS;
         u2 byteNb   :8*sizeof(u2)-JOURNAL_STATE_NB_BITS;
       #else
         u2 byteNb   :8*sizeof(u2)-JOURNAL_STATE_NB_BITS;
         u2 slotNb   :JOURNAL_STATE_NB_BITS;
       #endif
     } s;
} journal_state_index;
```

FIG. 4.36 – Influence de l'architecture du compilateur sur la disposition des champs

Les fonctions qui définissent les liens entre les champs de l'union doivent donc être soit définies par l'utilisateur, soit générées à partir des caractéristiques du compilateur et du processeur.

Nous avons développé, dans cet objectif, une bibliothèque utilisable pour la définition de ces fonctions, soit par l'utilisateur, soit par la génération automatique. Les calculs dans ces fonctions se ressemblent sensiblement, quelque soit l'union considérée. En effet, il s'agit toujours d'extraire plusieurs octets, ou bits, d'un ensemble d'octets. Ce genre de calculs peut être difficilement définissable dans le langage d'annotations, alors qu'il peut être facilement représentable dans un langage d'ordre supérieur comme celui de Coq. Nous avons donc développé une bibliothèque Coq permettant de représenter cette extraction de bits. Cette bibliothèque fournit les fonctions suivantes (voir Figure 4.37 pour des exemples) :

- `keep_tail : Z->Z->Z` : `(keep_tail z n)` rend les `n` bits de poids faible de la valeur de `z`
- `trunc_tail : Z->Z->Z` : `(trunc_tail z n)` rend la valeur de `z` à laquelle on a tronqué les `n` bits de poids faible.
- `add_tail : Z->-Z->Z` : `(add_tail z n)` rend `z` multiplié par 2^n.

```
  z = | aaaa aaaa | bbbb bbbb | cccc cccc | dddd dddd |

keep_tail(z,2*8)  = | aaaa aaaa | bbbb bbbb |
trunc_tail(z,3*8) = | dddd dddd |
add_tail(trunc_tail(z,3*8),1*8) = | 0000 0000 | dddd dddd |
```

FIG. 4.37 – Exemples d'utilisation des fonctions `keep_tail`, `trunc_tail` et `add_tail`

Ces fonctions vérifient la relation :

```
z = (keep_tail z n) + (add_tail (trunc_tail z n) n)
```

Ces trois fonctions suffisent à définir les fonctions de mise à jour des différents champs d'une union, même en présence de champs de bits. Par exemple, pour notre exemple type, on a :

```
jsi.s.slotNb==keep_tail(jsi.w,14)
jsi.s.byteNb==trunc_tail(jsi.w,2)
jsi.w==jsi.s.slotNb + add_tail(jsi.s.byteNb,2)
```

Ces fonctions de manipulation de bits peuvent être déclarées dans le langage d'annotation (bien que définies dans le langage de preuve) pour pouvoir être utilisées par le programmeur pour

définir les fonctions de mise à jour des unions qu'il définit (sous la forme de fonction C annotée, comme décrit dans la section précédente). Elles peuvent également être utilisées directement dans la partie logique pour une génération automatique des fonctions à partir des caractéristiques du compilateur et du processeur (la fonction de synchronisation est alors directement définie dans le langage de Why).

4.5.3 Modèle de Burstall-Bornat avec liens dynamiques

L'idée générale, comme nous l'avons dit, est de re-synchroniser les champs d'une union chaque fois que l'un d'entre eux a été modifié. Pour cela, chaque mise à jour d'un champ d'une union est suivie d'un appel à une fonction de synchronisation : toute affectation `e1.x=e2`, où `e1` est de type union, est interprétée par `(upd x e1 e2)` (comme avant), mais suivi d'un appel à la fonction `(sync_x e1)`, qui signifie qu'il faut mettre à jour tous les champs de l'union `e1` autres que `x`, en fonction de la valeur du champ `x` qui, elle, est à jour. Cette fonction n'est pas instanciée, seule sa spécification est donnée. Elle indique que les champs de l'union (autres que `x`) sont modifiés et donne leurs valeurs en fonction du champ `x` (grâce aux fonctions de la bibliothèque présentée dans la section précédente) :

> Si e_1 est de type `U`, où `U` est défini par `typedef union {`τ_1 c_1 `; ... ;` τ_n c_n `;} U ;`
> et si e_2 est de type τ_i, alors l'affectation e_1`.`c_i`=`e_2 est interprétée par :
>
> `{(upd x` e_1 e_2`) ; (sync_`c_i e_1`) ;}`
>
> où la fonction de synchronisation aurait la spécification suivante dans le langage de spécification de Caduceus :
>
> `/*@ assigns` e_1`.`c_1`, ...,` e_1`.`c_{i-1}`,` e_1`.`c_{i+1}`,...,` e_1`.`c_n
> `@ ensures U_invariant(`e_1`) */`
> `void sync_`c_i `(U` e_1`) ;`
>
> où le prédicat `U_invariant` décrit les liens entre les différents champs de l'union `U`.

Comme nous l'avons vu, cette solution ne suffit pas puisqu'une modification d'une sous-structure de l'union nécessite également la synchronisation des champs de l'union. L'exemple le plus simple est le suivant :

```
typedef struct {char x; char y;} point;
typedef union  {short s; point p;} mon_union;
mon_union u;
point* q = &u.p;
q->x=0;
```

Dans ce cas, il faut se souvenir que `q` est une sous-structure d'union, qui modifie donc les autres champs de l'union. Mais nous avons vu que la sous-structure peut également être modifiée dans une fonction définie de façon indépendante. En effet, si une fonction `f` prend en argument un pointeur sur un `point` et en modifie le champ `x`, alors lorsque cette fonction est appelée avec l'adresse `&u.p` du champ, les autres champs de l'union doivent être modifiés. Cela signifie qu'il faut savoir, lors de la modification du champ `p.x` dans le corps de la fonction `f`, si `p` est une sous-structure d'une union. Si ce n'est pas le cas, alors la modification peut être faite normalement. En revanche, si c'est le cas, il fait savoir à quelle union elle est liée et par quelle champ elle a été accédée, pour pouvoir mettre les autres champs de l'union à jour. Par conséquent, toute modification d'un champ `e1.x=e2` doit être suivie d'une synchronisation non seulement lorsque `e1` est une union, mais également lorsque `e1` est une adresse *liée* à une union par un champ donné. De plus, cela s'applique également aux mises à jours des valeurs pointées par des variables. En effet, si une fonction modifie la valeur pointée par un de ses arguments de type `char*`, alors elle

peut être appelée avec `&u.p.x`. Dans ce cas aussi une synchronisation est nécessaire pour mettre à jour le champ `u.s`.

La difficulté majeure, comme nous l'avons évoqué, provient du fait que ce *lien* entre une adresse et l'union à laquelle elle peut appartenir est *dynamique*. Il ne peut donc être mémorisé que dans une table dynamique, que nous noterons `link`, à l'image de la table d'allocation `alloc_table` qui mémorise de façon dynamique la taille des emplacements mémoire alloués. Chaque fois qu'une expression `e` est liée à une expression `u` de type union par le champ `c`, la paire `(u,c)` est ajoutée à la table `link` (ou remplace la paire déjà contenue dans la table si `e` était déjà liée). En particulier, les expressions qui n'apparaissent pas dans la table sont libres. La table est alors utilisée comme suit.

> L'affectation `e1.x=e2` est interprétée par `(upd x e1 e2)`, suivi de l'appel `sync(u,l)` si `link(e1)=(u,l)` ou de l'appel `sync(e1,x)` si `e1` est de type union. De même, `*e1=e2`, où `e1` est un pointeur d'entier, est interprété par `(upd intP e1 e2)`, suivi de `sync(u,l)` si `link(e1)=(u,l)`.

Par exemple, lors de la déclaration d'une variable `u` de type `mon_union`, l'expression `u.p` sera associée à `(u,p)` dans la table `link`, l'expression `u.p.x` sera associée à `(u,p)`, etc. Ainsi, si une variable `i`, de type `char*` prend la valeur `&u.p.x`, alors la modification `*i=e` de la valeur pointée sera suivie de l'appel à la fonction de synchronisation `(sync u p)` puisque `i` sera liée à `(u,p)`.

Une remarque importante est que le champ qui doit être modifié ne peut pas être connu statiquement, ce qui implique que la fonction de synchronisation prend en argument le champ modifié (on a `sync(u,p)` au lieu de `sync_p(u)`). Il n'y a donc plus qu'une seule fonction de synchronisation, qui prend en argument n'importe quelle variable de type union[1] et n'importe quel champ. La postcondition est donc définie par une analyse par cas sur le champ reçu en entrée. Mais surtout, cette fonction modifie potentiellement *tous* les champs de *toutes* les unions, et, par propagation, c'est le cas de toutes les fonctions modifiant le champ d'une structure ou la valeur d'un pointeur.

Parmi les différentes solutions que nous avons examinées, c'est la seule qui permette une interprétation correcte des structures, en conservant un modèle mémoire à la Burstall-Bornat. Cependant, elle va alourdir considérablement la phase de vérification, puisque la propagation des effets de bords de la fonction de synchronisation réduit fortement les avantages de la séparation en segments disjoints. Mais dans une grande partie des cas (comme dans notre cas d'étude), une analyse statique permettrait de déterminer, le plus finement possible, les champs d'unions qui doivent être mis à jour dans la fonction de synchronisation, réduisant ainsi ses effets de bords.

L'analyse statique peut agir selon deux axes : restreindre l'ensemble des champs à analyser et retarder la synchronisation lorsque c'est possible.

Le premier axe consiste à analyser tous les appels de fonction du programme[2] dans le but de minimiser la "pollution" du corps des fonctions. Dans le cas où une fonction n'est jamais appelée avec l'adresse d'un champ d'une variable de type union en argument, la fonction peut être interprétée sans analyse de lien puisqu'elle sera toujours appelée avec des arguments *libres* (non

[1] La fonction de synchronisation sera définie directement dans le langage d'entrée de Why, où tous les types unions correspondent au type pointeur, ce qui implique qu'une telle fonction est correctement typée.

[2] L'analyse statique est donc non modulaire.

liés à une union). Concernant les cas où des appels sont faits avec des arguments liés, l'analyse statique permet de calculer, pour chaque argument d'une fonction donnée, l'ensemble des champs d'unions auxquels cet argument est potentiellement lié. Lors de la modification de cet argument dans le corps de la fonction, cet ensemble peut être utilisé pour réduire les champs analysés par la fonction de synchronisation. Les effets de bords de la synchronisation pourront donc être réduits à cet ensemble. Il s'agit bien-sûr d'un sur-ensemble, qui peut éventuellement valoir l'ensemble de tous les champs de toutes les unions s'il n'est pas possible de réduire les cas statiquement. Autrement dit, pour toute fonction `f`, et pour tout argument `e` de la fonction, l'analyse statique détermine l'ensemble `poss_link(e)={c1,...,cn}` des champs d'union possiblement liés à cet argument lors d'un appel à la fonction. Puis la fonction de synchronisation `sync(u,c)` pourra prendre en compte le fait que `c` est forcément l'un des `ci`.

Il est également possible de définir deux versions de la fonction, une version *libre* et une version *liée*, pour ne pas polluer les appels de la fonction lorsque tous les arguments sont libres et ne nécessitent pas de synchronisation.

Notons que dans notre étude de cas, toutes les fonctions étaient appelées avec des arguments libres. C'est entre autres pour cela que l'implémentation d'une solution où le lien est connu statiquement était suffisante.

Le deuxième axe consiste à constater que la synchronisation peut être retardée dans certains cas. Dans notre exemple de la fonction `f` modifiant le champ `p.x`, il est possible de synchroniser l'union (par l'appel à `sync(u,p)`) uniquement après l'appel `f(&u.p)`, étant donné que le champ `u.s` n'a pas été lu avant cette synchronisation (i.e. dans le corps de la fonction). Cette approche revient à comparer une interprétation *défensive* où chaque modification serait suivie d'une synchronisation et une interprétation *offensive* où la synchronisation suivrait l'appel de la fonction :

```
T f_defensive(...,t e,...){... e.l=v; sync(link(e)); ...}   ...  f(&u.c);
T f_offensive(...,t e,...){... e.l=v; ...}                  ...  f(&u.c); sync(u,c);
```

La fonction de synchronisation a pour rôle de rétablir l'invariant assurant les liens entre les différents champs de l'union. Lorsqu'elle est retardée, l'invariant n'est pas vérifié jusqu'à son appel, et les valeurs des autres champs sont donc incohérentes. Par conséquent, la synchronisation ne peut être retardée que si la fonction ne lit (et *a fortiori* ne modifie) aucun des autres champs de l'union (dont la valeur est incohérente). L'analyse des effets de la fonction peut servir à identifier tous les champs potentiellement lus par la fonction. Plus précisément, soient `{d1,...,dp}` les variables Why lues (et modifiées) par la fonction, donné par le calcul d'effets de Caduceus. Chacune de ces expressions peut contenir une valeur gauche, pour laquelle l'analyse statique a calculé un ensemble de champs d'union auxquels elle est potentiellement liée. L'union $\mathcal{U}$ de tous ces ensembles contient donc tous les champs d'union potentiellement lus par la fonction. Dès lors, l'affectation `e.l=v ;` (ou `*e=v ;`), où `poss_link(e)={c1,...,cn}`, ne peut être "post-synchronisée" que si aucun `ci` n'appartient à la même union qu'un des champs de $\mathcal{U}$. Par exemple, dans le programme :

```
mon_union u;
int f(point *q){q->x=0; return u.s;}
...
f(&u.p);
```

l'affectation `q->x=0 ;` doit être immédiatement suivie de sa synchronisation, pour que la valeur de `u.s` soit cohérente. En effet, l'analyse statique indiquerait que l'argument de `f` est potentiellement lié au champ `p`. Or, la fonction lit l'expression `u.s`, dont l'analyse statique dirait qu'elle est liée au champ `s`. Par conséquent, étant donné que les champs `p` et `s` appartiennent tous deux à l'union

`mon_union`, la synchronisation ne peut être retardée. Dans cet autre exemple :

```
void g(point*q, short* t){q->x=0; *t=1;}
mon_union u;
g(&u.p,&u.s);
```

la fonction lit `t`, qui est potentiellement lié au champ `s`, donc l'affectation `q.x=0;`, où `q` est potentiellement lié au champ `p` (qui appartient à la même union que `s`) ne peut être post-synchronisée non plus.

En conclusion, une utilisation de cette solution d'interprétation des unions dans le modèle à la Burstall-Bornat, à l'aide d'une table dynamique de liens, peut être accompagnée d'une analyse statique permettant de réduire la "pollution" non nécessaire de l'interprétation. En effet, une utilisation naïve aurait des inconvénients comparables à ceux d'une approche bas-niveau. Plus l'analyse statique est fine, moins l'interprétation est polluée.

Une implémentation de la solution a été développée dans le cas où le lien entre la sous-structure et l'union est connu statiquement. Ce cas de figure est celui qui apparait dans le code source du système d'exploitation que l'on a étudié (voir Chapitre 6). Cette implémentation permet donc de vérifier notre cas d'étude.

Remarque. Comme nous l'avons évoqué dans la Section 4.4, la modification de l'interprétation de l'affectation et du passage de paramètre d'expressions de type structure doit être également appliquée aux expressions de type union. Par exemple, une affectation globale `mon_union u = v;` doit être interprétée par l'affectation des champs de l'union. Toutefois, dans le cas des unions, l'affectation d'un seul champ suffit puisqu'elle sera suivie d'une synchronisation qui mettra à jour les autres champs. Une autre possibilité est de mettre à jour tous les champs, comme dans le cas des structures, mais sans faire de synchronisation.

Expliquons à présent comment utiliser la solution présentée dans la section précédente pour l'interprétation des casts.

4.5.4 Casts

Commençons par noter que les casts d'entiers peuvent être traités simplement par une conversion consistant à tronquer certains octets d'un entier ou, au contraire, à ajouter des octets nuls à un entier. Les fonctions définies dans la Section 4.5.2 proposent justement l'extraction ou l'ajout de bits à un entier donné et peuvent donc être utilisées pour cette conversion. Par exemple, le cast `(char)i`, où `i` est de type `int` peut être interprété par `keep_tail(i,8)`. Inversement, le cast `(int)c`, où `c` est de type `char` peut être interprété par `add_tail(c,8*3)`. Plus généralement, si `e` est de type τ, alors le cast `(`τ'`)e` est interprété soit par `keep_tail(i,8*nb_of_byte_in_`τ'`)` si la taille du type τ' est plus petite que celle du type τ, soit par `add_tail(i,8*nb_of_byte_in_`τ' `- nb_of_byte_in_`τ`)` dans le cas contraire. Le nombre d'octets des différentes types entiers doit bien sûr être calculé en fonction des caractéristiques du compilateur utilisé.

Montrons à présent comment les casts de pointeurs peuvent être interprétés à l'aide d'unions, même en présence d'arithmétique de pointeur, et ceci qu'il s'agisse de casts n'impliquant que des entiers ou de casts comportant des structures. La première remarque est que l'on se ramène au cas où le cast est contenu dans une affectation `e1=(`τ_1`*)e2` entre deux variables, quitte à introduire des nouvelles variables pour gérer des expressions comme `((char*)i) + 4`.

L'idée générale est d'interpréter une affectation `e1=(`τ_1`*)e2`, où `e1` est de type τ_1`*` et `e2` de type τ_2`*`, à l'aide d'une union où le premier champ est de type τ_1`*` et le deuxième est de type τ_2`*`. Plus précisément, au lieu de faire pointer les deux variables `e1` et `e2` dans le même bloc mémoire, qui doit gérer deux arithmétiques différentes, on considère deux blocs distincts, un pour chaque variable, avec son arithmétique propre, et l'*alias* entre les deux variables est assuré par la fonction de synchronisation. Par exemple, considérons les instructions suivantes :

```
char *c = malloc(sizeof(char)*12);
int *i;
i = (int*)c;
```

Notre approche est de considérer un bloc mémoire pour la variable `c` (le bloc qui a été alloué) et un nouveau bloc, que l'on suppose alloué, pour la variable `i`, comme l'illustre la Figure 4.38. Le lien entre les deux blocs est assuré par l'utilisation d'une union :

```
union { char *cc; int* ii; } u;
```

Le champ `u.cc` de cette union est initialisé avec le pointeur `c`. Ceci a pour effet, d'une part, de rendre le pointeur `u.cc` valide sur un intervalle de taille 12 et, d'autre part, de mettre à jour le champ `u.ii`, par l'intermédiaire de l'appel à la fonction de synchronisation. Le champ `u.ii` doit pour cela être supposé valide, sur un intervalle de taille 3. Plus précisément, on suppose que ce champ, qui est de type pointeur, a été alloué :

`alloc_stack(u.ii, alloc1, alloc2)`

où `alloc1` est la table avant l'allocation de `u.ii` et `alloc2` celle après cette allocation. La table `alloc2` associe à l'addresse de base de `u.ii` la taille du bloc où le champ a été alloué. Ce bloc doit contenir toutes les valeurs du bloc contenant `c` mais converties en type `int`, donc il faut supposer que :

`block_length(alloc2,u.ii)=block_length(alloc1,c)*sizeof(c)/sizeof(u.ii)`

Puis l'affectation du champ `i` qui utilise le cast est remplacée par l'affectation `i=u.ii ;`. En d'autres termes, la modification consiste à remplacer l'affectation `i=(int*)c ;` par :

```
union { char *cc; int* ii; } u;
u.cc=c;
<hypothèses sur l'allocation de u.ii>
i=u.ii;
```

Notons que cette transformation est purement locale et que le reste du programme est inchangé. En effet, à la suite de ces instructions, toutes les modifications de la valeur pointée par `c` correspondent à une modification du champ `u.cc`, qui implique une synchronisation du champ `u.ii`, qui à son tour modifie la valeur pointée par `i`. Et inversement, les mises à jour de la valeur pointée par `i` sont répercutées sur celle pointée par `c` (voir Figure 4.38).

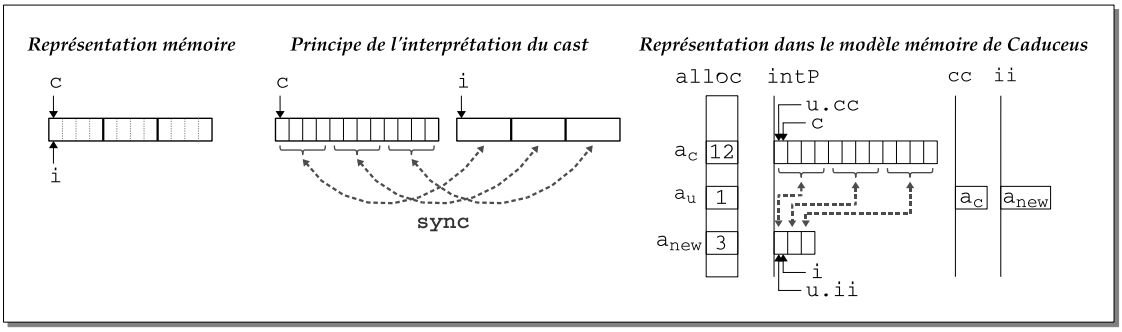


FIG. 4.38 – Approche pour l'interprétation d'un Cast de Pointeurs

Étudions à présent ce qu'il se passe dans le cas d'arithmétique de pointeur. Étant donné que le pointeur `c` a été alloué sur un bloc de taille 12, il peut être incrémenté et décrémenté à l'intérieur de ce bloc, pour en modifier ou en lire les valeurs. C'est pourquoi le champ `u.cc` de l'union ne représente pas uniquement la valeur pointée par `c` au moment du cast, mais *toutes* les valeurs du bloc où `c` a été alloué. Par conséquent, le champ `u.cc` est initialisé non pas au pointeur `c` mais plus exactement au début du bloc contenant `c` (au cas où `c` aurait été incrémenté avant le cast, comme dans le prochain exemple). Autrement dit, le champ `u.cc` est initialisé au pointeur `c-offset(c)` (ce qui entraîne que `base_addr(u.cc)=base_addr(c)` et `offset(u.cc)=0`). De la même façon, le champ `u.ii` pointe sur le début du nouveau bloc, ce qui signifie qu'il faut supposer que `offset(u.ii)=0`. Le pointeur `i` n'est alors pas initialisé au pointeur `u.ii` mais à l'index correspondant au décalage de `c`, converti en type `int` :

