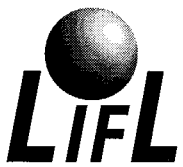


Laboratoire d'Informatique
Fondamentale de Lille

Numéro d'ordre : ????

THÈSE

Nouveau Régime

Présentée à

L'UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE

Pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

Gilles GRIMAUD

CAMILLE :
UN SYSTÈME D'EXPLOITATION OUVERT POUR CARTE À MICROPROCESSEUR

Thèse soutenue le 12 décembre 2000, devant la Commission d'Examen :

Président	:	Jean-Marc GEIB	Université de Lille 1
Rapporteurs	:	Bertil FOLLIOT	Universite de Paris VII
		Daniel HAGIMONT	INRIA Rhône-Alpes
Examinateurs	:	Pierre PARADINAS	Société Gemplus
		Vincent CORDONNIER	Université de Lille 1

Avant propos

Lorsque j'avais huit ans, le Père Noël m'a offert un livre passionnant sur les ordinateurs. J'y ai lu que la machine comprenait un langage de programmation par l'intermédiaire duquel on pouvait lui faire faire ce que l'on voulait. J'ai appris ce langage pour écrire mes propres programmes. Mais j'étais toujours déçu de me rendre compte que, quels que soient mes efforts, je ne parvenais pas à faire des logiciels aussi satisfaisants que ceux proposés par des professionnels. Jusqu'au jour, où, en m'intéressant de plus près à la manière dont ils étaient réalisés, j'ai découvert que le livre m'avait trompé. Mon ordinateur faisait des efforts démesurés pour comprendre les programmes que je lui transmettais. J'ai appris le langage de la machine et j'ai réussi mon premier programme véritablement satisfaisant. Lorsque je parlais d'informatique, je testais mon interlocuteur en disant « assembleur, *VBL*, ou *SoundTrack* » mais toujours on me parlait de langage de programmation et de système d'exploitation, je pensais « tout cela est bien pensé mais c'est loin de la réalité de mon ordinateur ». Ce n'est que bien plus tard, grâce à la patience et à l'intelligence de mes professeurs d'informatique, que j'ai pu comprendre le bien-fondé et les qualités de l'algorithmique, et des langages aussi « abstraits » que *Prolog* ou *Smalltalk*. Il m'est néanmoins resté une marque (probablement indélébile) de cette époque où l'on pouvait tout savoir et utiliser au mieux le matériel à notre disposition dans un ordinateur. Bien sûr, l'informatique a évoluée et il n'est guère raisonnable d'espérer tout faire soi-même aujourd'hui. Il n'en reste pas moins vrai que les utopies d'autrefois, celles qui prônaient l'avènement d'un langage et d'un système unique, parfaitement construit, adapté à tous les usages, en toute occasion, cèdent la place à une mosaïque de plus en plus diversifiée d'environnements qui ne font que refléter la richesse, sous toutes ses facettes, de l'informatique contemporaine.

La carte à microprocesseur

La carte à microprocesseur, nommée dans le langage courant carte à puce, est en fait un véritable système informatique embarqué. Elle connaît un essor retentissant depuis une vingtaine d'années, mais on ne peut véritablement lui reconnaître le statut de micro-ordinateur que depuis dix ans. Elle a dès lors été pressentie comme une technologie vouée à évoluer [Cor92]. En temps que système programmable elle a connu, et elle connaît encore, des développements informatiques. Les programmes placés sur ce support sont assez comparables à ceux exécutés sur des ordinateurs plus traditionnels. Il s'agit d'indexer, de mémoriser, de rechercher et de traiter des informations liées à un utilisateur. Cependant

ce composant est assujetti à des contraintes propres. Ces contraintes physiques, portant principalement sur des limitations des composants matériels, sont la source de recherches publiques et privées intensives. En effet, alors que la carte aborde aujourd'hui les mêmes problématiques que nos stations de travail, ces contraintes rendent inadaptées l'utilisation des solutions élaborées dans le contexte des ordinateurs classiques. En résumé, la carte à puce est probablement le plus contraint des systèmes embarqués qui est aussi le seul à avoir des ambitions comparables à celles des ordinateurs conventionnels. L'essentiel des travaux entrepris jusqu'à ce jour ont consisté à gommer cette frontière.

Orientation du projet

Alors que depuis sa création la carte à microprocesseur était cantonnée à des applications ponctuelles et fermées, ou toutes les décisions technologiques étaient à la charge du fabriquant, ces dernières années ont été pour cet objet discret une véritable révolution. Sous la pression de nouvelles applications les producteurs de carte ont finalement renoncé à assurer eux-mêmes la conception des logiciels.

Leur intégration dans des systèmes informatiques les rendent plus présentes et plus proches des courants de pensée de l'informatique classique [Cor96].

Le modèle technologique et économique qui supporte depuis vingt ans le développement des cartes à puces a atteint ses limites. Les cartes génériques (capables de charger de nouveaux programmes à tout instant) commencent à être industrialisées – alors qu'elles avaient été pressenties depuis plusieurs années [Pel95, Van95, Van97] comme des sources de progrès essentielles. Leur émergence a profondément modifié les « métiers de la carte », mais elles ont finalement soulevé autant d'interrogations qu'elles ont apporté de solutions.

En effet si les systèmes d'exploitation génériques placés dans ces nouvelles générations de carte [Sun96, Mic98, Mao98] ont finalement prouvé leurs vertus, ils ont aussi rapidement placés leurs concepteurs devant de nouveaux verrous technologiques. Le problème est que la conception d'une carte générique suppose l'existence d'un consensus sur le modèle d'abstraction du matériel proposé aux programmeurs. Non seulement aucun consensus ne s'est dégagé des différentes propositions envisagées, mais l'adaptation de nouvelles technologies propres au contexte des systèmes d'exploitations, tel que les supports de mémoires transactionnels [Lec98], ont conclus sur la nécessité de proposer différentes stratégies d'implantation en fonction des applications envisagées.

C'est dans ce contexte que mes travaux ont débuté. L'objectif fixé était de réussir à dépasser le modèle des cartes génériques pour proposer des structures systèmes moins rigides, c'est-à-dire capables de supporter différents modèles d'abstraction du matériel pour permettre à diverses applications de coexister (et même de coopérer) sur un seul support matériel. L'étude de cette problématique à été décomposée en trois étapes :

1. l'étude des caractéristiques et des besoins des applications des cartes de demain, ainsi que les technologies des systèmes d'exploitations qui répondent à ces besoins sur des supports moins contraints ;

2. la proposition une architecture logicielle pour concevoir un système d'exploitation qui soit adapté au contexte de la carte et à ses ambitions actuelles ;
3. l'évaluation de la validité de la proposition retenue de ses enseignements, pour que d'autres architectures puissent être expérimentées.

Le troisième point nous a amené à effectuer deux démarches complémentaires. Pour montrer la correction de notre démarche nous en avons formalisé les principes. Cette étude formelle a abouti à la génération automatique d'une preuve du bien-fondé de notre système. Une fois cette première étape accomplie, il reste à évaluer le potentiel de notre démarche. Nous avons donc réalisé un certain nombre d'expériences pour caractériser notre maquette et en montrer la pertinence.

Ce document reprend point par point l'ensemble des études réalisées durant les trois années de mes travaux. Ils ne sont pas le fruit d'une démarche solitaire mais bien d'un travail d'équipe (et même de plusieurs équipes en l'occurrence).

Sommaire

Ce document ce décompose en sept chapitres :

Le **Premier chapitre** présente le contexte informatique des cartes à microprocesseur. Le lecteur qui ne possède pas de connaissances particulières dans ce domaine y trouvera un rapide historique ainsi qu'une description détaillée des matériels et architectures dédiés. Le chapitre conclut sur les ambitions de la carte en rapport avec les travaux présentés dans la suite.

Le **chapitre 2** traite des systèmes d'exploitation. L'objectif de ce chapitre est de présenter les éléments de réflexion récents autour des systèmes d'exploitation ainsi que la manière dont ils peuvent être perçus, dans le contexte particulier des cartes à microprocesseurs. Les différents mécanismes de protection qui assurent la sécurité des applications les unes par rapport aux autres seront détaillés. Cette fonction du système d'exploitation est essentielle pour les cartes à puces qui sont pensées comme des systèmes de sécurité. Le chapitre conclut sur l'intégration dans les cartes des mécanismes systèmes récents.

Le **chapitre 3** présente les principes de base qui ont guidé la réalisation du projet. Il expose l'architecture logicielle globale, le rôle de chaque entité et les structures de base placées dans le noyau encarté.

Le **chapitre 4** décrit le langage intermédiaire, FACADE, ainsi que le mécanisme d'inférence de type qui assure la sécurité des logiciels encartés. Le contrôle de type, déjà largement exploités en informatique, est cependant délicat à transposer sur le matériel disponible dans une carte. Le chapitre 4 présente une solution pour y parvenir. Les éléments de formalisation qui ont permis de prouver la validité de notre proposition sont exposés.

Le **chapitre 5** détaille le processus de chargement dans la carte. Il montre l'infrastructure logicielle déployée afin d'apporter l'efficacité et la flexibilité qui sont les ambitions premières de cette architecture.

Le **chapitre 6** décrit une première implantation de nos propositions. Cette implantation est finalement la meilleure preuve de l'adéquation de notre démarche avec les contraintes propres aux cartes à puces. Ensuite deux expériences utilisant notre prototype sont détaillées et discutées.

Enfin **le chapitre 7** conclut sur les leçons qui peuvent d'or et déjà être tirées de notre travail d'implantation. Quelques limites imposées par l'approche que nous retenue sont commentées. Elles pour ouvrir sur de nouveaux éléments de recherche.

Lecture

Ce document présente la synthèse des différentes expériences menées durant mes travaux de thèse. Il est conçu pour être lu comme un tout, aussi est-il préférable de lire les chapitres dans l'ordre établi. Les acronymes malheureusement difficiles à éviter sans alourdir inutilement le texte, sont définis par des notes en bas de page lors de leurs premier emploi. Un glossaire, situé à la fin du document (annexe C) reprend ces notes à toutes fins utiles. Il reporte aussi certains termes et expressions consacrées qui sont fréquemment utilisés. Les schémas, lorsqu'ils portent sur des structures orientées objets, respectent la notation UML [Rat97, Amb00]. Bien que les informations fournies par les chapitres 1 et 2 (qui forment un l'état de l'art) soient systématiquement réutilisées par la suite, les chapitres 3, 4 et 5 ont été rédigés de manière à pouvoir être abordés (quasi-)indépendamment les uns des autres, aussi différentes lectures du document sont envisageables.

Le lecteur qui cherche des informations relatives à la carte à microprocesseur et à l'implantation de systèmes d'exploitation dans ce contexte, pourra avantageusement lire les **chapitre 1** et **2**. Le premier lui apportera la description des composants physiques avec lesquels il faut compter et dans le second il trouvera une synthèse du sens que prend aujourd'hui la recherche en systèmes d'exploitation dans le contexte de la carte à microprocesseur.

Le lecteur rompu aux pratiques systèmes de la carte à microprocesseur et qui souhaite comprendre l'approche générale retenue dans les travaux présentés ici pourra se reporter au **chapitre 3** pour avoir une première vue d'ensemble de l'approche retenue. Le **chapitre 5** lui permettra de découvrir plus précisément la conception du composant central du système présenté ici : « le chargeur de code ». Différents résultats expérimentaux obtenus à partir de cette architecture sont présentés dans le **chapitre 6**.

Celui qui s'intéresse plus particulièrement aux détails du mécanisme de sécurité-innocuité retenu dans notre architecture peut ce référer à la section 2.4 du **chapitre 2** pour avoir un état de l'art sur le sujet (une sélection d'articles est présentée dans la table 2.1 page 42), puis consulter directement le **chapitre 4** pour avoir le détail de la technique retenue et de sa mise en œuvre dans la carte. La section 4.3.4 présente la formalisation de cette technique mais les articles [GLV99, GJ00a, RCG00] peuvent être des sources d'informations complémentaires. Les éléments de réflexions présentés par la partie *Limites de l'approche* de la conclusion peuvent se présenter comme un complément du **chapitre 4** lorsqu'elle

traitent les limites de l'utilisation du langage intermédiaire FACADE, notamment au sujet des relations entre *optimisation de code intermédiaire* et *inférence de types assistée.*

Chapitre 1

Carte et applications à grande échelle

« Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher. Au terme de son évolution, la machine se dissimule. »

Terre des hommes, A. de Saint-Exupéry

Ce chapitre a pour objet de fournir au lecteur des informations relatives à la carte à microprocesseur. La connaissance des composants matériels et logiciels d'une carte à puce est en effet nécessaire à la compréhension des choix et des résultats présentés par les parties suivantes. Il rappelle l'utilisation de la carte à puce et en précise les fondements éthiques et économiques. Les différentes applications (stéréotypées) sont rapidement présentées, dans le but d'éclairer le lecteur sur les particularités de ce domaine de l'informatique ; le matériel embarqué est alors décrit. Les limitations de ce matériel font la spécificité des développements logiciels pour la carte. Enfin la dernière partie présente une vision plus générale de l'informatique de demain, dans laquelle la carte peut être introduite à bon escient. L'objet de la conclusion est d'identifier et de préciser les obstacles technologiques qui freinent aujourd'hui cette introduction.

1.1 Introduction

La carte à puce est un produit de l'informatique de pointe. Son évolution, coordonnée à celle de l'informatique traditionnelle, est extrêmement rapide et vise des objectifs toujours plus ambitieux. Son premier succès industriel porta sur la production de télécartes destinées à donner accès aux cabines téléphoniques publiques. Dès l'origine la carte était donc pensée comme un outil d'intégration de l'informatique géographiquement répartie. Elle avait l'avantage de sécuriser le système d'information déployé. La problématique d'une informatique décentralisée, encore naissante lorsque les télécartes envahirent nos portefeuilles, devint l'un des moteurs de la recherche. Aussi la carte à puce, précurseur des

solutions massivement distribuées, s'appuya sur ce créneau novateur et très prometteur pour étendre ses domaines d'application. La carte bancaire, la carte santé, puis la carte SIM[1] (du GSM[2]) ne sont que des exemples convenus pour souligner le rayonnement de cette informatique singulière.

La carte à microprocesseur est une alternative à des solutions centralisées. Elle nécessite une approche différente dans la modélisation des systèmes d'information [Pel95, Hoe97] ainsi que le développement d'outils spécifiques. Il s'agit aussi bien de procédés matériels [Cor94, CCG95], que de procédés liés aux systèmes d'exploitation [Gri91, Lec98] ou encore de procédés d'intégration de la distribution des applications [Van97, Car98].

Cette alternative trouve une motivation éthique. À l'heure de l'information, reine des nouvelles technologies et de la nouvelle économie, je reste surpris de la facilité avec laquelle les internautes (notamment) exposent des informations privées, voire confidentielles, sur le réseau. L'utilisation et la distribution ultérieure de ces informations échappe à la protection du droit français. Et quand bien même, ces masses d'informations centralisées sur des serveurs privés sont régulièrement soumises à l'indiscrétion des employés ou au pillage des pirates. Fondamentalement, la carte à puce propose une alternative élégante en matérialisant l'information de son porteur : « si ma carte est dans mon porte-feuille, l'information qu'elle contient ne peut être utilisée ailleurs. ». Impossible en effet de consulter des informations numériques privées sans que le possesseur de droit n'en soit conscient. Les mécanismes qui viennent ensuite assurer l'authenticité du porteur, et l'intégrité des données, ne servent qu'à étendre et renforcer cette idée première. Une carte est un support d'information dédiée à un porteur. Elle préserve et sécurise les données qui lui appartiennent. Elle assure la médiation des informations privées vis-à-vis des systèmes d'information qui les sollicitent.

La carte à microprocesseur est avant tout un objet grand public. En France elle a été adoptée par le plus grand nombre en un temps record. Et pourtant, la carte à puce est un véritable ordinateur, un ordinateur dissimulé dans un morceau de plastique. Les cartes à puces ne possèdent pas d'interfaces homme-machine apparentes. On les utilise simplement en les manifestant, de la même manière qu'on présente un document d'identité traditionnel par exemple. Cette simplification à l'extrême résume bien l'esprit de la carte. C'est cet esprit qui est depuis maintenant vingt-cinq ans, le garant de son succès. Un ordinateur personnel auquel tout a été retiré, jusqu'à ce que la machine se dissimule derrière le service qu'elle rend. Elle assure pourtant une fonction complexe de médiation entre des prestataires de services et des utilisateurs. Elle simplifie à l'extrême le mécanisme d'accès aux services déployés géographiquement (à l'échelle d'un pays ou même d'avantage). La carte apparaît alors comme une clef pour la distribution.

1. SIM : Subscriber Identity Module, module d'identification de l'abonné sur les réseaux GSM.
2. GSM : Global System for Mobile communications, *i.e* norme européenne de gestion des réseaux de téléphonie mobile.

1.2 Une clef pour la distribution

La carte à puce est née en 1974 de la volonté de son inventeur, Rolland Moréno, de distribuer le système d'information bancaire dans les poches de ses utilisateurs. Cependant, cette idée parut trop avant-gardiste pour être retenue telle quelle par les instituts bancaires. Finalement c'est le réseau des cabines téléphoniques publiques qui fut le premier succès de masse de l'utilisation de la carte à puce. Elle gère le décompte des unités consommées et elle personnalise le terminal (e.g. numéros d'appel préférés, touche bis, ...). C'est l'avènement de la télécarte.

1.2.1 La télécarte

Les télécartes possèdent un micro-circuit imprimé sur du silicium. Le circuit, comme celui de toute carte à puce, est fixé à l'aide d'une colle dans une cavité pratiquée sur le support plastique de la carte. La *pastille de contact* (la pièce de métal que l'on voit sur toutes les cartes à puce) est placée sur le micro-circuit. Elle assure la connexion électrique et numérique qui permet l'échange d'information entre la carte et le terminal. On dit que le fabriquant *encarte* le micromodule.

Dans le cas des télécartes, la puce placée sous les contacts reste un pur produit de l'électronique. Cette orientation micro-électronique marque profondément le matériel embarqué. Une carte à microprocesseur est soumise à des contraintes spécifiques définies par les normes ISO[3] [ISO87, ISO88]. Ces normes visent à assurer la résistance de la carte à puce en fixant des contraintes de torsion et de flexibilité élevées. En effet les cartes à puce sont des objets destinés au grand public et ne doivent pas nécessiter d'attention particulière de la part de leurs porteurs. Dans leur assemblage, le silicium est le composant le moins flexible ; pour pouvoir dissimuler les cartes à puce parmi les autres papiers de nos portefeuilles, les normes citées précédemment prévoient une surface de silicium maximale de 27 mm^2 et une épaisseur maximale de 0,28 mm.

D'autres applications ont utilisé le même principe de fonctionnement que les télécartes. Le terme « carte porte-jeton » ou encore « carte porte-fusible » désigne de manière générique toutes les cartes qui sont utilisées selon le principe de la télécarte. Cependant, d'autres rôles peuvent être joués par la carte à puce, qui ne rentrent pas dans ce cadre générique ; il s'agit notamment de la carte bancaire.

1.2.2 La carte bancaire

Les cartes bancaires (dites à puces, tel que les définissent les consortium MasterCard et Visa[MV96]) que nous utilisons aujourd'hui jouent en fait le rôle d'une clef; elles partagent un secret avec les services bancaires. Il s'agit d'une clef de chiffrement de type RSA[4], d'une centaine de chiffres (décimaux). L'autentification de la carte ce fait en deux étapes. Tout

3. ISO : International Standardization Organisation : organisme de standardisation internationale.
4. RSA : algorithme de chiffrement à clef privée de Rivest, Shamir et Adelman.

dabord le porteur s'authentifie après de sa carte, c'est la saisie du code secret. Ensuite la carte utilise sa clef pour chiffrer un nombre transmis par le terminal. Le terminal vérifie que le nombre chiffré correspond au nombre initial. Il peut ainsi s'assurer que la carte connaît le secret (la clef de chiffrement) et qu'elle est donc authentique. L'échange nombre/nombre chiffré constitue un *certificat d'authenticité* qui est utilisé par le terminal de paiement pour justifier les transactions bancaires (paiement du client) qu'il sollicite. Cette technique présente l'avantage de ne pas nécessiter une connexion systématique entre le terminal et les serveurs bancaires. Ainsi les terminaux sont moins coûteux à l'usage, car ils n'effectuent des connexions aux réseaux bancaires[5] pour « vider » leurs tampons de transactions, tous les soirs par exemple.