```
i = u.ii[offset(c)*sizeof(c)/sizeof(u.ii)]
```

Et enfin la fonction de synchronisation met à jour *toutes* les valeurs du bloc. Par exemple, la fonction de synchronisation `sync(cc,u)` aura une postcondition de la forme :

`\forall int k ; \valid(u.ii,k) -> u.ii[k]==`$\sum_{j=0}^{3}$ `u.cc[j+4*k]*`2^{8*j}

A titre d'exemple, considérons les instructions :

```
1  int k;
2  char *c = malloc(sizeof(char)*12);
3  int *i;
4  for (k=0;k<12;k++) { c[k]=0; }
5  c = c+8;
6  i = (int*)c;
7  c = c-1;
8  i = i-1;
9  *c=1;
```

Dans cet exemple, la valeur pointée par `i` à la fin de l'exécution des instructions est 2^{24}, comme le montre la représentation mémoire de la Figure 4.39.

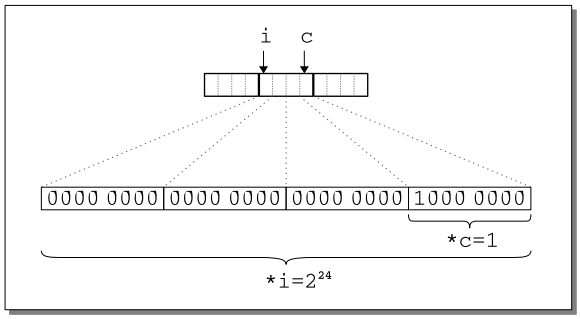


FIG. 4.39 – Représentation mémoire d'un cast avec arithmétique de pointeur

Avec l'interprétation des casts qui nous proposons, l'affectation de la ligne 6 est remplacée par la déclaration d'une union `u` du même type que précédemment, où le champ `u.cc` est initialisé à `c-8`. Cette initialisation a pour conséquence d'appeler la fonction `sync(cc,u)` qui assure, dans notre cas, que `u.ii[0]=u.ii[1]=u.ii[2]=0` puisque tous les octets du bloc ont été mis à zéro à la ligne 4. Puis, le pointeur `i` est initialisé à `u.ii[8*1/4]` soit `u.ii[2]`. L'état du modèle mémoire de Caduceus après la ligne 6 est représenté dans la Figure 4.40.

Puis, l'affectation de la ligne 9 est suivie d'une synchronisation de l'union puisque la variable `c` est liée à l'union `u` par le champ `cc`. Cette synchronisation assure, entre autre, que :

`*i = u.ii[offset(i)] = u.ii[1] = u.cc[4]+u.cc[5]*`2^8`+u.cc[1]*`2^{16}`+u.cc[3]*`2^{24} = 2^{24}

Cette interprétation des casts permet donc de prouver que `*i` vaut 2^{24}, comme le montre la représentation mémoire de la Figure 4.40.

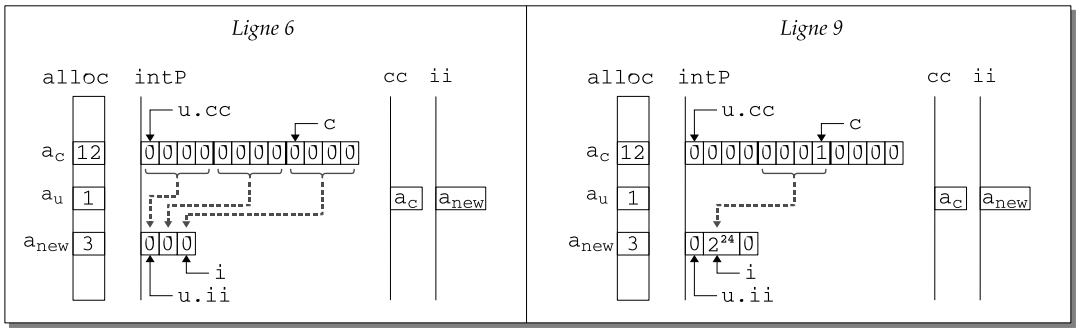


FIG. 4.40 – Modèle mémoire lors de l'interprétation d'un cast avec arithmétique de pointeur

Interprétation des casts dans le cas général

Toute affectation `e1=(`τ_1`*)e2`, où `e1` est de type τ_1`*` et `e2` de type τ_2`*`, est interprétée par :

```
union { τ1* p1 ; τ2* p2 ; } u ;
u.p2 = e2-offset(e2) ;
e1 = u.p1[offset(e2)*sizeof(τ2)/sizeof(τ1)] ;
```

avec l'ajout des propriétés suivantes :

```
alloc_stack(u.p1, alloc1, alloc2)
block_length(alloc2,u.p1)=block_length(alloc1,e2)*sizeof(τ2)/sizeof(τ1)
offset(u.p1)=0
```

Les trois instructions sont données dans une syntaxe C pour plus de lisibilité, mais elles seront définies en Why, comme l'indique l'utilisation des prédicats `offset` et `sizeof`. Ce dernier prédicat n'est d'ailleurs pas défini dans l'outil. Dans le cas des entiers, il correspond à la constante `nb_of_byte_in_`τ évoquée au début de la section. De manière générale, il correspond à la taille du type donné en argument, en nombre d'octets, et doit être calculé en fonction des caractéristiques du compilateur utilisé.

Restriction. La solution proposée pour interpréter un cast `e1=(`τ_1`*)e2`, où la taille de τ_1 est plus grande que celle de τ_2, *sans en être un multiple*, ne permet pas de représenter le dernier octet de `e1`. Autrement dit, la valeur pointée par `e1` est "tronquée" pour avoir la taille `block_length(e2)*sizeof(`τ_2`)/sizeof(`τ_1`)`.

En effet, si la taille du bloc alloué par le pointeur `c` n'est pas un multiple de 4, alors les octets dont l'index dépasse le plus grand multiple de 4 inférieur à la taille du bloc, ne pourront pas être accédés. Par exemple, si le bloc contenant `c` est de taille 5 (comme dans l'exemple donné dans les limitations du modèle mémoire, page 125, et redéfini ci-dessous), alors le champ `u.ii` de l'union sera dans un bloc de taille 1. Il sera donc impossible d'accéder à `u.ii[1]`. Cela parait logique puisque seul le premier octet de cet entier est défini. Cependant si un nouveau cast retransforme cet entier en type `char`, il devrait être possible d'accéder à sa valeur. L'exemple donné page 125 était le suivant :

```
char* d;
char tab[5]={1,2,3,4,5};
int *i=(int *)tab;
i++;
d = (char*)i;
```

où `*d` doit valoir 5 à la fin de l'exécution. Dans cet exemple, une première union `u` sera utilisée pour le premier cast. Cette union a le même type que précédemment, sauf qu'ici le bloc contenant `u.ii` n'est que de taille 1. Par conséquent, la variable `i`, après incrémentation, pointe en dehors du bloc (dans le modèle). Or, le deuxième cast utilise une autre union :

```
union { int *jj; char* dd; } v;
```

où `v.jj` est initialisé à `i-1` et `d` à `v.dd[4]` qui n'est pas valide. L'accès `*d` sera donc considéré comme non valide dans le modèle.

Casts impliquant des structures. A titre d'illustration, montrons que la solution donnée permet de gérer également les casts faisant intervenir un type structure. Pour cela, prenons l'exemple suivant :

```
1  int k;
2  typedef struct {char x; char y;} point;
3  point *p = malloc(sizeof(point)*4);
4  int *i;
5  for (k=0;k<4;k++) {p[k].x=2*k+1; p[k].y=2*k+2;}
6  i = (int*) p;
7  *i=0;
```

dont la représentation en mémoire, à la ligne 6, est donnée dans la Figure 4.41. Avec la solution proposée pour l'interprétation des casts, l'instruction d'affectation de la ligne 6 sera interprétée par (illustré dans la Figure 4.42) :

```
union { int* p1 ; point* p2 ; } u ;
u.p2 = p ;                              (puisque offset(p) vaut 0)
i = u.p1[0] ;
```

où le nouveau bloc contenant `u.p1` est de taille 2. L'affectation `u.p2=p ;` fait appel à la fonction de synchronisation qui met à jour toutes les valeurs du bloc de `u.p1`, à savoir `u.p1[0]` (qui est également la valeur pointée par `i`) et `u.p1[0]` (qui correspond à la valeur pointée par `i+1`). Puis l'affectation de la ligne 7 de notre exemple est suivie, dans le modèle, par un appel à `sync(u,p1)` qui, dans notre cas, met à jour les champs de `u.p2[0]` et `u.p2[1]`. Il est donc possible de démontrer qu'à la fin de l'exécution, les champs `p[0].x`, `p[0].y`, `p[1].x` et `p[1].y` sont tous nuls.

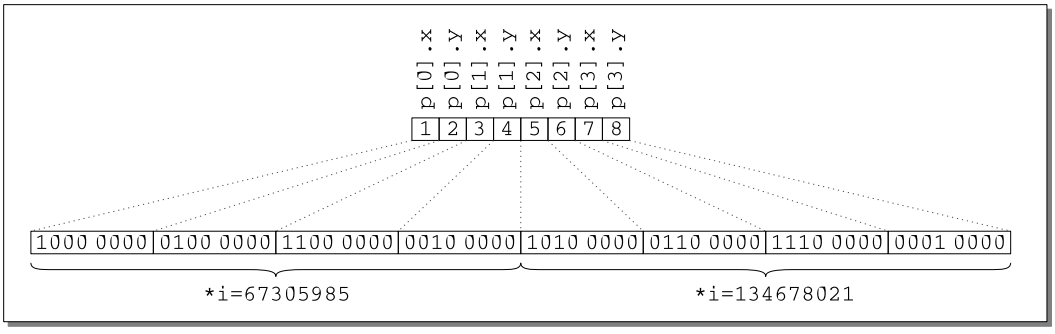


FIG. 4.41 – Représentation mémoire d'un cast faisant intervenir une structure

Notons que la Figure 4.42 met bien en évidence que le pointeur `i` peut être incrémenté pour mettre à zéro les champs des autres points, ce qui prouve que la solution marche également dans

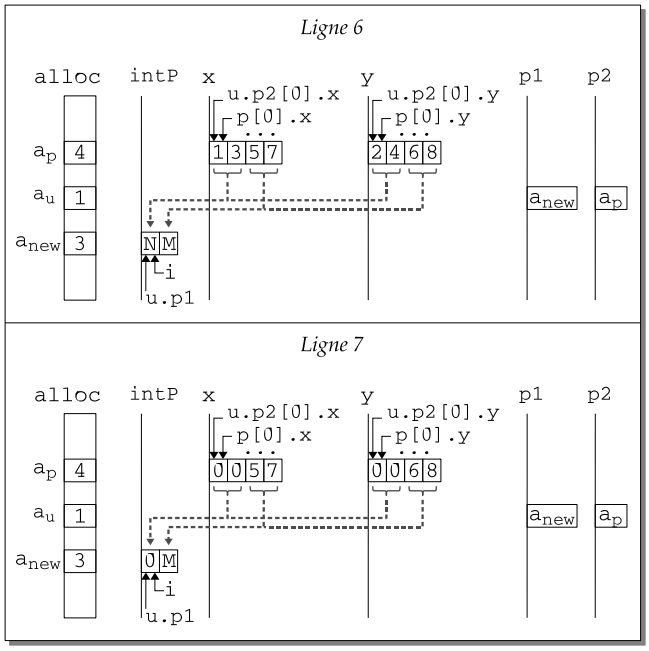


FIG. 4.42 – Modèle mémoire lors de l'interprétation d'un cast faisant intervenir une structure

les cas d'utilisation décrits dans la Section 4.3.4, où un pointeur sur une structure est casté en un pointeur d'entier pour la mise à zéro de tous ses champs.

Quant à l'exemple où une telle utilisation est faite sans passer par des casts, comme dans :

```
point p;
char* i = &p.x;
*i=0;
i++;
*i=0;
```

elle peut être traitée par le cast `char* i = (char*)&p.x ;`, qui n'aurait aucun effet en C (puisque l'expression `&p.x` est déjà de type `char*`), mais qui ici rend valide l'accès à `i` après son incrémentation. Par ailleurs, si `i` pointe directement à l'intérieur de la structure, avec `char* i = &p.y ;`, alors une variable supplémentaire sera utilisée pour remplacer cette affectation par `char* j = (char*)&p.x ; i=j+1 ;`.

4.6 Arrachage

Étant donné que nous nous intéressons à la vérification de programmes qui sont embarqués dans des cartes à puce, il est important de pouvoir modéliser les propriétés du programme en cas d'interruption soudaine de son exécution. En effet, comme nous l'avons souligné dans la Section 1.1.5.3, le retrait prématuré de la carte du terminal, appelé un *arrachage*, doit être considéré comme un comportement possible du programme. Or, la carte n'étant alimentée que par l'intermédiaire du terminal, son retrait provoque sa mise hors tension et l'interruption du programme qui était en cours d'exécution. La spécification d'un programme embarqué sur une carte à puce doit donc préciser son comportement en cas d'interruption. Dès lors, la vérification de la correction du programme vis-à-vis de sa spécification doit inclure la preuve que le programme

a le comportement attendu lors d'une interruption soudaine. Cette vérification nécessite, d'une part, la modélisation de l'interruption du programme et, d'autre part, la possibilité de spécifier formellement le comportement du programme en cas d'interruption. L'outil Caduceus, que nous avons utilisé pour la vérification fonctionnelle du code source d'un système d'exploitation embarqué, n'est pas développé dans un objectif de vérification de programmes de carte à puce. Par conséquent, il ne permet pas de modéliser et de vérifier les propriétés d'un programme en cas d'interruption de son exécution. Cette section propose cette extension de l'outil Caduceus.

L'interruption d'un programme est un changement soudain du flot de contrôle, qui provoque la sortie de la fonction et qui doit être *propagé* dans les fonctions appelantes (si une fonction appelée est interrompue, la fonction appelante l'est aussi). Cette description correspond exactement à la sémantique des *exceptions*. C'est pourquoi une interruption peut être modélisée par une exception, qui ne serait jamais rattrapée. Plus précisément, une interruption peut être représentée par un appel à une fonction qui peut, de façon non déterministe, lever une exception d'interruption. Spécifier le comportement du programme en cas d'interruption revient alors à spécifier la propriété en cas de la levée de l'exception d'interruption.

Nous avons donc étendu l'outil Caduceus de manière à modéliser l'interruption d'un programme par l'appel à une fonction pouvant lever une exception d'interruption. Plus précisément, cette extension consiste à :

- définir une exception d'interruption ;
- définir une fonction d'interruption, qui ne fait rien à part éventuellement lever l'exception d'interruption ;
- ajouter, en tout point du programme, un appel à la fonction d'interruption ;
- ajouter une clause `ensures_interrupt` au langage de spécification, dont la sémantique est de définir la propriété en cas de la levée de l'exception d'interruption.

À cette liste devrait s'ajouter le point suivant :

- faire en sorte que tous les objets stockés en mémoire volatile aient une valeur indéfinie après l'interruption.

Ce dernier point fait partie des travaux futurs à mener et nécessite de pouvoir identifier tous les objets volatiles en un point donné du programme. Pour le moment, nous faisons l'hypothèse que seuls les objets persistants sont mentionnés dans la clause d'interruption.

Pour ce qui est des autres points, le premier problème à résoudre est que le mécanisme d'exception n'existe pas dans le langage C. En revanche, il existe dans le langage d'entrée de Why. L'exception d'interruption et la fonction d'interruption sont donc définies dans le langage d'entrée de Why :

```
exception POExc
parameter tearing_parameter:
    tt:unit ->
    { }
    unit raises POExc
    {true | POExc=>true}
```

De même, l'appel à cette fonction a été ajoutée, non pas dans le programme C, mais dans sa traduction dans le langage de Why. Pour finir, le langage de spécification de Caduceus a été étendu avec la nouvelle clause `ensures_interrupt`, qui est traduite dans le langage de Why par la propriété en cas de levée de l'exception d'interruption. Autrement dit, la spécification de programme C suivante :

```
/*@ requires Pre
  @ assigns  a1, ..., am
  @ ensures  Post
  @ ensures_interrupt Posti */
T f(t1 p1, ..., tn pn);
```

sera traduite par le programme Why suivant :

```
parameter f_parameter : p1:t1 -> ...  -> pn:tn ->
   { Pre }
   T reads ... writes ... raises POExc
   { Post and assign(...)
    | POExc => Posti }
```

Le deuxième problème consiste à définir le niveau de granularité d'une interruption, à savoir ce que l'on entend par "en tout point du programme". En effet, une fonction peut être interrompue entre deux instructions, mais également au sein même d'une instruction. Par exemple, lors de l'exécution d'une instruction de la forme `if-then-else`, l'interruption peut se produire pendant l'évaluation de l'expression conditionnelle. De même, si une interruption survient lors de l'exécution d'une instruction telle que `((*b1|*b2++)==*b1++)` (utilisée pour savoir si le buffer `b1` peut être écrit dans le buffer `b2` sans effacement en mémoire Flash), alors les valeurs de `b1` et `b2` au moment de l'interruption dépendent du moment exact où celle-ci s'est produite. Or, cette instruction est un abrégé, ou *sucre syntaxique*, de plusieurs instructions, à savoir `((*b1|*b2)==*b1) ; b2++ ; b1++ ;`. Par conséquent, considérer une granularité au niveau des instructions implique que deux programmes qui ont la même sémantique (tels que les deux programmes ci-dessus) ne seraient pas interprétés de la même façon. C'est pourquoi l'expression "*désucrage*" (*desugar* en anglais) est parfois utilisée pour représenter l'action qui consiste à enlever le sucre syntaxique d'un programme pour se ramener à un langage minimal. Ces différents niveaux de granularité sont appelés, dans [66] (voir les travaux similaires en fin de section), le niveau *syntaxique* lorsqu'une interruption est insérée entre chaque instruction, et le niveau *sémantique* lorsqu'une interruption est considérée comme le résultat possible de toutes les opérations pouvant modifier la valeur des variables du programme. Le niveau sémantique est bien sûr le plus précis, mais son implémentation est plus complexe et la vérification devient beaucoup plus lourde (le nombre d'obligations générées pour la vérification de la propriété à vérifier en cas d'interruption est proportionnel au nombre de points du programme où une interruption est considérée comme possible).

L'approche plus directe, mais moins précise, d'un niveau syntaxique a été choisie dans un premier temps pour notre modélisation de l'interruption. Un appel à la fonction d'interruption est donc ajouté après chaque instruction (i.e. après chaque "point-virgule") du corps de la fonction. Notre but, dans des travaux futurs, est d'affiner cette interprétation. Une première façon de l'affiner est de n'ajouter l'appel qu'à la suite d'instructions modifiant les variables du programme, afin d'éviter la génération d'un trop grand nombre d'obligations de preuves inutiles. Le niveau sémantique peut également être envisagé, mais seulement pour des fonctions critiques où une telle précision est nécessaire, car la vérification est alors très lourde. Enfin, nous nous intéressons surtout à offrir la possibilité à l'utilisateur d'indiquer en quel point du programme il souhaite vérifier la propriété spécifiée. En effet, cela permettrait de se concentrer sur certaines parties critiques du programme où une propriété donnée doit impérativement être vérifiée. Dans ce cas, l'utilisateur pourrait même définir ses exceptions propres, précisant les propriétés à vérifier en divers points critiques du code.

A titre d'illustration, mentionnons un exemple issu de notre cas d'étude de la gestion d'une mémoire Flash embarquée (qui a été présenté dans la Section 1.3.4 et dont la vérification sera

décrite dans le Chapitre 6). Dans ce module, la fonction

```
void _journal_secureErase(unsigned char jid)
```

a pour but d'effacer le journal d'identifiant `jid`. Ce journal est stocké à l'adresse

```
JournalRegistry[jid].pBaseAddress
```

et sa taille est donnée par le nombre de secteurs qu'il occupe, multiplié par la taille d'un secteur :

```
JournalRegistry[jid].bNbSector*FLASH_SECTOR_SIZE
```

La postcondition de cette fonction est donc :