En fait, l'objet de ces cartes est de proposer une alternative numérique aux clefs conventionnelles. Elles peuvent servir aussi bien à autoriser l'ouverture d'une porte dans des locaux sécurisés que l'accès à un serveur informatique ou à un réseau spécialisé (ce que font les cartes bancaires). Elles remplacent alors les traditionnels systèmes d'identification par nom et mot de passe ou tout autre identifiant numérique « en clair ». La carte implique un système de sécurité plus fiable car elle combine (ou remplace) une information connue (un mot de passe) et (par) un objet possédé: la carte. Pour qu'elle puisse jouer ce rôle, le composant électronique a dû embarquer en son sein un système de chiffrement élaboré. L'usage des systèmes cryptographiques symétriques et asymétriques [Sch89] bien connus permet d'ouvrir la carte à puce à de nouveaux emplois. Elle se présente alors comme une véritable clef numérique. En quelques échanges, la carte et le service local peuvent s'authentifier mutuellement.

L'acceptation immédiate des cartes à microprocesseur par le grand public a attiré l'attention des programmeurs d'autres applications; la plupart n'envisageaient pas la carte comme une clef ou un porte fusible, mais comme un véritable support de données numériques attaché à chaque utilisateur. Cette sorte de carte est bien plus complexe à concevoir. On parle alors de dossiers portables, au nombre desquels on compte la carte étudiant ou encore la carte santé.

1.2.3 La carte santé

La carte santé et dans de nombreux pays (Canada, Allemagne, ...) un exemple abouti de l'utilisation de la carte à microprocesseur. En France c'est le G.I.E.[6] Sesam Vitale qui a défini les normes de la cartes Sesam Vitale[SV96] et elle préserve et garantit la confidentialité des dossiers médicaux du patient, en l'identifiant auprès des systèmes informatiques des assurances. En résumé elle joue véritablement le rôle d'un système informatique.

Toutefois, le frein principal à l'exploitation des cartes pour ce type d'usage est lié au délai de production d'un nouveau produit. La réalisation d'un nouveau type de carte, qui embarque un nouveau système d'information, nécessitait, il y a encore peu de temps, plus

5. réseau numérique dédiés aux transaction bancaires.

6. G.I.E. Groupement d'Intérêt Économique.

de dix-huit mois. Cette durée entre la date de commande et la date de livraison était principalement due au temps de masquage[7] des micromodules.

Pour comprendre ce délai, il faut savoir que les cartes à puce passent entre les mains de différentes industries avant d'atteindre nos poches. La production d'une carte à microprocesseur commence avec la spécification d'un besoin par **(1) l'émetteur** qui est le prestataire de service souhaitant utiliser des cartes (*e.g.* Groupement d'Intérêt Économique Bancaire); ensuite, **(2) le fabriquant de cartes** produit le logiciel destiné à être embarqué dans la carte. Le code à encarter[8] est alors transmis **(3) au fondeur** qui l'imprime sur le substrat de silicium dans le micromodule. Le fabriquant produit alors des cartes avec ces nouveaux micromodules (*c.f.* section 1.2.1 page 9). Les cartes sont alors envoyées a l'émetteur qui les distribue aux **(4) porteurs**, c'est-à-dire à l'utilisateur final.

Un nombre grandissant d'applications ont intégré la carte en temps que support persistant, personnel et sécurisé, de données confidentielles. L'objectif des dossiers portables est de généraliser ces différentes utilisations et de réduire le délai de livraison. L'idée d'introduire des cartes plus génériques, capables d'héberger des données spécifiées *a posteriori* par un nombre grandissant d'émetteurs s'imposa naturellement. Il s'agissait aussi de permettre à des applications plus ponctuelles (et visant un public plus restreint) de voir le jour.

La norme ISO7816-4 [ISO94] définit la carte comme un support de fichiers auxquels sont associés des droits d'accès. La norme ISO 7816-7 [ISO99] va plus loin et reprend les travaux accomplis autour de CQL[9][Gri91] afin de permettre de manipuler et de structurer les informations confiées aux cartes de la même manière que SQL[10] permet d'interroger les informations placées dans des bases de données classiques.

Ces deux normes ont permis de définir des cartes capables d'embarquer des structures de données quelconques. Ces structures de données ne sont alors spécifiées que lorsque le prestataire de service les a complètement identifiées. Les dossiers portables permettent une spécification des données contenues dans les cartes après le masquage du logiciel. Ainsi une même carte produite à des millions d'exemplaires peut devenir une carte d'abonné vendue à un cinéma, ou une carte santé vendue à un assureur. Le cycle de production ne nécessite plus de passer par le fondeur **après** spécification par le prestataire de services. Le délai d'attente en est d'autant réduit. Le deuxième avantage de cette approche est de permettre l'utilisation de cartes à puce pour des marchés plus limités et ponctuels que le marché de masse des télécartes. Mais aujourd'hui, ce modèle de carte ne contente plus les émetteurs qui souhaitent pouvoir reprogrammer à tout moment le comportement des cartes émises. C'est cette motivation qui a, par exemple, guidé la conception de la carte Java-SIM.

7. masquage: dans ce contexte il s'agit du procédé d'impression du logiciel sur le support de silicium des cartes à microprocesseur commandées par les fabricants.

8. encarter: normalement utilisé pour désigner le procédé de collage du micromodule, mais qui par extension désigne aussi le placement des programmes dans la carte.

9. CQL: Card Query Language, Langage dédié à la carte qui permet de formuler des requêtes d'interrogation sur les informations qu'elle contient.

10. SQL: Structured Query Language: Langage normalisé de manipulation de bases de données.

1.2.4 La carte Java-SIM

La structuration des données dans une carte à puce satisfait aujourd'hui 80% des besoins des applications conventionnelles. Cependant ces solutions techniques se sont montrées incapables de répondre aux exigences croissantes des cartes SIM. Ce sont les fournisseurs de réseaux mobiles ont plébiscités la conception de cartes plus puissantes pour ne pas être dépendants des fabricants de téléphones portables. Ces cartes sont aujourd'hui les plus puissantes mises sur le marché, mais ce sont aussi les cartes les plus transparentes (les plus dissimulées et les moins connues) aux yeux de leurs utilisateurs. Elles sont présentes dans tous les téléphones portables (de type GSM) utilisés. Chaque carte représente la relation commerciale entre un fournisseur de réseau de téléphonie mobile et un client. En fait, elles peuvent cumuler toutes les fonctions des cartes précédemment présentées :

1. cartes porte-jeton rechargeables lorsqu'elles gèrent des unités pré-payées pour téléphoner ;
2. cartes d'accès qui utilisent le chiffrement pour sécuriser l'accès au réseau de téléphonie sans fil du fournisseur ;
3. cartes dossier portable qui contiennent notamment les informations de configuration des abonnements et le répertoire téléphonique de l'abonné.

Toutefois, le secteur très concurrentiel de la téléphonie mobile incite les opérateurs à proposer régulièrement de nouveaux services à leurs clients. Ce sont des services additionnels que les porteurs peuvent acquérir sans changer d'abonnement, et donc de carte. Pour que cela soit possible, plus que de nouvelles structures de données, la carte doit pouvoir charger de nouveaux traitements. Ces nouveaux traitements sont représentés, dans le cas des cartes JAVA-SIM, sous la forme de classes Java qui peuvent à tout moment être ajoutées à celles déjà présentes.

Ces dernières années ont vu l'émergence d'un nombre croissant de cartes qui proposent la possibilité de charger du code après leur émission. Elles sont parfois nommées *Cartes ouverte*. Trois des plus significatives sont les cartes Java [Sun98], les *Windows for SmardCard* [Mic98] et les cartes *MULTOS* [Mao98] :

Les **JavaCard**[11] permettent de charger des programmes réalisés avec le langage du même nom. Cependant, les spécifications de la machine virtuelle Java encartée sont différentes des spécifications d'une machine virtuelle Java standard. Les différences essentielles portent sur la gestion des opérations arithmétiques et logiques, la gestion de la sécurité entre les applications, la persistance des objets ainsi que leurs création et libération. Aussi une opération de conversion est-elle nécessaire avant de transmettre les programmes à la carte. De plus, la plupart des bibliothèques de base du langage sont redéfinies ou inutilisables ;

Les **SmartCard for Windows** sont une solution proposée par la société Microsoft. Ces cartes permettent de charger un P-Code qui peut être généré à l'aide d'outils proposés par le fabriquant ;

11. JavaCard est un nom déposé pour les cartes qui supporte le langage Java.

Les **cartes MULTOS** sont spécifiées par la société Maosco Ldt. Un format de code exécutable appelé MEL[12] est défini pour charger des programmes dans les cartes. Un compilateur génere du code MEL à partir d'un code source C-ANSI qui permet de programmer les cartes à volonté.

Ce nouveau degré de flexibilité dans l'utilisation des cartes à microprocesseur est particulièrement attrayant mais reste très limité en pratique. Ces limitations constituent de nouveaux verrous technologiques qui sont détaillés dans la section 1.5.3, page 26.

Nous pouvons cependant conclure qu'avec ces derniers développements la carte à microprocesseur se présente, aux yeux des programmeurs d'applications carte, comme un ordinateur à part entière. Il convient toutefois, pour comprendre le sujet de ce mémoire, de connaître les spécificités de ce matériel embarqué fortement contraint.

1.3 Matériel embarqué fortement contraint

Force est de constater que, depuis ses premières spécifications, le matériel encarté n'a guère évolué. La section 1.2.1, page 9 a présenté une norme qui impose une surface de silicium encartée inférieure à $27mm^2$. Il est alors aisé de comprendre que cette limite entraîne une architecture matérielle minimaliste.

1.3.1 Architecture matérielle minimaliste

La carte à microprocesseur est un micromodule monolithique qui contient six composants fonctionnels représentés sur la figure 1.1. Ce micromodule est le plus souvent réalisé avec la technologie CMOS[13] $3,5\mu$ qui, dans le contexte de la carte à microprocesseur, a l'avantage d'avoir une faible consommation et surtout une faible sensibilité aux perturbations électriques.

Deux critères fondamentaux guident la conception de l'architecture matérielle d'une carte à microprocesseur.

Le premier est relatif à la sécurité. Les cartes à microprocesseur en tant qu'éléments de sécurité, sont soumises à des risques d'attaques physiques[AK96]; Le micromodule doit être monolithique pour rendre impossible toute tentative d'espionnage du bus de données. En conséquence toutes les ressources matérielles de la puce doivent pouvoir être placées sur le même module. De plus, un bloc de sécurité composé de détecteurs de variation de la luminosité, de la tension, de la fréquence ,... est placé sur le seul est unique substrat de silicium pour garantir l'intégrité physique de la carte. Ce bloc de sécurité a pour objectif de rendre impossible l'utilisation malveillante de la puce (en la sous-alimentant par exemple).

12. MEL: MULTOS Executable Language: langage spécifiant le format des instructions exécutables par les cartes MULTOS.

13. CMOS: *Complementary Metal-Oxide Semiconductor.*

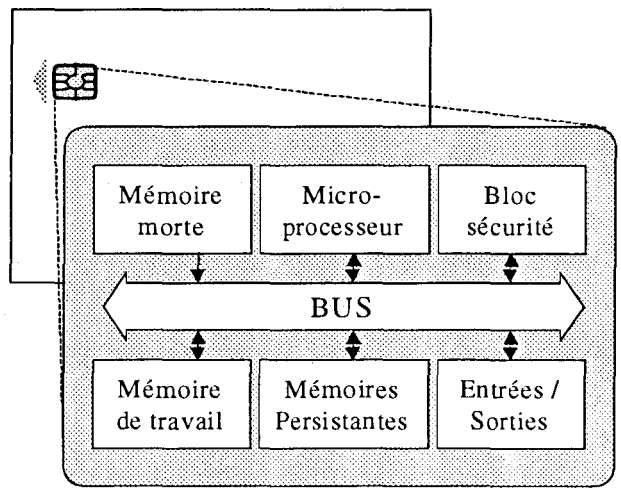


FIG. 1.1 – *Architecture logique d'un micromodule carte*

Le second critère est relatif à la taille du circuit imprimé. L'architecture générale des cartes vise toujours à minimiser cette taille. Une idée éculée consiste à proposer de s'abstraire de ce critère de taille pour proposer d'avantage de ressources aux programmeurs de cartes; elle n'a cependant pas aboutit à ce jour pour de multiples raisons. Bien sûr, en outrepassant les limites de $27mm^2$ fixées par les normes ISO, le microcontrôleur monolithique risque de ne plus supporter les contraintes de flexion et de torsion que le porteur lui fait subir tous les jours. Il est vrai aussi qu'un micromodule monolithique peut être espionné par des appareils spécialisés lorsqu'il est trop volumineux. Toutefois un « non-dit » essentiel dans le milieu de la carte est que le coût de production dépend principalement de la surface occupée par la puce sur le silicium ; et c'est souvent grâce à des prix de vente extrêmement réduits que la carte à microprocesseur devient attractive face à d'autres solutions technologiques. Aussi le critère de la taille des circuits guide les choix opérés dans l'architecture matérielle des cartes.

Une description détaillée des différents microprocesseurs et une synthèse des différents types de mémoires utilisés dans les cartes seront l'objet des deux sections consécutives 1.3.2 et 1.3.3. Pour ce qui est des entrées/sorties, elles se résument aujourd'hui à une liaison série bi-directionnelle parfois couplée à un registre à décalage de 8 bits qui décharge alors le microprocesseur de l'échantillonnage et de la sérialisation et de la désérialisation des données. Il est normalement utilisé à une vitesse de 9 600 BPS[14] mais il peut sans difficulté atteindre des vitesse de 192 000 BPS ou plus. La vitesse de transmission des données entre la carte et le monde extérieur n'est actuellement pas un obstacle à l'efficacité des applications encartées ; des pénalités bien plus importantes sont induites par les accès en écriture à certains types de mémoires des cartes à microprocesseur.

14. BPS: Bits Par Secondes.

1.3.2 Mémoires des cartes à microprocesseur

Les deux critères qui guident la conception de l'architecture matérielle des cartes, guident aussi la nature et la quantité des mémoires encartées. Pour comprendre les limitations des mémoires encartés, il faut considérer la taille qu'occupe sur le silicium un point mémoire[15] de RAM[16], de ROM[17], d'EEPROM[18] ou encore de FlashRAM[19]. En effet, toutes ces mémoires ne sont pas égales devant la place qu'elles occupent sur le silicium du micromodule. La table 1.1 montre les rapports de surface entre ces différents éléments. Il faut encore préciser que le point mémoire de la RAM est vingt fois plus élevé que celui la ROM, en partie parce que la mémoire de travail (la RAM) utilisée dans une carte à microprocesseur est de type RAM statique (bascules). La mémoire DRAM (capacité) n'est pas utilisée car elle nécessite un système de rafraîchissement matériel volumineux sur le silicium.

Type de mémoire	point mémoire	capacité typique	délais écriture	grain
ROM	*unité de base*	32→64 Ko	*impossible*	1 octet
Flash	× 2-3	16→64 Ko	2,5ms	16-128 octets
EEPROM	× 4	4→32 Ko	4ms	1→4 octets
RAM	× 20	128→4096 Octets	$< 0.2\mu s$	1 octet

TAB. 1.1 – *Caractéristiques moyennes constatées sur les mémoires des cartes (1998-2000)*

L'EEPROM et la mémoire Flash sont souvent présentées comme le disque de la carte à puce. Cela n'est que partiellement vrai. Dans la pratique, ces mémoires persistantes ont, en lecture, les mêmes caractéristiques que la mémoire de travail. Il est donc possible de lire dans de l'EEPROM sans délai d'accès notable. Aussi un programme n'a pas à être chargé en RAM pour être exécuté. Par contre, les accès en écriture sont. eux, particulièrement pénalisés, non seulement et terme de délais d'écriture, mais aussi à fiabilité de fonctionnement. En effet, les pages de mémoires persistantes ne supportent qu'un nombre limité de réécriture (de 10^4 à 10^5 selon la qualité des mémoires produites). Ensuite il devient impossible de les modifier. C'est pourquoi la mémoire persistante ne peut pas véritablement être utilisée comme une mémoire de travail. Pourtant, les faibles quantités de RAM présentes dans le micromodule contraignent profondément l'implantation d'algorithmes issus de l'informatique classique. Cette contrainte contribue fortement à la « culture carte » et même

15. Point Mémoire : circuit électrique qui engendre la mémorisation d'un bit (ou d'une unité élémentaire d'informations).

16. RAM : Random Access Memory.

17. ROM : Read Only Memory.

18. EEPROM : Electric Erasable Programmable Read Only Memory.

19. FlashRAM : la mémoire FlashRAM (aussi appelée Flash) est une mémoire EEPROM dont l'architecture à été modifiée pour gérer une granularité d'écriture moins fine, mais des délais d'écriture moins importants.

le haut niveau d'abstraction introduit par les systèmes d'exploitation des cartes ouvertes (*c.f.* section 1.2.4), ne masque pas complètement ces contraintes.

Cet état de fait pourrait bien changer dans les années à venir. La recherche en électronique a proposé des innovations importantes dans le domaine des technologies de mémoires persistantes. Ainsi la FeRAM[20] présente des caractéristiques largement supérieures à celles des autres technologies. La taille de son point mémoire est proche de celui de la ROM et elle ne connaît aucun délai d'écriture particulier.

Finalement, la seule contrainte est que toute donnée lue par le microprocesseur est perdue par la mémoire. Elle doit donc être immédiatement réécrite pour être préservée. Dans la pratique, ce type de propriété peut être détourné par les pirates afin de réinitialiser la valeur lue à un instant donné par le microprocesseur.

Plus généralement, ce type de mémoire soulève de nouveaux problèmes de sécurité. Aussi fait-il aujourd'hui encore, et déjà depuis deux ans, l'objet d'une évaluation approfondie par l'industrie de la carte à puce. À ce jour, son utilisation est sérieusement compromise pour les cartes qui nécessitent un certain degré de sécurité. Notons encore que le processus de fabrication de la FeRAM implique une seule couche d'aluminium[AMO+98]. Cela soulève deux nouveaux problèmes : d'une part, les composants avec une couche d'aluminium sont plus faibles vis à vis des attaques physiques ; d'autre part, l'intégration de composants mono-couche aluminium n'est guère compatible avec la technologie en double couche d'aluminium nécessaire à la fabrication et la sécurisation des microprocesseurs dédiés aux cartes.

1.3.3 Microprocesseurs dédiés aux cartes

Le chef d'orchestre de l'activité d'une carte est bien sûr le microprocesseur. Certes, les contraintes importantes imposées sur l'architecture du micromodule monolithique influencent le choix de l'unité de traitement embarquée ; toutefois l'écart de technologie entre les microprocesseurs encartés et les microprocesseurs des stations de travail n'est pas comparable avec le gouffre qui sépare les mémoires des cartes des mémoires conventionnelles. La table 1.2 présente les caractéristiques de trois microprocesseurs aujourd'hui utilisés dans des cartes.

Dénomination	68H05	AVR	ARM7*Thumb*
Architecture	CISC	CISC/RISC	RISC
BUS de données	8 bits	8/16 bits	32 bits
Registres	2, 8 bits	32, 8 bits	16, 32 bits
Fréquence max.	4,77 Mhz	4,77 Mhz	4,77 à 28,16 Mhz

TAB. 1.2 – *Caractéristiques de trois microprocesseurs encartés (1998-2000)*

20. FeRAM : RAM Ferro-Electrique.

Le premier est le microprocesseur 6805 de la société Motorola, parfois décliné en 68H05, qui est alors équipé d'un multiplicateur « $8 \times 8 \rightarrow 16$ bits » . Il s'agit d'un microprocesseur CISC[21] qui a un long passé puisqu'il date de la fin des années 1970. Vis-à-vis de l'informatique moderne, le plus surprenant est peut-être l'absence de registre de pointage autre que le pointeur de programme. Dans la pratique, ce microprocesseur est utilisé pour les cartes « bas de gamme » ou pour des produits à très faible coût.

Le second est le microprocesseur AVR de la société ATMEL. Il comporte un nombre important de registres et a été conçu en considérant les compilateurs C comme une source privilégiée de code machine. Cette remarque des concepteurs du jeu d'instructions est en partie contestable. Les opérations de décalage logique, par exemple, ne sont que partiellement implantées alors qu'elles sont partie intégrante de l'arithmétique du langage C (les opérateurs >> et <<). Les seules instructions en fait disponibles sont le décalage logique de 1 bit à droite, signé et non-signé. Pourtant, des opérations de manipulation de 1 bit, difficilement exploitables par les compilateurs de ce langage, sont présentes. Le jeu d'instructions reconnu par l'AVR, avec des registres spécialisés et des opérations complexes limitées à un sous-ensemble de registres, est finalement plus éloignée des préceptes qui guident la réalisation d'une architecture RISC[22] que d'une architecture CISC [PH97]. C'est ce microprocesseur qui a été retenu pour l'évaluation des solutions proposées dans ce mémoire. Il a l'avantage de prouver la validité des solutions proposées avec une architecture matérielle classique utilisée par industrie.

Le troisième microprocesseur est le ARM7 *Thumb* qui est un RISC 32 bits conçu par la société ARM. L'utilisation de cette architecture RISC est en fait issue d'un projet européen ambitieux: Cascade[ESP98]. Il représente un progrès considérable par rapport aux machines qui étaient alors utilisées dans les cartes. Cette gamme de microprocesseurs exécute un code machine où toutes les instructions ont un format régulier et sont codées sur 32 bits. L'introduction de microprocesseur RISC ne fut pas pas un choix évident. Il est vrai que l'homogénéité du jeu d'instruction engendre un circuit logique plus simple et donc moins volumineux sur le silicium. Toutefois on constate qu'avec les architectures RISC le volume du code est généralement vingt pourcent plus important qu'avec une architecture CISC. Pour pallier ce problème, le microprocesseur ARM7 décliné en « *Thumb* » est capable d'exécuter un second jeu d'instructions, codées sur 16 bits et définies pour obtenir que le code machine soit compact. Ce microprocesseur est particulièrement performant. Il est capable d'exécuter des algorithmes cryptographiques complexes tels que RSA sans utiliser de coprocesseur spécialisé. Il existe maintenant d'autres unités de traitements encartables qui sont comparables dans leurs caractéristiques à cette machine. Même si leur utilisation industrielle reste cependant très marginale, elles sont le manifeste de la montée en puissance du matériel encarté.

21. CISC : *Complex Instruction Set Computer.*
22. RISC : *Reduce Instruction Set Computer.*

1.3.4 Montée en puissance du matériel encarté

Historiquement, la première carte à microprocesseur fut construite par la société Bull en 1979 autour d'une architecture CISC 8 bits. Les architectures CISC 8 bits étaient alors les seules assez compactes (et bon marché) pour être encartées. Cependant l'apparition de nouvelles technologies, plus compactes et performantes, bâties autour de microprocesseurs RISC, a ouvert de nouvelles possibilités. La gamme des microprocesseurs utilisés dans les cartes semble indiquer un glissement progressif des architectures « historiques » vers des architectures spécialement pensées et conçues pour les cartes. Cependant, le coût du produit reste le premier critère de sélection du matériel devant ses autres performances.

Aussi, le constat de Moore[23] qui notait un doublement de la puissance des ordinateurs tous les dix-huit mois n'est pas vérifié dans le contexte des cartes à puce. La table 1.3 nous invite plutôt à observer un doublement de la capacité de la mémoire de travail tous les 36 mois, et un doublement de la puissance de calcul plus lent encore. Les progrès d'intégration et d'optimisation des circuits réalisés par les fondeurs sont plus souvent utilisés par les industriels pour diviser les coûts de production par deux, que pour multiplier par deux la puissance de ces ordinateurs.

année	taille du bus	Fréquence	RAM	mémoire persistante
1981	8 bits	4,77 Mhz	36 octets	1Ko EPROM
1985	8 bits	4,77 Mhz	128 octets	2Ko EEPROM
1990	8 à 16 bits	4,77 Mhz	256 octets	8Ko EEPROM
1996	8 à 32 bits	4,77 à 28,16 Mhz	512 octets	32Ko Flash
2000	8 à 32 bits	4,77 à 28,16 Mhz	1536 octets	32+32Ko Flash+EEPROM

TAB. 1.3 – *Évolution des caractéristiques moyennes des cartes entre 1981 et 2000*

Dans ces conditions, la différence de puissance entre les cartes à microprocesseur et l'informatique conventionnelle, loin de se résorber, s'accentue chaque année d'avantage. Aussi, aujourd'hui encore, adapter des technologies logicielles pour la carte à microprocesseur ne peut être mené sans une réflexion appropriée. Dans les faits, les contraintes liées au matériel des cartes ont une influence considérable sur les programmes encartés. Pour dépasser ces contraintes il est nécessaire de proposer des architectures logicielles dédiées.

1.4 Architectures logicielles dédiées

Concevoir un logiciel pour carte a, pendant de longues années, été un métier à part entière. Seule une poignée d'experts (quelques centaines dans le monde en 1997 selon des sources industrielles) programmaient les logiciels encartés. Cette activité consistait à trouver « le plus court chemin » entre l'architecture matérielle et les fonctionnalités

23. Le constat de Moore est par la suite devenu « la loi de Moore ».

souhaitées par le client. Les logiciels encartés se présentaient alors sous la forme d'un programme monolithique unique placé en ROM, dont l'architecture changeait entièrement d'un produit à l'autre. Il s'agissait d'applications clef en main. Ce n'est que récemment, depuis de debut de l'année 1997, que les efforts d'ouverture ont successivement débouchés sur des architectures de plus en plus souples et stables. Ces efforts ont abouti à la conception d'architectures logicielles encartés (principalement aux systèmes d'exploitation) ainsi qu'à l'intégration des différents acteurs du déploiement des cartes.

1.4.1 Acteurs du déploiement des cartes

En généralisant le déploiement des applications qui impliquent une carte, on peut identifier cinq acteurs distincts :

1. l'utilisateur de l'application (c.a.d. le porteur de la carte) ;
2. la carte à puce qui contient la partie mobile de l'application ;
3. le terminal qui est la partie de l'application géographiquement déployée ;
4. le réseau qui permet au terminal de communiquer avec l'opérateur ;
5. et enfin le serveur d'applications qui centralise les informations propres à l'opérateur.

Les acteurs 4 et 5 sont facultatifs. Cependant la carte à microprocesseur devient de plus en plus souvent un complément local des prestataires de services distribués sur différents réseaux. Aussi l'architecture de déploiement présentée par la figure 1.2 est la plus fréquemment constatée dans les applications incluant l'usatilisation de cartes.

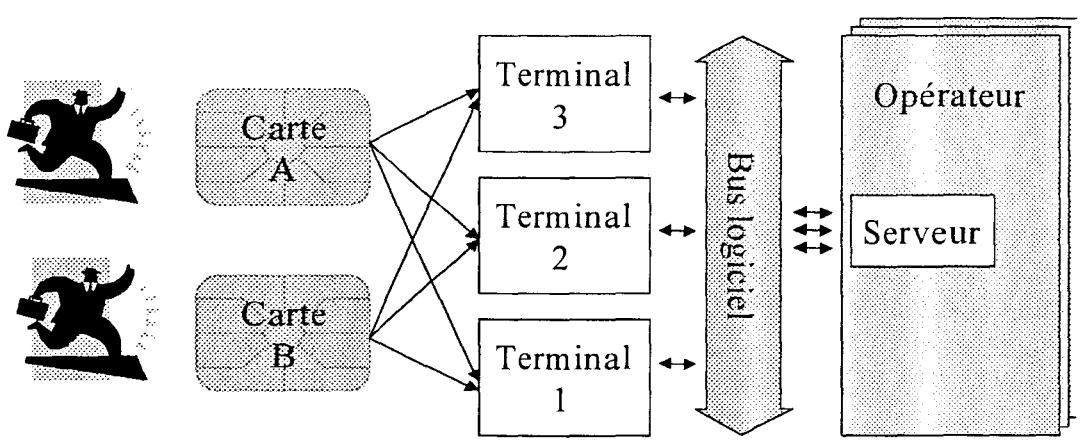


FIG. 1.2 – *Relations entre les acteurs d'applications qui utilisent des cartes*

Il faut noter que sur ce schéma la carte peut devenir un précieux atout de tolérance aux pannes, lorsque le réseau de communication entre le terminal et l'opérateur est hors service. Il peut même s'agir d'un fonctionnement annexe de l'application. Les applications bancaires par exemple peuvent fonctionner sur un modèle minimaliste appelé « hors ligne »

en n'utilisant que les ressources de la carte. Ainsi une connexion permanente entre le terminal de paiement (d'un restaurateur par exemple) et la banque n'est pas nécessaire. Aussi la relation carte/terminal a toujours joué un rôle central dans les architectures de déploiement des applications.

Le modèle de communication normalisé par l'ISO[ISO89] présuppose un usage orienté serveur, de la carte par le terminal : le terminal pilote l'activité de la carte. Le mécanisme de connexion de la carte est présenté par la figure 1.3. De fait, les TPDU [24], équivalents d'une trame de TCP/IP [25] échangées entre le terminal et la carte, décomposent la communication sous la forme de différents éléments de requêtes émis par le serveur, suivis d'éléments de réponse rendus par la carte. L'un des avantages de ce protocole est de n'impliquer aucun risque de collision de messages entre la carte et le terminal.

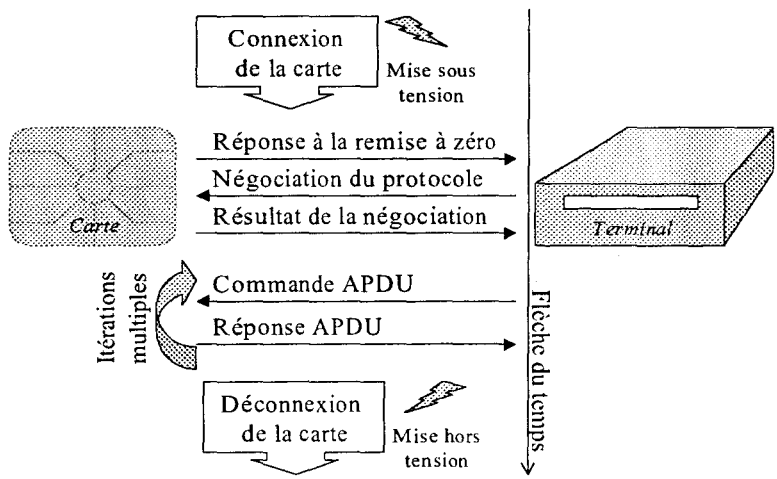


FIG. 1.3 – *Mécanisme de connexion carte/terminal*

Certaines applications encartées (concernant principalement les cartes Java-SIM) restent à l'étroit dans cette architecture où la carte apparaît toujours comme un serveur. Ces applications utilisent la carte non plus comme un serveur de traitements et de données sensibles, mais comme un client qui sollicite de son propre chef les infrastructures logicielles qui l'entourent. Pour cela le mécanisme de communication de la carte doit être renversé. En fait, il est possible de superposer la couche liaison (telle que la définit le modèle OSI [26] de l'ISO.) qui place la carte dans un modèle serveur avec une couche application qui montre aux applications encartées, les terminaux comme des clients potentiels. Il suffit pour cela que la première requête du terminal soit de la forme: « Quelle requête dois-je exécuter? ». La réponse de la carte est donc une requête déguisée. Le terminal enchaine

24. TPDU: Transport Protocol Data Unit.

25. TCP/IP: *Transmission Control Protocol / Internet Protocol.*

26. OSI: *Open System Interconnexion*, modèle de référence pour la communication entre des systèmes informatiques interconnectés.

ensuite des requêtes de la forme « La réponse à ta requête est ... Quelle requête dois-je exécuter? » avec la carte. Les cartes qui utilisent ce mécanisme sont appelées pro-actives. Finalement, ce qu'il faut retenir du protocole de communication spécifié par les normes ISO [ISO89] c'est qu'un mécanisme de va et vient entre la carte et le terminal rythme les transmissions en assurant l'absence de collisions. La sémantique des paquets transmis est de moindre importance, même si elle a malheureusement trop souvent influencé les architectures d'intégration des cartes.

1.4.2 Architectures d'intégration des cartes

D'un point de vue opérationnel, les serveurs de l'opérateur sont souvent amenés à dialoguer avec les cartes. Dans cette optique, la carte à puce doit être représentée sur le bus logiciel au même titre que son hôte (le terminal auquel elle est temporairement connectée). Des travaux de recherches [Van97] ont permis de développer un Objet d'Adaptation dédié aux Cartes pour bus logiciel nommé COA[27]. Le COA permet à la carte à microprocesseur, connectée à un terminal, de recevoir des requêtes provenant d'un serveur distant. Il joue le rôle d'une passerelle entre le bus logiciel (CORBA en l'occurrence) et le protocole de communication des cartes: les APDU[28]. Le terminal ne sert plus alors que de point de connexion entre la carte et le monde extérieur. La figure 1.4 présente l'intégration de la carte dans CORBA.

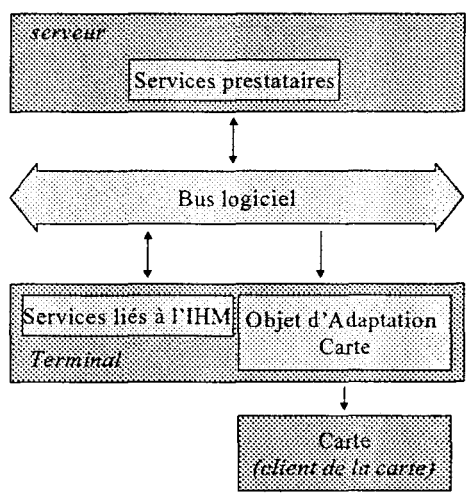


FIG. 1.4 – *Intégration de la carte dans l'architecture CORBA*

Ce modèle soulève de nouveaux problèmes. Les cartes à microprocesseur qui, par le biais d'un COA, deviennent des objets visibles et manipulables comme tout autre service

27. COA : Card Object Adapter, Objet d'Adaptation de la Carte sur un bus IIOP.
28. APDU : Application Protocol Data Unit, les APDU s'appuient sur les TPDU pour définir les couches 5,6 et 7 du modèle OSI.

distribué, ne sont cependant généralement connectées que durant les quelques instants où l'utilisateur accède au service. Pourtant des applications longues - C'est-à-dire les applications dont l'exécution ne se termine qu'après plusieurs connexions/déconnexions de la carte auprès de différents terminaux [LD97, Lec98, section 1.3.2, page 53] - nécessitent de disposer d'une représentation permanente des services de la carte d'un utilisateur mobile. Pour cela, des travaux de recherche [CLT96, Car98] ont proposé l'utilisation d'agents mobiles. Ces agents peuvent être utilisés pour éviter les problèmes de disponibilité de la carte. Dans le cadre de transactions entre la carte et des services distribués par exemple, l'arrachement de la carte avant la fin du protocole de validation à deux phases entraîne une file de messages qu'elle doit pouvoir recevoir lors de sa prochaine connexion, faute de quoi la transaction en cours ne pourra, ni être validée, ni être abandonnée. Ce concept a été généralisé par D. Carlier, P. Trane et S. Lecomte [CLT97]. Ici le terminal devient un support d'exécution pour agents mobiles. Les agents assurent la continuité des services et représentent les utilisateurs lorsque les cartes sont déconnectées. Ce support se présente alors comme la partie externe de l'architecture des cartes ouvertes.

1.4.3 Architecture des cartes ouvertes

Parallèlement aux efforts d'intégration des cartes à microprocesseur sur les réseaux distribués, d'autre efforts ont été consentis par les constructeurs pour en faciliter la programmation. Pour que ces cartes soient capables de charger dynamiquement de nouveaux logiciels, il a été nécéssaire de définir précisément le contenu du logiciel de base : le système d'exploitation. Ainsi, les cartes capables d'exécuter des programmes Java proposent un découpage des logiciels en quatre couches. L'architecture générale de ces cartes, capables de charger et d'executer plusieurs services distincts et mutuellement méfiants, est présentée par la figure 1.5.

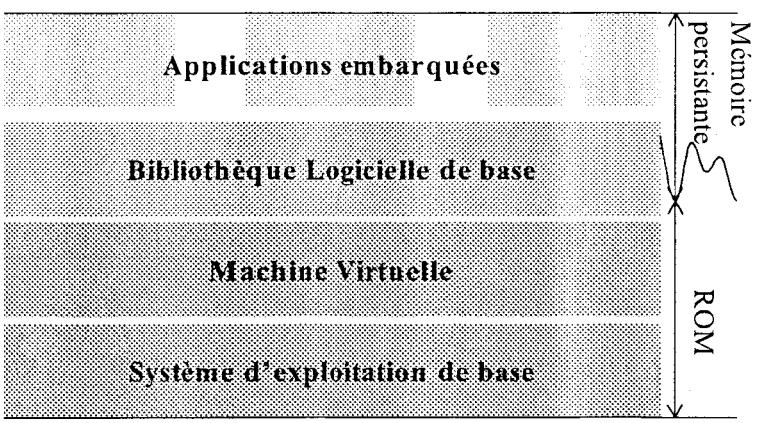


FIG. 1.5 – *Architecture logicielle des cartes JAVA*

Cette architecture se décompose en quatre couches logicielles superposées. Au plus haut niveau figurent les applications. Elles sont généralement confinées dans des espaces

d'exécution distincts. Ces applications s'appuient sur des bibliothèques JAVA dédiées à l'exploitation des ressources cartes. Il s'agit en fait d'un sous-ensemble réduit des API[29], spécifiées par le langage JAVA (car les cartes à microprocesseur sont trop contraintes pour pouvoir assumer l'ensemble des spécifications usuelles du langage). Le code des applications ainsi que le code des API, est exprimé avec un pseudo-code (appelé *bytecode* pour les cartes Java) dédié aux cartes [Sun96]. Ce bytecode est interprété par la machine virtuelle sous-jacente. Elle assure d'une part l'évaluation des opérations élémentaires et d'autre part l'appel aux fonctions du système de base qui assurent la gestion des protocoles de communication, des mécanismes cryptographiques et le support des normes ISO avancées[ISO94, ...].

En fait, comme bien souvent dans l'informatique traditionnelle, cette architecture est issue d'un empilement des systèmes successivement introduits dans la carte. Elle n'a cependant rien à envier aux architectures des autres systèmes d'exploitation pour cartes ouvertes qui sont très sensiblement identiques. Pour faire progresser ce type d'architectures, il faut identifier les critères de performance qui permettront de les évaluer. Ces critères dépendent bien évidement de leurs usages, et plus généralement de la manière de penser la carte de demain.

1.5 Penser la carte de demain

L'objectif de cette section est de proposer une prospective des futures utilisations de la carte à microprocesseur. Les motivations qui animent aujourd'hui encore les activités de recherche autour cet objet sont intimement liées au devenir de l'informatique, de plus en plus largement déployée, et de ses utilisateurs, de plus en plus « nomade » et exigeants vis à vis des systèmes d'informations. Aussi, pour comprendre les objectifs des futures cartes, il faut tout d'abord présenter la problématique de l'informatique omniprésente[30] pour utilisateurs mobiles.

1.5.1 Informatique omniprésente pour utilisateurs mobiles

L'informatique change de visage. Les quelques gros serveurs qui centralisaient tous les traitements se sont transformés en des stations de travail bien plus nombreuses et personnalisées pour chaque utilisateur. Aujourd'hui la miniaturisation des microprocesseurs initie un nouveau changement: plutôt qu'une seule machine centralisant toutes nos applications, c'est une multitudes de systèmes spécialisés qui répondent aux exigences croissantes de chaque utilisateur. C'est en s'appuyant sur cette constatation que Mark Weiser[31], l'un des pères de l'informatique omniprésente, fonda sa vision de l'informatique de

29. API: Application Programming Interface, ensemble d'interfaces logicielles qui à pour but de faciliter la programmation des applications.