```
@ ensures ErasedMemory(JournalRegistry[jid].pBaseAddress,
@                      JournalRegistry[jid].bNbSector*FLASH_SECTOR_SIZE)
```

où le prédicat `ErasedMemory(src,length)` indique que le bloc mémoire à l'adresse `src` et de taille `length` est effacé.

Lorsqu'un arrachage a lieu, l'effacement du journal peut être interrompu, le laissant dans un état incohérent. De manière à pouvoir s'en rendre compte et effacer de nouveau le journal, un *état d'effacement* indique si un effacement est en cours. Étant donné que cet état est stocké en mémoire Flash, sa gestion est *journalisée* : le tableau `JournalStateRegistry[jid]` contient les états d'effacement successifs du tableau d'identifiant `jid`. Ce tableau est à l'origine effacé, et les cases du tableau sont utilisées successivement pour stocker l'état d'effacement. Autrement dit, l'état d'effacement courant du journal est le premier qui ne soit pas "terminé". Lors d'une remise sous tension, l'analyse du tableau des états permet de savoir si l'effacement du journal a été interrompue. Si le tableau est *cohérent*, i.e. formé d'une succession d'états "terminés" suivie d'une succession d'états "non utilisés" (effacés), alors aucun effacement du journal n'a été interrompu. Dans le cas contraire, le journal doit de nouveau être effacé.

La spécification de la fonction d'effacement peut donc être complétée par une précondition indiquant que le journal est cohérent lorsque la fonction est appelée :

```
/*@ requires is_consistent(JournalStateRegistry[jid])
```

et par une postcondition d'interruption indiquant que si le tableau est cohérent au moment de l'interruption, cela signifie que le tableau a été effacé :

```
@ ensures_interrupt
@   is_consistent(JournalStateRegistry[jid]) =>
@   ErasedMemory(JournalRegistry[jid].pBaseAddress,
@                JournalRegistry[jid].bNbSector*FLASH_SECTOR_SIZE) */
```

La spécification de cette propriété en cas d'arrachage nous permettra, comme nous le verrons dans la Section 6.4, de prouver que l'effacement de journal défini dans le module de gestion de la mémoire Flash a la propriété d'*anti-tearing*, c'est à dire qu'il assure la cohérence des données en cas d'arrachage de la carte.

Travaux similaires. Étant donné que le langage Java Card est dédié aux cartes à puce, certains travaux de spécification et de vérification de programmes Java Card se sont intéressés à la formalisation de l'arrachage, en particulier les travaux présentés dans [14] et ceux exposés dans [66]. L'objectif commun est la preuve de la correction du mécanisme de *transaction* fournit par Java Card. Ce mécanisme assure que les instructions comprises entre un début de transaction et une fin de transaction seront exécutées de façon atomique. En particulier, si une transaction est interrompue par un arrachage, les données persistantes retrouvent les valeurs qu'elles avaient au début de la transaction et les données volatiles sont perdues.

Dans [14], une extension de la logique de spécification définie dans le projet Key [82, 2] est proposée pour la définition d'invariants *forts*, i.e. vérifiés en *tout* point du programme, et donc également en cas d'interruption. Cette extension permet de spécifier les propriétés qui sont

vérifiées par toutes les méthodes d'une application, y compris en cas d'arrachage de la carte. Cependant, elle ne permet pas de spécifier, méthode par méthode, les propriétés d'une exécution normale de la méthode et les propriétés en cas d'interruption, comme nous l'avons proposé dans notre approche.

L'extension que nous proposons est très proche de celle présentée dans [66]. Ces travaux s'inscrivent dans la vérification de programmes Java Card annotés en JML. L'idée est de modéliser le mécanisme de transaction, en représentant son comportement en cas d'arrachage. Une première partie consiste à modéliser l'arrachage par l'appel à une méthode qui lève une exception d'arrachage (ce qui est plus direct qu'en C puisque les exceptions sont définies dans le langage Java). Il est alors possible de spécifier, en JML, les propriétés de la méthode en cas d'interruption. Ces travaux vont plus loin, en modélisant le fait que les données volatiles sont perdues en cas d'arrachage : le code est transformé de manière à ce que lorsque l'exception d'arrachage est levée, elle soit rattrapée en fin de méthode pour donner aux variables volatiles une valeur indéfinie. Les transactions sont alors modélisées en "désucrant" le mécanisme de Java Card, i.e. en décrivant, dans le langage Java, les effets de ce mécanisme. Le programme est là aussi modifié, de manière à mémoriser la valeur des données persistantes au début de la transaction (à l'aide de nouvelles variables) et à les utiliser si un arrachage a eu lieu.

4.7 Conclusion

Nos travaux de vérification fonctionnelle de programmes C embarqués nous ont amené à devoir analyser et compléter les travaux de recherche dans le domaine de la formalisation du langage C. Ce chapitre a démontré les limitations d'un modèle mémoire de "haut niveau" pour la modélisation de constructions de bas niveau. Plus précisément, un modèle où la mémoire est séparée en emplacements disjoints (suivant une approche de Burstall et Bornat), permet de rendre la vérification praticable en minimisant le nombre d'obligations de preuve générées. Mais les constructions de bas niveau telles que les unions ou les casts enfreignent un tel modèle car elles correspondent à des blocs de mémoire qui sont confondus dans la réalité mais disjoints dans le modèle. Nous proposons donc de modéliser ces constructions en préservant des blocs séparés, mais dont on assure la correction par une synchronisation lors de toute modification. Cette synchronisation n'est en général pas définissable statiquement, le lien entre les blocs devant être établi de façon dynamique. Toutefois, dans le cas d'étude de la mémoire Flash, qui sera présenté dans le Chapitre 6, les liens peuvent toujours être déterminés statiquement. Par conséquent, une modification de l'outil Caduceus, implémentant une solution statique des unions, a suffi pour la vérification de notre cas d'étude. Dans le cas général, la solution dynamique ne doit être envisagée qu'accompagnée d'une analyse statique réduisant la "pollution" inutile des fonctions pour lesquelles les liens sont définissables statiquement.

Nous avons également étendu l'outil Caduceus de manière à pouvoir spécifier et vérifier des propriétés en cas d'interruption soudaine du programme. Cette extension permet la modélisation de l'arrachage de la carte à puce et a été utilisée pour la vérification de propriétés d'*anti-tearing* du module de gestion de mémoire Flash.

Outre la vérification fonctionnelle décrite dans ce chapitre, nous nous sommes également interéssés à la vérification de propriétés dites de *haut niveau*. Une telle vérification nécessite un modèle du programme sous la forme d'un système de transitions. Étant donné que le lien formel entre le code source et le modèle, sur lequel les propriétés sont vérifiées, est une de nos priorités, nous avons choisi une approche où le modèle est généré à partir du code. Nous avons donc défini une méthodologie d'extraction automatique d'un système de transitions, présentée dans le chapitre suivant.

Chapitre 5

Méthodologie de vérification haut niveau de code source

Résumé

Dans ce chapitre, nous proposons une méthodologie de vérification de propriétés de haut niveau à partir d'un code source de bas niveau écrit en langage C (méthodologie également présentée dans [7]). Le principe général est d'utiliser le modèle formel implicitement généré par l'outil de vérification de programme *Caduceus*, pour en extraire un système de transitions, pour lequel des propriétés globales ou temporelles complexes pourront être définies et vérifiées. Les avantages de l'approche reposent sur le lien formel entre le modèle et le code et la possibilité de plonger le modèle dans une logique expressive telle que la logique d'ordre supérieur.

Sommaire

5.1 Contexte et principe de la méthode

5.1.1 Motivation

L'objectif que nous cherchons à atteindre est toujours le même : prouver formellement des propriétés de programmes C d'un système d'exploitation embarqué. Dans le chapitre précédent,

nous avons présenté une méthode de vérification *fonctionnelle*. Cette méthode utilise l'outil Caduceus (voir [52, 53]) qui génère des obligations de preuve à partir de programmes C annotés. Nous avons expliqué comment adapter cet outil pour la vérification de code de bas niveau, comme celui de la gestion d'une mémoire de carte à puce. Dans cette approche, les propriétés visées sont définies, pour chaque fonction du programme, dans le langage de spécification de l'outil. Il s'agit d'un langage de premier ordre, très similaire au langage de spécification JML pour les programmes Java (voir [86]). Ce type de langage permet de décrire le comportement de chacune des fonctions, cependant il n'est pas conçu pour exprimer des propriétés globales, faisant intervenir plusieurs fonctions, avec une notion temporelle sous-jacente. Voici quelques exemples de telles propriétés :

- *"si l'état avant l'appel à la fonction f satisfait P, alors il existe un état résultant d'une séquence finie d'appels à f qui satisfait Q";*
- *"si la fonction g est appelée après la fonction f, alors la propriété P est vérifiée".*

Ce genre de propriétés de *haut niveau* nécessite une modélisation du programme sous la forme d'un *système de transitions*. Autrement dit, chaque fonction f du programme est modélisée par une relation binaire $f\_transition$ sur les états du programme : $\mathbf{x}$ est en relation avec $\mathbf{x}$' par $f\_transition$ (noté ($f\_transition$ $\mathbf{x}$ $\mathbf{x}'$)) si $\mathbf{x}$' est l'état résultant de l'exécution de la fonction f à partir de l'état $\mathbf{x}$. Illustrons ceci par un exemple issu de notre cas d'étude de la mémoire Flash.

5.1.2 Exemple d'une propriété d'*anti-tearing*

Dans notre cas d'étude de gestion de mémoire Flash, qui a été présenté dans la Section 1.3.4 et dont la vérification sera décrite dans le Chapitre 6, nous nous intéressons à la propriété d'*anti-tearing* de l'opération d'effacement. Autrement dit, nous voulons prouver formellement que l'opération d'effacement d'un journal assure que le journal sera effectivement effacé, y compris si la carte a été arrachée du terminal pendant l'effacement. Cette propriété repose sur deux fonctions : la fonction `void _journal_secureErase(unsigned char jid)` qui efface le journal d'identifiant `jid` et la fonction `void _journal_abortErase(unsigned char jid)` qui est appelée à chaque remise sous tension de la carte et qui efface de nouveau le journal `jid` si un arrachage a interrompu un effacement de ce journal.

La propriété que nous voudrions montrer est la suivante : si la fonction `_journal_abortErase` est appelée à chaque remise sous tension, alors un appel à la fonction `_journal_secureErase` permet d'effacer le journal même si la carte a été arrachée du terminal. Cette propriété peut être représentée comme illustré dans la Figure 5.1, où PO désigne le fait qu'un arrachage (*Power Off*) a eu lieu.

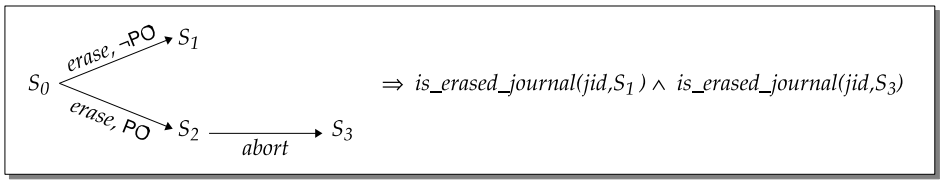


FIG. 5.1 – Représentation de la propriété d'*anti-tearing*

Notons que la fonction d'abandon peut également être interrompue. La propriété à vérifier est donc plus exactement que si un effacement a débuté, alors le journal sera effacé même en cas d'une séquence finie d'arrachage. Cette propriété est illustrée dans la Figure 5.2.

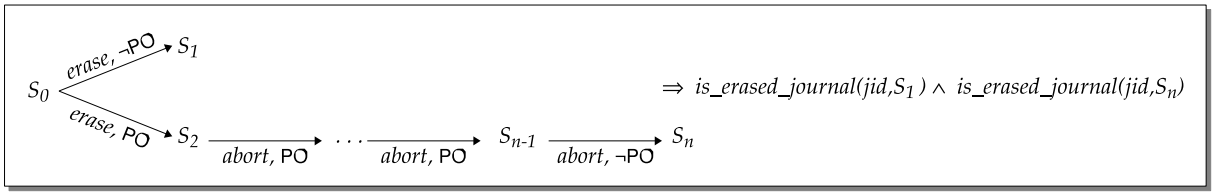


FIG. 5.2 – Représentation de la propriété d'*anti-tearing* dans le cas d'une séquence finie d'arrachages

Ces figures illustrent bien le fait que la propriété nécessite une modélisation des fonctions sous la forme de relations de transition ($erase\_transition\ jid\ \mathbf{x}\ \mathbf{x}'$) et ($abort\_transition\ jid\ \mathbf{x}\ \mathbf{x}'$). Plus précisément, la définition des transitions doit permettre de distinguer le cas d'une terminaison normale et le cas d'un arrachage. La transition sera donc définie sous la forme :

($erase\_transition\ jid\ \mathbf{x}\ \mathbf{x}'\ status$)

où *status* vaut soit PO soit ¬PO (de même pour $abort\_transition$). Une séquence finie d'abandons interrompus, qui se termine par un abandon non interrompu, peut alors être définie de façon *inductive* comme la *clôture transitive* de la transition d'abandon :

$$(abort\_seq\ jid\ \mathbf{x}\ \mathbf{x}') \equiv (abort\_transition\ jid\ \mathbf{x}\ \mathbf{x}'\ \neg\mathsf{PO}) \vee (\exists\, \mathbf{x}_{aux}.\ (abort\_transition\ jid\ \mathbf{x}\ \mathbf{x}_{aux}\ \mathsf{PO}) \wedge (abort\_seq\ jid\ \mathbf{x}_{aux}\ \mathbf{x}'))$$

et l'opération d'effacement est définie par :

$$(erase\_op\ jid\ \mathbf{x}\ \mathbf{x}') \equiv (erase\_transition\ jid\ \mathbf{x}\ \mathbf{x}'\ \mathsf{PO}) \vee (\exists\, \mathbf{x}_{aux}.\ (erase\_transition\ jid\ \mathbf{x}\ \mathbf{x}_{aux}\ \neg\mathsf{PO}) \wedge (abort\_seq\ jid\ \mathbf{x}_{aux}\ \mathbf{x}'))$$

La propriété visée s'exprime alors sous la forme :

$$\forall \mathbf{x}.\ \forall \mathbf{x}'.\ (erase\_op\ jid\ \mathbf{x}\ \mathbf{x}') \Rightarrow (is\_erased\_journal\ jid\ \mathbf{x}')$$

5.1.3 Travaux existants

La vérification de propriétés de haut niveau sur du code source a été étudiée dans [67, 11, 15]. Ces travaux proposent une approche qui consiste à exprimer des propriétés de haut niveau au sein d'un langage de spécification. Plus précisément, la méthode proposée (implémentée dans un outil appelé *JAG*) permet de traduire des propriétés temporelles d'*innocuité* ("rien de mauvais n'arrive") et de *vivacité* ("quelque chose de bon finit par arriver") en formules du langage de spécification JML (voir [86]) pour des programmes Java (ou Java Card). Cette approche permet de prouver que la propriété est vérifiée par le code source, à l'aide d'outils existants de vérification de programmes Java annotés.

Les propriétés temporelles d'innocuité étudiées dans ces travaux concernent essentiellement le mécanisme de transaction de Java Card. Elles expriment, par exemple, qu'il ne peut y avoir de transactions imbriquées, i.e. que si la méthode de début de transaction a déjà été appelée, elle ne peut pas l'être de nouveau jusqu'à ce que la méthode de fin ou d'abandon de transaction soit appelée. Notons que ce genre de propriété a du sens pour une méthode d'API, puisqu'elle spécifie son comportement dans tous les cas possibles d'utilisation. Dans la vérification du système d'exploitation, les cas d'utilisation sont connus et nous nous intéressons donc à analyser certains scénarios donnés.

En ce qui concerne les propriétés de vivacité, elles permettent de vérifier des spécifications de *cycles de vie*, ce qui est très proche de ce que nous recherchons. En effet, la propriété donnée en exemple précédemment peut être vue comme une propriété du cycle de vie de l'opération d'effacement, comme le montre la Figure 5.3. Notons que dans cette figure, la "boucle" d'interruptions devrait en fait être remplacée par une séquence finie d'interruptions. La principale difficulté de cette approche repose sur la définition d'un variant, par l'utilisateur, assurant la terminaison du cycle. Notons que sans notre exemple, la "boucle" d'interruptions représentée dans la figure devrait en fait être remplacée par une séquence finie d'interruptions, dont il faudra prouver la terminaison.

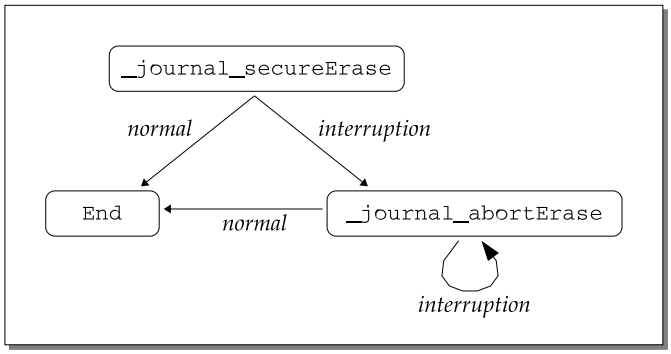


FIG. 5.3 – Représentation de l'opération d'effacement sous la forme d'un cycle de vie

La traduction des propriétés temporelles dans le langage de spécification passe par l'utilisation de *variables de modèles* (*ghost variables*) qui permettent de représenter les différents états du programme au sein des annotations. Par exemple, la propriété de non-imbrication des transactions utilise une variable de modèle `TRANSACT`, qui vaut 0 si aucune transaction n'est en cours et 1 dans le cas contraire. La propriété est alors définie par l'ajout de la précondition `/*@ requires TRANSACT==0 */` et de la postcondition `/*@ ensures TRANSACT==1 */` à la spécification de la fonction de début de transaction. Plus généralement, une variable booléenne `m_called` peut être définie pour toute méthode `m` du programme. Cette variable sera initialisée à *faux* et `/*@ set m_called = true*/` sera ajouté à la première ligne de la méthode `m`. Il sera alors possible d'exprimer des propriétés comme "la méthode `m'` n'est jamais appelée si la méthode `m` a été appelée".

Cette approche implique de définir le modèle mémoire du programme au sein des annotations, par l'intermédiaire de variables de modèles, en fonction de la propriété visée. Notre approche est différente. Nous ne voulons pas "polluer" les annotations par la définition des états mémoire du programme. Les annotations sont considérées comme le modèle *local* et nous voulons que la preuve des propriétés de haut niveau soit effectuée sur un système de transitions où les états de la mémoire sont représentés. Comme nous allons l'expliquer, l'idée clé est de *se servir du modèle local existant pour générer le système de transitions.*

5.1.4 Notre approche

Nous voulons modéliser chaque fonction f du programme C par une relation de transition ($f\_transition\ \mathbf{x}\ \mathbf{x}'$) entre les états mémoire $\mathbf{x}$ et $\mathbf{x}$' du programme. Pour cela, un modèle mémoire doit être choisi pour le calcul des états $\mathbf{x}$ et $\mathbf{x}$'. Il est possible de le faire à la main, en considé-

rant qu'un état contient les valeurs des variables manipulées par le programme et en calculant manuellement l'effet de la fonction sur l'état global du programme. Prenons l'exemple trivial suivant :

```
void f(int *i) { (*i)--; }
```

et supposons que l'on veuille montrer que, pour tout entier positif i, il est possible de le rendre négatif en lui appliquant la fonction f un nombre fini de fois. Dans cet exemple, un état $\mathbf{x}$ du programme contient, entre autres, la valeur de l'entier i. Une fonction d'accès $(val\ i\ \mathbf{x})$ permet d'obtenir cette valeur. La relation $f\_transition$ peut alors être définie par :

$$(f\_transition\ i\ \mathbf{x}\ \mathbf{x}') \quad \equiv \quad (\ (val\ i\ \mathbf{x}') = (val\ i\ \mathbf{x}) - 1\ )$$

ou même par :

$$(f\_transition\ i\ \mathbf{x}\ \mathbf{x}') \quad \equiv \quad (\ (val\ i\ \mathbf{x}') < (val\ i\ \mathbf{x})\ )$$

qui suffit à prouver la propriété recherchée. Une séquence finie d'appels à la fonction est alors définie de façon inductive par la clôture transitive de la relation :

$$\begin{array}{lcl}(f\_exec\ i\ \mathbf{x}\ \mathbf{x}') & \equiv & (f\_transition\ i\ \mathbf{x}\ \mathbf{x}') \\ & & \vee \quad (\ \exists\ \mathbf{x}_{aux}.\ (f\_transition\ i\ \mathbf{x}\ \mathbf{x}_{aux})\ \wedge\ (f\_exec\ i\ \mathbf{x}_{aux}\ \mathbf{x}')\ )\end{array}$$

Enfin, la propriété visée s'exprime sous la forme :

$$\forall\ j.\ \forall\ \mathbf{x}.\ \forall\ \mathbf{x}'.\ (val\ j\ \mathbf{x}) \geq 0 \quad \wedge \quad (f\_exec\ j\ \mathbf{x}\ \mathbf{x}') \Rightarrow\ (val\ j\ \mathbf{x}') \leq 0$$

Les inconvénients d'un calcul manuel des d'états d'un programme donné sont multiples. Tout d'abord, la définition d'un modèle mémoire pour les programmes C, permettant de modéliser une fonction comme une transformation d'états mémoire du programme, est *intrinsèquement complexe*, comme l'illustre le modèle mémoire de Caduceus présenté dans le chapitre précédent. Le fait que ce calcul doit être recommencé pour chaque nouveau programme n'en est que plus fastidieux. Enfin, un problème majeur de cette approche repose sur l'incertitude du lien entre la relation de transition et le comportement réel du programme. Par exemple, comment être sûr que la propriété $(val\ i\ \ \mathbf{x}') < (val\ i\ \mathbf{x})$ est une abstraction correcte du code de la fonction f (dans un cas où le corps de la fonction serait plus complexe) ?

Un lien *formel* entre le modèle, sur lequel la vérification est effectuée, et le code source du programme est primordial, comme l'illustre la conclusion sur nos travaux de vérification de la plate-forme Java Card, présentée dans le Chapitre 3. L'enseignement tiré de ces premiers travaux nous a incité à rechercher une méthode de vérification permettant d'avoir un lien formel entre le modèle et le code. Comme nous l'avons expliqué dans la Section 2.1.6, une première approche consiste à construire un modèle formel et à le raffiner successivement jusqu'à générer le code source du programme. Mais la génération de code n'est pas assez optimisée, à l'heure actuelle, pour générer du code source de bas niveau, comme celui d'une gestion de mémoire Flash embarquée. De plus, cette approche correspond à une méthode de *conception* de programme. Or, nous nous intéressons ici à une méthode de *vérification* d'un code *existant*. Une approche possible est donc *générer automatiquement (et formellement) le modèle à partir du code source.* Mais notre but n'est surtout pas de définir un nouvel outil d'extraction de modèles à partir de programmes C. Notre approche est au contraire de constater que l'on recherche un moyen de calculer les états mémoire d'un programme et de modéliser les fonctions du programme sous la forme de transitions entre ces états mémoire, et que tout ce travail est effectué implicitement par les outils de vérification de programmes. L'objectif de ces outils est de vérifier des propriétés fonctionnelles à partir du code source, mais ils utilisent pour cela un modèle formel du programme, construit à partir du code et de sa spécification. Ce modèle est *implicite*, dans le sens que

l'utilisateur ne voit que le résultat de la vérification, à savoir les conditions à prouver pour établir la correction.

Nous proposons d'utiliser le modèle formel implicite généré par l'outil de vérification de programme, pour en extraire un système de transitions permettant d'exprimer des propriétés de haut niveau.

Plus précisément, un outil de vérification de programme prend en entrée un programme annoté de sa spécification, construit un modèle formel de ce programme et une traduction de sa spécification, et tente de construire une preuve que la spécification formelle est vérifiée par le modèle formel du programme. Toutes les étapes de la preuve que l'outil n'a pas pu produire automatiquement constituent les *obligations de preuves*. Celles-ci sont soumises soit à un assistant de preuve, pour une preuve interactive, soit à une procédure de décision, pour une preuve automatique, et sont les seuls "fragments" du modèle accessibles à l'utilisateur.

L'idée de la méthodologie que nous proposons est de dériver un système de transitions à partir de la traduction de la spécification de la fonction, utilisant les états mémoire calculés. La spécification peut alors être vue comme une modélisation ou une abstraction du code. Le point clé de l'approche est que *le lien formel entre l'abstraction et le code est assuré par la preuve des obligations générées par l'outil*. Par exemple, la fonction triviale citée précédemment pourrait être modélisée par :