30. traduction du terme anglo-saxon *Ubiquitous Computing.*

31. chercheur du *Xerox Parc*, aujourd'hui décédé.

demain[Wei93, WB97, WGB99]. Le graphique 1.6 présente l'évolution du marché des ordinateurs tel qu'il la prévoyait. Aujourd'hui une multitude d'outils informatiques ont envahi nos bureaux, nos maisons, nos voitures, mais aussi et surtout notre attention.

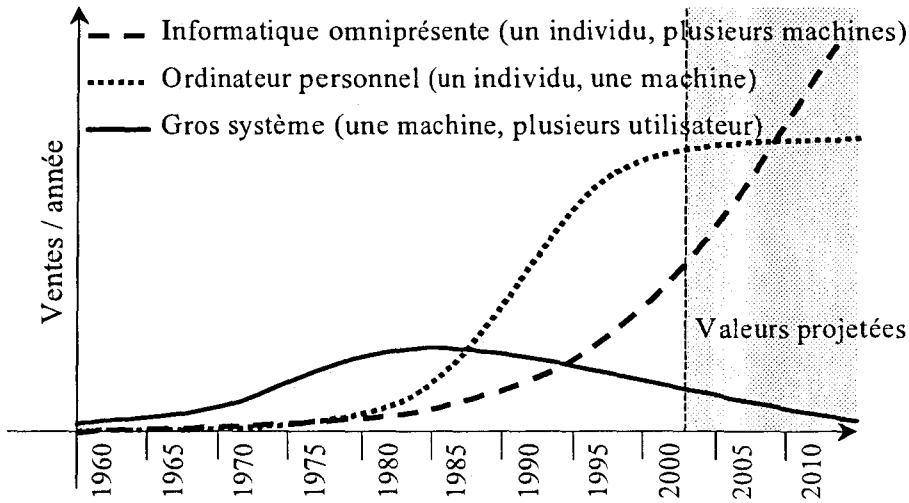


FIG. 1.6 – *Evolution du marché de l'informatique*

Cette évolution soulève cependant de nouveaux problèmes. Il est en effet regrettable que « *plutôt que de devenir un objet avec lequel nous travaillons, et qui devrait donc disparaître de notre conscience, l'ordinateur requière pour lui-même l'essentiel de notre attention.* »[32]. On peut penser qu'il s'agit d'une étape normale lors de l'émergence de nouvelles technologies. Cependant, il faut bien reconnaître qu'à ce jour les évolutions successives de l'informatique ne se sont pas accompagnées d'un effort d'harmonisation de leur mise en œuvre vis à vis de leurs utilisateurs. Force est de constater que l'utilisateur doit s'adapter aux machines qui l'entourent.

L'inadéquation entre les services informatiques et leurs utilisateurs ne se résume pas simplement à un problème d'ergonomie de l'interface homme-machine. Quelle est l'interface graphique d'un service bancaire, téléphonique ou médical informatisé? Les supports visuels et matériels du service changent selon les lieux et les utilisations qui en sont faites. Trop souvent lorsqu'un utilisateur se déplace, il doit pallier au manque de continuité des services qu'il possède. Les utilisateurs mobiles ont à s'occuper de tâches de gestion non triviales (transfert, vérification, duplication, conversion, sécurisation) de l'information. Ces tâches devraient, à terme, être prises en charge automatiquement, sans que l'utilisateur n'ait à s'en préoccuper. Dans le cas contraire on peut craindre que la complexité croissante des structures informatiques les rendent inexploitables. Au terme de son évolution, l'informatique doit se dissimuler. Ce n'est qu'à ce prix qu'elle peut devenir omniprésente

32. extrait traduit de l'article [Wei93, page 76 ligne 17-20].

et véritablement utile pour ses usagers. Cette profonde réforme de la manière de penser l'informatique est nécessaire pour que demain soit enfin *la nuit des temps*[33].

1.5.2 Pour que demain soit enfin *la nuit des temps*

Lorsque Rolland Moréno a publié en 1974 les quatre brevets fondateurs de la carte à puce, il était convaincu d'avoir une vision profondément novatrice des relations entre les utilisateurs et les systèmes automatisés. Il a reconnu, plus tard, que cette vision était probablement issue d'un livre de science fiction de R. Barjavel : *la nuit des temps*. Il y est fait allusion à un système électronique appelé « la clef » ; il s'agit d'un petit anneau que chaque habitant possède et qui assure plusieurs fonctions, au nombre desquelles on compte l'identification et l'authentification des individus ainsi que la personnalisation de l'accès à tous les services automatiques disponibles... [34].

Aujourd'hui la carte à microprocesseur n'est utilisée que sur un nombre limité de services déployés à grand échelle. La section 1.2 page 9 montre que la carte à microprocesseur est alors une solution performante. Elle permet de gérer la mobilité de l'utilisateur, la sécurité des données et la personnalisation des services utilisés, même dans des conditions réelles difficiles (défaillances des réseaux, malveillance des pirates ...). Le succès de son intégration dans les téléphones portables nous montre qu'elle a un rôle à jouer dans l'utilisation d'objets de plus en plus petits et « intelligents ». Pour que demain la carte devienne la clef de l'omniprésence de l'informatique, il faut parvenir à généraliser ses activités et son fonctionnement; il faut donner aux concepteurs d'applications et de services les outils qui leur permettent simplement de placer la carte au cœur de leurs architectures logicielles pour y gérer la mobilité, la sécurité et l'interopérabilité avec d'autres applications.

Pour être notre représentant électronique universel, la carte ouverte doit être le point de confluence des différents services et des applications que nous sollicitons [GJ99, GJ00b]. En effet, les informations critiques présentes dans les cartes ne devraient pas avoir à être extraites pour être partagées par différentes applications lorsqu'elles s'inscrivent dans une opération globale, au nom du porteur. Le stéréotype de la coopération inter-services est aujourd'hui l'opération de fidélité; l'émetteur de la carte permet le chargement d'autres programmes qui coopèrent pour gérer des réductions forfaitaires communes sur leurs services. La carte parait alors être le meilleur « lieu » pour réaliser ces opérations car chaque service y possède les informations clés nécessaires au partenariat – c'est-à-dire les informations personnelles, associées à chaque usager, et dont la confidentialité et l'intégrité sont essentielles pour le fonctionnement du système d'information. Cette utilisation de la carte implique qu'elle soit à même de supporter en son sein l'interopérabilité à tous les niveaux entre des programmes, dans un contexte de sécurité nominal.

Ainsi la carte deviendra l'objet imaginé par R. Barjavel. Toutefois, ces nouvelles exigences soulèvent un certain nombre de problèmes que l'on ne sait pas traiter correctement

33. *la nuit des temps*, R. Barjavel, ISBN 2266023039, édition Pocket.

34. Dans l'imagination de R. Barjavel « la clef » a une troisième fonction. Rattacher cette dernière au rôle des cartes restera probablement pour toujours une utopie !

et qui constituent finalement les véritables verrous technologiques des cartes ouvertes.

1.5.3 Verrous technologiques des cartes ouvertes

Il a été montré dans les sections précédentes que les contraintes matérielles importantes ont introduit une manière bien spécifique de penser le logiciel encarté. Les cartes ouvertes à la programmation de tous, à chaque moment de leur cycle de vie, restent cloisonnées dans des limitations drastiques. Les contraintes de la programmation des cartes se déclinent en termes :

- de **portabilité** des programmes sur les différents microprocesseurs encartés ;
- d'**hétérogénéité** des langages de programmations utilisables ;
- de **sécurité-innocuité** des applications et de leurs données ;
- d'**interopérabilité** des programmes dans la carte et entre la carte et *l'extérieur* ;
- de **flexibilité/extensibilité** du système d'exploitation et des abstractions proposées ;
- d'**efficacité** d'exécution des programmes chargés ;
- de **compacité** des structures de données et des traitements encartés.

Aujourd'hui les cartes ouvertes *JavaCard*, *Windows for Smartcard* et *MULTOS* proposent des solutions à trois de ces critères : la **portabilité**, la **compacité** et la **sécurité-innocuité** des applications. Dans ces trois produits, la portabilité est toujours assurée par l'entremise d'une machine virtuelle. En organisant cette machine virtuelle autour d'une pile, on assure aussi un certain niveau de compacité. Pour la sécurité-innocuité, deux solutions sont utilisées : la première, qui n'en est pas véritablement une, repose sur la confiance mutuelle des partenaires contrôlée par la carte à l'aide d'un mécanisme de signature digitale ; dans ce cas les cartes ne peuvent supporter que des services dont les prestataires ont préalablement conclus des partenariats ; et elle perdent leur vocation « universelle ». L'alternative proposée par l'industrie à cette semi-solution est de cloisonner les programmes dans des espaces mémoires distincts. C'est alors la machine virtuelle, qui contrôle avant d'exécuter chaque opération élémentaire d'un programme, qu'elle n'est pas nuisible pour autrui.

Pour les autres critères, beaucoup reste à faire. L'interopérabilité entre les programmes encartés et les programmes hors cartes a tout de même été le sujet d'études particulières [VV98]. L'interopérabilité entre programmes au sein d'une même carte a été partiellement abordée dans les spécifications *JavaCard*. La gestion de la sécurité-innocuité dans ces conditions reste cependant insatisfaisante. Le critère d'efficacité d'exécution des traitements n'est absolument pas abordé dans la carte; les machines virtuelles encartées se montrent particulièrement contre-performantes avec, par exemple, seulement quelques milliers de bytecode java exécutés chaque secondes ; quant à l'hétérogénéité des langages de programmations supportés et à la flexibilité des systèmes d'exploitation, rien n'est discuté ni dans la littérature scientifique spécifique aux cartes à microprocesseur ni dans les documentations techniques industrielles. Beaucoup reste à faire dans ce domaine pour que se rencontrent véritablement cartes et systèmes d'exploitation adaptables.

Chapitre 2

Cartes et systèmes d'exploitation adaptables

« *Avoir un système borne son horizon ; n'en avoir pas est impossible. Le mieux est d'en posséder plusieurs.* »

Journaux, Raymond Queneau

L'objet de ce chapitre est de donner au lecteur les éléments de réflexion suffisants pour comprendre les principes qui ont guidé les travaux présentés par la suite. Des ouvrages de référence (notamment les références [Kra89, Tan89]) peuvent être utilisés pour retrouver les fondements du domaine des systèmes d'exploitation. Les sujets abordés par ce chapitre sont associés, lorsque cela est possible, à des considérations relatives aux cartes à microprocesseur. Ils sont néanmoins issus des recherches récentes menées par la communauté scientifique propre aux systèmes d'exploitation. L'objectif est de montrer comment se concrétise, dans le contexte bien spécifique de la carte à puce, les interrogations propres aux recherches dans les systèmes d'exploitation.

La première section rappelle les différents modèles d'implantation des systèmes d'exploitation. L'objectif est d'identifier ce qui fait un système d'exploitation.

La seconde section porte une attention particulière à un modèle de système d'exploitation pensé en tant que gestionnaire du matériel : les Exo-noyau[1].

La troisième section s'intéresse aux récents développements de la problématique de l'extensibilité des supports d'exécution des logiciels en décrivent les propositions qui constituent les fondements de l'architecture des « machines virtuelles virtuelles ».

La quatrième section rappelle l'état de l'art sur ce que sont aujourd'hui, indépendamment de toute architecture système, les solutions envisageables pour assurer la sécurité des logiciels qui coexistent sur un même support. Cette problématique est particulièrement

1. Exo-Noyau: traduction du terme *ExoKernel* dans la littérature anglo-saxonne

sensible dans le contexte des cartes à microprocesseur où il a été montré que la sécurité est essentielle.

Enfin la dernière partie introduit la notion de système d'exploitation dédiée aux cartes à microprocesseur. Elle présente les difficultés inhérentes à l'intégration de concepts avancés dans le contexte des cartes dans lesquelles ont déjà été implantés différents modèles de systèmes d'exploitation.

2.1 Différents modèles de systèmes d'exploitation

Aujourd'hui les systèmes d'exploitation répondent à trois besoins dissociés.

1. **Les systèmes d'exploitation gèrent le matériel.** L'utilisation du matériel nécessite en général d'exécuter certains traitements logiciels qui n'ont rien à voir avec le but dans lequel on sollicite le materiel. Dans une carte à microprocesseur, par exemple, l'accès en écriture à une page de mémoire Flash n'est pas une opération élémentaire du microprocesseur. Une routine dédiée charge les « pompes d'écriture ». Elle veille à respecter certains « délais de garde » durant l'effacement de la mémoire, puis elle écrit. De plus en cas d'écritures répétées la page de Flash peut être « stressée » et devient alors inutilisable. Le système d'exploitation gère le matériel de telle sorte que toutes ces considérations matérielles ne soient pas prises en charge par les applications encartées.
2. **Les systèmes d'exploitation fournissent une abstraction fonctionnelle du matériel.** La mémoire persistante des cartes est par exemple souvent représentée au programmeur sous la forme d'un système de fichier. C'est le cas des normes ISO 7816-4 ou des *SmartCard for Windows*. Elle est parfois aussi présentée comme un support d'objets persistants (dans les cartes JAVA). L'objectif est de fournir au concepteur d'applications une manière conviviale d'utiliser le support physique. Aucune règle particulière ne guide en fait le modèle d'abstraction qui est défini par le système d'exploitation. Il en va de même pour les entrées/sorties (qui dans la carte sont présentés sous la forme d'APDU) et même du microprocesseur qui est maintenant masqué par la notion de machine virtuelle.
3. **Les systèmes d'exploitation protègent les ressources fournies aux différentes applications.** Cette tâche a pour but d'assurer la pérennité des applications. Lorsque plusieurs programmes coexistent sur un même support d'exécution, l'activité de l'un peut nuire au fonctionnement de l'autre. Dans une carte, un programme qui gère un porte-monnaie électronique stocke en mémoire persistante le code secret qui permet son utilisation. Si cette ressource mémoire peut être atteinte (simplement lue en l'occurrence) par un autre programme chargé ultérieurement, alors ce programme nuit au fonctionnement de l'application porte-monnaie. Pour pallier à ce problème, le système d'exploitation est alors pensé comme un médiateur entre les différentes applications : il devient le noyau de confiance grâce auquel les ressources accordées à chaque programme sont sécurisées.

Tous les systèmes d'exploitation répondent à ces trois critères. Chacun apporte des approches différentes qui amènent des architectures différentes. Finalement, le critère qui les distingue les uns des autres porte sur la frontière qu'ils définissent avec les applications sur les trois axes que sont : la gestion du matériel, l'abstraction du matériel et la sécurité des programmes. Cette frontière entre système et applications se manifeste le plus souvent par des pénalités qui augmentent le temps machine consommé et qui limitent l'utilisation des ressources du matériel. Ces pénalités sont généralement induites par certains mécanismes de protection choisis par les systèmes. Mais il est aussi possible de refuser cette séparation. Tel est le cas des systèmes d'exploitation intégrés.

2.1.1 Systèmes d'exploitation intégrés

Historiquement, avant que le concept de systèmes d'exploitation ne soit clairement identifié, les programmes réalisaient indifféremment des tâches applicatives et des tâches de gestion du matériel. Il était impossible de distinguer dans le programme ce qui relevait de la gestion du matériel, de ce qui relevait des traitements applicatifs. Au mieux, quelques routines de gestion du matériel étaient systématiquement réutilisées. Dans ce cas, les bibliothèques constituées par ces routines représentaient le système d'exploitation. De fait, on constate dans les systèmes intégrés que la fonction d'abstraction disparaît partiellement ou complètement. L'application, conçue en même temps que le système, s'appuie directement sur les routines de gestion du matériel. L'une des conséquences est que la conception de nouveaux logiciels devient une tâche ardue, aussi le délai de conception est particulièrement important. Cette approche, très ancienne dans les systèmes informatiques conventionnels, a dès les années 1964-1965, été remplacée de véritables systèmes d'exploitation.

Dans le contexte des cartes à puce, cette technique est longtemps restée la seule manière de faire. Elle s'y est même développée plus qu'ailleurs car elle présente plusieurs avantages. L'absence de couche d'abstraction accroît de manière très sensible les performances des applications tout en diminuant d'autant la taille totale des logiciels encartés, ce qui est essentiel pour réduire les coûts de productions des fabricants. Le travail du programmeur est de trouver « le plus court chemin » entre les services accomplis par son logiciel et le matériel dont il dispose. Des systèmes formels[Lan98] et des techniques de test ont été spécialement développées pour prouver, avant leur émission, la viabilité des applications à systèmes intégrés qui étaient encartées. Jusqu'à la fin des années 90, les spécialistes de la carte parlaient de système d'exploitation pour désigner le tout que formaient les logiciels encartés. Ce n'est qu'avec l'accroissement de la puissance du matériel disponible dans les cartes que les applications intégrées ont été remplacées progressivement par les premiers véritables systèmes d'exploitation monolithiques.

2.1.2 Systèmes d'exploitation monolithiques

Les systèmes d'exploitation monolithiques sont pensés comme un tout. Dans le meilleur des cas ils gèrent le matériel, en proposent une abstraction et assurent la sécurité (en

protégeant les ressources fournies aux applications). Ils constituent une couche logicielle placée entre le matériel et les applications. Le système de fichiers est l'exemple le plus conventionnel d'abstraction que propose un système d'exploitation monolithique. La figure 2.1 présente l'architecture interne de la partie « gestion de fichier » (sur le principe de la norme ISO7816-4) dans une carte à microprocesseur. La couche supérieure assure l'interface entre le système d'exploitation et les applications. Elle ne correspond pas au cœur du système d'exploitation. Cependant on distingue sur le graphique le système se subdivise lui-même en sous-couches. Au plus bas on trouve les routines de gestion du matériel. Elles gèrent l'allocation des unités physiques de mémoire et leurs accès. Sur cette base s'appuie la gestion du regroupement des unités physiques en unité logique et de l'accès aux unités logiques indépendamment les une des autres. Traditionnellement c'est sur les unités logiques qu'est placé un mécanisme de contrôle d'accès pour garantir la sécurité-innocuité des applications.

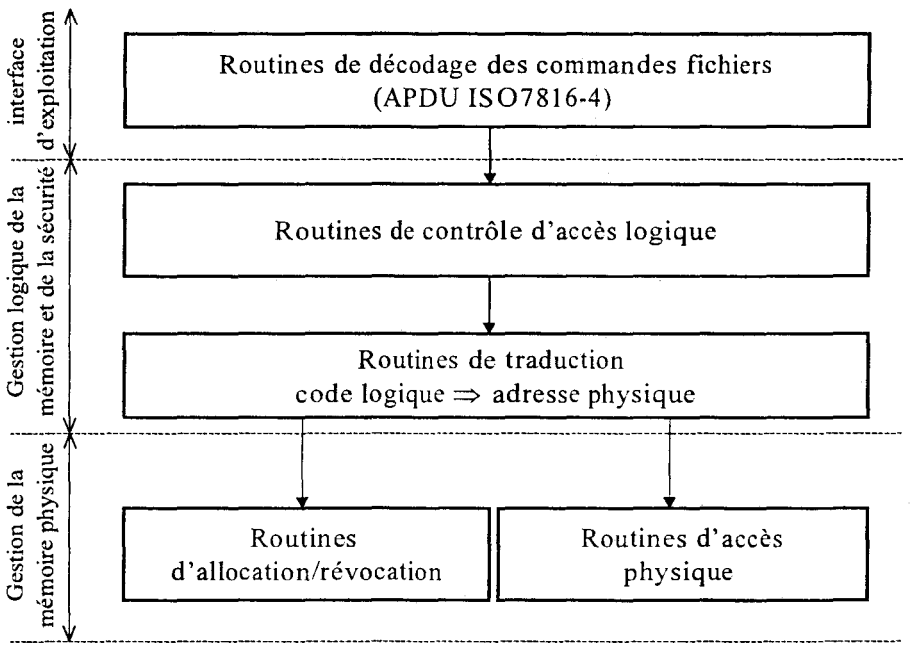


FIG. 2.1 – *Architecture d'un système de gestion de fichiers encarté*

Les couches internes de la gestion d'un système monolithique ne doivent pas nous faire perdre de vue que le propre de ce type de système est de présenter la gestion du matériel au travers d'une abstraction unique. Les sous-couches sont bien évidemment inaccessibles aux applications, faute de quoi les mécanismes de sécurité pourraient être contournés. Le propre d'un système monolithique est d'être une « offre tout-en-un » : **une gestion du matériel + une abstraction + un système de sécurité.** Dans la pratique, cette intégration des différents aspects permet de réaliser un bloc fonctionnel plus performant que s'ils étaient dissociés. Toutefois, l'abstraction unique qu'ils proposent aux applications se montre parfois

totalement inadaptée. Les applications orientés bases de données se trouvent par exemple fortement pénalisées lorsqu'elles sont réalisées « au-dessus » de systèmes qui représentent la mémoire persistante sous forme de fichiers[SS94]. Finalement, on retrouve à un niveau moindre le même phénomène que celui des applications à systèmes intégrés : la complexité croissante des tâches assumées par le noyau des systèmes monolithiques les rend de plus en plus difficiles à maintenir; elle les rend aussi de moins en moins réutilisables. Pour maîtriser cette complexité, la communauté des systèmes d'exploitation a cherché à réduire la taille du noyau, ce qui a abouti au concept de micro-noyau.

2.1.3 Micro-noyau

L'objectif principal des micro-noyaux [GJ91, RAA+92] est de parvenir à maîtriser la complexité croissante des logiciels qui constituent un système d'exploitation. Pour les concepteurs ont cherchés à rendre plus synthétiques, plus homogènes les abstractions qui sont proposées. Aussi l'une des principales caractéristiques d'un micro-noyau est la taille réduite de son code. Comme ils sont plus petits que les systèmes monolithiques, ils sont aussi plus facilement compris et leur maintenance est simplifiée. La taille d'un micro-noyau peut aller d'un kilo octets et demi à une centaine de kilos octets. Malgré cette première caractéristique attrayante dans un contexte minimaliste, ces architectures n'ont jamais su retenir l'attention des concepteurs de systèmes des cartes à microprocesseur. Mais la réduction de la taille du noyau d'un système a aussi pour conséquence de limiter le nombre de services proposés. En fait, cette approche montre que les programmeurs système renoncent à implanter toutes les abstractions que les applications peuvent réclamer.

Les micro-noyaux cherchent à proposer un découpage des ressources matérielles et une abstraction sécurisée plus simple et plus flexible que celle proposée par les systèmes monolithiques. Les applications peuvent ensuite réutiliser les interfaces du micro-noyau pour définir leurs propres modèles de mémoires persistantes, par exemple.

La principale difficulté pour réaliser un micro-noyau est de réussir à proposer un modèle de base véritablement réutilisable et performant. Il existe aujourd'hui deux familles de micro-noyaux correspondant à des stratégies différentes.

La première stratégie consiste à proposer des mécanismes permettant d'étendre le contenu du micro-noyau. Ainsi, chaque application qui souhaite un service inconnu du noyau peut l'y ajouter. Les performances des composants ajoutés sont supérieures à ce qu'elles seraient si elle étaient placées du côté de l'application car elles ne subissent plus les pénalités de performance lorsqu'elles sollicitent les interfaces de base du micro-noyau. La difficulté est alors de maintenir un niveau de sécurité suffisant alors que les problèmes peuvent venir du noyau du système lui-même. Les systèmes SPIN[BSP+95] et Vino[SESS96] exposent les problèmes rencontrés dans l'extension du logiciel placé dans le noyau.

La seconde stratégie consiste à proposer des micro-noyaux qui exposent des interfaces d'exploitation plus proches du matériel géré[CD94, HHL+97]. Ainsi l'implantation de nouveaux mécanismes n'est plus contrainte que par le matériel. Les routines de base proposées par le système d'exploitation ne se présentent plus comme une couche de traitements, par-

fois inutiles (voire néfastes), mais imposées de fait par les ressources physiques qui sont manipulées. Dans cette approche, la mémoire persistante, par exemple, n'est plus présentée comme un support de fichiers, mais comme un ensemble de pages avec une sémantique d'exploitation proche de celle des secteurs de disque manipulés pour les représenter. Libre alors à chacun d'exploiter ces pages pour implanter des fichiers, des objets Java ou des lignes d'une table CQL. Chacun peut exploiter différemment les ressources matérielles.

2.1.4 Exploiter différemment les ressources matérielles

Finalement, aujourd'hui, deux domaines d'investigation scientifique différents se dessinent derrière la conception d'un système d'exploitation.

Un système d'exploitation, en tant que support d'exécution, doit pouvoir répondre aux exigences croissantes des applications. Il devient impératif de parvenir à faire la synthèse des besoins que les programmeurs manifestent vis à vis des plates-formes qui exécutent leurs applications. La section 2.3 présente au travers d'une architecture novatrice comment il est possible de formuler une réponse cohérente aux attentes de plus en plus diversifier des concepteurs d'application. Cependant, un système d'exploitation gère avant tout des ressources physiques. Les reproches adressés aux systèmes existants portent presque toujours sur des limitations qui sont induites par l'abstraction du matériel qui est proposé par le système. C'est en partant de cette constatation [EK95] qu'une équipe de chercheur du MIT[2] a chercher à définir une nouvelle architecture de systèmes d'exploitation plus efficaces, fiables et extensibles qui ce place au plus près du matériel : les exo-noyaux.

2.2 Au plus près du matériel : les exo-noyaux

L'ambition de ces systèmes d'exploitation est de donner la vue la plus complète possible des ressources matérielles, plutôt que de les masquer derrière des interfaces abstraites. Pour les concepteurs des exo-noyaux, l'abstraction est nuisible à l'extensibilité. Aussi cherchent-ils la supprimer entièrement du cœur de leurs système. C'est au premier abord l'ambition des Exo-noyaux.

2.2.1 L'ambition des Exo-noyaux

Les Exo-noyaux se proposent de donner aux applications un contrôle fiable des ressources matérielles. Pour cela, les exo-noyaux cherchent à séparer, chaque fois que cela est possible, les mécanismes de protection des ressources des mécanismes de gestion (de l'abstraction) de ces ressources. Toutes les fonctions nécessaires à la protection résidant dans l'exo-noyau, le contrôle sur tous les autres aspects du matériel est permis aux applications. Idéalement, les applications peuvent exécuter de façon efficace et sécurisée toutes les opérations qu'un système d'exploitation peut solliciter vis-à-vis des matériels

2. MIT : Massachusetts Institute of Technology

qu'il gère. L'objectif n'est pas d'obliger tous les programmeurs d'application à gérer eux-mêmes le matériel, mais plutôt de permettre à chaque application d'utiliser, sous forme de bibliothèques, le gestionnaire de ressources matérielles le plus approprié. La figure 2.2 montre la décomposition logicielle de deux applications qui s'appuient sur le découpage en couches d'un exo-noyau.

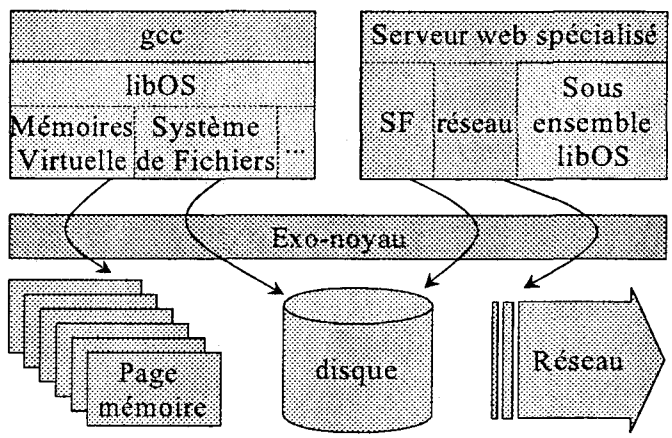


FIG. 2.2 – *Applications et exo-noyau*

Le noyau de ce type d'architecture se présente comme la couche de logiciel la plus fine possible entre le matériel et les applications. L'objectif est la transparence du système vis-à-vis du matériel qu'il protège. Trois prototypes distincts ont été conçus à ce jour. Leurs conception repose sur des règles de base qui constituent les principes des Exo-noyaux.

2.2.2 Les principes des Exo-noyaux

Dawson R. Engler définit dans sa thèse [Eng98], intitulé *The Exokernel Operating System Architecture*, un ensemble de sept principes fondamentaux qui guident les décisions prises pour concevoir un exo-noyau :

1. **séparer la protection et la gestion du matériel**. Ce principe est un fondement de la notion des exo-noyaux. Les seules routines gérées par le noyau sont celles qui permettent de manière sécurisée d'allouer, de révoquer, de partager et de déterminer le possesseur de chaque ressource matérielle. Les applications peuvent allouer un secteur sur le disque, le rendre, le partager ... L'utilisation faite de cette ressource (partie d'un fichier, ligne d'une base de données, ...), reste du ressort de l'application ;
2. **exposer le matériel**. Ce principe impose que le noyau ne masque rien du matériel. Le plus grand nombre possible d'aspects d'un matériel donné doivent être exposés par le noyau, c'est-à-dire utilisables par les applications. L'objectif est de permettre des utilisations aussi variées que possible de chaque ressource matérielle ;

3. **protéger à grain fin.** L'idée avancée par les concepteurs des exo-noyaux est qu'en réduisant la taille de l'unité de protection on réduit le coût de la projection d'abstractions de haut niveau sur les unités de ressources réelles;
4. **exposer les mécanismes de nommage.** Les exo-noyaux utilisent chaque fois que possible les identifiants physiques des ressources (adresse mémoire réelle plutôt que virtuelle). L'introduction de niveaux d'indirection dans l'exo-noyau ne ferait que s'ajouter à ceux des bibliothèques systèmes placées dans les applications. Lorsqu'un système de nommage est néanmoins utilisé, les applications doivent pouvoir retrouver les propriétés intrinsèques des unités de ressources matérielles manipulées (par exemple la proximité physique de deux blocs de mémoire sur un disque);
5. **exposer les mécanismes d'allocation.** Les exo-noyaux doivent permettre aux applications de maîtriser le processus d'allocation des ressources. Elles doivent pouvoir décider quand et quelle unité doit être allouée;
6. **exposer les mécanismes de révocation.** Les Exo-noyaux doivent permettre aux applications de maîtriser le processus de libération des ressources. Les applications qui ont alloué une ressource peuvent la rendre explicitement;
7. **exposer les informations systèmes.** Les exo-noyaux exposent le plus souvent possible les informations systèmes qu'ils manipulent. Cela doit permettre aux bibliothèques systèmes des applications de connaître l'état global de d'utilisation des ressources afin de choisir leurs stratégies de gestion. En effet, avec les exo-noyaux les librairies systèmes des applications ne sont plus à même de collecter les informations systèmes globales (taux d'activité du microprocesseur, taux de sollicitation des disques, ...).

Après plus de cinq ans de travaux et trois implantations du concept d'exo-noyau, un certain nombre de leçons ont été tirées des exo-noyaux [Eng98, section 7.2][KEG+97].

2.2.3 Les leçons à tirer des exo-noyaux

Nous reprenons ici les principaux enseignements présentés par les concepteurs de cette architecture système. Certaines restent valables dans le contexte de la carte à microprocesseur, d'autres deviennent irrecevables:

- **certaines informations pertinentes peuvent être perdues en implantant des composants systèmes au dessus d'un exo-noyau.** L'optimisation de certains composants systèmes est meilleure lorsqu'elle maîtrise de l'information relative à l'ensemble des applications. Cette remarque est notamment recevable pour les mécanismes de gestion de caches qui sont moins performants lorsqu'ils sont distribués entre chaque application que lorsqu'ils sont communs à toutes les applications;
- **il est utile d'exposer les structures du noyau.** Cette remarque est beaucoup moins pertinente dans le contexte de la carte. Les données du noyau peuvent être utilisées par des applications malveillantes, éventuellement en les combinant à des attaques physiques du composant (mise hors tension par exemple), pour nuire à la

sécurité globale du système. Rappelons que le seul élément de mesure de l'activité des applications encartées, le temps de réponse a été exploité par le passé pour mettre en défaut la sécurité de certaines cartes[DKL+98];

- **les bibliothèques sont plus simples à concevoir que les noyaux.** Cette constatation est importante car le temps de développement du logiciel de base dans une carte à microprocesseur est un critère de première importance. Or l'évolution des besoins des applications ne peut pallier l'inflexibilité des systèmes d'exploitation, ni par l'augmentation de la puissance de calcul, ni par l'augmentation de la puissance de stockage des logiciels;
- **concevoir l'interface de base d'un exo-noyau est une entreprise difficile.** Le matériel élémentaire présent dans les cartes à microprocesseur devrait simplifier ce problème. L'absence de mémoire virtuelle, de disques, et de toute autre architecture matérielle reconnue difficile à exposer par les exo-noyaux facilite énormément la tâche;
- **fournir des bibliothèques d'extension du système par défaut efficaces est une bonne stratégie.** Cette remarque est particulièrement pertinente dans le cas de la carte. La section 1.3.2 page 15 montre que le point mémoire de la ROM est très avantageux. Aussi, identifier et ajouter au système de base de la carte, un ensemble de « bibliothèques par défaut » est une stratégie payante car elles n'occuperont pas, plus tard un espace mémoire (plus coûteux) sur EEPROM;
- **les systèmes de typage de bas niveaux sont utiles.** La sécurisation de l'exposition des ressources matérielles peut trouver de nouvelles solutions avec des techniques de protection encore mal exploitées par les systèmes d'exploitation traditionnels.

Finalement, les concepteurs des exo-noyaux s'interrogent sur la viabilité de leur approche. Leurs inquiétudes portent sur la capacité des programmeurs d'application à exploiter la liberté qui leur est donnée dans le but de tirer un parti optimal du matériel. Les habitudes que les programmeurs ont prises avec les interfaces systèmes standardisées (persistance assurée par l'utilisation exclusive de fichiers, réseau vu au travers du prisme unique de TCP/IP, ...), reconnues comme étant contre-performantes pour certaines applications et malgré tout de plus en plus systématiquement utilisées, nuisent profondément à l'innovation et à la recherche en système.

Il est clair aujourd'hui que pour une majorité d'applications le sujet de préoccupation principal des programmeurs n'est plus lié à l'exploitation nominale d'un matériel déjà surpuissant. Ceux qui écrivent les nouvelles applications perçoivent de plus en plus souvent l'ordinateur comme une machine abstraite dont la sémantique est définie par le langage qu'ils utilisent. La généralisation récente de l'utilisation du concept ancien des machines virtuelles (qui simule le fonctionnement d'une machine abstraite plus simple à utiliser que la machine réelle) permet d'aborder sous un angle différent les fonctions d'abstraction du matériel dont la conception est le « second métier » des programmeurs système. Une démarche novatrice place aujourd'hui la réflexion sur les plates-formes d'exécution au plus près des applications: les machines virtuelles virtuelles.

2.3 Au plus près des applications : les machines virtuelles virtuelles

Les MVV[3] sont le fruit d'une réflexion sur les nouveaux besoins qu'expriment les applications (et leurs concepteurs) vis à vis du système d'exploitation. La généralisation de l'utilisation des machines virtuelles (Java, O-Caml, Smalltalk, Prolog ...) n'a pas apportée la réponse globale aux problèmes de génie logiciel visés. Le concept de la machine virtuelle Universelle a cédé la place à une foule de machines virtuelles incompatibles des unes avec les autres. Alors que la complexité croissante des applications les amène de plus en plus souvent à interopérer, leurs supports d'exécution distincts les confinent dans des espaces dissociés. Et finalement il faut reconnaître que ces nouvelles plate-formes d'exécution n'apportent qu'une réponse partielle aux exigences des programmeurs. Identifier et répondre à ces nouvelles exigences, tel est l'ambition des machines virtuelles virtuelles.

2.3.1 L'ambition des machines virtuelles virtuelles

Les MVV s'appuient sur l'identification de cinq fonctions distinctes, utiles aux nouvelles applications[FPR97, FPR98, BP99] :

- **portabilité des programmes**. Les applications sont amenées à se déployer et à s'exécuter sur des matériels de plus en plus diversifiés (*c.f.* section 1.5.1 page 23). Le système d'exploitation doit proposer des solutions à ce problème. Cette remarque est tout aussi pertinente dans le contexte des cartes, où les microprocesseurs utilisés sont bien plus variés, dans leur nature (Hitachi, Siemens, Motorola, Texas Instruments, Philips, Atmel, ...) et dans leur architecture (toute la gamme du CISC au RISC, 8bits à 32bits) que ne le sont les microprocesseurs de l'informatique traditionnelle (*c.f.* session 1.3.3, page 16) ;
- **hétérogénéité des langages de programmations**. Les programmeurs ont à leur disposition des langages très variés et adaptés à différentes sortes d'applications. Le système d'exploitation doit pouvoir s'adapter simplement et efficacement aux différents langages. Cette remarque reste vraie dans le contexte de la carte ou différents langages sont d'ores et déjà utilisables pour programmer les applications. En fait elle devient particulièrement pertinente en considérant que tous les codes chargés dans la carte sont exprimés pour une machine virtuelle ;
- **interopérabilité des programmes**. De plus en plus souvent les programmes sont amenés à coopérer pour fournir plus de services à leurs utilisateurs. Cette fonction est d'autant plus compliquée à réaliser lorsque les applications sont écrites dans des langages hétérogènes. L'interopérabilité est pour la carte (dans le cadre présenté par la section 1.5, page 23), un élément clef. Elle doit être abordée dans un contexte de méfiance mutuelle entre les applications, contrairement à ce qui est proposé par défaut par les exo-noyaux[Eng98, Section 2.2];

3. MVV : Machines Virtuelles Virtuelles

- **sécurité-innocuité.** Cette problématique est un objectif fondamental des systèmes d'exploitation. Le problème généralement rencontré est que le système d'exploitation impose un modèle de sécurité donné avec un grain (une taille minimale de données sécurisées) le plus souvent différent de ceux des différents langages de programmation. Le problème de la sécurité devient particulièrement sensible dans le contexte de l'interopérabilité des cartes où chaque application définit une politique de coopération spécifique vis-à-vis des d'autres applications, et où la méfiance mutuelle entre les applications est importante ;
- **efficacité d'exécution.** L'efficacité, propriété traditionnellement jugée essentielle pour évaluer les performances d'un système d'exploitation, n'est plus jugée aussi fondamentale ici. Elle reste néanmoins un critère important pour lequel des études ont apporté des éléments de réponse encourageants[PR98].

À partir de ces cinq critères de base, les articles [FPR97, FPR98] proposent une architecture globale.