```
| /*@ ensures  *i<\old(*i) */
```

mais dans notre approche, la preuve des obligations assure que cette abstraction est vérifiée par le code source de la fonction.

Un autre avantage majeur de cette approche est que le modèle formel peut être plongé dans une logique d'ordre supérieur, comme celle du système de preuves Coq, offrant toute l'expressivité nécessaire à une définition simple et intuitive de propriétés complexes de haut niveau.

Cette méthodologie est possible avec l'outil Caduceus, où le modèle généré est accessible, comme nous allons le décrire dans la section suivante.

5.2 Description détaillée de la méthode

5.2.1 Extraction du système de transitions

L'outil Caduceus prend en entrée un programme C annoté, calcule automatiquement les états mémoire du programme, suivant son modèle mémoire interne, et modélise toutes les fonctions et leurs spécifications en termes de ces états. Ces modèles sont utilisés pour générer les obligations de preuves assurant la correction du programme, mais ils sont également accessibles, lorsque Caduceus est utilisé avec l'assistant de preuves Coq. Plus précisément, le modèle d'une fonction et de sa spécification est donné par un terme T de type τ, où T est le modèle du corps de la fonction et τ sa spécification. Or, pour toute fonction f d'un programme C, le paramètre Coq suivant est fourni :

$$f\_parameter \ : \ \forall \mathbf{x}.\ Pre_f(\mathbf{x}) \ \rightarrow \ \exists \mathbf{x}'.\ Post_f(\mathbf{x}, \mathbf{x}')$$

Le type de ce terme correspond à la traduction de la *spécification* de f. Le terme est, quant à lui, explicité uniquement dans l'assistant de preuve Coq, sous la forme du *terme de validation* (voir Section 4.2.1).

Il existe donc deux manières de définir une relation de transition ($f\_transition$ $\mathbf{x}$ $\mathbf{x}'$) à partir de la formalisation de Caduceus. La première est d'utiliser le modèle du corps de la fonction (ce qui implique que ce n'est possible que dans l'assistant de preuve Coq). Dans ce cas, l'état $\mathbf{x}$ est en relation avec $\mathbf{x}'$ si $\mathbf{x}'$ est le "témoin existentiel", i.e. l'état final donné par le terme de validation. La modélisation est alors *complète*, en ce sens que ($f\_transition$ $\mathbf{x}$ $\mathbf{x}'$) $\Leftrightarrow$ $x' = \bar{f}(x)$ où $\bar{f}$ représente le modèle de la fonction f. Le problème est que cette approche peut devenir extrêmement lourde si le corps de la fonction est complexe.

Une deuxième approche consiste à considérer que, dès lors que les obligations ont été prouvées, la spécification de la fonction représente le comportement du code source et peut donc être utilisée pour modéliser la fonction. La relation de transition est alors définie par :

$$(f\_transition\ \mathbf{x}\ \mathbf{x}') \quad \equiv \quad Pre_f(\mathbf{x}) \ \wedge \ Post_f(\mathbf{x}, \mathbf{x}')$$

Cette approche permet d'utiliser les annotations comme une abstraction du code, dont la correction est assurée par la preuve des obligations. Si les annotations sont très précises, un plus grand nombre de propriétés pourront être prouvées avec les mêmes annotations (qui peuvent d'ailleurs être les mêmes que celles définies pour la vérification fonctionnelle). Mais les annotations peuvent également décrire un modèle plus abstrait, pour une preuve plus simple d'une propriété donnée. Les annotations sont alors dépendantes de la propriété à vérifier. Par exemple, la spécification `/*@ ensures *i<\old(*i)-1 */` d'une fonction f permet de prouver que tout entier positif peut être rendu négatif par un séquence finie d'appels à f, alors que la spécification `/*@ ensures \old(*i)>0 -> *i>=0 */` pourrait permettre de montrer que tout entier strictement positif n'est rendu négatif par aucune fonction du programme. Les deux propriétés pourraient être vérifiées avec la spécification précise `/*@ ensures *i==\old(*i)-1 */`.

Dans cette deuxième approche, l'état final n'est pas forcément celui calculé par le programme, mais tout état qui vérifie la postcondition. La classe de propriétés prouvables dans cette approche est donc restreinte. En effet, des propriétés où l'état final est quantifié de façon universelle seront effectivement vérifiées par le code, puisque si la propriété est vraie pour tout état vérifiant la postcondition, alors elle est vraie en particulier pour l'état calculé par le programme. En revanche, la vérification de propriétés comportant des quantifications existentielles sur l'état final ne garantit pas que la propriété est vérifiée par le programme. Par exemple, la fonction suivante :

```
/*@ ensures *i<\old(*i) */
void g(int *i) *i=*i-2;
```

sera modélisée par la transition ($g\_transition\ i\ \mathbf{x}\ \mathbf{x}'$) $\equiv$ ($(val\ i\ \mathbf{x}') < (val\ i\ \mathbf{x})$) et il est possible de prouver que $\forall\ j.\ \forall\ \mathbf{x}.\ \exists\ \mathbf{x}'.\ (g\_transition\ j\ \mathbf{x}\ \mathbf{x}') \ \wedge \ (val\ j\ \mathbf{x}') = 0$ alors que la fonction g ne vérifie pas cette propriété. Plus généralement, les perspectives de ces travaux doivent inclure l'identification de la classe de formules logiques dont la validité peut être transférée du système de transitions au code source du programme.

Le principe de la méthodologie que nous proposons est donc de considérer les annotations comme le modèle local du programme (qui peut éventuellement constituer une abstraction du code source) et d'en extraire automatiquement un système de transitions. Ce système est défini par l'ensemble des relations de transition ($f\_transition\ \mathbf{x}\ \mathbf{x}') \equiv Pre_f(\mathbf{x}) \wedge Post_f(\mathbf{x}, \mathbf{x}')$, pour chaque fonction f du programme. L'idée majeure étant de bénéficier du calcul automatique des états mémoire $\mathbf{x}$ et $\mathbf{x}$' et de la traduction Pre_f et $Post_f$ des spécifications en termes de modification de ces états.

Plus précisément, Pre_f est calculé à partir de la clause `requires` de l'annotation de f, et

éventuellement des invariants globaux du programme s'ils concernent une variable lue par la fonction. $Post_f$ est calculé à partir des clauses `assigns` et `ensures`, et des invariants globaux qui impliquent une variable modifiée par la fonction. Les états mémoire `x` et `x'` sont, quant à eux, calculés à partir des paramètres de la fonction, des variables globales du programme et des variables représentant les segments mémoire : pour chaque fonction f prenant en argument une liste $\overrightarrow{a}$ de paramètres, Caduceus calcule l'ensemble $\overrightarrow{z}$ des variables "uniquement lues" et l'ensemble $\overrightarrow{t}$ des variables "lues et modifiées" (par variables, on entend les variables du programme et les variables d'états représentant les segments mémoire). La traduction de la spécification fournie par Caduceus a donc la forme suivante :

$$f\_parameter \ : \ \forall \overrightarrow{a}.\ \forall \overrightarrow{z}.\ \forall \overrightarrow{t@}.\ Pre_f(\overrightarrow{a}, \overrightarrow{z}, \overrightarrow{t@}) \ \rightarrow \ \exists result.\ \exists \overrightarrow{t}.\ Post_f(result, \overrightarrow{a}, \overrightarrow{z}, \overrightarrow{t@}, \overrightarrow{t})$$

La relation de transition est donc définie par :

$$(f\_transition\ result\ \overrightarrow{a}\ \overrightarrow{z}\ \overrightarrow{t@}\ \overrightarrow{t}\ ) \ \equiv \ Pre_f(\overrightarrow{a}, \overrightarrow{z}, \overrightarrow{t@}) \ \wedge \ Post_f(result, \overrightarrow{a}, \overrightarrow{z}, \overrightarrow{t@}, \overrightarrow{t})$$

ce qui peut être exprimé par une expression simple dépendant uniquement de $f\_parameter$.

A titre d'illustration, considérons la fonction annotée suivante :

```
/*@ requires \valid(i)
  @ assigns *i
  @ ensures *i==\old(*i)-1 */
void f(int *i) { (*i)--; }
```

Pour cette fonction, une seule variable d'état $intP$ est introduite, pour représenter le segment mémoire où le pointeur i est alloué. La seule variable lue par la fonction est la table d'allocation $alloc$ et la seule variable lue et modifiée est $intP$. La modélisation de la fonction dans le système de transitions est donc :

$$\begin{aligned}(f\_transition\ i\ alloc\ intP@\ intP) \ \equiv \\ & (\ valid(alloc, i) \ ) \\ \wedge \ & (\ (\ acc(intP, i) = acc(intP@, i) - 1 \ ) \wedge \ assigns(alloc, intP@, intP, pointer\_loc(i)) \ )\end{aligned}$$

Le système de transitions peut alors être plongé dans la logique d'un assistant de preuves. Si cette logique est d'ordre supérieur, alors des propriétés complexes de haut niveau (comme des clôtures transitives) pourront être exprimées et vérifiées sur un modèle, qui est formellement lié au code source (grâce à la vérification des obligations de preuve).

5.2.2 Vérification de propriétés de haut niveau

Le but de cette section est d'illustrer l'utilisation de la modélisation du programme sous la forme d'un système de transitions pour la vérification de propriétés dites de *haut niveau*.

Exemple de clôture transitive. Commençons par l'exemple trivial, défini dans les sections précédentes, d'une fonction `f` qui prend en argument un pointeur d'entier `i` et dont la spécification indique que la valeur pointée par `i` a diminué strictement. La modélisation de cette fonction sous la forme d'une relation de transition est définie à la fin de la section précédente. La propriété que nous voulions prouver pour cette fonction était que tout entier positif pouvait être rendu négatif en lui appliquant un nombre fini de fois la fonction `f`. Une séquence finie d'exécutions de la fonction peut être définie très simplement à partir de la relation de transition, dans un assistant de preuve d'ordre supérieur comme Coq, par un prédicat inductif f_exec :

$$(f\_exec\ i\ alloc\ intP_1\ intP_2) \equiv (f\_transition\ i\ alloc\ intP_1\ intP_2) \vee (\exists\ intP_3.\ (f\_transition\ i\ alloc\ intP_1\ intP_3) \wedge (f\_exec\ i\ alloc\ intP_3\ intP_2))$$

La propriété visée peut alors être exprimée par le théorème suivant :

$$\forall\ i.\ \forall\ alloc.\ \forall\ intP_1.\ \forall\ intP_2.\ (acc\ intP_1\ i) \geq 0 \wedge (f\_exec\ i\ alloc\ intP_1\ intP_2) \Rightarrow ((acc\ intP_2\ i) \leq 0).$$

Ce théorème peut être prouvé en quelques dizaines de lignes en Coq. Notons que nous avons bien ici une quantification universelle, qui garantit que la propriété sera vérifiée par le code source de la fonction.

Propriété d'*anti-tearing*. Concernant notre cas d'étude de la gestion d'une mémoire Flash embarquée, la propriété d'*anti-tearing* de l'opération d'effacement, expliquée dans la Section 5.1.4, a pu être énoncée sous la forme :

$$\forall \mathbf{x}.\ \forall \mathbf{x'}.\ (erase\_op\ jid\ \mathbf{x}\ \mathbf{x'}) \Rightarrow (is\_erased\_journal\ jid\ \mathbf{x'})$$

où les états **x** et **x'** sont ceux calculés par l'outil et où la clôture transitive *erase_op* de la fonction d'abandon est définie de façon immédiate par un type inductif dans le langage de Coq. La propriété a été prouvée en quelques dizaines de lignes en Coq (voir la Section 6.4 pour plus de détails). De nouveau, cette propriété est prouvée pour toute trajectoire, donc en particulier pour celle définie par le code source de l'opération d'effacement.

Propriété de mise à jour. Des propriétés plus fonctionnelles, mais impliquant une notion temporelle, peuvent également être exprimées simplement à l'aide d'un système de transitions. Un autre exemple de propriété qui a pu être prouvée avec cette méthodologie concerne les fonctions d'accès et de mise à jour d'un *état d'effacement* d'un journal (qui indique si un effacement est en cours). Comme nous l'avons déjà évoqué à diverses reprises, un état est formé de deux bits, contenus dans un tableau d'octets, associé au journal. Un état dans ce tableau peut être accédé grâce à la fonction :

```
unsigned char getJournalState(unsigned char jid, journal_state_index index)
```

où `jid` est l'identifiant du journal et où `index` contient à la fois l'index de l'octet dans le tableau et l'emplacement des deux bits dans cet octet, qui doivent être retournés par la fonction. De même, la fonction :