2.3.2 Architecture globale

Une plate-forme regroupe toutes les fonctions propres à la notion de machine virtuelle virtuelle. La figure 2.3 présente les différents composants nécessaires à la réalisation des MVV. On y retrouve les éléments (abstraits) d'un système informatique classique :

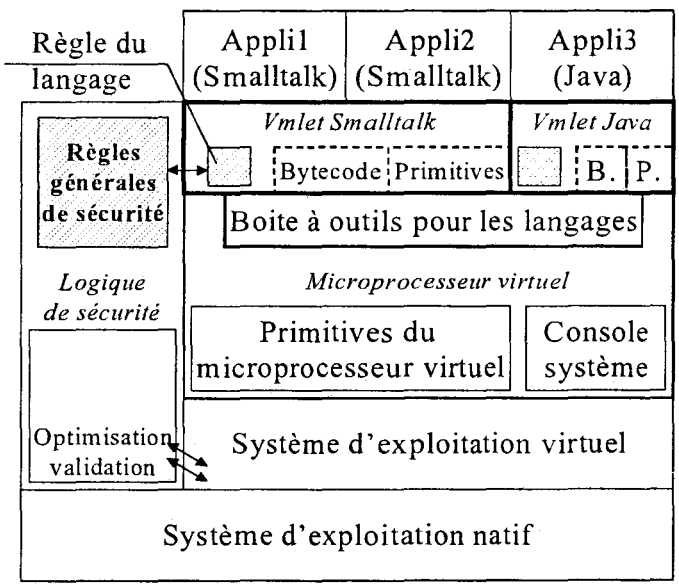


FIG. 2.3 – *Architecture d'une Machine Virtuelle Virtuelle*

1. **microprocesseur virtuel.** Il représente le cœur du moteur d'exécution de la plateforme. Il définit un jeu d'instructions élémentaires qui est une synthèse minimale

des jeux d'instructions des différentes machines virtuelles (Java, Smalltalk, Prolog, O-Caml,...). Cette machine à pile n'est cependant capable d'exécuter directement aucun des codes des machines traditionnelles. L'idée est plutôt de pouvoir étendre le jeu d'instructions de base en définissant de nouveaux codes par une séquence des codes de base de la machine ;

2. **système d'exploitation virtuel**. Il fournit une abstraction du système d'exploitation réel sur lequel se trouve installée la MVV. Il a la charge de gérer la protection des ressources et le partage du temps d'activité de chaque application prise en charge par le microprocesseur virtuel ainsi que de sa reconfiguration en fonction du code d'origine de chaque application ;
3. **boite-à-outils pour les langages**. Il s'agit d'un ensemble d'extensions de base pour le microprocesseur virtuel. Elles facilitent la tâche de l'introduction de nouveaux langages sur la plate-forme en définissant des fonctions de base souvent utiles (clonage d'objets, instanciation,héritage,...) ;
4. **VMlets**. Ces composants sont dynamiquement chargés par la MVV. Ils décrivent la sémantique d'un nouveau jeu d'instructions élémentaires. Lorsqu'une MVV est utilisée avec un nouveau type de code (défini par une machine virtuelle qu'elle ne connaît pas encore), elle reçoit une description de cette machine virtuelle sous la forme d'une VMlet. La VMlet définit le jeu d'instructions de l'application à partir de celui (plus élémentaire) du microprocesseur virtuel de la MVV ;
5. **module de sécurité**. Le module de sécurité agit en interaction avec le système d'exploitation virtuel. Il permet à chaque VMlet de définir la politique de sécurité spécifique au langage de chaque application. Les règles de sécurité décrivent la manière dont le microprocesseur virtuel de la plate-forme gère l'interprétation d'un code binaire ;
6. **applications**. Elles sont chargées dans la MVV sous la forme d'un code binaire auquel est associée une VMlet qui en définit la sémantique. Le chargement transforme le code de l'application en un code associé au microprocesseur virtuel (tel que le définit la VMlet). Une fois chargées, toutes les applications se trouvent donc exprimées dans le même code binaire. le moteur d'exécution est donc commun à toutes les applications. Les bénéfices (en termes d'interopérabilité et de performance) du concept de machine virtuelle universelle est donc rétabli.

À ce jour, plusieurs propositions d'implantation des MVVs ont été menées[PR98, BP99] ; et il existe maintenant différentes stratégies de mise en œuvre et des premiers résultats.

2.3.3 Premiers Résultats

Selon l'évaluation de la pertinence de chaque critère deux différentes approches sont d'ores et déjà évaluées.

Des travaux récents présentent le concept de MVV sous la forme d'une Machine Vir-

tuelle Récursive (par la suite notée MVR[4]). Il s'agit de donner une visibilité logicielle sur tous les aspects internes de la machine virtuelle (représentation du code, des méthodes, des objets, ...). Cette approche a montré sa viabilité dans le contexte des réseaux actifs. Elle permet de réaliser un environnement d'exécution dans un langage de haut niveau (en quelques centaines de lignes) tout en affichant des performances parfois supérieures à celles obtenues avec un compilateur C.

La seconde stratégies est de « compiler » une VMlet pour générer une machine virtuelle dédiée et performante: une MVA[5]. La figure 2.4 présente cette mise en œuvre. La MVA ne repose plus sur un cœur de machine virtuelle minimaliste, mais directement sur le jeu d'instructions du microprocesseur réel visé.

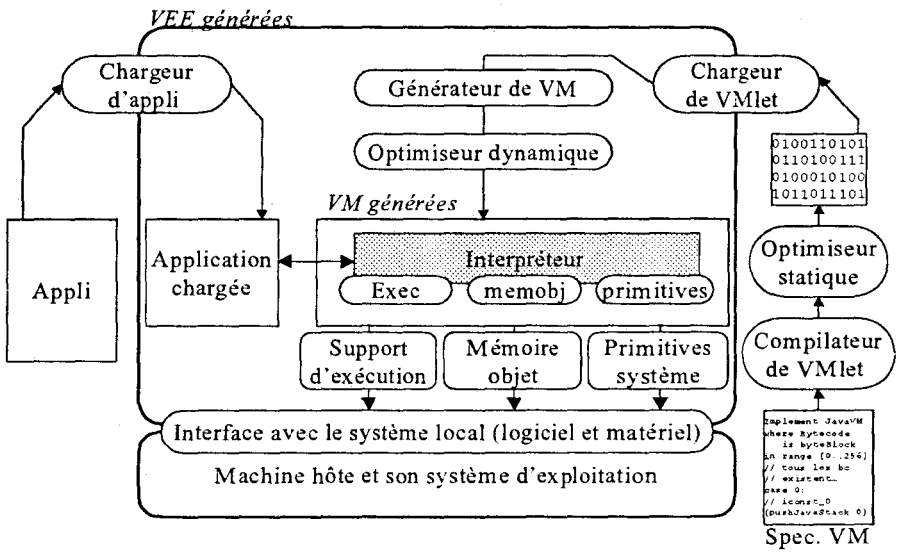


FIG. 2.4 – *Environnement d'Exécution Virtuel*

Les premiers résultats obtenus sont plus qu'encourageants, mais de nombreux travaux restent à faire. Ils s'étaleront probablement encore sur plusieurs années. Dans le contexte de la carte à microprocesseur, la démarche initiée par les MVV apporte une réponse bien fondée à bon nombre des problèmes contemporains du logiciel encartés (*c.f.* section 1.5.2 et 1.5.3). Deux aspects fondamentaux de la carte restent cependant non traités à ce jour: Il s'agit d'une part de l'impact des contraintes matérielles extrêmes de ce support embarqué – Ces contraintes (de mémoire et de performance du microprocesseur) considérées bien souvent comme négligeables font toute la spécificité des recherches de la carte; d'autre part, il est une question, cruciale dans le domaine de la carte, qui ne peut être laissée en suspends: Il s'agit de la sécurité-innocuité des programmes.

4. MVR: Machine Virtuelle Récursive.
5. MVA: Machine Virtuelle Adaptative.

2.4 Sécurité-innocuité des programmes

Dans les cartes dédiées à une application, la production du logiciel était réalisée avec un souci constant de garantir sa sécurité. Les problèmes de fiabilité du comportement des applications ont introduit, plus facilement qu'ailleurs, l'utilisation des méthodes formelles[Lan00] et des mécanismes de tests élaborés, qui sont devenus le savoir faire des industries de la carte. Il s'agit alors pour le logiciel de valider la fiabilité d'un comportement vis-à-vis des commandes transmises à la carte. On parle de test en boîte noire, car le logiciel et le matériel de la carte à puce sont vus comme un tout.

Cependant, l'avènement des cartes ouvertes, qui permettent le chargement de logiciel à tout moment, modifie profondément cette approche. Il devient impossible pour les constructeurs de carte de déployer leur arsenal de tests et de méthodes formelles pour valider la fiabilité des nouvelles applications. Ils n'ont d'ailleurs plus le monopole de cette production. Mais surtout, les applications encartées deviennent attaquables « de l'intérieur », c'est à dire par d'autres applications qui jouent le rôle d'un cheval de Troie. On parle alors d'innocuité des applications, les unes vis à vis des autres. Aussi devient-il essentiel que le système d'exploitation assure la continuité au niveau logiciel des efforts consacrés au niveau matériel (*c.f.* section 1.3.1) afin d'assurer la sécurité des applications encartées.

La gestion de la sécurité par les systèmes d'exploitation carte soulève des problèmes déjà bien connus dans le contexte de l'informatique traditionnelle. La sécurité-innocuité des programmes se décline en trois facettes distinctes :

- la **confidentialité**, qui garantit qu'il est impossible de connaître les données et les traitements d'une application sans son consentement ;
- l'**intégrité**, qui garantit l'inviolabilité des données et des traitements propres à une application ;
- la **disponibilité**, qui garantit l'accessibilité des données et des traitements à une application lorsqu'elle en a besoin.

Nous n'aborderons pas dans les lignes qui suivent les problèmes de sécurité en terme de disponibilité. La disponibilité ne concerne pas le propos de ce mémoire et elle est le plus souvent gérée par des mécanismes différents de ceux utilisés pour garantir la confidentialité et l'intégrité.

Dans le domaine de la confidentialité et de l'intégrité des programmes (que nous appelons abusivement problèmes de « sécurité-innocuité » des programmes) de nombreuses techniques de protection ont été mises au point. Afin d'en avoir une vue d'ensemble la section suivante propose une classification des modèles de sécurité.

2.4.1 Classification des modèles de sécurité

L'identification des différents modèles de sécurité nécessite le choix de de critères discriminants. Deux critères essentiels permettent de catégoriser les nombreux mécanismes de

protection et d'isolation des applications. Il s'agit de:

1. l'unité d'espace, la protection est alors associée aux:
   - espaces d'adressage mémoire,
   - structures de données;
2. l'unité de temps, la protection est alors associée au:
   - mécanisme d'exécution,
   - mécanisme de chargement.

Les mécanismes de sécurité qui s'appuient sur les adresses des données manipulées par les programmes sont dits **mécanismes de protections par adresse** (ou par adressage), alors que les mécanismes de sécurité qui reposent sur les déclarations des structures de données manipulées par les programmes sont dits **mécanismes de protection par les types** (ou par typage). De plus, les mécanismes de protection impliquant une activité (logicielle ou matérielle) spécifique durant l'exécution des programmes sont dits **mécanismes de protection dynamique**, alors que ceux qui s'appuient sur un traitement logiciel exécuté durant le chargement sont dits **mécanismes de protection statique**.

La recherche dans le domaine des mécanismes de sécurité-innocuité des traitements est particulièrement prolifique. Le tableau 2.1 présente les quatre familles de protections en fonction des deux critères précédemment identifiés. Pour chaque famille, des références bibliographiques significatives sont indiquées. Il faut préciser que ces critères ne sauraient définir des frontières rigides pour tous les mécanismes proposés à ce jour. La plupart des modèles statiques impliquent par exemple quelques points qui sont traités dynamiquement, et les capacités (associées dans ce tableau à un mécanisme par typage) ont été intégrées au mécanisme d'adressage dans certains microprocesseurs spécialisés. Cependant ces deux critères permettent une taxonomie globalement satisfaisante des différents mécanismes de protection. Notons encore que le support d'exécution du code (machine virtuelle ou microprocesseur) n'est pas un critère significatif. La plupart des techniques proposent des implantations destinées à des traitements exprimés par le biais d'un langage machine, et d'autres destinées à des traitements exprimés dans un langage intermédiaire.

Les sections suivantes détaillent les principes fondamentaux de ces différentes familles de solutions. Il s'agit aussi de préciser leurs mises en œuvre dans le contexte des cartes (pour celles qui y ont été implantées), en commençant par les plus répandues: les techniques de contrôle d'accès dynamique.

2.4.2 Contrôle d'accès dynamique

Cette technique consiste à définir des espaces mémoire associés à chaque programme et isolés les uns des autres: on dit aussi que les applications sont confinées dans des espaces distincts. Deux approches sont possibles. Soit (1) les programmes sont relogés dans des espaces d'adressage virtuels de la mémoire différents. Dans ce cas, quelle que soit l'adresse

Mécanismes de protection				
Unité d'espace de protection	Unité de temps de protection	Principales technologies	Références générales	Références cartes
par adressage	Statique	Isolation logicielle de défaillance	[WLAG93]	*néant*
	Dynamique	Confinement matériel	*Systèmes courants : Linux, Windows NT, ...*	[CCG95]
		Confinement logiciel	[Gon97]	[Mao98]
			[Sun00]	[Mic98]
par typage	Statique	Langage assembleur typé	[MWCG98a]	*néant*
			[GM99]	
			[MWCG98b]	
		Code comprenant sa preuve	[NL96]	*néant*
			[Nec97]	
		Bytecode JAVA	[LY96]	[RR98]
		Code intermédiaire commun typé	[Sha97]	[GLV99]
	Dynamique	Matériel à base de capacité	[JW75]	*néant*
			[Fab74]	
		Système à base de capacité	[SFS96]	*néant*
			[Lev84]	
			[HCC+98]	
		Capacité et systèmes distribués	[HHM97]	[HV00]
			[HI97]	

TAB. 2.1 – *Les différentes familles de protection selon leurs stratégies*

manipulée par un programme, elle ne peut référencer une zone de mémoire appartenant à un autre programme. Aussi est-il impossible pour un programme d'altérer des données associées à d'autres programmes. Cela implique que chaque adresse virtuelle utilisée soit convertie en une adresse physique lors de chaque accès à la mémoire par le programme. Soit (2) les espaces d'adressage des applications sont contigus, dans un seul espace d'adressage. Cela implique alors que lors de chaque accès à la mémoire un test de droit d'accès soit effectué afin de déterminer si le programme a le droit d'atteindre cette adresse. Seul le système d'exploitation est à même de modifier le droit d'accès à la mémoire ou le mécanisme de conversion de mémoire virtuelle. Si un programme transgresse ces règles, le système d'exploitation reçoit une interruption et décide de l'action à mener. Les systèmes Unix, par exemple, terminent le processus fautif en vidant le contenu de sa mémoire de travail dans un fichier « core ». La plupart des systèmes d'exploitation grand public, tel que Linux, Windows NT, BeOS, etc ...utilisent aujourd'hui ce mécanisme de protection pour isoler les différents processus et ainsi éviter qu'une défaillance de l'un entraîne une défaillance globale du système.

Ces techniques nécessitent que les contrôles d'accès soient effectués lors de l'exécution de chaque instruction d'accès à la mémoire de chaque processus. Certaines machines virtuelles implantent donc des instructions en réalisant systématiquement ces tests; une perte d'efficacité à l'exécution est alors inévitable. Pour la minimiser les techniques utilisées par les systèmes d'exploitation précédemment cités s'appuient le plus souvent sur des mécanismes de gestion de la mémoire matérielle : les MMU[6]. Dans le contexte de l'informatique grand public, les premiers microprocesseurs qui proposaient des uités de gstion de la mémoire adéquats à ce type d'usage sont les i80286 et MC68030. On distingue alors deux niveaux de privilèges : un niveau application[7] et un niveau système[8]. Lorsqu'un programme s'exécute avec les privilèges « systèmes » il peut accéder à toute la mémoire, mais lorsqu'il s'exécute avec les privilèges « utilisateurs », s'il accède à une zone de la mémoire qui ne lui est pas dédiée, une exception est déclenchée et l'opération de lecture ou d'écriture est abandonnée.

Qu'elles soient câblées ou implantées par des machines virtuelles, ces techniques de confinement soulèvent plusieurs problèmes :

1. en termes de performance d'accès à la mémoire. Tous les accès mémoires sont ralentis par les tests de limitation et le niveau d'indirection qui est requis entre la mémoire et le support d'exécution. L'utilisation de circuits câblés réduit considérablement ce coût ;
2. en termes d'accès aux routines système. Tous les accès aux routines système nécessitent une commutation de contexte. Il faut changer le niveau de privilège du code exécuté par la routine système, mais aussi le modèle d'indirection de la mémoire afin que le système puisse intervenir sur les zones de mémoire des applications ;

6. MMU : Memory Management Unit, ces mécanismes peuvent servir à accélérer matériellement la gestion des droits d'accès et/ou la gestion des mémoires virtuelles.

7. le niveau de privilège des applications est souvent nommé *User* dans les documents techniques.

8. le niveau de privilège du système est généralement appelé *Supervisor* ou *master* dans les documents techniques.

3. en termes de flexibilité du système. Les systèmes d'exploitation pour lesquels la sécurité-innocuité des traitements repose entièrement sur ces techniques de confinement ne peuvent pas permettre à des mécanismes apportés par les applications de s'exécuter avec les privilèges du système, faute de quoi une défaillance du module d'extension pourrait entraîner une défaillance globale de toutes les applications. Aussi ces « briques » apportées dans les systèmes d'exploitation extensibles restent confinées au niveau de privilège des applications, ce qui limite d'autant leur efficacité ;
4. en termes de performance des partages de données. Les applications qui souhaitent partager des données sont amenées à dupliquer cette information dans des zones mémoire dites « partagées ». Aucune garantie sur l'intégrité des données placées dans ces zones mémoire ne peut être apportée par le système. Il incombe donc à l'application de veiller à ce qu'aucun dysfonctionnement ne puisse être introduit par ce biais ;
5. en termes de granularité du partage de données. Les mécanismes de confinement segmentent la mémoire en pages de taille fixe. Chaque pages est associée à un jeu de droits d'accès de telle sorte que seulle l' (ou les) application(s) autorisée(s) puisse(nt) atteindre cette page. Aussi lorsqu'une application réserve ses pages, elle consomme plus de mémoire qu'elle n'en a réellement besoin, à cause des arrondis entre l'espace mémoire demandé par l'application et la taille de l'espace mémoire sécurisable par le système. De plus, les applications sont obligées d'allouer de nouvelles pages pour chaque donnée qu'elles souhaitent partager avec des droits d'accès différents vis-à-vis des autres applications.

Le confinement dynamique (aussi appelé technique des bacs à sable) ne peut généralement pas s'appuyer sur la présence d'une unité de gestion de la mémoire câblée dans des cartes. Les microprocesseurs aujourd'hui encartés n'en disposent pas. Des travaux de recherche sur le matériel spécifique aux cartes à microprocesseur [CCG95] ont abouti à la définition d'un matériel dédié. Toutefois cette technologie reste encore assez peu répandue. Malgré cela, le confinement est aujourd'hui la solution la plus généralement mise en oeuvre par les systèmes d'exploitation des cartes à microprocesseur pour se prémunir des défaillances (ou des attaques) des codes chargés dynamiquement. En l'absence de matériel dédié, c'est la machine virtuelle qui, en exécutant le code mobile, vérifie les pages mémoires accédées par le logiciel. Tel est le cas des machines virtuelles du système d'exploitation de Maosco: MULTOS [Mao98] et du système d'exploitation de Microsoft : *Smartcard for windows*[Mic98].

Il n'en reste pas moins vrai que ces techniques de protection sont dynamiques. Elles impliquent donc un surcroît d'activité lorsque les programmes s'exécutent. Et c'est là l'une des raisons pour lesquelles les machines virtuelles encartées sont peu efficaces (quelques milliers de bytecodes Java par seconde pour les machines virtuelles dites défensives). Les mécanismes statiques permettent normalement de reporter le coût de la sécurité-innocuité du système lors de la compilation ou durant le chargement des traitements. Les SFI[9]

9. *SFI : Software Fault Isolation*

sont une solution statique de confinement par adressage. On parle alors d'une technique d'isolation des défaillances logicielles.

2.4.3 Isolation des défaillances logicielles

Ce mécanisme s'appuie sur l'édition de liens effectuée lors de la création d'un processus. L'édition de liens reloge le code chargé en déplaçant les adresses des données manipulées pour qu'elles soient mises en correspondance avec la localisation réelle du programme (*c.f.* [Kra89, Section 6.3 « liaison des programmes et des données »]). L'idée consiste à compléter ce traitement afin de vérifier que la liaison de données est établie dans les limites des zones adressables par le traitement. Ainsi tous les contrôles qui ont pu être réalisés lors du chargement n'ont plus à être réitérés lors de l'exécution.

Pour réaliser ce contrôle le chargeur doit cependant effectuer un traitement bien plus compliqué que le simple acte de « liaison de code ». Il doit scruter le code natif qu'il charge, afin d'identifier les opérations qui accèdent à la mémoire. Les opérations d'accès à la mémoire sont classées en trois catégories :

1. les accès directs ;
2. les accès indirects décidables ;
3. les accès indirects indécidables.