```
void setJournalState(unsigned char jid, journal_state_index index,
                     unsigned char newstate)
```

retourne un état où l'état correspondant à l'index donné en argument, dans le tableau du journal `jid`, a pris la valeur `newstate`. De façon simplifiée, ces deux fonctions sont modélisées par les transitions suivantes :

$$(getJournalState\_transition\ jid\ index\ intP\ result)$$
$$(setJournalState\_transition\ jid\ index\ intP\ newstate\ intP')$$

Une propriété classique que l'on souhaite vérifier, lorsque l'on définit ce type de couple de fonctions, est que si l'on accède à un état après l'avoir mis à jour, alors sa valeur est la nouvelle valeur :

$$(setJournalState\_transition\ jid\ index\ intP\ newstate\ intP') \wedge (getJournalState\_transition\ jid\ index\ intP'\ result) \Rightarrow result = newstate.$$

et si les valeurs des autres états n'ont pas changé :

$index \neq index' \wedge$
$(getJournalState\_transition\ jid\ index\ intP\ result) \wedge$
$(setJournalState\_transition\ jid\ index\ intP\ newstate\ intP') \wedge$
$(getJournalState\_transition\ jid\ index'\ intP'\ result') \Rightarrow$
$result' = result.$

La généralisation de cette méthodologie à d'autres propriétés sécuritaires et l'étude du type de propriétés pour lequel l'avantage de cette approche est significatif par rapport à d'autres approches, restent à faire. Néanmoins, la simplicité d'expression et de vérification de propriétés combinant des fonctions ou impliquant des clôtures transitives, dans le cadre de notre cas d'étude, est prometteuse.

5.3 Conclusion

Les méthodes de vérification de programmes varient en fonction du type de propriétés visées. Ces propriétés peuvent être fonctionnelles et de bas niveau, à vérifier sur le code source, ou des propriétés globales de haut niveau, à vérifier sur un modèle du programme, sous la forme d'un système de transitions.

Nous avons expliqué dans ce chapitre une façon de combiner plusieurs approches, afin d'utiliser une modélisation locale sous la forme d'annotations (éventuellement développée précédemment pour une vérification fonctionnelle du programme), pour en extraire *automatiquement* un système de transitions pour une vérification dite de haut niveau.

L'idée principale consiste à constater que la modélisation de haut niveau nécessite un choix de modèle mémoire, un calcul des états mémoire du programme et une modélisation des fonctions sous la forme de transitions entre ces états mémoire, et que tout ce travail est effectué implicitement par les outils de vérification de programmes. Nous proposons donc d'utiliser l'outil de vérification de programme Caduceus pour prouver la correction d'un programme vis-à-vis de propriétés fonctionnelles annotées, et ensuite d'utiliser la modélisation de la spécification, dont on a formellement prouvé qu'elle était vérifiée par le code, pour en extraire un modèle de chaque fonction sous la forme d'un système de transitions. Ce modèle de haut niveau peut alors être utilisé pour exprimer et prouver des propriétés globales de haut niveau, élargissant le domaine d'application de l'outil Caduceus, conçu à l'origine pour la vérification fonctionnelle.

Outre l'automaticité de l'extraction, la valeur ajoutée majeure provient du lien *formel* entre le modèle et le code source, assuré par preuve de correction du modèle local à l'aide d'outils de vérification de programmes. Ce lien formel est également assuré par les outils de génération de code à partir de modèles formels (comme ceux utilisant la méthode B), mais notre approche est une approche de vérification d'un code *existant* et non une approche de conception de programme. D'une part cela permet de vérifier un code source très optimisé, écrit par des programmeurs experts, tel que celui d'une gestion de mémoire sur une carte à puce (contrairement à la génération de code qui ne permet pas, à l'heure actuelle, de générer des programmes de bas niveau tels que ceux de systèmes d'exploitation embarqués). D'autre part, notre approche semble plus adaptée aux milieux industriels pour lesquels l'intégration des méthodes formelles est lente, en particulier dans le processus de développement. En effet, le remplacement du développement de programmes par le développement de modèles formels à partir desquels les programmes seront générés est une frontière bien plus délicate à franchir que la vérification formelle de code existant, apportant une

garantie sécuritaire supplémentaire quant au code embarqué.

Un autre bénéfice de cette méthodologie concerne la possibilité de plonger le système de transitions dans une logique d'ordre supérieur, pour une plus grande expressivité et la définition d'un plus grand nombre de propriétés.

La question qui reste ouverte concerne l'identification d'un langage logique pour lequel la validité des propriétés vérifiées par le système de transitions peut être transférée au code source.

Notre méthodologie a été utilisée pour la vérification du module de gestion de mémoire Flash, que nous décrivons en détail dans le chapitre suivant.

Chapitre 6

Étude de cas : module de gestion de mémoire Flash

Résumé

La méthode de vérification fonctionnelle, utilisant Caduceus, présentée dans le Chapitre 4, et l'approche de vérification de haut niveau, présentée dans le Chapitre 5, sont illustrées dans ce chapitre par la vérification formelle du module générique de gestion de mesures d'*anti-tearing* en mémoire Flash, présenté en Section 1.3.4. Ce module est représentatif des particularités et des difficultés d'un code source de bas niveau d'un système d'exploitation embarqué. En particulier, nous présentons ici une fonction cruciale permettant d'assurer que l'opération d'effacement de journaux vérifie, elle-même, la propriété d'*anti-tearing*. Nous présentons la spécification formelle de cette fonction, la validation de sa correction, le modèle de haut niveau qui en est extrait automatiquement et la preuve formelle de la propriété d'*anti-tearing* sur ce modèle.

Sommaire

6.1 Une opération *anti-tearing* d'effacement de journal

Lorsque nous nous sommes intéressés à étendre la modélisation et la vérification formelles aux programmes du système d'exploitation, nous avons choisi un module représentatif des pré-

occupations en termes de correction et des difficultés en termes de modélisation de code de bas niveau. La mémoire Flash, dont la gestion a tout d'abord consisté à la faire passer pour de l'EEPROM auprès des algorithmes existants, a ensuite été gérée par des algorithmes spécifiques et optimisés, qui tentent d'adapter les techniques connues, en particulier de gestion de l'arrachage. Établir la correction de ces algorithmes est une réelle valeur ajoutée, localement pour le module étudié, et globalement pour l'utilisation de la mémoire Flash.

Nous avons donc choisi d'étudier le module générique de gestion de mesures *anti-tearing* en mémoire Flash, présenté en Section 1.3.4. Une vérification fonctionnelle, utilisant l'outil Caduceus et les extensions présentées dans le Chapitre 4, a permis de spécifier formellement et de vérifier la correction de chaque fonction du module. Puis la méthodologie présentée dans le Chapitre 5 a été utilisée pour modéliser le module sous la forme d'un système de transitions. Ce modèle a permis de définir et de prouver des propriétés globales de haut niveau, comme la propriété d'*anti-tearing* de l'opération d'effacement.

Rappelons que le module générique étudié permet de gérer toute mesure d'*anti-tearing*, à savoir une mesure associée à une opération sensible à l'arrachage de la carte et qui permet de retrouver un état cohérent lorsque cette opération a été interrompue. Chaque mesure est associée à un journal, où chaque entrée du journal correspond à une exécution de l'opération sensible et permet de mémoriser les données nécessaires au retour à un état cohérent en cas d'arrachage. La Figure 6.1 rappelle l'architecture générale de ce module, dans le cas où un seul journal, identifié par son numéro `jid`, est représenté.

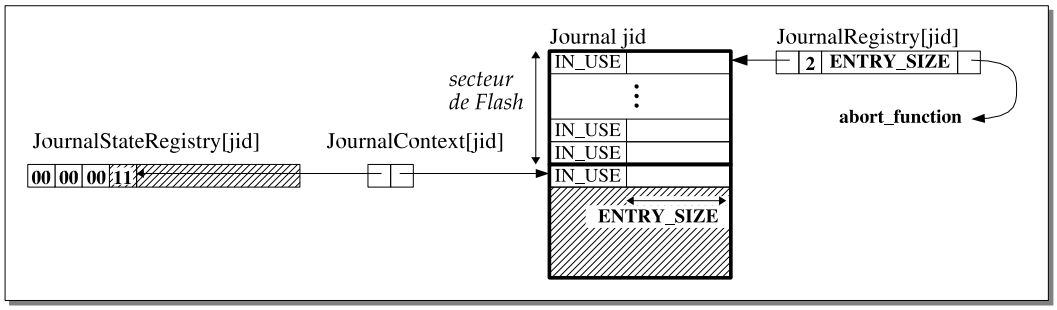


FIG. 6.1 – Vue générale de l'interface générique pour un journal donné

Nous allons nous intéresser ici plus particulièrement à la façon dont l'effacement est géré en cas d'arrachage. En effet, cette interface a pour but d'assurer, entre autre, que l'opération d'effacement d'un journal a la propriété d'*anti-tearing*, à savoir que le journal sera effacé même si un arrachage a eu lieu. Cette propriété est assurée par deux mécanismes :

- Tout d'abord une variable globale `JournalStateRegistry` est utilisée pour indiquer, pour chaque journal, si ce journal est en cours d'effacement ou non. Plus précisément, le tableau `JournalStateRegistry[jid]` donne l'*état d'effacement* du journal d'identifiant `jid`. Cet état peut être soit *en cours* (associé à la valeur `01`), soit *terminé* (associé à la valeur `00`), soit *non utilisé*, lorsqu'aucun effacement n'a lieu (associé à la valeur `11`). Rappelons qu'étant donné que la valeur de cet état doit être connue après une interruption, elle est stockée en mémoire Flash. Elle est donc à son tour *journalisée*, dans le tableau `JournalStateRegistry[jid]`, ce qui signifie qu'une nouvelle case du tableau est utilisée pour chaque nouvel effacement. Par conséquent, le tableau `JournalStateRegistry[jid]` contient non seulement l'état d'effacement *courant* du journal, mais également l'historique de tous les états d'effacement pré-

cédents. L'index de l'état d'effacement courant dans le tableau est stocké dans la variable `JournalContext`, qui est stockée en mémoire RAM, puisque cet état courant est souvent mis à jour.
L'effacement d'un tableau est alors géré de la façon suivante (voir la Figure 6.2) : avant de commencer l'effacement, l'état d'effacement courant du journal (qui est censé être non utilisé) prend la valeur indiquant qu'un effacement est en cours. Puis, l'effacement du journal est effectué, suivi de la mise à jour de l'état d'effacement du journal, qui indique alors que l'effacement est terminé. Enfin, l'état d'effacement courant du journal est "décalé" vers la case suivante de `JournalStateRegistry[jid]` (qui est effacée) pour indiquer, de nouveau, qu'aucun effacement n'est en cours. Cet effacement est assuré par la fonction `_journal_secureErase`.

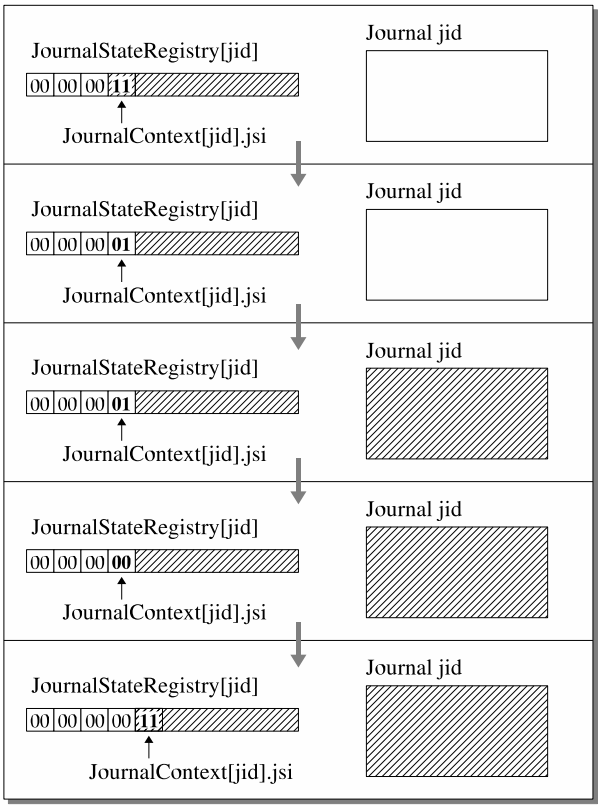


FIG. 6.2 – Effacement d'un journal

- Par ailleurs, lors de toute remise sous tension (donc y compris après un arrachage), une fonction de *montage* est appelée avant toute autre instruction. Cette fonction a pour but de vérifier si la carte est dans un état cohérent et, le cas échéant, d'effectuer toutes les opérations nécessaires pour retrouver un état cohérent. Lors de ces opérations, les journaux dont l'effacement a été interrompu, i.e. dont l'état d'effacement courant vaut `01`, doivent être de nouveau effacés. Seulement, la valeur de l'index de l'état d'effacement courant a été perdue lors de l'arrachage puisqu'elle est stockée en mémoire RAM. Le tableau `JournalStateRegistry[jid]` est donc analysé, pour tous les journaux, et un critère de *cohérence* est défini, censé représenter le fait qu'aucun effacement n'était en cours au moment de l'interruption. Le tableau `JournalStateRegistry`[jid] est dit *cohérent* s'il est composé d'une suite d'états d'effacement "terminés", suivie d'une suite d'états d'effacement "non utilisés" (voir Figure 6.3).

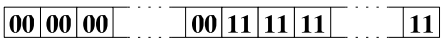


FIG. 6.3 – Tableau d'états d'effacement cohérent

Cet algorithme de vérification de cohérence est assuré par la fonction `_journal_abortErase`. Cette fonction va également réinitialiser la valeur de l'index indiquant où se trouve l'état courant dans ce tableau, qui a été perdue lors de la mise hors tension. La Figure 6.4 donne le schéma général de l'execution de la fonction, dont l'algorithme informel est donné dans la Figure 6.5 et l'argument informel de la correction en Figure 6.6.

Dans la suite, nous allons tout d'abord donner le code source des composants impliqués dans l'effacement. Puis nous allons décrire la spécification formelle de la fonction `_journal_abortErase`, qui inclut une formalisation de la notion de cohérence, suivie de la description de la preuve de la correction du code source vis-à-vis de cette spécification. Puis, la fin du chapitre sera consacrée à la modélisation et la vérification de haut niveau.

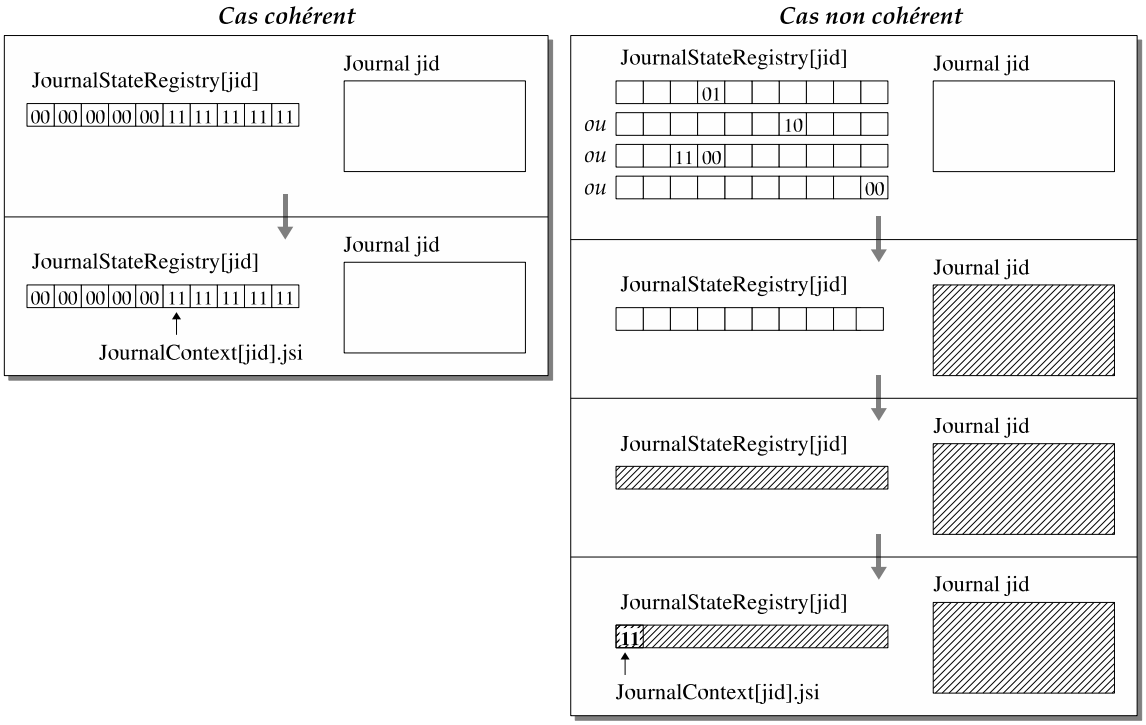


FIG. 6.4 – Opération "d'abandon" de l'effacement

6.2 Code source

6.2.1 Code source des variables globales

Commençons par décrire le code source des variables globales du module, à savoir les structures de données utilisées pour les représenter dans le langage C.

Algorithme d'abandon de l'effacement d'un journal :

- Variables lues et modifiées :
  - une variable `Journal` représentant un bloc de mémoire journalisé ;
  - un tableau `JSR` de taille `MAX` contenant des *états d'effacement* du journal (où un état d'effacement est soit `COMMITTED` soit `FAKE` soit `ONGOING` soit `UNUSED`) ;
  - une variable `curr_index` représentant l'index dans `JSR` de l'état courant du journal.
- Variables locales :
  - un index `i` pour le parcours du tableau `JSR`
  - l'état `currst` contenu dans le tableau `JSR` à l'index `i`
  - l'état `prevst` contenu dans le tableau `JSR` à l'index `i-1`
- Spécification : si `JSR` est *cohérent*, alors `curr_index` est l'index du premier état valant `UNUSED` et le reste est inchangé, sinon cet index vaut zéro et `Journal` et `JSR` sont effacés.

  `JSR` est dit *cohérent* si $\exists\ 0 \leq N < \texttt{MAX}.\ (\ \forall\ 0 \leq j < N.\ \texttt{JSR[j]=COMMITTED} \wedge \forall\ N \leq k < \texttt{MAX}.\ \texttt{JSR[k]=UNUSED}\ )$

- Algorithme :

```
1    i=0 ;
2    prevst=COMMITTED ;
3    while i<MAX
4    { currst = JSR[i]
5      if ( (currst==FAKE) ||
6           (currst==ONGOING) ||
7           (currst==COMMITTED && (prevst!=COMMITTED || i==MAX-1)) )
8      then
9      { Erase(Journal) ;
10       Erase(JSR) ;
11       curr_index=0 ;
12       return ;
13     }
14     else /* currst==UNUSED ||
15             (currst==COMMITTED && prevst==COMMITTED && i!=MAX-1) */
16     { if (currst==UNUSED && prevst=COMMITTED) then curr_index=i ;
17       prevst=currst ;
18       i++ ; }
19   }
```

FIG. 6.5 – Algorithme d'abandon de l'opération d'effacement d'un journal

L'algorithme d'abandon est correct vis-à-vis de sa spécification car :

- si `JSR` est cohérent, alors :
  - lorsque $0 \leq i < N$, `currst` et `prevst` valent tous deux `COMMITTED` (même lorsque i vaut 0 puisque `prevst` est initialisé à `COMMITTED`), donc le corps de la boucle est dans le branchement "else" et la condition de la ligne 16 n'est pas vérifiée. Par conséquent, aucune variable globale n'est modifiée, seul l'index est incrémenté et l'état `prevst` mis à jour.
  - lorsque $i = N$, `currst` vaut `UNUSED`, donc le corps de la boucle est dans le branchement "else" et la condition de la ligne 16 est vérifiée (car $\forall\ 0 \leq j < N$. `JSR[j]=COMMITTED`). Par conséquent, `curr_index` est mis à jour avec la valeur de `i`, qui correspond bien au premier état valant `UNUSED` et les autres variables globales ne sont pas modifiées. Puis l'index est incrémenté et l'état `prevst` mis à jour.
  - enfin lorsque $N < i < MAX$, `currst` vaut `UNUSED` (et `pervst` aussi), donc le corps de la boucle est dans le branchement "else" mais la condition de la ligne 16 n'est pas vérifiée. Par conséquent, aucune variable globale n'est modifiée, seul l'index est incrémenté et l'état `prevst` mis à jour.

  En conclusion, si `JSR` est cohérent, alors seule la variable globale `curr_index` est modifiée, pour valoir l'index du premier état valant `UNUSED`.
- si `JSR` n'est pas cohérent : la définition de cohérence s'énonce informellement par "le tableau ne contient que des états `COMMITTED` et `UNUSED` et tout état `UNUSED` est suivi uniquement d'états `UNUSED`". La négation de la cohérence est donc :
  - soit il existe un état `ONGOING` ou `FAKE`,
  - soit il existe un état `UNUSED` suivi d'un état `COMMITTED`.
  - soit le tableau ne contient que des états `COMMITTED`

  Ces trois cas sont clairement les conditions du "if-then-else" principal de la boucle. En effet, le premier cas est explicite. Le deuxième cas est équivalent à : il existe un état `COMMITTED` précédé d'un état `UNUSED`. Étant donné que les cas `ONGOING` et `FAKE` ont déjà été exclus, cela revient également à dire qu'il existe un état `COMMITTED` précédé d'un état qui n'est pas `COMMITTED` (ce qui est exprimé dans la condition). Enfin, si tous les états sont `COMMITTED`, le corps de la boucle est dans le branchement "else" pour tout les $i < MAX-1$ et la condition de la ligne 16 n'est pas vérifiée. Donc l'index est simplement incrémenté jusqu'à ce que i vaille MAX-1, ce qui correspond à la dernière conjonction de la condition du "if". En conclusion, si `JSR` n'est pas cohérent, il existe un tour de boucle pour lequel la condition du "if-then-else" principal est vérifiée, donc `Journal` et `JSR` sont effacés et `curr_index` mis à zéro.

FIG. 6.6 – Argument informel de la correction de l'algorithme d'abandon

JournalStateRegistry

Le tableau `JournalStateRegistry[jid]` contient l'historique des états d'effacement du journal dont l'identifiant est `jid`. Comme expliqué à la Section 4.3.1.1, et plus précisément à la page 117, étant donné que seuls deux bits suffisent pour représenter un état et que la plus petite granularité pour un tableau en C est l'octet, le tableau `JournalStateRegistry[jid]` est un tableau d'octets, où chaque case du tableau contient quatre états (voir Figure 6.7) :

```
typedef unsigned char   u1;     /* unsigned  8 bit integer (1 byte) */
#define JOURNAL_STATE_POOL_NB_BYTES  (FLASH_PAGE_SIZE)
typedef u1 journal_state[JOURNAL_STATE_POOL_NB_BYTES];
journal_state JournalStateRegistry[JOURNAL_REGISTRY_DIM];
```

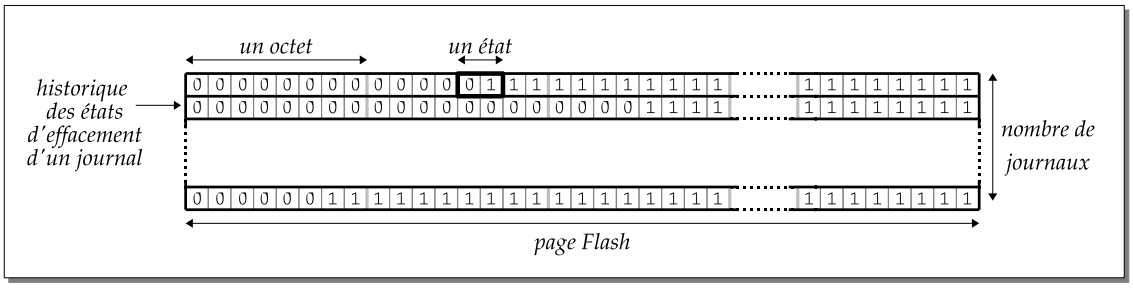


FIG. 6.7 – Représentation du code source de la variable `JournalSTateRegistry`

JournalContext

Cette variable contient, pour chaque journal, un pointeur sur la queue du journal et l'index de l'état d'effacement courant du journal, dans le tableau `JournalStateRegistry` (comme représenté dans la vue globale Figure 6.1) :

```
typedef struct {
    journal_state_index jsi;  /* journal state index */
    u1* pCurrentLog;          /* current log address */
} journal_context;
journal_context JournalContext[JOURNAL_REGISTRY_DIM];
```

Comme nous l'avons également vu à la page 117, un état dans `JournalStateRegistry[jid]` est repéré à la fois par l'index de l'octet dans lequel il est contenu et le numéro indiquant lequel est-ce parmi les quatre états de l'octet (voir Figure 6.8). Ceci est représenté par une union entre un entier et une structure contenant des champs de bits (pour des raisons que nous avons exposées) :

```
typedef unsigned short  u2;    /* unsigned 16 bit integer (2 bytes) */
#define JOURNAL_STATE_NB_BITS  2
typedef union {
    u2  w;
    struct {
        u2 byteNb   :8*sizeof(u2)-JOURNAL_STATE_NB_BITS;
        u2 slotNb   :JOURNAL_STATE_NB_BITS;
    } s;
} journal_state_index;
```

JournalRegistry

Toutes les données intrinsèques à chaque journal sont mémorisées dans cette variable. Comme indiqué dans la vue générale, cela comprend l'emplacement où est stocké le journal (l'adresse de base et la taille de l'espace alloué), la taille de ses entrées et un pointeur sur la fonction d'abandon correspondant à l'opération gérée par ce journal :

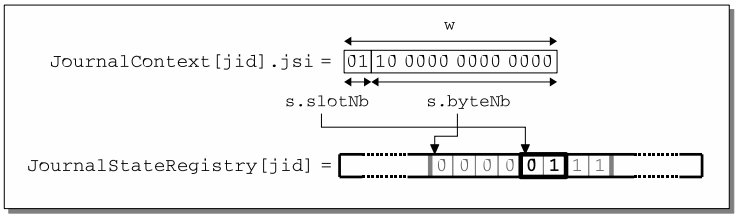


FIG. 6.8 – Représentation d'un index d'état dans `JournalContext`

```
typedef struct {
    u1* pBaseAddress;       /* journal base address */
    u1 bNbSector;           /* journal size */
    u2 wLogSize;            /* journal entry size (without header) */
    PCVFUNCTION abort;      /* abort function of application */
} journal_registry_entry;
journal_registry_entry JournalRegistry[JOURNAL_REGISTRY_DIM];
```

Étant donné que nous décrivons ici uniquement la fonction de gestion de l'effacement d'un journal en cas d'arrachage, nous ne nous servirons que des champs `pBaseAddress` et `bNbSector`, qui permettent d'identifier quel est l'emplacement à effacer.

6.2.2 Code source de `_journal_abortErase`

Le code source de la fonction `void _journal_abortErase(u1 jid)` est donné en Figure 6.9. Rappelons que cette fonction a pour but de déterminer si un arrachage a interrompu l'effacement du journal dont l'identifiant est `jid`. L'idée est d'analyser le tableau `JournalStateRegistry[jid]` et d'effacer de nouveau le journal si ce tableau n'est pas cohérent, c'est-à-dire s'il n'est pas composé d'une suite d'états "terminés" suivie d'une suite d'états "non utilisés". La valeur de l'état d'effacement courant du journal, stockée dans `JournalContext[jid].jsi`, est quant à elle mise à jour à l'index du premier état "non utilisé" rencontré.

Plus précisément, une variable locale `jsi` permet de parcourir le tableau des états d'effacement `JournalStateRegistry[jid]` et une variable locale `bPreviousState` permet de conserver la valeur de l'ancienne case du tableau (voir Figure 6.10). Chaque état à l'index `jsi` est calculé par une fonction auxilliaire `getJournalState` dont la spécification est la suivante :

```
/*@ requires  ( 0<=jid<JOURNAL_REGISTRY_DIM ) &&
  @           ( 0<=index.s.byteNb<JOURNAL_STATE_POOL_NB_BYTES )
  @ assigns \nothing
  @ ensures \result ==  GetState(JournalStateRegistry[jid][index.s.byteNb],
  @                              index.s.slotNb) */
static u1 getJournalState(u1 jid, journal_state_index index);
```

où la fonction logique `GetState(b,p)` extrait les deux bits en position `p` dans l'octet `b`.

À chaque case du tableau, si l'état est en cours (s'il vaut $(01)_b$) ou s'il est impossible (s'il vaut $(10)_b$) alors le journal doit être effacé. S'il est terminé (s'il vaut $(00)_b$), mais que l'état précédent ne l'est pas (c'est-à-dire s'il est non utilisé, puisque les autres cas ont déjà été exclus), alors le journal doit également être effacé. Dans les autres cas, à savoir lorsque l'état est terminé et son précésseur l'est également, ou lorsqu'il est non utilisé, alors le parcours du tableau continue. Lorsque l'état est non utilisé et que son prédécesseur était terminé, alors la variable `JournalContext[jid].jsi` est mise à jour à cet état puisqu'il s'agit du premier état non utilisé du tableau.

```
void _journal_abortErase(u1 jid)
{
    journal_state_index jsi;
    u1 bState;
    u1 bPreviousState;
    u1 bErase;
    u1 i;

    /* scan journal state pool */
    jsi.w = 0;
    bPreviousState =0;
    bErase = 0;     /* erase not needed */

    while (jsi.w < JOURNAL_STATE_POOL_NB_STATES) {
        bState = getJournalState(jid,jsi);
        switch (bState) {
            case STATE_JOURNAL_ERASE_FAKE:
                bErase = 1;
                break;
            case STATE_JOURNAL_ERASE_ONGOING:
                bErase = 1;
                break;
            case STATE_JOURNAL_ERASE_COMMITTED:
                if ((bPreviousState != STATE_JOURNAL_ERASE_COMMITTED)
                    || (jsi.w == (JOURNAL_STATE_POOL_NB_STATES - 1))) {
                    bErase = 1;
                }
                bPreviousState = STATE_JOURNAL_ERASE_COMMITTED;
                break;
            case STATE_JOURNAL_ERASE_UNUSED:
                if (bPreviousState == STATE_JOURNAL_ERASE_COMMITTED) {
                    JournalContext[jid].jsi.w = jsi.w;
                }
                bPreviousState = STATE_JOURNAL_ERASE_UNUSED;
                break;
        }

        /* test if a tearing occured */
        if (bErase) {
            for (i=0; i < JournalRegistry[jid].bNbSector; i++) {
                _flash_eraseSector(JournalRegistry[jid].pBaseAddress
                                   + i*FLASH_SECTOR_SIZE);
            }
            _flash_erasePage(JournalStateRegistry[jid]);
            JournalContext[jid].jsi.w = 0;
            return;
        }
        ++jsi.w;
    }
 }
```

FIG. 6.9 – Code source de la fonction `_journal_abortErase`

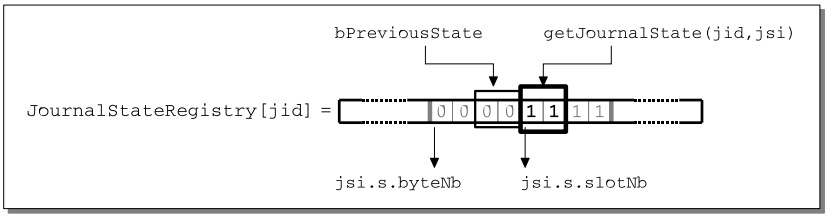


FIG. 6.10 – Parcours du tableau par la fonction `_journal_abortErase`

Les effacements du journal et du tableau d'états sont effectués par des fonctions d'effacement primitives de la mémoire Flash. Comme déjà mentionné, il existe deux granularité d'effacement de mémoire Flash : un effacement par *page* et un effacement par *secteur* (ensemble de plusieurs pages). Il existe donc deux fonctions primitives d'effacement, dont les spécifications sont les suivantes :

```
/*@ assigns pDest[0 .. FLASH_SECTOR_SIZE-1]
  @ ensures ErasedMemory(pDest, FLASH_SECTOR_SIZE) */
extern void _flash_eraseSector  (u1* pDest);
/*@ assigns pDest[0 .. FLASH_PAGE_SIZE-1]
  @ ensures ErasedMemory(pDest, FLASH_PAGE_SIZE) */
extern void _flash_erasePage    (u1* pDest);
```

où le prédicat `ErasedMemory` indique que chaque octet du bloc donné en argument est dans un état effacé pour la mémoire Flash (i.e. tous ses bits sont à 1).

En résumé, soit le journal des états d'effacement était cohérent, auquel cas seule la variable `JournalContext[jid].jsi` est mise à jour, soit il ne l'était pas, alors le journal est effacé, puis le journal des états est lui même également effacé et la variable `JournalContext[jid].jsi` pointe sur le premier état du tableau.

6.3 Vérification fonctionnelle

6.3.1 Étape de spécification

Nous avons choisi de présenter la fonction `_journal_abortErase` car le code source du corps de la fonction est suffisamment complexe pour justifier la preuve de sa correction, i.e. la preuve qu'il assure effectivement ce pour quoi il a été conçu. En effet, il n'est pas absolument clair que le journal est de nouveau effacé si le journal des états n'était pas cohérent.

Nous allons donc spécifier formellement cette fonction dans le langage d'annotation de Caduceus, en suivant la méthode générale présentée en Section 4.2.2, qui préconise une description complète du comportement de la fonction. D'après cette méthode, la première étape est d'identifier les variables potentiellement modifiées par la fonction. Dans notre cas, si aucun arrachage n'a eu lieu, la fonction ne fait que mettre à jour la variable `JournalContext[jid].jsi.w` pour qu'elle pointe vers le premier état non utilisé du journal. Mais si un arrachage a eu lieu, le journal est de nouveau effacé, i.e. tout le bloc mémoire commençant à la tête du journal (donnée par le pointeur `JournalRegistry[jid].pBaseAddress`) et dont la taille est celle du journal (donnée par le nombre de secteurs `JournalRegistry[jid].bNbSector` multiplié par la taille d'un secteur). De plus, dans le cas d'un arrachage, le journal des états d'effacement `JournalStateRegistry[jid]`

est également effacé (puisqu'il était incohérent) et la variable `JournalContext[jid].jsi.w` pointe sur le premier état du tableau. La clause `assigns` de la fonction est donc la suivante :

```
@ assigns JournalStateRegistry[jid][0 .. JOURNAL_STATE_POOL_NB_BYTES-1] ,
@         JournalContext[jid].jsi.w ,
@         JournalRegistry[jid].pBaseAddress
@                 [0 .. JournalRegistry[jid].bNbSector*FLASH_SECTOR_SIZE-1]
```

La méthode générale de l'étape de spécification propose ensuite de décrire les valeurs attendues, à la fin de l'exécution, de toutes les variables potentiellement modifiées par la fonction. Dans notre exemple, il y a deux cas : soit un arrachage a eu lieu, soit aucun arrachage n'a eu lieu, ceci étant repéré par la cohérence du tableau des états d'effacement. La cohérence du tableau signifie qu'il existe un numéro d'octet et un numéro d'état dans cet octet tels qu'avant cet état, tous les états sont *terminés*, et qu'après cet état (cet état y compris), tous les états sont *non utilisés* (voir Figure 6.11). Un prédicat `is_consistent` est défini dans le langage d'annotation pour représenter la cohérence d'un tableau :

```
/*@ predicate is_consistent (int * tab, int b, int s) {
  @   (\forall int j; 0<=j<b => is_committed_byte(tab[j])) &&
  @   (\forall int k; b<k<JOURNAL_STATE_POOL_NB_BYTES => is_unused_byte(tab[k])) &&
  @   (\forall int m; 0<=m<s =>
  @         (GetState(tab[b],m)==STATE_JOURNAL_ERASE_COMMITTED)) &&
  @   (\forall int n; s<=n<4 =>
  @         (GetState(tab[b],n)==STATE_JOURNAL_ERASE_UNUSED))
  @ }
*/
```

avec

```
/*@ predicate is_committed_byte (int i)
  @   \forall int n; 0<=n<4 => (GetState(i,n)==STATE_JOURNAL_ERASE_COMMITTED)  */
/*@ predicate is_unused_byte (int i)
  @   \forall int n; 0<=n<4 => (GetState(i,n)==STATE_JOURNAL_ERASE_UNUSED)  */
```

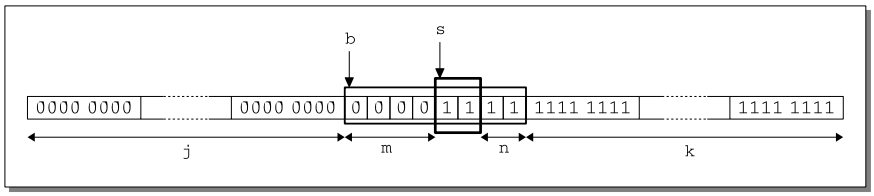


FIG. 6.11 – Définition de la cohérence d'un tableau d'états d'effacement

Avec cette définition de la cohérence, la postcondition de la fonction peut alors être définie par deux cas. Dans le premier cas, le tableau d'états était cohérent à l'appel de la fonction, auquel cas il n'est pas changé par la fonction et le journal non plus. En revanche l'état courant pointé par `JournalContext[jid].jsi` indique le premier état non utilisé, c'est-à-dire celui qui définit la cohérence du tableau :

```
\exists int b; \exists int s;
 is_consistent(\old(JournalStateRegistry[jid]),b,s) &&
 JournalContext[jid].jsi.byteNb == b &&
 JournalContext[jid].jsi.slotNb == s
```

Ce qui peut être simplifié par :

```
is_consistent(\old(JournalStateRegistry[jid]),
              JournalContext[jid].jsi.byteNb,
              JournalContext[jid].jsi.slotNb)
```

Le deuxième cas correspond au fait qu'un arrachage a eu lieu, ce qui signifie que le tableau était incohérent à l'appel de la fonction. En termes du prédicat de cohérence, cela signifie qu'il n'existe pas de `b` et de `s` rendant le tableau cohérent. Dans ce cas, le tableau d'états et le journal sont effacés et l'état courant pointé par `JournalContext[jid].jsi` est initialisé à zéro.

La postcondition peut donc être définie comme suit :

```
@ ensures
@ ( is_consistent(\old(JournalStateRegistry[jid]),
@                 JournalContext[jid].jsi.byteNb,
@                 JournalContext[jid].jsi.slotNb) &&
@   (\forall int k; 0<=k<JOURNAL_STATE_POOL_NB_BYTES =>
@       JournalStateRegistry[jid][k] == \old(JournalStateRegistry[jid][k]))&&
@   (\forall int k; 0<=k<JournalRegistry[jid].bNbSector*FLASH_SECTOR_SIZE =>
@       JournalRegistry[jid].pBaseAddress[k] ==
@       \old(JournalRegistry[jid].pBaseAddress[k]) ) &&
@   (JournalContext[jid].jsi.s.w==\old(JournalContext[jid].jsi.s.w))
@ )
@ ||
@ ( !(\exists int b; \exists int s;
@      0<=b<JOURNAL_STATE_POOL_NB_BYTES => 0<=s<4 =>
@      is_consistent(\old(JournalStateRegistry[jid]),b,s)) &&
@   (ErasedMemory(JournalStateRegistry[jid], JOURNAL_STATE_POOL_NB_BYTES))&&
@   (ErasedMemory(JournalRegistry[jid].pBaseAddress,
@                 JournalRegistry[jid].bNbSector*FLASH_SECTOR_SIZE))&&
@   (JournalContext[jid].jsi.w == 0)
@ )
```

où le prédicat `ErasedMemory`, déjà rencontré, indique que chaque octet du bloc donné en argument est dans un état effacé pour la mémoire Flash.

Concernant la précondition, elle se résume à :