Lorsque le chargeur rencontre un accès direct à la mémoire, il peut vérifier, une fois pour toutes lors du chargement, que l'adresse (explicitement définie) est dans l'espace d'adressage autorisé. Pour les accès indirects, la tâche est plus difficile. En fait il existe deux cas de figure. Soit l'accès indirect peut être statiquement ramené à un intervalle d'adresses ; c'est le cas par exemple lors de l'accès à des variables locales depuis le pointeur de pile, lors de l'accès à des structures de données globales ou système et aux constantes. Dans ce cas, le chargeur peut vérifier statiquement le code chargé en recherchant le point fixe des valeurs placées dans les registres d'adresse utilisés pour l'accès à la mémoire. Toutefois il est évident que dans le cas général, c'est impossible. Aussi lorsqu'il est impossible d'établir statiquement la preuve de l'innocuité d'un accès à la mémoire, le chargeur de code « injecte » dans le code chargé des opérations supplémentaires. Ces opérations vérifierons dynamiquement que l'adresse pointée est inscrite dans les bornes fixées avant l'exécution de l'accès à la mémoire.

Nous avons montré que cette approche n'apporte que partiellement les bénéfices escomptés par un mécanisme statique. Bien souvent, lorsque le programme est issu d'un compilateur non-dédié (du langage C par exemple) il est impossible de faire la preuve statiquement de l'innocuité du programme. Enfin, les SFI n'apportent aucune solution au problème de la granularité de la mémoire et se montrent absolument inadaptées pour gérer l'évolution des droits dans le temps sur des zones de mémoire partagées.

La recherche d'invariants sur les registres d'adresse est une opération bien trop coûteuse en temps et en espace pour pouvoir être implantée directement dans une carte. De plus,

au vue des difficultés rencontrées et des bénéfices discutables obtenus dans la communauté des systèmes d'exploitation, ce mécanisme statique n'a encore été envisagé dans aucune carte à microprocesseur. Finalement les carences en termes d'efficacité, de granularité et de flexibilité des mécanismes de confinement par adressage ont amené les recherches, dans le domaine des cartes ouvertes, à s'orienter vers des mécanismes de contrôle des types.

2.4.4 Contrôle des types

Alors que les mécanismes de protection par adresse s'intéressent aux zones de mémoire traitées par les programmes, les mécanismes de protection par typage s'intéressent à la nature des données qui sont manipulées par les programmes. Les types définissent des règles d'utilisation pour les variables auxquelles ils sont associés. Une variable du type *Entier* peut être utilisée pour effectuer des opérations arithmétiques et logiques. Une variable de type *TableauDEntier* peut être utilisée pour effectuer des opérations d'accès à un tableau. Ces accès donnent en retour des *Entier*s qui peuvent être rangés dans des variables de type *Entier* ... Un programme est correctement typé s'il ne transgresse aucune des règles d'utilisation des types. Ainsi un programme correctement typé n'effectue aucune opération d'accès à un tableau en utilisant des variables du type *Entier*. Contrairement aux mécanismes de confinement présentés précédemment le mécanisme de contrôle des types n'assure pas par lui-même l'innocuité des programmes. Un programme correctement typé pourra atteindre les données sensibles d'autres programmes. Pour qu'un système de typage puisse assurer la sécurité-innocuité il doit valider certaines conditions (assez intuitives) en plus les règles de type. Ces conditions visent à réduire les droits d'accès aux instances des différents types. Le système doit garantir trois propriétés fondamentales :

1. la production de nouvelles références est une activité fondamentale du système d'exploitation. La sécurité du système global dépend de l'intégrité des références manipulées. Il est par exemple écrit dans les spécifications du langage Java (qui utilise les types comme mécanisme de sécurité) : « il est illégal de forger des pointeurs » ;
2. la libération d'une instance ne doit être possible que s'il n'existe plus aucune référence vers cette instance. Si le système d'exploitation ne peut pas assumer cette contrainte, alors, par effet de bord, un programme peut disposer d'une référence d'un type sur un objet d'un autre type ;
3. la création d'une nouvelle instance ne se fait jamais sur un emplacement mémoire déjà alloué pour une autre instance. De la fiabilité des mécanismes d'allocation élémentaires de la mémoire dépendent l'intégrité et la sécurité des données.

La sécurité par le contrôle des types peut servir à garantir statiquement ces points en typant les méthodes système qui réalisent la création et la destruction de nouvelles instances. Forger de nouveaux pointeurs peut être impossible si aucun traitement système ne permet de réaliser des opérations dans ce sens (pas d'opérations arithmétiques et logiques sur une référence, garantie de l'intégrité des manipulations appliquées sur des références « nulle »). Cependant il ne faut pas oublier que la sécurité-innocuité des programmes

correctement typés repose sur la fiabilité des types fondamentaux définis par le noyau du système. Le contrôle des types est une manière de s'assurer statiquement que les règles de sécurité fixées par le système d'exploitation sont respectées. Rappelons encore que les classes dans les langages orientés objets sont une manière simple et efficace de rendre flexible et extensible la notion de type. Dans ce cas les programmes peuvent définir leurs propres types et les règles d'utilisation qui leur sont associées. Ainsi le contrôle de types garantit l'intégrité des données partagées entre applications. Le programme qui a obtenu la référence sur l'instance partagée ne pourra appliquer sur cette instance que les opérations décrites par sa classe (son type).

Le mécanisme de protection par les types est statique. Il consiste donc en un traitement effectué avant que le programme soit exécuté. Pour pouvoir vérifier qu'un programme manipule les données qu'il possède en accord avec les règles de types définies par le système, le mécanisme de contrôle des types doit disposer de l'information portant sur les types des variables (arguments, variables locales, ...) et des structures de données. Ainsi lorsqu'un programme utilise un argument, le contrôleur de type est à même de confirmer ou non que cette utilisation est correcte vis-à-vis du type de l'argument. Selon la nature du langage utilisé pour exprimer le programme, l'information de type est implicitement disponible ou pas. Lorsque le programme est exprimé à l'aide de bytecode Java par exemple, toute l'information de typage utile au contrôle est présente dans le code binaire[LY96] (les fichiers .class). Malheureusement, en règle générale les informations de type ne sont disponibles que dans le code source des programmes. Les compilateurs ne conservent pas cette information dans les codes objets qu'ils génèrent. Aussi des langages intermédiaires génériques ont-ils été définis de telle sorte qu'ils puissent préserver l'information de type. Des travaux particulièrement significatifs sur ce type de langage ont mené à la définition du langage FLINT[Sha97]. Ces solutions ne sont pas suffisantes pour permettre la vérification statique de code machine. Une équipe de l'université de Cornell a proposé l'introduction d'informations de type dans le langage assembleur TAL [10][GM99]. Mais cette approche n'était pas complète sans l'introduction des types dans la pile d'appel des traitements[MWCG98b]. En fait, toutes les approches statiques se présentent comme des instances d'un concept plus général : le code comprenant sa preuve ou PCC [11]. Cette idée s'appuie sur le constat que, dans les modèles de code mobile, la source du code (le compilateur et la machine qui sert la compilation) sont à même d'assurer des contraintes de sécurité que le receveur (la machine qui supporte l'exécution du code) ne peut plus assurer (statiquement). Afin de résoudre ce problème le code généré est enrichi avec la trace de sa preuve (exprimée dans la logique du premier ordre)[Nec97].

L'approche générique du code comprenant sa preuve soulève deux problèmes principaux.

D'une part, la complexité (algorithmique et spatiale) du processus de vérification est toujours élevée. Dans le cas général, celui de PCC, elle est liée à la complexité d'un moteur d'inférences dans la logique du premier ordre qui s'applique sur des axiomes reflètant les

10. TAL *Typed Assembly Language*: Langage assembleur typé.
11. PCC: *Proof Carrying Code*: Code comportant sa preuve.

contraintes de sécurité à valider. Pour que le moteur d'inférences ne soit pas amené à traiter une explosion combinatoire démesurée de règles, il s'appuie sur la trace de la preuve adjointe au code par le générateur du code. Mais, la taille de cette preuve est une fonction exponentielle de la taille du code qu'elle prouve. Ceci limite fortement l'utilisation de cette technique.

D'autre part, Il est évident qu'aucune approche statique ne saurait permettre à elle seule l'évolution des droits d'accès au cours de l'activité des traitements. Selon les besoins cette limitation de l'utilisation des contrôles de type peut être négligeable, mais en général ce n'est pas le cas.

La vérification statique de type peut être envisagée dans la carte, lors du chargement d'applications exprimées à l'aide de bytecode JAVACARD. Cependant les contraintes physiques de la carte rendent impossible l'inférence de type normalement réalisée par les machines virtuelles Java lors du chargement de nouvelles classes. La complexité algorithmique et la quantité de mémoire de travail nécessaire resteront, pour longtemps encore, un obstacle infranchissable. En conséquence Eva Rose propose dans ses travaux [RR98] une nouvelle instance de PCC dédiée à ce problème. L'idée est de simplifier la tâche du vérifieur encarté en ajoutant au code une preuve (« à la PCC ») du programme.

Les limitations du contrôle de type identifiées plus haut restent les mêmes dans la carte. Cette approche n'est pas adaptée pour la gestion de l'évolution des droits d'accès dans le temps. Pourtant, dans des modèles d'utilisation où la carte à microprocesseur est pensée comme un pivot de coopération entre des services mutuellement méfiants, il est évident qu'au cours de leurs activités les différentes applications seront amenées à faire évoluer leurs droits d'accès sur des objets (au sens large) qu'elles se partagent. Pour ne pas obliger le développeur à réaliser lui-même la contrepartie dynamique qui permet aux droits d'accès d'évoluer selon des critères de partage, d'autres mécanismes associés au langage sont nécessaires : les capacités.

2.4.5 Capacités

Dans un système reposant sur l'utilisation de cette technique, les valeurs sont distinguées des références qui sont appelées capacités. Une capacité définit à la fois une référence vers des données, mais aussi des droits d'accès associés à ce pointeur. Dans les premiers modèles les droits d'accès portaient sur le droit de lecture, le droit d'écriture et le droit d'exécution du bloc mémoire pointé par la référence de la capacité[WLH81]. Par la suite il a été étendu aux méthodes associées à une référence d'objet. Un prototype de SLK[12] a été réalisé a partir du langage Java[HCC+98]. Le principal intérêt de cette approche est de permettre aux programmes de s'échanger des références, et ainsi de partager des informations, tout en contrôlant, pour chaque échange, les droits accordés. Ce mécanisme permet un contrôle très fin des droits d'accès, aussi les capacités sont présentées comme la solution idéale pour gérer la sécurité de l'interopérabilité des applications. Leur utilisation

12. SLK : *Safe Language Kernel*, nom d'un noyau expérimental.

a d'ailleurs été étendue jusqu'au bus logiciel en permettant la coopération sécurisée de programmes distribués [HHM97].

Pour qu'un système de capacités soit viable, le système d'exploitation doit pouvoir assurer qu'une capacité n'est pas gérée comme une valeur. Pour ce faire, selon les disponibilités des mécanismes matériels et les stratégies de protection logicielles, plusieurs solutions d'implantation peuvent être envisagées. Certains systèmes utilisent des bits de tag devant chaque donnée de la mémoire pour distinguer explicitement celles qui représentent des références de celles qui représentent des valeurs. L'interprétation de chaque instruction est alors soumise à conditions en fonction de ce bit. D'autres systèmes utilisent des zones mémoires dissociées pour stocker les capacités ou les valeurs. Ainsi il est possible de bénéficier des mécanismes de contrôle d'accès matériels pour éviter qu'une capacité soit manipulée de la même manière qu'une valeur par les applications. Enfin, lorsque la stratégie de protection des programmes repose sur le contrôle de types, les capacités peuvent se présenter comme des types particuliers avec des règles de manipulations spécifiques. Tous ces mécanismes apparaissent comme des solutions de contrôle d'accès dynamiques. En fait, il est possible de réaliser des versions statiques des capacités lorsqu'elles s'appuient sur un système de sécurité à base de type. Elles se présentent alors comme des types qui définissent un sous-ensemble limité des méthodes associées aux instances d'un type partagé par le programme. Cette approche peut être réalisée dans le langage Java par l'utilisation de la notion d'interface. Dans ce cas, plutôt que de partager le type de base, l'application présente l'instance sous la forme d'une de ses interfaces qui ne déclare que les opérations qu'elle souhaite partager. L'avantage de cette approche est d'être particulièrement simple et « natuelle » pour le programmeur qui souhaite l'utiliser.

Il a été montré dans le passé que les systèmes à base de capacités à grain fin sont inefficaces lorsqu'ils s'appuient sur un mécanisme matériel de protection par page mémoire [WLH81, JW75]. En fait, le regain d'intérêt que les mécanismes de capacités ont soulevé dans la communauté scientifique spécialisée est principalement dû aux nouvelles possibilités apportées par la présence de mécanismes d'inférence de types performants au cœur du système d'exploitation, et sur les bus logiciels. Dans la pratique les capacités apparaissent comme un complément essentiel aux mécanismes de sécurité par les types. Et cette remarque est d'autant plus vraie lorsqu'elles permettent la création et la révocation dynamique de droits. Possibilité qui échappe totalement aux solutions statiques.

L'introduction des capacités dans la carte à microprocesseur est récente. Elle a dès l'origine été pressentie comme un support de sécurité de haut niveau [Gri99] apportant un complément dynamique à la protection par les types. L'objectif de ce complément est avant tout de fournir au programmeur un outil simple, robuste et puissant pour définir et gérer les partages que son application accorde aux autres. D. Hagimont et J.-J. Vandewalle ont proposé un tel système de base pour cartes[HV00].

2.5 Système de base pour cartes

L'ouverture de la carte à microprocesseur aux problématiques typiquement « systèmes » (gestion du matériel, abstraction du matériel et sécurité) est née avec la possibilité de charger à chaque instant de son cycle de vie de nouveaux logiciels. Cette évolution, passage obligé pour que la carte de demain s'intègre dans l'informatique omniprésente, soulève de nouveaux problèmes. Ils n'ont pas, pour la plupart trouvé de réponse satisfaisante dans les systèmes ouverts aujourd'hui disponibles sur le marché (*JavaCard* et *SmartCard for windows* par exemple). En fait, les cartes ouvertes conservent pour l'instant des architectures internes rigides. L'évolution des fonctions du système n'est possible qu'en changeant de carte. Les services offerts sont extrêmement limités et les contre-performances logicielles des machines virtuelles embarquées sont une source de pénalité intrinsèque pour le développement de nouvelles utilisations de la carte. Ces cartes restent des gadgets à la sécurité médiocre, souvent assurée par des solutions cryptographiques mal utilisées et dont les récents évenements[13] autour des cartes bancaires à puce nous montrent les limites.

La conception d'un système d'exploitation pour carte est assujettie à quelques règles fondamentales liées aux matériels embarqués :

1. **le code du système de base est le plus souvent placé en ROM.** L'avantage est évident aux vues des informations des différentes tailles de point mémoire (Section 1.3.2 page 15). Cependant ce type de mémoire n'est pas réinscriptible, le code du noyau est donc absolument immuable. Les mécanismes d'extension, pour outrepasser cette limitation, doivent inscrire les différents éléments du noyau dans un *framework* qui définit des points d'entrés pour des routines chargées hors de la ROM (en Mémoire Flash par exemple). Cette frontière entre la mémoire qui supporte le système et la mémoire qui supporte les autres logiciels n'introduit toutefois aucune pénalité d'appel transfrontalier. La protection du noyau est la conséquence directe d'un support physique ;
2. **la mémoire de code (autre que la ROM) est différente de la mémoire de donnée.** Cette distinction est fréquente dans les cartes. Les deux mémoires sont parfois adressées par deux bus différents selon un concept initié par les concepteurs des microprocesseurs RISC. C'est le cas notamment du microprocesseur AVR utilisé pour les expérimentations présentées par la suite. Dans la carte, les raisons de cette distinction sont toutes autres. La mémoire dédiée au code n'est normalement pas aussi fréquemment sollicitée en écriture que la mémoire des données. Or il est possible de réaliser des circuits plus compacts (sur le silicium) mais moins « réinscriptibles ». Ils peuvent être utilisés avantageusement pour le stockage de programmes chargés dynamiquement ;
3. **la mémoire persistante n'est pas un disque dur.** Cette affirmation est évidente en ce qui concerne les capacités des deux supports. Cependant la distinction principale concerne le fait qu'une mémoire Flash ou EEPROM est accèdée en lecture

13. La clef privée de carte bancaire française à été « cassée » par une personne déterminée mais seule et avec des moyens et des connaissances accessibles au grand public.

à la même vitesse que la RAM. Les seules pénalités concerne l'écriture. D'un point de vue fonctionnel la mémoire persistante peut donc être utilisé pour le stockage d'informations utilisées par les programmes. La RAM n'est utile que pour accélérer l'utilisation de données de traitement fréquemment modifiées (cette remarque est recevable autant pour la conception du système que pour celle des applications) ;

4. **l'activité du microprocesseur peut être interrompue à tout moment par son utilisateur**. Les normes ISO prévoient qu'une carte peut être arrachée (déconnectée) à tout instant. Cette action a normalement une sémantique implicite : « l'opération en cours doit être annulée ». Après une déconnection la carte doit être capable d'effectuer une reprise sur panne. Cette considération qui peut être masquée aux concepteurs des applications par des gestionnaires de mémoire recouvrables[DGL98] pénalise les performances du système[LGD99]. Tout système d'exploitation pour une carte doit être conçu pour être extrêmement tolérant à ces défaillances et pour assurer la reprise sur panne et son idempotence. Cette contrainte pénalise généralement la performance des algorithmes système;
5. **le matériel encarté est généralement pauvre**. Le raisonnement tenu par les fondeurs est que l'unité de contrôle est capable de réaliser logiciellement les tâches de base pour un moindre coût (en surface de silicium) que le circuit câblé. En conséquence le système d'exploitation a une partie logicielles dédiée à la manipulation du matériel, plus importante (toute proportion gardée) que celle des systèmes classique. Le matériel ne propose le plus souvent que de très médiocres commodités câblées utiles aux systèmes (pas de mémoire virtuelle, pas ou peu d'interruption, ...).

Ces éléments constituent les fondements de toutes réflexions système adapté au contexte spécifique de la carte à microprocesseur. C'est en les gardant à l'esprit qu'il est possible être bâtis les fondements d'un système de cartes ouvert.

Chapitre 3

Architecture nouvelle pour carte ouverte

« C'est le langage qui noue les choses. »

Pilote de guerre, A. de Saint-Exupéry

Les sections 1.5.3 et 2.5 de la première partie ont identifié un certain nombre de verrous qui cantonnent l'utilisation de la carte à des domaines d'applications isolés. Les cartes de demain ont pourtant un rôle prépondérant à jouer dans toutes les formes de l'informatique distribuée et omniprésente. Malheureusement elles s'avèrent d'un usage trop limité dans la pratique. Leurs faibles performances lors de l'exécution de traitements les confinent dans des tâches où elles ont une activité très limitée et ponctuelle. L'inflexibilité de leurs systèmes d'exploitation leur interdit d'embarquer des applications exprimées dans des langages hétérogènes. De plus la faible interopérabilité des applications (même lorsqu'elles sont exprimées dans le même langage) limitent leur utilisation en temps que médiateur personnalisé de confiance entre différentes applications.

Ce chapitre décrit l'architecture globale que nous avons retenue pour deployer dans et autour de la carte les composants logiciels apportant une réponse cohérente aux différents problèmes qui aujourd'hui la privent d'un usage universel. Les objectifs de base qui ont motivés la conception de notre architecture, dénommée *Camille*, sont détaillés dans la première section. Les éléments clefs de la stratégie retenue pour atteindre ces objectifs sont ensuite discutés. L'architecture générale qui a finalement été adoptée est alors présentée dans la troisième section. Deux aspects de cette architecture sont davantage détaillés dans les sections suivantes. Il s'agit de la chaîne de compilation/chargement des logiciels encartés et du cœur du système placé dans la carte. Le noyau encarté constitue le socle de sécurité (dans un contexte d'interopérabilité), d'efficacité et de flexibilité choisi. La conclusion de ce chapitre annonce deux des aspects de l'architecture qui ont débouché sur des résultats particulièrement innovants et montre en quoi nous apportons une solution aux besoins identifiés (ce qui constitue finalement notre motivation).