```
@ requires (0 <= jid < JOURNAL_REGISTRY_DIM)
```

En effet, c'est le seul accès à un tableau dans le corps de la fonction. De plus, les seuls appels de fonctions sont ceux aux fonctions d'effacement et l'appel à la fonction `getJournalState`. La précondition de cette dernière fonction requiert deux conditions sur des index de tableau : l'une des conditions est celle ci-dessus, et l'autre stipule que `jsi.s.byteNb` soit compris entre `0` et `JOURNAL_STATE_POOL_NB_BYTES`, où `jsi` est la variable locale parcourant le tableau. Or, cette variable est initialisée à zéro et est incrémentée jusqu'à atteindre `(JOURNAL_STATE_POOL_NB_BYTES-1)`, donc cette deuxième condition est toujours satisfaite.

Remarque. En toute rigueur, la définition de cohérence donnée par le prédicat `is_consistent` a été utilisée pour une question de lisibilité, mais ce prédicat ne peut pas prendre en argument une expression contenant le mot clé `\old` (comme dans la postcondition de la fonction d'abandon). En effet, au lieu de mentionner les cases du tableau à l'appel de la fonction, à savoir `\old(tab[jid][k])`, le prédicat spécifie `\old(tab[jid])[k]`, qui est inchangé dans le modèle mémoire de Caduceus. Par conséquent, la postcondition ne peut être définie comme présenté, avec le prédicat `is_consistent`. Elle doit explicitement décrire les propriétés du prédicat dans la spécification de la fonction.

6.3.2 Étape de validation fonctionnelle

La vérification formelle que le code source implémente la spécification qui a été définie dans la section précédente consiste à prouver les obligations générées par l'outil Caduceus. Comme nous

l'avons décrit dans la Section 4.2, cette étape nécessite la définition d'invariants de boucle, qui peut parfois être délicate. Cette section n'a pas pour vocation d'expliquer les détails techniques de la définition de l'invariant ou de la preuve des obligations, mais plutôt de d'essayer de dégager les leçons d'une telle vérification, par la description des principales difficultés rencontrées pour la preuve de cette fonction. La Figure 6.12 présente quelques chiffres concernant la vérification de la fonction `_journal_abortErase`, que nous allons commenter.

Nombre de lignes de code	45
Nombre de lignes de spécification	35
Nombre de lignes d'annotations de boucle	40
Nombre d'obligations de preuve générées	15
Nombre de lignes de preuves Coq (hors énoncés)	~ 450

FIG. 6.12 – Quelques chiffres de la vérification fonctionnelle de la fonction `_journal_abortErase`

La première constatation concerne le nombre important d'obligations de preuves générées. Ceci est dû à la présence du `switch` dans le corps de la fonction, qui induit un grand nombre de noeuds dans le flot de contrôle. Par exemple, le fait que l'invariant de la boucle `while` est satisfait à chaque tour de boucle doit être prouvé dans les cas où l'état est "en cours", "non utilisé", "mal formé" et "terminé". Étant donné que peu d'éléments changent dans ces différents cas, les preuves sont très similaires. Par exemple, la preuve de l'invariant dans le cas d'un état "mal formé" est strictement identique à celle du cas où il est "en cours". Par conséquent, sur les 450 lignes de preuves, environ 60 pourcents sont recopiés d'une partie existante de la preuve. Le nombre d'obligations n'est donc pas représentatif de la difficulté des preuves.

La deuxième remarque est qu'une grande partie des preuves permettent uniquement d'établir les clauses `assigns` (de la fonction et des boucles). En effet, dans notre exemple, si l'on réduit les annotations uniquement à ces clauses, les preuves représentent déjà 230 lignes de Coq, soit plus de la moitié des preuves totales. Ceci est lié au fait que les fonctions ont beaucoup d'effets de bord, qu'il faut prouver à chaque noeud du flot de contrôle. Cette caractéristique est spécifique (entre autre) aux programmes C de gestion de mémoire, où un grand nombre de variables globales sont utilisées pour manipuler divers composants et où l'objectif principal des fonctions est non pas calculatoire, mais la modification de ces variables globales. Par conséquent, la taille des preuves n'est pas seulement proportionnelle à la difficulté des propriétés à montrer, mais est également très dépendant du nombre d'effets de bord de la fonction.

Notons enfin que l'énoncé de chaque obligation de preuve contient en hypothèse, en plus des invariants globaux, l'ensemble des propriétés de validité générées automatiquement par Caduceus. Par exemple, pour toute variable globale de type tableau, une hypothèse est ajoutée indiquant que les accès aux cases du tableau sont valides pour des index compris entre zéro et la taille du tableau. De même, pour toute paire de variables globales de type tableau, une hypothèse indique qu'elles ne sont pas dans le même segment mémoire (qu'elles n'ont pas la même *adresse de base*). Par conséquent, dans notre cas d'étude, le grand nombre de variables globales, de type tableau voire tableau de tableau, engendre une disjonction considérable de propriétés, ajoutée en hypothèse à toutes les obligations de preuves. À cela s'ajoutent les éventuelles postconditions de fonctions appelées, qui peuvent également être sous la forme d'une disjonction ou conjonction de clauses. Une décomposition à la main de ces disjonctions et conjonctions est très fastidieuse et la décomposition brutale, par la tactique `intuition`, prend quant à elle plus d'une minute à chaque obligation, ce qui alourdit considérablement le développement de la preuve.

Pour conclure, la phase de validation, pour une spécification très précise, et pour une fonction non triviale (flot de contrôle complexe, deux possibilités pour la postcondition, etc) s'avère fastidieuse, mais pour des raisons qui ne sont pas liées uniquement au type des propriétés visées. Sur les 450 lignes de preuves dans notre exemple, seules 85 environ concernaient des preuves de la postcondition ou de l'invariant, et qui soient "nouvelles" (non recopiées). Le cas d'étude du module est donc prometteur pour une généralisation de la méthode de vérification fonctionnelle aux programmes C embarqués, sous réserve d'une étude sur le passage à l'échelle et les améliorations possibles en termes d'automatisation et d'aisance d'utilisation. Notons à ce titre que la version de l'outil qui est en cours de développement propose des progrès dans ce sens, testés sur d'autres exemples industriels, qui sont prometteurs pour le déploiement d'une vérification fonctionnelle efficace.

6.4 Vérification de la propriété d'*anti-tearing* de l'effacement

Nous avons présenté dans le Chapitre 5 une approche pour la vérification de propriétés globales sur un modèle formellement lié au code. Nous avons illustré la méthode avec, entre autres, la propriété d'*anti-tearing* de l'opération d'effacement. Nous présentons dans cette section les détails de sa modélisation et de sa vérification dans le système de preuve Coq.

La propriété d'*anti-tearing* de l'opération d'effacement signifie que si une opération d'effacement d'un journal commence, alors elle se termine obligatoirement par un état où le journal est effacé, même si un ou plusieurs arrachages ont lieu pendant l'opération. Comme nous l'avons vu, la formalisation de cette propriété nécessite une modélisation de chaque fonction sous la forme de transition. Ce système de transitions est extrait à partir de la traduction des spécifications des fonctions dans le modèle mémoire de Caduceus. Chaque fonction f est modélisée par une transition sur les états mémoires calculés par l'outil :

$$(f\_transition\ result\ \overrightarrow{a}\ \overrightarrow{z}\ \overrightarrow{t@}\ \overrightarrow{t}\ ) \equiv Pre_f(\overrightarrow{a}, \overrightarrow{z}, \overrightarrow{t@}) \wedge Post_f(result, \overrightarrow{a}, \overrightarrow{z}, \overrightarrow{t@}, \overrightarrow{t})$$

Les deux fonctions `_journal_secureErase` et `_journal_abortErase` sont donc modélisées par des transitions `erase_transition` et `abort_transition` respectivement, calculées à partir de leurs spécifications. La spécification de `_journal_abortErase` est celle donnée dans la section précédente. Quant à celle de `_journal_secureErase`, elle indique simplement que le tableau d'états doit être cohérent à l'appel de la fonction et que le journal est effacé après l'exécution de la fonction.

Cependant, ces spécifications ne sont pas suffisantes, car le comportement des fonctions doit également être spécifié en cas d'arrachage de la carte, de manière à pouvoir formaliser la propriété d'*anti-tearing*. Grâce à l'extension de Caduceus, présentée dans la Section 4.6, une clause `ensures_interrupt` est ajoutée à la chaque spécification. La fonction de transition a alors la forme suivante :

$$\begin{aligned}&(f\_transition\ result\ \overrightarrow{a}\ \overrightarrow{z}\ \overrightarrow{t@}\ \overrightarrow{t}\ status\ ) \equiv \\ &\quad Pre_f(\overrightarrow{a}, \overrightarrow{z}, \overrightarrow{t@}) \wedge PostGen_f(result, \overrightarrow{a}, \overrightarrow{z}, \overrightarrow{t@}, \overrightarrow{t}, status)\end{aligned}$$

où *status* est de type $InterruptStatus \equiv Val \mid Exn$ et la postcondition $PostGen_f$ est de la forme :

$$\begin{aligned}PostGen_f(result, \overrightarrow{a}, \overrightarrow{z}, \overrightarrow{t@}, \overrightarrow{t}, status) \equiv\ & match\ status\ with \\ & \mid Val \Rightarrow Post_f(result, \overrightarrow{a}, \overrightarrow{z}, \overrightarrow{t@}, \overrightarrow{t}) \\ & \mid Exn \Rightarrow Posti_f(result, \overrightarrow{a}, \overrightarrow{z}, \overrightarrow{t@}, \overrightarrow{t})\end{aligned}$$

où $Post_f$ correspond à la traduction de la clause `ensures` de la fonction et *Posti* à celle de la clause `ensures_interrupt`.

La postcondition d'interruption de la fonction `_journal_secureErase` indique que si le tableau des états est cohérent au moment de l'arrachage, alors l'effacement a été effectué. En effet, la première instruction de la fonction est de changer l'état d'effacement courant du journal pour indiquer qu'un effacement est en cours. Le tableau est donc incohérent jusqu'à ce que l'effacement soit terminé et que l'état courant soit de nouveau modifié. Quant à la fonction `_journal_secureErase`, sa postcondition d'interruption est presque identique à sa postcondition normale : si le tableau était cohérent à l'appel, la fonction ne modifie pas le journal, donc il n'est pas modifié même si un arrachage survient. En revanche, si le tableau était incohérent, et que la fonction est interrompue dans son exécution, ce que l'on peut affirmer est que si le tableau est de nouveau cohérent au moment de l'arrachage, alors le journal a été de nouveau effacé (car le tableau n'est de nouveau cohérent que lorsqu'il est effacé, ce qui est effectué après l'effacement du journal).

Les relations de transition peuvent à présent être définies à partir de ces spécifications et des états calculés par Caduceus. Les états mémoire calculés par Caduceus, pour les deux fonctions, sont : le paramètre `jid` de la fonction, les variables globales `JournalContext`, `JournalRegistry` et `JournalStateRegistry`, la table d'allocation `alloc`, ainsi que les états représentant les segments mémoire du modèle. Ces états comprennent les champs de structures (comme `jsi`, `w`, `pBaseAddress`, etc) et le segment `intP` des pointeurs d'entier, contenant le journal et le tableau d'états d'effacement. Parmi ces variables, celles qui sont modifiées sont `w`, puisque le champ `JournalContext[jid].jsi.w` est mis à jour, et `intP` puisque le journal et le le tableau d'états sont potentiellement modifiés par les fonctions. La transition modélisant la fonction `_journal_secureErase` est donc la suivante :

$$
\begin{array}{l}
(erase\_transition\ jid\ ...\ intP_1\ w_1\ intP_2\ w_2\ res)\ \equiv \\
\quad (\ (0 \leq jid < 3)\ \wedge\ (is\_consistent\ alloc\ intP_1\ (...\ JournalStateRegistry\ ...))\ \wedge\ ...\ )\ \wedge \\
\quad (match\ res\ with \\
\qquad |\ Val\ \Rightarrow\ (ErasedMemory\ alloc\ intP_2\ (...\ pBaseAddress\ ...))\ \wedge\ ... \\
\qquad |\ Exn\ \Rightarrow\ (\ (is\_consistent\ alloc\ intP_2\ (...\ JournalStateRegistry\ ...)) \Rightarrow \\
\qquad\qquad (ErasedMemory\ alloc\ intP_2\ (...\ pBaseAddress\ ...))\ )\ )
\end{array}
$$

De même, la transition qui modélise la fonction `_journal_abortErase` est la suivante :

$$
\begin{array}{l}
(abort\_transition\ jid\ ...\ intP_1\ w_1\ intP_2\ w_2\ res)\ \equiv \\
\quad (\ (0 \leq jid < 3)\ \wedge\ ...\ )\ \wedge \\
\quad (match\ res\ with \\
\qquad |\ Val\ \Rightarrow\ (\ (is\_consistent\ alloc\ intP_1\ (...\ JournalStateRegistry\ ...)) \wedge \\
\qquad\qquad (acc\ intP_1\ (...\ pBaseAddress\ ...)) = (acc\ intP_2\ (...\ pBaseAddress\ ...))\ \wedge\ ...\ ) \\
\qquad\qquad \vee\ (\ \neg(is\_consistent\ alloc\ intP_1\ (...\ JournalStateRegistry\ ...)) \wedge \\
\qquad\qquad (ErasedMemory\ alloc\ intP_2\ (...\ pBaseAddress\ ...))\ \wedge\ ...\ ) \\
\qquad |\ Exn\ \Rightarrow\ (\ (is\_consistent\ alloc\ intP_1\ (...\ JournalStateRegistry\ ...)) \wedge \\
\qquad\qquad (acc\ intP_1\ (...\ pBaseAddress\ ...)) = (acc\ intP_2\ (...\ pBaseAddress\ ...))\ \wedge\ ...\ ) \\
\qquad\qquad \vee\ (\ \neg(is\_consistent\ alloc\ intP_1\ (...\ JournalStateRegistry\ ...)) \wedge \\
\qquad\qquad (\ (is\_consistent\ alloc\ intP_2\ (...\ JournalStateRegistry\ ...)) \Rightarrow \\
\qquad\qquad\qquad (ErasedMemory\ alloc\ intP_2\ (...\ pBaseAddress\ ...))\ )\ )\ )
\end{array}
$$

Dans ces deux définitions, n'apparaissent que les propriétés qui nous intéressent pour la preuve d'*anti-tearing*. Par exemple, la valeur de `JournalContext[jid].jsi.w`, qui est modifiée par les

deux fonctions, fait partie des propriétés non mentionnées.

L'opération d'effacement peut maintenant être définie à l'aide de ces relations de transition. Elle représente le fait qu'un effacement a commencé, suivi d'une suite finie d'arrachage de la carte. Étant donné que la fonction d'abandon est appelée après chaque arrachage, l'opération est soit un effacement non interrompu, soit un effacement interrompu, suivi d'une séquence d'abandons interrompus se terminant par un abandon non interrompu. Cette opération est définie par :

$$
\begin{aligned}
&(erase\_op\ jid\ ...\ intP_1\ w_1\ intP_2\ w_2\ res)\ \equiv \\
&\qquad (erase\_transition\ jid\ ...\ intP_1\ w_1\ intP_2\ w_2\ Val) \\
&\vee\ (\ \exists\ intP_3.\ \exists\ w_3.\ \ (erase\_transition\ jid\ ...\ intP_1\ w_1\ intP_3\ w_3\ Exn) \\
&\qquad\qquad\qquad \wedge (abort\_seq\ jid\ ...\ intP_3\ w_3\ intP_2\ w_2\ )\ )
\end{aligned}
$$

où une séquence finie d'appels à la fonction d'abandon est définie de façon inductive par :

$$
\begin{aligned}
&(abort\_seq\ jid\ ...\ intP_1\ w_1\ intP_2\ w_2)\ \equiv \\
&\qquad (abort\_transition\ jid\ ...\ intP_1\ w_1\ intP_2\ w_2\ Val) \\
&\vee\ (\ \exists\ intP_3.\ \exists\ w_3.\ \ (abort\_transition\ jid\ ...\ intP_1\ w_1\ intP_3\ w_3\ Exn) \\
&\qquad\qquad\qquad \wedge (abort\_seq\ jid\ ...\ intP_3\ w_3\ intP_2\ w_2\ )\ )
\end{aligned}
$$

La propriété d'*anti-tearing* de l'opération d'effacement peut alors être définie par :

$$
\begin{aligned}
&Theorem\ anti\_tearing\_erase\ : \\
&\quad \forall\ jid.\ \ ...\ \ \forall\ intP_1.\ \ \forall\ w_1.\ \ \forall\ intP_2.\ \ \forall\ w_2. \\
&\quad (erase\_op\ jid\ ...\ intP_1\ w_1\ intP_2\ w_2) \Rightarrow (ErasedMemory\ alloc\ intP_2\ (...\ pBaseAddress\ ...))
\end{aligned}
$$

Cette propriété a été démontrée en une vingtaine de lignes de preuve en Coq.

Remarque : Si seul le théorème d'*anti-tearing* est visé, les spécifications des fonctions peuvent être réduites aux propriétés nécessaires à la preuve de ce théorème. Par exemple, les propriétés sur la variable `JournalContext[jid].jsi.w` ne sont pas nécessaires (ce qui signifie que les parties non mentionnées dans les relations de transitions définies précédemment sont vides). En d'autres termes, la spécification peut représenter une *abstraction* du comportement du code, dans le but de prouver une propriété donnée. La preuve de la propriété n'est pas nécessairement simplifiée, mais la preuve de *simulation* de l'abstraction, i.e. la preuve des obligations, peut quant à elle être fortement allégée. Cette démarche perd en revanche ses avantages dès lors que plusieurs propriétés sont visées, car plusieurs abstractions différentes sont nécessaires, avec autant de preuves de simulation. Dans ce cas, une spécification "complète", dont la correction est prouvée une seule fois, peut être utilisée pour toutes les propriétés.

Le fait que ce théorème est transférable au niveau du code source est justifié, d'une part, par la quantification universelle sur les états finaux, et, d'autre part, par la précision de la spécification qui assure que la relation de transition décrit exactement l'état calculé par le programme.

Conclusion

D'un défi, l'utilisation des méthodes formelles dans le monde industriel a évolué en nécessité. De nombreux travaux de recherche ont pour vocation l'étude de différentes approches pour une meilleure intégration et accessibilité des méthodes formelles. Cette thèse s'inscrit dans cet objectif global, dans le monde de la carte à puce.

La technologie de la carte à puce a nettement évolué ces quinze dernières années, avec l'apparition des nouvelles générations de plate-formes, ouvertes et multi-applicatives. Ces plate-formes offrent plus de flexibilité et de portabilité au logiciel embarqué, mais complexifient considérablement le modèle sécuritaire de la carte. La technologie Java Card, la plus répandue des ces nouvelles plate-formes, a fait l'objet de nombreuses études, soit au niveau de la plate-forme elle-même (fournie par les fabricants de cartes), soit au niveau des applications écrites dans le langage Java Card (définies par les fournisseurs d'applications).

La première partie de nos travaux a contribué à démontrer formellement que la plate-forme Java Card propose un environnement sécurisé pour l'hébergement et l'exécution de diverses applications embarquées dans une carte à puce. En particulier, nous avons prouvé que la plate-forme assure l'*isolation* des données d'applications chargées sur une même carte. Nos travaux se démarquent des autres études sur le partage des données et les flux d'information entre applications par le fait que la vérification est effectuée au niveau de la plate-forme, i.e. du côté des fabricants de carte, qui veulent assurer la protection des données des applications hébergées. En effet, les travaux existants sur la propriété d'isolation proposent une analyse du code source des applications, en supposant la plate-forme correcte. Quant aux travaux sur la plate-forme, ils se concentrent sur les propriétés du vérificateur de *bytecode*, comme la sûreté de typage, ou sur une certification sécuritaire, par les Critères Communs. Nos travaux apportent donc une nouvelle contribution à la vérification formelle de la technologie Java Card.

Les limitations de notre étude, mis à part l'exclusion du cas des interfaces partageables, concernent l'impossibilité de vérifier certaines conditions de l'isolation, du fait du niveau d'abstraction du modèle formel. Par ailleurs, le lien informel entre le modèle et une implémentation donnée de la plate-forme dans un produit ne permet pas d'affirmer que les propriétés vérifiées par le modèle le sont par le code source. Cette dernière limitation a été au coeur de nos préoccupations dans la deuxième partie de nos travaux.

Dans cette deuxième partie, nous avons proposé d'étendre le domaine d'application des méthodes formelles aux couches logicielles de plus bas niveau du système d'exploitation, généralement supposées correctes dans les études formelles existantes. Nous nous sommes intéressés aux préoccupations et aux points critiques du moment, ce qui nous a amené à choisir un module de gestion de mémoire Flash comme cas d'étude. En effet, l'utilisation de cette nouvelle technologie dans la carte à puce a nécessité l'adaptation ou le développement d'algorithmes complexes, tels que ceux de la gestion de l'arrachage. Or, aucune modélisation formelle n'existait pour cette

nouvelle technologie. Nous nous sommes donc proposés d'utiliser les méthodes formelles pour démontrer la correction de ces algorithmes et celle des adaptations, à la mémoire Flash, du savoir-faire des développeurs. En particulier, la preuve formelle de la propriété d'*anti-tearing* de l'opération d'effacement en mémoire Flash, est un apport essentiel dans l'objectif de garantir un bonne utilisation de ce type de mémoire. Les bénéfices de ces travaux pourront être étendus à la preuve d'autres propriétés de la mémoire Flash ou la preuve de la propriété d'*anti-tearing* d'autres opérations.

Au delà du contexte de la carte à puce, nos travaux de vérification du système d'exploitation ont nécessité l'extension d'outils existants et la définition de méthodes de vérification formelle, dont les résultats s'appliquent à tous les domaines de modélisation et de preuve de propriétés de programmes.

Comme nous l'avons mentionné, un de nos objectifs était de conserver un lien formel entre le code source et le modèle sur lequel la vérification était faite. Pour cela, nous avons commencé par la vérification de propriétés locales et fonctionnelles, à l'aide de l'outil Caduceus. Cet outil permet une preuve sur le code source, mais n'accepte pas certaines constructions du langage C. Nous nous sommes donc penchés sur la formalisation du langage C et nous avons proposé des solutions pour modéliser formellement les types unions et l'opération de cast. Il s'agit de constructions dont la formalisation est délicate et qui ne sont pas considérées dans les formalisations existantes du langage. Nous avons défini une solution générale de gestion dynamique des casts et des unions, dans un modèle où la mémoire est séparée en emplacements disjoints. Cette solution n'a été implémentée que dans le cas où une gestion statique est possible. La solution dynamique doit quant à elle être accompagnée d'une analyse statique la rendant praticable. Cette solution reste à être validée par une implémentation et des expérimentations de son utilisation en pratique.

L'outil Caduceus a également été étendu de manière à pouvoir représenter et prouver les propriétés d'un programme en cas de l'interruption soudaine de son exécution. Cette extension permet de définir la propriété d'*anti-tearing*, essentielle dans le monde de la carte à puce où un état cohérent doit être assuré en cas d'arrachage de la carte du terminal. Déjà étudié pour les programmes Java Card, spécifiques aux cartes à puces, l'arrachage est ici modélisé pour les programmes C. Cette modélisation ne permet de modéliser, pour le moment, que les données persistantes de la carte.

Par ailleurs, nous avons également défini une méthodologie permettant la vérification de propriétés temporelles globales sur un modèle formellement lié au code. Cette méthodologie innove par le fait qu'elle permet de gérer un code existant et que le modèle peut être plongé dans des logiques d'ordre supérieur. Elle utilise l'outil Caduceus pour l'extraction automatique d'un système de transitions à partir d'annotations vérifiées par le programme. Cette méthodologie a été utilisée dans le cadre de nos travaux, mais l'étude de son application à différents types de propriétés et à d'autres domaines reste à faire.

Les différentes perspectives de nos travaux se distinguent en deux aspects : d'une part la consolidation et l'étude des extensions théoriques des travaux, et d'autre part la mise en pratique des résultats à plus grande échelle dans la carte à puce.

La solution générale que nous avons proposée pour définir un modèle mémoire du langage C avec une gestion dynamique des unions et des casts ne peut être mise en application sans une

analyse statique réduisant au maximum la lourdeur d'une telle approche. Cette solution doit donc impérativement être implémentée afin de tester si elle est viable. En d'autres termes, des expérimentations doivent être menées pour déterminer s'il est possible de définir des analyses statiques suffisamment fines pour l'utilisation de cette solution en pratique.

La modélisation de l'arrachage doit également être complétée de manière à représenter le fait que les données volatiles sont perdues lors de l'arrachage. Cela permettrait de prouver que leurs valeurs (indéfinies après un arrachage) ne sont pas utilisées.

Concernant la méthodologie proposée pour la vérification de propriétés de haut niveau à partir du code source, elle mérite une analyse approfondie de ses avantages et ses inconvénients, par rapport aux autres approches existantes. Déterminer quel genre de propriétés est plus facilement exprimable à l'aide de cette méthode permettrait de lier la méthode à une logique temporelle, où l'utilisateur pourrait plus aisément exprimer les propriétés visées. Cette problématique est liée à la question ouverte de l'identification d'une logique pour laquelle la validité des propriétés vérifiées par le système de transitions peut être transférée au code source. Par ailleurs, une étude comparative avec la vérification de propriétés par *model-checking*, où le modèle est extrait du code source, permettrait également de mieux positionner notre approche. Notons également que la méthode pourrait être étendue à d'autres langages de programmation, comme Java, par l'intermédiaire d'outils de vérification de programme, comme Krakatoa (similaire à Caduceus mais pour les programmes Java). Enfin, l'application à la certification peut également être étudiée, afin de rendre formel le lien entre le modèle de plus bas niveau et l'implémentation.

D'un point de vue plus pratique, le module de gestion de mémoire Flash n'était qu'un cas d'étude pour le déploiement de la vérification formelle de programmes C du système d'exploitation. Le passage à l'échelle, l'amélioration de l'automatisation et le développement de bibliothèques pour une plus grande facilité de vérification font partie de nos objectifs futurs.

En particulier, initier un transfert vers les équipes de développement permettrait l'utilisation de techniques formelles dès la phase de spécification. Le choix de l'outil Caduceus, outre les raisons déjà évoquées dans ce mémoire, était fortement motivé par son langage de spécification syntaxiquement et sémantiquement proche du langage de programmation. En effet cela permet au langage, à l'image de JML, d'être facilement utilisable par les développeurs. Notre objectif d'introduire ce langage dans le processus de spécification des programmes embarqués est double. Tout d'abord, indépendemment de tout vérification, nos travaux ont prouvé la valeur ajoutée d'une spécification formelle par rapport aux commentaires informels insérés dans le code, en termes de cohérence, d'exactitude, d'absence d'ambiguïté, etc. Le travail de formalisation a permis, en collaboration avec les développeurs, de lever certaines incertitudes sur le comportement des programmes et d'identifier les propriétés cruciales à vérifier. De plus, une spécification formelle constitue une documentation précise du code, très utile pour la compréhension, la réutilisation ou le partage du code. Notre deuxième objectif est justement de pouvoir bénéficier de cette première étape de formalisation pour la vérification formelle du système. En effet, une partie non négligeable du temps passé à la vérification du module de gestion de la Flash a été dédiée à la compréhension du code et à la définition de sa spécification. Il est clair que beaucoup des bénéfices de la vérification formelle sont perdus si les propriétés sont définies par rapport à une analyse du code plutôt qu'à partir de ce qui est attendu de la fonction au moment où elle est programmée.

Ce transfert de la phase de spécification peut être accompagnée d'une phase de vérification de propriétés "simples" utilisant des méthodes automatiques (comme Simplify). Nous n'avons pas testé de méthodes automatiques dans nos travaux, du fait du type de propriétés que nous voulions prouver. Une analyse de faisabilité reste donc à faire pour la vérification automatique

de propriétés simples du code source du système d'exploitation embarqué.

Enfin, l'amélioration de l'automatisation des preuves, qu'elles soient fonctionnelles ou de haut niveau, permettrait de profiter au maximum des bénéfices de la vérification formelle, qui pourrait alors être considérée comme une technique avancée de la validation.

Bibliographie

[1] Jean-Raymond Abrial. *The B-book : Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 2004. Online First issue, to appear in print.

[3] June Andronick and Boutheina Chetali. Proving the Source Code of a Smart Card Operating System using Formal Methods. In *International Conference on Research in Smart Cards (Esmart'04)*, 2004.

[4] June Andronick, Boutheina Chetali, and Olivier Ly. Formal Modelling and Verification of Java Card Applet Isolation Properties. *Submitted at Journal of Computer Security*.

[5] June Andronick, Boutheina Chetali, and Olivier Ly. Formal Verification of the Integrity Property in Java Card Technology. In *International Conference on Research in Smart Cards (Esmart'03)*, 2003.

[6] June Andronick, Boutheina Chetali, and Olivier Ly. Using Coq to Verify Java Card Applet Isolation Properties. In D. A. Basin and B. Wolff, editors, *International Conference on Theorem Proving in Higher Order Logics (TPHOLs'03)*, volume 2758 of *LNCS*, pages 335–351. Springer-Verlag, 2003.

[7] June Andronick, Boutheina Chetali, and Christine Paulin-Mohring. Formal Verification of Security Properties of Smart Card Embedded Source Code. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *Formal Methods, International Symposium of Formal Methods Europe (FM'05)*, volume 3582 of *LNCS*, pages 302–317. Springer-Verlag, 2005.

[8] Ade Azurat and Wishnu Prasetya. A Survey on Embedding Programming Logics in a Theorem Prover. Technical report, Institute of Information and Computing Sciences, Utrecht University, 2002.

[9] René Barjavel. *La Nuit des Temps*. 1968.

[10] Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass - Java with Assertions. In K. Havelund and G. Rosu, editors, *Workshop on Runtime Verification 2001*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.

[11] Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Mariela Pavlova. Enforcing High-Level Security Properties For Applets. In *Sixth Smart Card Research and Advanced Application IFIP Conference (CARDIS'04)*, 2004.

[12] Gilles Barthe, Guillaume Dufay, Line Jakubiec, and Simão Melo de Sousa. A Formal Correspondence between Offencive and Defensive JavaCard Virtual Machines. In A. Cortesi, editor, *Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI'02)*, volume 2294 of *LNCS*, pages 32–45. Springer-Verlag, 2002.

[13] Gilles Barthe, Guillaume Dufay, Line Jakubiec, Bernard Paul Serpette, and Simão Melo de Sousa. A Formal Executable Semantics of the JavaCard Platform. In D. Sands, editor, *European Symposium on Programming Languages and Systems (ESOP '01)*, volume 2028 of *LNCS*, pages 302–319. Springer-Verlag, 2001.

[14] Bernhard Beckert and Wojciech Mostowski. A Program Logic for Handling JAVA CARD's Transaction Mechanism. In M. Pezzè, editor, *Fundamental Approaches to Software Engineering (FASE'03)*, volume 2621 of *Lecture Notes in Computer Science*, pages 246–260. Springer, 2003.

[15] Françoise Bellegarde, Julien Groslambert, Marieke Huisman, Olga Kouchnarenko, and Jacques Julliand. Verification of Liveness Properties with JML. Technical Report RR-5331, INRIA, 2004.

[16] Gérard Berry. The Foundations of Esterel. pages 425–454, 2000.

[17] Didier Bert, Sylvain Boulmé, Marie-Laure Potet, Antoine Requet, and Laurent Voisin. Adaptable Translator of B Specifications to Embedded C Programs. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *International Symposium of Formal Methods Europe (FME 2003)*, volume 2805 of *LNCS*, pages 94–113. Springer-Verlag Heidelberg, 2005.

[18] Gustavo Betarte, Boutheina Chetali, Eduardo Giménez, and Claire Loiseaux. Formavie : Formal Modelling and Verification of the JavaCard 2.1.1 Security Architecture. In *E-SMART 2002*, pages 213–231, 2002.

[19] Pierre Bieber, Jacques Cazin, Pierre Girard, Jean Louis Lanet, Virginie Wiels, and Guy Zanon. Checking Secure Interactions of Smart Card Applets : Extended Version. *Journal of Computer Security*, 10(4) :369–398, 2002.

[20] Sandrine Blazy and Xavier Leroy. Formal Verification of a Memory Model for C-like Imperative Languages. In *International Conference on Formal Engineering Methods (ICFEM 2005)*, volume 3785 of *LNCS*, pages 280–299. Springer-Verlag, 2005.

[21] The BOM project. `http://lifc.univ-fcomte.fr/~tatibouet/WEBBOM/`.

[22] Richard Bornat. Proving Pointer Programs in Hoare Logic. In *International Conference on Mathematics of Program Construction (MPC'00)*, pages 102–126. Springer-Verlag, 2000.

[23] Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John van Tassel. Experience with Embedding Hardware Description Languages in HOL. In V. Stavridou and T. F. Melham and R. T. Boute, editor, *International Conference on Theorem Provers in Circuit Design : Theory, Practice and Experience (IFIP TC10/WG 10.2)*, volume A-10 of *IFIP Transactions A : Computer Science and Technology*, pages 129–156, Nijmegen, The Netherlands, 1992. North-Holland/Elsevier.

[24] Chetali Boutheina, Claire Loiseaux, Eduardo Gimenez, and Olivier Ly. An Interpretation of the Common Criteria EAL7 level : Formal Modeling of the Java Card Virtual Machine. In *3rd International Common Criteria Conference (ICCC'02)*, 2002.

[25] Boutheina Chetali and Marie-Noëlle Lepareux and Quang-Huy Nguyen. An Automated Testing Experiment for Multi-layered Embedded C Code. In *IEEE ISoLA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation*, 2005.

[26] Cees-Bart Breunesse, Néstor Cataño, Marieke Huisman, and Bart Jacobs. Formal Methods for Smart Cards : an Experience Report. *Science of Computer Programming*, 55(1–3) :53–80, 2005.

[27] Lilian Burdy, Jean Louis Lanet, and Antoine Requet. Java Applet Correctness : A Developer-Oriented Approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *International Symposium of Formal Methods Europe (FME'03)*, volume 2805 of *LNCS*, pages 422–439. Springer-Verlag, 2003.

[28] Lilian Burdy and A. Requet. Jack : Java Applet Correctness Kit, 2002.

[29] Rod Burstall. Some Techniques for Proving Correctness of Programs which Alter Data Structures. *Machine Intelligence*, 7 :23–50, 1972.

[30] Egon Börger. The Origins and the Development of the ASM Method for High Level System Design and Analysis. *Journal of Universal Computer Science*, 8(1) :2–74, 2002.

[31] Ludovic Casset. *Construction Correcte de Logiciels pour Carte à Puce*. PhD thesis, Aix-Marseille II University, 2002.

[32] Ludovic Casset. Development of an Embedded Verifier for Java Card Byte Code Using Formal Methods. In L.-H. Eriksson and P. Lindsay, editor, *International Symposium of Formal Methods Europe (FME'02)*, volume 2391 of *LNCS*, pages 290–309. Springer-Verlag, 2002.

[33] The Caveat Project. `http://www-drt.cea.fr/Pages/List/lse/LSL/Caveat/index.html/`.

[34] Zhiqun Chen. *Java Card Technology for Smart Cards : Architecture and Programmer's Guide*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[35] Yoonsik Cheon and Gary T. Leavens. A Simple and Practical Approach to Unit Testing : The JML and JUnit Way. In B. Magnusson, editor, *European Conference on Object-Oriented Programming (ECOOP'02)*, volume 2374 of *LNCS*, pages 231–255. Springer, 2002.

[36] Boutheina Chetali and Nicolas Rousset. Formal Verification of an Epurse. In *International Conference on Research in Smart Cards (Esmart'05)*, 2005.

[37] Edmund M. Clarke and Jeannette M. Wing. Formal Methods : State of the Art and Future Directions. *ACM Computing Surveys*, 28(4) :626–643, 1996.

[38] Common Criteria. `http://www.commoncriteria.org/`.

[39] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *International Conference on Software Engineering (ICSE'00)*, pages 439–448. ACM Press, 2000.

[40] Simão Melo de Sousa. *Outils et Techniques pour la Vérification Formelle de la Plate-forme JavaCard*. PhD thesis, INRIA/Université de Nice-Sophia Antipolis, 2003.

[41] Claude Delannoy. *Langage C*.

[42] Jürgen Dethloff. Identification System Safeguarded Against Misuse, 1976. Brevet n°US4105156, déposé le 6 septembre 1976, publié le 8 août 1978. Inventeur : Jürgen Dethloff. Déposant : Jürgen Dethloff.

[43] Werner Dietl, Peter Müller, and Arnd Poetzsch-Heffter. A Type System for Checking Applet Isolation in Java Card. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *LNCS*, pages 129–150. Springer-Verlag, 2004.

[44] Lydie du Bousquet and Hugues Martin. Automatic Test Generation for Java-Card Applets. In I. Attali and T. P. Jensen, editors, *International Workshop on Java on Smart Cards : Programming and Security (JavaCard'00)*, volume 2041 of *LNCS*, pages 121–136. Springer-Verlag, 2001.

[45] Guillaume Dufay. *Vérification Formelle de la Plate-forme Java Card*. PhD thesis, Université de Nice Sophia-Antipolis, 2003.

[46] Jules K. Ellingboe. Active Electrical Card Device, 1970. Brevet n°US3637994, déposé le 19 octobre 1970, publié le 25 janvier 1972. Inventeur : Jules K. Ellingboe. Déposant : TRW Inc.

[47] Marc Éluard and Thomas P. Jensen. Secure Object Flow Analysis for Java Card. In *Smart Card Research and Advanced Application Conference (CARDIS'02)*, pages 97–110. USENIX, 2002.

[48] ESC/Java. `http://research.compaq.com/SRC/esc/`.

[49] ESC/Java2. `http://www.sos.cs.ru.nl/research/escjava`.

[50] Jean-Christophe Filliâtre. The Why Verification Tool. `http://why.lri.fr/`.

[51] Jean-Christophe Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4) :709–745, 2003.

[52] Jean-Christophe Filliâtre and Claude Marché. Multi-Prover Verification of C Programs. In *Sixth International Conference on Formal Engineering Methods (ICFEM)*, volume 3308 of *LNCS*, pages 15–29, Seattle, 2004. Springer-Verlag.

[53] Jean-Christophe Filliâtre, Claude Marché, and Thierry Hubert. The Caduceus tool for the Verification of C Programs. `http://why.lri.fr/caduceus/`.

[54] Formal Methods Home Page. `http://www.afm.sbu.ac.uk`.

[55] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37(2) :138–163, 2005.

[56] Joseph A. Goguen and Jose Meseguer. Security Policy and Security Models. In *Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, 1982.

[57] Joseph A. Goguen and Jose Meseguer. Unwinding and interference control. In *Symposium on Security and Privacy*, pages 75–86. IEEE Computer Society Press, 1984.

[58] Gilles Grimaud. *Camille : un Système d'Exploitation Ouvert pour Carte à Microprocesseur*. PhD thesis, Université de Lille, 2000.

[59] John Wolfgang Halpern and William Ward. Digital Data Carrying Component and Associable Data Transfer Device, 1969. Brevet n°BG1314021, déposé le 28 février 1969, publié le 18 avril 1973. Inventeur : John Wolfgang Halpern, William Ward. Déposant : TRW Inc.

[60] Klaus Havelund and Thomas Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTP)*, 2(4) :366–381, 2000.

[61] Charles Antony Richard Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10) :576–580, 1969.

[62] HOL. `http://www.cl.cam.ac.uk/Research/HVG/HOL/HOL.html`.

[63] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions Software Engineering (TSE)*, 23(5) :279–295, 1997.

[64] Gerard J. Holzmann. From Code to Models. In *International Conference on Application of Concurrency to System Design (ACSD'01)*, pages 3–10. IEEE Computer Society, 2001.

[65] Gerard J. Holzmann and Margaret H. Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2) :72–87, 2000.

[66] Engelbert Hubbers and Erik Poll. Reasoning about Card Tears and Transactions in Java Card. In M. Wermelinger and T. Margaria, editors, *Fundamental Approaches to Software Engineering (FASE'04)*, volume 2984 of *LNCS*, pages 114–128. Springer, 2004.

[67] Marieke Huisman and Kerry Trentelman. Extending JML Specifications with Temporal Logic. In *Algebraic Methodology And Software Technology (AMAST'02)*, volume 2422 of *LNCS*, pages 334–348. Springer-Verlag, 2002.

[68] Intel Corporation. Understanding the Flash Translation Layer (FTL) Specification. Technical report, Intel Corporation, 1998.

[69] Intel Corporation. What Is Flash Memory ? Technical report, Intel Corporation, 2002.

[70] International Organization for Standardization (ISO). International Standard ISO 7810 : Identification Cards – Physical Characteristics, 1995. *(Cette norme décrit les principales caractéristiques physiques des cartes dans puce et définit les formats de cartes ID-1, ID-2 et ID-3).*

[71] International Organization for Standardization (ISO). International Standard ISO 7810 : Identification Cards – Recording Technique, 1995-2001. *(Cette norme définit les caractéristiques de la piste magnétique et des caractères estampés).*

[72] International Organization for Standardization (ISO) and International Electronic Committee (IEC). International Standard ISO 10 536 : Identification Cards – Contactless Integrated Circuit(s) Cards, 1995-2000. *(Cette norme décrit les cartes sans contact dont le champ d'application est limité à un contact direct avec le terminal).*

[73] International Organization for Standardization (ISO) and International Electronic Committee (IEC). International Standard ISO/IEC 10 373 : Identification Cards – Test Methods, 1995-2001. *(Cette norme fondamentale pour le test des cartes contient des descriptions précises des méthodes de tests pour les corps de carte et pour les corps de cartes avec puce implantée.).*

[74] International Organization for Standardization (ISO) and International Electronic Committee (IEC). International Standard ISO/IEC 7816 : Identification Cards – Integrated Circuit(s) Cards with Contacts, 1995-2001. *(Cette norme est la plus importante norme des cartes à microprocesseurs. Les trois premières parties de la norme définissent les caractéristiques de la carte, de ses contacts et des protocoles de communication. Les douze parties restantes sont dédiées aux applications et au système d'exploitation de la carte).*

[75] Isabelle. `http://isabelle.in.tum.de`.

[76] ISO/IEC 7816-1, Identification Cards – Integrated Circuit(s) Cards with Contacts – Part 1 : Physical characteristics, 1998.

[77] ISO/IEC 7816-2, Identification Cards – Integrated Circuit(s) Cards with Contacts – Part 2 : Dimensions and location of the contacts, 1999.

[78] ISO/IEC 7816-3, Identification Cards – Integrated Circuit(s) Cards with Contacts – Part 3 : Electronic signals and transmission protocols, 1997.

[79] ISO/IEC 7816-4, Identification Cards – Integrated Circuit(s) Cards with Contacts – Part 4 : Inter-industry Commands for Interchange, 1995.

[80] Jass. `http://csd.informatik.uni-oldenburg.de/~jass/`.

[81] Java Card. `http://java.sun.com/products/javacard/`.

[82] The Key Project. `http://www.key-project.org/`.

[83] Gerwin Klein. *Verified Java Bytecode Verification.* PhD thesis, Institut für Informatik, Technische Universität München, 2003.

[84] Gerwin Klein and Tobias Nipkow. Verified Bytecode Verifiers. *Theoretical Computer Science*, 3(298) :583–626, 2003.

[85] Jean Louis Lanet. Are Smart Cards the Ideal Domain for Applying Formal Methods ? In J. P. Bowen, S. Dunne, A. Galloway, and S. King, editors, *International Conference of B and Z Users on Formal Specification and Development in Z and B (ZB'00)*, LNCS, pages 363–373. Springer-Verlag, 2000.

[86] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML : Notations and Tools Supporting Detailed Design in Java. In *OOPSLA 2000 Companion*, pages 105–106. ACM, 2000.

[87] Xavier Leroy. Java Bytecode Verification : Algorithms and Formalizations. *Journal of Automated Reasoning*, 30(3–4) :235–269, 2003.

[88] The LOOP Tool. `http://www.sos.cs.ru.nl/research/loop`.

[89] Maosco Ltd. Multos. `http://www.multos.com/`.

[90] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The Krakatoa Tool for Java Program Verification, 2002. `http://krakatoa.lri.fr/`.

[91] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA Tool for Certification of JAVA/JAVACARD Programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2) :89–106, 2004.

[92] Jörg Meyer, Peter Müller, and Arnd Poetzsch-Heffter. The Jive System—Implementation Description, 2000. `http://www.sct.inf.ethz.ch/publications`.

[93] Sun Microsystems. *Java Card 2.1.1 Specification.* `http://java.sun.com/products/javacard/specs.html`.

[94] Model-Based Software Testing Home Page. `http://www.geocities.com/model_based_testing/`.

[95] Roland Moreno. Procédé et Dispositif de Commande Électronique, 1974. Brevet n°GB1504196 (numéro original : FR2266222), déposé le 25 mars 1974, publié le 15 mars 1978. Inventeur : Roland Moreno. Déposant : Société Innovatron.

[96] Stéphanie Motré. Formal Model and Implementation of the Java Card Dynamic Security Policy, 2000.

[97] George C. Necula. Proof-Carrying Code. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, 1997.

[98] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured : Type-Safe Retrofitting of Legacy Software. *ACM Trans. Program. Lang. Syst.*, 27(3) :477–526, 2005.

[99] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL : Intermediate Language and Tools for Analysis and Transformation of C Programs. In R. Nigel Horspool, editor, *Computational Complexity*, volume 2304 of *LNCS*, pages 213–228. Springer, 2002.

[100] George C. Necula, Scott McPeak, and Westley Weimer. CCured : Type-Safe Retrofitting of Legacy Code. In *Symposium on Principles of Programming Languages (POPL'02)*, pages 128–139, 2002.

[101] Tobias Nipkow and David von Oheimb. Machine-checking the Java Specification : Proving Type-Safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, pages 119–156. Springer-Verlag, 1999.

[102] Tobias Nipkow, David von Oheimb, and Cornelia Pusch. μJava : Embedding a Programming Language in a Theorem Prover. In Friedrich L. Bauer and Ralf Steinbrüggen, editors, *Foundations of Secure Computation*, volume 175 of *NATO Science Series F : Computer and Systems Sciences*, pages 117–144. IOS Press, 2000.

[103] Tobias Nipkow and Martin Wildmoser. Certifying Machine Code Safety : Shallow versus Deep Embedding. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *International Conference on Theorem Proving in Higher Order Logics (TPHOLs'04)*, volume 3223 of *LNCS*, pages 305–320. Springer-Verlag, 2004.

[104] Michael Norrish. *Formalising C in HOL*. PhD thesis, Computer Lab., University of Cambridge, 1998.

[105] David von Oheimb. *Analyzing Java in Isabelle/HOL : Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.

[106] David von Oheimb, Volkmar Lotz, and Georg Walter. A Formal Security Model of the Infineon SLE 88 Smart Card Memory Management. In E. Snekkenes and D. Gollmann, editors, *European Symposium on Research in Computer Security (ESORICS)*, volume 2808 of *LNCS*, pages 217–234. Springer, 2003.

[107] The PVS system. `http://pvs.csl.sri.com/`.

[108] Wolfgang Rankl and Wolfgang Effing. *Smart Card Handbook, 3rd Edition*. John Wiley & Sons, 2003.

[109] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1) :26–52, 1992.

[110] John Rushby. Noninterference, Transitivity, and Channel-Control Security Policies. Technical report, 1992.

[111] John Rushby. Formal Methods and their Role in the Certification of Critical Systems. Technical Report SRI-CSL-95-01, Computer Science Laboratory, SRI International, 1995.

[112] Andrew S. Tanenbaum. *Modern Operating Systems, 2nd Edition*. Prentice Hall, 2001.

[113] The Coq Development Team LogiCal Project. *The Coq Proof Assistant Reference Manual*. `http://coq.inria.fr`.

[114] Michel Ugon. Support d'Information Portatif Muni d'un Microprocesseur et d'une Mémoire Morte Programmable, 1977. Brevet n°US4211919 (numéro original : FR2401459), déposé le 26 août 1977, publié le 8 juillet 1980. Inventeur : Michel Ugon. Déposant : CII-HB (Compagnie Internationale pour l'Informatique - Honeywell Bull).

[115] Joachim van der Berg, Bart Jacobs, and Erik Poll. Formal Specification and Verification of JavaCard's Application Identifier Class. In I. Attali and T. Jensen, editors, *Java on Smart Cards : Programming and Security*, volume 2041 of *LNCS*, pages 137–150. Springer-Verlag, 2001.

[116] Joachim van der Berg, Bart Jacobs, and Erik Poll. Formal specification of the java card api in jml : the apdu class. *Computer Networks*, 36(4) :407–421, 2001.

[117] David Woodhouse. JFFS : The Journalling Flash File System. In *Ottawa Linux Symposium*. RedHat Inc., 2001.

[118] Zeitcontrol Cardsystems. Basic Card. `http://www.basiccard.com/`.

Résumé

Les travaux présentés dans ce mémoire ont pour objectif de renforcer le niveau de sûreté et de sécurité des systèmes embarqués dans les cartes à puce, grâce à l'utilisation des Méthodes Formelles. D'une part, nous présentons la vérification formelle de l'isolation des données de différentes applications chargées sur une même carte à puce. Plus précisément, nous décrivons la preuve formelle, dans le système de preuve *Coq*, que le contrôle dynamique d'accès aux données, implémenté par la plate-forme Java Card, assure les propriétés de confidentialité et d'intégrité. D'autre part, nous nous sommes intéressés à la correction et à l'innocuité du code source bas niveau d'un système d'exploitation embarqué. Cette étude est illustrée par un module de gestion de mémoire *Flash* par *journalisation*, assurant la cohérence des données de la mémoire en cas d'*arrachage* de la carte du terminal. La vérification de propriétés fonctionnelles et locales de ce module a été développée à l'aide de l'outil de vérification de programmes *Caduceus*. Cet outil n'acceptant pas certaines constructions de bas niveau du langage C, telles que les unions et les casts, nous proposons une analyse et différentes solutions pour la formalisation de ces constructions. Nous proposons également une extension de Caduceus permettant de spécifier et de vérifier le comportement d'une fonction en cas d'interruption soudaine de son exécution. Puis nous introduisons une méthodologie de vérification de propriétés globales de haut niveau visant l'expression et la preuve de ce type de propriétés sur un modèle formellement lié au code source. Plus précisément, nous décrivons l'extraction automatique d'un système de transitions formel, à partir d'annotations vérifiées par le code source. Ce système de transitions peut alors être plongé dans une logique d'ordre supérieur, apportant toute l'expressivité nécessaire à la définition et la preuve de propriétés complexes.

Mots-clés: Carte à puce, Méthodes Formelles, Sécurité, Isolation d'Applets, Vérification de Programme, Langage C, Modèle Mémoire, Mémoire Flash.

Abstract

The work presented in this thesis aims at strengthening the security and safety level of smart card embedded systems, with the use of Formal Methods. On one hand, we present the formal verification of the isolation of the data belonging to different applets loaded on the same card. More precisely, we describe the formal proof, in the *Coq* proof system, that the run-time access control, performed by the Java Card platform, ensures data confidentiality and integrity. On the other hand, we study the correctness and the safety of low level source code of an embedded operating system. Such source code is illustrated by a case study of a *Flash* memory management module, using a *journalling* mechanism and ensuring the memory consistency in the case of a *card tear*. The verification of functional and local properties has been developed using the *Caduceus* program verification tool. Since this tool does not support some low level constructions of the C language, such as the unions and the casts, we propose an analysis and some solutions for the formalisation of such constructions. We also propose an extension of Caduceus that allows to specify and verify the behaviour of a function in the case of sudden interruption of its execution. Then, we introduce a methodology for the verification of high level and global properties, which is meant for the expression and proof of this kind of properties on a model formally linked to the source code. More precisely, we describe an automatic extraction of a transition system from the annotations that are verified by the source code. This transition system can then be translated into a higher-order logic, with all the expressiveness for the definition of complex properties.

Keywords: Smart Card, Formal Methods, Security, Applet Isolation, Program Verification, C Language, Memory Model, Flash Memory.