3.1 Motivations

Aujourd'hui les cartes ouvertes restent d'un usage limité. Elles se montrent incapables d'apporter une réponse satisfaisante aux ambitions en termes d'interopérabilité, de flexibilité et d'efficacité présentées dans la première partie. La grande hétérogénéité des formats de code des différentes applications, leur sécurité et leur interopérabilité les rendent trop complexes pour être toutes gérées dans une même carte. Les verrous technologiques (présentés section 1.5.3 page 26) constituent un frein majeur à l'utilisation des cartes dans un contexte ouvert. Les travaux présentés par la suite sont une première proposition pour que la carte puisse relever les défis à venir. Les critères essentiels qui ont motivé la conception de cette nouvelle architecture logicielle pour les cartes à microprocesseur sont ceux identifiés comme des verrous technologiques du code encarté à la fin du premier chapitre. Trois d'entre eux ont tout particulièrement été retenus. Il s'agit de l'efficacité, de l'interopérabilité et tout d'abord de la flexibilité.

3.1.1 Flexibilité

Les cartes ouvertes ont une capacité d'adaptation très limitée. Il faut distinguer deux aspects dans les problèmes de flexibilité des systèmes pour cartes ouvertes : l'hétérogénéité des langages supportés et l'abstraction que le système donne du matériel.

Les cartes ouvertes vendues furent, à l'origine, envisagées comme des supports d'exécution universels. Nous constatons aujourd'hui que cet objectif n'est que partiellement atteint. La figure 3.1 montre l'évolution réelle qu'a engendrée l'introduction de machine virtuelle. Avant l'introduction des cartes ouvertes, les applications encartées étaient strictement dédiées à un support physique (premier cadre) d'exécution. Différents systèmes d'exploitations pour cartes ont eu pour ambition de proposer un support d'exécution générique (*c.f.* cadre numéroté 3 de la figure 3.1). Leurs systèmes d'exploitation devait prendre en charge d'une part les différents matériels possibles, mais aussi, d'autre part, toutes les applications envisageables, quelles que soient leurs origines et leurs rôles. Le but était de permettre de placer à la demande toutes sortes d'applications sur une carte « universelle ».

Malheureusement, les machines virtuelles n'ont rempli que partiellement cet objectif. Les cartes Java n'acceptent que les applications exprimées avec un bytecode JavaCard, les cartes MULTOS n'acceptent que le code MEL, etc. On se retrouve dans la situation présentée par le cadre n°2, figure 3.1. Deux applications exprimées dans deux langages différents ne peuvent pas être embarquées dans la même carte. Cela est dû, principalement, au fait que ces langages reposent sur des abstractions incompatibles du matériel. Chacune d'entre elles repose sur la définition d'une machine virtuelle spécifique.

Il est évident que, dans ce contexte, la carte ne peut pas être présentée comme le point de liaison entre différentes applications. En conséquence il devient impossible de supporter deux programmes qui présupposent des supports d'exécution différents au sein d'une seule et même carte. Un programme MEL ne peut pas être introduit dans une JavaCard par exemple. Or l'un des bénéfices de l'introduction d'une machine virtuelle

(escompté notamment par des recherches précédentes [Van97]) devait être l'universalité du support d'exécution. Perdre ce bénéfice retire aujourd'hui une bonne partie de son sens à cette démarche. Le seul avantage restant concerne la portabilité des programmes destinés à des cartes qui utilisent des microprocesseurs variées.

Cette catégorie de problèmes n'est pas propre au milieu de la carte à puce. Nous avons présenté dans le second chapitre, section 2.3 page 36, une analyse qui part de ce même constat dans le cadre plus conventionnel de l'informatique classique. Les travaux présentés ici se distinguent par le fait qu'ils portent sur un environnement d'exécution particulièrement contraint. Ces contraintes ne permettent pas de transposer littéralement l'approche des Machines Virtuelles Virtuelles. C'est en partie pourquoi une réflexion particulière a été entreprise ici.

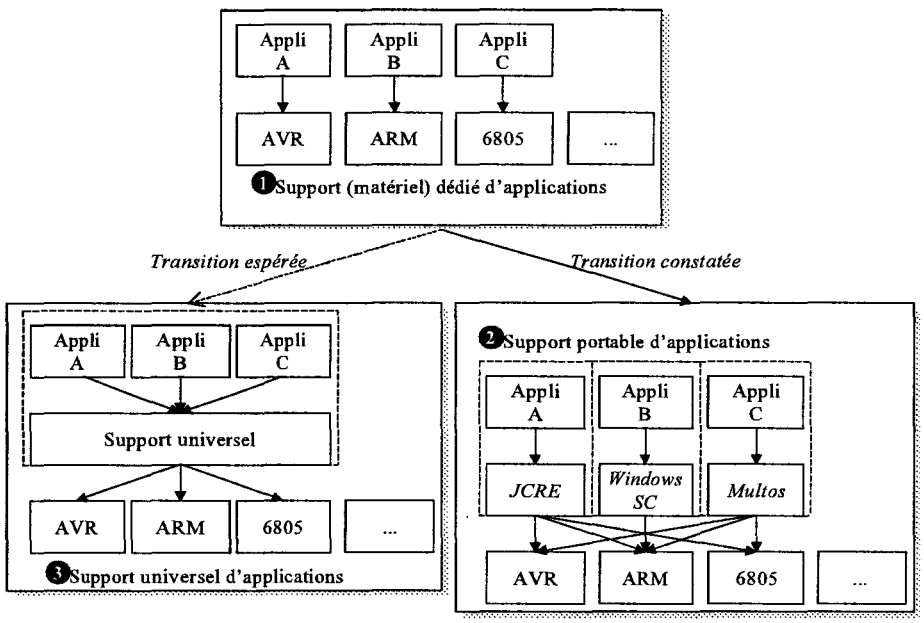


FIG. 3.1 – *Évolutions espérées et évolutions constatées de la programmation des cartes*

Cependant, les dissemblances entres les différents systèmes d'exploitation ne se limitent pas au support d'exécution (au sens de machine virtuelle) des applications. Les structures de base des systèmes encartés s'avèrent dans la pratique très hétérogènes. Alors que l'immense majorité des systèmes d'exploitation grand public proposent par exemple des systèmes de fichiers analogues – il n'est pas difficile de faire fonctionner sur un système Windows un programme qui, à l'origine, été destiné à manipuler des fichier Unix – on ne trouve qu'une gamme restreinte de cartes ouvertes qui gèrent des fichiers. Et lorsque tel est le cas, la manière de les utiliser est tellement différente d'un cas à l'autre que les applications qui seraient conçues pour un système ne pourraient que très difficilement être portées sur un autre. Les fichiers au sens de la norme ISO7816-4 ne fonctionnent pas de la

même manière que les fichier d'une *SmartCard for Windows*. Ils ont des modèles de nommage incompatibles, des structures d'organisation (notion de répertoires, contrôle d'accès, modalité de lectures,...) différentes.

En fait les cartes ouvertes que l'on peut aujourd'hui acquérir dans le commerce, fournissent une abstraction du matériel qui leur est spécifique mais aussi indispensable. Envisager d'utiliser la gestion du matériel proposée par un système pour réaliser (sous la forme d'une bibliothèque par exemple) d'autres abstractions n'est absolument pas envisageable. Les cartes *SmartCard for Windows* par exemple représentent la mémoire persistante au travers d'un système de gestion de fichiers. Pour écrire des informations persistantes il faut donc écrire dans un fichier. Cependant un système de fichiers est particulièrement contre-performant pour héberger des objets persistants (les objets des cartes Java par exemple) ou une base de données (à la manière des cartes CQL). En conséquence, même si un effort était entrepris pour permettre de charger des objet Java dans une carte où la mémoire est structurée avec un système de fichiers, les résultats seraient décevants.

Le problème tient principalement au fait que pour réduire la taille du système d'exploitation, un choix est fait quant à l'organisation précise de la mémoire (et des autres ressources matérielles). Les concepteurs des cartes actuelles ont cherché le « plus court chemin » entre l'abstraction fournie par une machine virtuelle et le matériel des cartes. Cette démarche, héritée des habitudes acquises dans le passé (*c.f.* section 2.1.1 page 29), est imposée par les contraintes du matériel. Néanmoins elle nuit à la vocation universelle de cet objet grand public. Finalement ce qui est le plus dommageable à cette seconde forme de flexibilité ce sont les contre-performances flagrantes des systèmes encartés. Pour pouvoir proposer différentes bibliothèques qui offrent des abstractions variées du matériel il faut pouvoir exprimer des traitements efficaces directement liés aux ressources physiques manipulées. Il faut pouvoir construire un système de base de données ou un système de fichier, par exemple, à partir de la ressource physique « mémoire persistante » plutôt qu'à partir de toute autre abstraction fonctionnelle de ce support. C'est pourquoi les bibliothèques de base du système sont proches du matériel. Elles doivent permettre à de nouvelles extensions de s'implanter sur ce que le matériel permet de faire alors qu'elles sont prisonnières d'un système d'exploitation trop directif.

Le manque de flexibilité des systèmes d'exploitation actuels est aussi lié à un autre critère essentiel : les routines d'extension du système sont des primitives extrêmement sollicitées par les applications. En conséquence, elles doivent pouvoir être exécutées avec efficacité.

3.1.2 Efficacité

Les machines virtuelles encartées se montrent particulièrement inefficaces. Cette remarque constitue un frein majeur au développement de bibliothèques système dans le cadre des JavaCard. Construire une base de données « au dessus » d'une machine virtuelle JavaCard par exemple, peut permettre de valider un concept [JDL00], mais la base ne peut

pas véritablement être envisagée pour un usage professionnel car il n'est pas possible d'en fournir une version performante.

Il n'est pas simple de rendre l'exécution d'un programme encarté performant car il est vrai que le microprocesseur (réel) des cartes est généralement peu performant lui-même. Les machines virtuelles destinées aux cartes n'implantent néanmoins pas les optimisations utilisées sur des plates-formes plus conventionnelles. Les techniques contemporaines d'expansion de code virtuel[1] ou de compilation à la volée de langages intermédiaires tels que le bytecode Java sont en fait trop complexes pour être réalisées dans le contexte de la carte à microprocesseur. La contrainte principale est la quantité réduite de mémoire de travail disponible. En conséquence, les traitements que le système d'exploitation effectue sur le code chargé doivent se présenter sous la forme d'un traitement de flot. Le code est chargé, décodé et stocké, bloc par bloc, instruction par instruction. Tout parcours non séquentiel du code est à proscrire. Or, l'inférence des types (sur un langage intermédiaire classique) ou la génération de code optimisé ne sont pas, normalement, pensées comme des algorithmes de traitements de flots.

La performance des machines virtuelles est pourtant une clef pour l'utilisation de cet outil dans le contexte de l'informatique omniprésente (*c.f.* chapitre 1, section 1.5.1). Si jusqu'à présent la performance d'exécution des codes chargés est restée médiocre c'est que les problèmes d'efficacité y ont trouvé des solutions ponctuelles. Les cartes bancaires par exemple utilisent des micromodules qui comportent un circuit câblé. Ce circuit réalise des opérations spécifiques (principalement cryptographiques) que le logiciel ne peut supporter efficacement. Néanmoins les applications dans lesquelles l'usage de la carte à microprocesseur ne parvient pas à se développer sont celles où elle pourrait jouer un rôle véritablement actif. La carte de demain, pour être un objet utilisé universellement, doit être active. Malheureusement, les programmeurs sont immédiatement découragés lorsqu'ils envisagent d'y placer des codes critiques qui doivent être exécutés efficacement. Aussi les efforts déployés pour rendre ce support de mobilité utilisable par tous restent vains.

Il faut néanmoins replacer le rôle du critère d'efficacité dans le contexte de la carte. Les performances sont, dans le domaine des systèmes d'exploitation, le principal instrument d'évaluation des résultats obtenus. Les recherches entreprises autour des machines virtuelles virtuelles montrent d'ailleurs que d'autres critères moins quantifiables (tels que la qualité de l'environnement d'exécution pour de développement de logiciels, le potentiel d'interopérabilité des applications...) ont probablement été trop négligés jusqu'à présent.

Dans la carte comme ailleurs, on constate que quatre-vingt pourcent du temps machine est consommé par vingt pourcent du code machine. S'il est nécessaire pour certaines tâches qu'un code encarté s'exécute de manière extrêmement performante (*e.g.* traitement numérique par flots, recherche d'informations dans une structure de données complexe, composants système pour gérer la mémoire de manière recouvrable, ...) cela n'a pas de sens dans le cas général. La taille du code encarté est le plus souvent un critère plus important que la vitesse à laquelle il est évalué. Il doit donc être possible, lorsqu'on introduit

1. généralement connues sous le nom de *direct-threading*.

un nouveau programme dans une carte à puce, de définir quels sont les traitements qui doivent être efficaces et quels sont ceux (majoritaires) pour lesquels ce n'est pas essentiel.

Les propriétés de flexibilité et d'efficacité ont pour but de permettre aux cartes de recevoir au cours de leur cycle de vie de nouvelles applications. Cependant il ne faut pas oublier que cet objectif a pour but de présenter la carte comme un support de sécurité et de coopération entre les applications. Pour cela il faut assurer, aussi, leur interopérabilité.

3.1.3 Interopérabilité

L'interopérabilité influence l'architecture logicielle des cartes à microprocesseur déjà existantes. Les cartes peuvent se connecter à un parc de terminaux généralement très important. De nouvelles cartes, comportant un matériel différent et de nouveaux logiciels sont émises régulièrement. Pourtant l'interopérabilité (que je nomme parfois interopérabilité externe) entre ces deux composants matériel (la carte et le terminal) est restée parfaite jusqu'à ce jour.

Ce sont les normes ISO qui ont été les garantes de cette compatibilité, et par ce biais de l'interopérabilité, entre différentes cartes produites. Rappelons qu'elles définissent :

- les éléments technologiques et logiques de la communication entre la carte est le terminal (position des contacts, porteuse du signal, protocole de communication T=...) [ISO88, ISO89] ;
- les formats des commandes que le terminal peut transmettre à la carte. Leurs signification et leur fonctionnement sont spécifiés par les normes ISO [ISO94] ;
- les modèles de gestion de la mémoire persistante. La structuration en termes de fichiers et de lignes de base de données sont fixés par les références [ISO94, ISO95, ISO96].

Toutes ces spécifications décrivent néanmoins une abstraction unique et rigide. En conséquence les constructeurs sont amenés petit à petit à enrichir les fonctions de base définies par ces normes pour répondre aux exigences croissantes de leurs clients. Les garanties d'interopérabilité des cartes apportées par le système sont alors remises en question. Cette normalisation interdit finalement une interopérabilité extensive des systèmes encartés qui sont, soit cantonnés à des utilisations trop restrictives, soit privés de l'interopérabilité avec les systèmes existants. Finalement ce qui remet en cause l'interopérabilité aujourd'hui existante, c'est une normalisation trop rigide et abstraite, qui enferme l'utilisation de la carte dans des scénarios réducteurs.

Pourtant l'interopérabilité est un élément primordial pour que la carte puisse être perçue comme le représentant électronique universel de son porteur, où les applications encartées peuvent coopérer « au nom » d'un individu, dans un contexte de sécurité. Pour cela ces logiciels doivent pouvoir s'appuyer sur des bases compatibles. Cela définit un nouveau besoin d'interopérabilité (que j'appelle interopérabilité interne) localisée entre les programmes

chargés dans la carte. Cette forme d'interopérabilité doit être dissociée d'un langage particulier pour être véritablement utilisable par toutes les applications. Elle doit donc être pensé comme partie intégrante de l'architecture présentée ici. Des interfaces (de programmation) spécialisées doivent pouvoir être définies dans les langages de haut niveau pour permettre aux concepteurs des applications de les utiliser.

En ce sens, le critère d'interopérabilité des applications encartées influence aussi la conception du système d'exploitation ouvert. Le mécanisme de sécurité adopté doit permettre simplement la coopération entre des applications. Il doit pouvoir être utilisé comme base de confiance pour que les abstractions de chaque application puissent l'utiliser pour implanter leur propre raffinement. L'objectif d'universalité de la carte de demain implique que ces coopérations se fassent en règle générale dans un contexte de méfiance mutuelle – contrairement aux applications de nos stations de travail qui, lorsqu'elles collaborent se font une confiance aveugle[2]. Les applications encartées contiennent pour l'essentiel des informations personnelles dont le détail est confidentiel. Dans le cas contraire le surcoût induit par l'utilisation d'une carte comme support de stockage incite le programmeur à placer l'information ailleurs. Dans ce cas la carte est abandonnée au profit d'autres supports de gestion de la mobilité de l'utilisateur avec une autre stratégie.

3.2 Stratégie

En fait les motivations présentées dans la section précédente rejoignent celles des autres projets entrepris en système. Les exo-noyaux et les MVV, dont les ambitions ont été présentées dans le chapitre 2 abordent plus particulièrement ce type de problématique. Cependant les contraintes propres à la carte à microprocesseur imposent en règle générale d'adopter une démarche particulière et d'imaginer des solutions pour qu'il soit possible d'offrir le même niveau de fonctionnalité dans la carte que dans l'informatique usuelle. Cette démarche pourrait se résumer en une simple phrase : « Déporter hors de la carte tout ce qui peut l'être sans nuire aux fonctions de la carte ! ».

Deux axes distincts sont finalement le moteur des travaux présentés dans ce mémoire : (1) l'hétérogénéité des logiciels coopérants qui sont encartés et (2) l'efficacité du noyau devant impérativement être localisé dans la carte. Pour parvenir à répondre à ces exigences, il est nécessaire de replacer la carte dans son contexte de déploiement. Une carte en activité est nécessairement connectée à un terminal, (ordinairement beaucoup plus puissant qu'elle) et, éventuellement, à travers lui à des serveurs distants. L'architecture proposée doit utiliser au mieux les différents supports physiques qui sont disponibles et sur lesquels il est possible de répartir les composants du système carte.

2. Constat notamment repris par les architectures en exo-noyau « *Optimize for the common case: Mutual trust* » [Eng98, section 2.2.1 *Protected sharing* page 23].

3.2.1 Répartir les composants du système carte

L'ambition de l'architecture présentée ici est de permettre à une seule carte (et donc à une seule plate-forme d'exécution encartée) d'accueillir différentes abstractions selon les applications qu'elle dessert. Le système d'exploitation se présente comme la liaison entre le matériel et les applications.

En conséquence il possède deux visages.

Le premier donne accès aux ressources matérielles minimales qui sont disponibles dans la carte. Il est intimement lié à la carte et ne peut donc pas être placé ailleurs.

Le second visage est polymorphe afin de supporter toutes les formes d'échanges que la carte peut être amenée à établir avec le monde extérieur. Il s'agit par exemple d'un mécanisme de connections interactives avec les programmeurs. Ce sont alors des outils du système tels qu'une console (qui joue le même rôle qu'un *Shell* Unix par exemple) que de connections réseaux ou encore d'échange de codes applicatifs qui permettent à la carte d'héberger et d'exécuter des codes issus de différents langages intermédiaires.

Ces deux aspects du système peuvent être distribués entre la carte et le terminal qui l'héberge. Selon leur nature les outils systèmes sont placés à l'intérieur où à l'extérieur de la carte. Dans le chapitre 1, nous avons vu que le terminal héberge déjà des logiciels non-applicatifs tels que l'Objet d'Adaptation de la Carte (figure 1.4 page 21) qui gère la représentation abstraite de la carte sur un bus Corba. La stratégie retenue par la suite sera de placer dans le terminal une véritable infrastructure capable de supporter les différents outils nécessaires à l'utilisation de la carte par le monde extérieur. Dans la carte ne figure plus alors qu'un noyau minimal de gestion du matériel et le mécanisme de sécurité élémentaire qui garantit l'intégrité de l'utilisation des ressources. Ce sont ces éléments qui constituent le noyau encarté.

3.2.2 Le noyau encarté

Le noyau de base est vu comme un support pour toutes sortes d'extensions associées aux différents langages. Deux possibilités sont envisageables pour le définir.

Les MVV définissent une micro-machine de base qui est la « souche » à partir de laquelle seront définies toutes les autres machines virtuelles. Il est en effet possible de trouver un ensemble de points communs à toutes les machines virtuelles utilisées (*e.g.* pile d'exécution, mémoire d'objet[3], primitive de saut et d'appel, ...). Cette souche pourrait être vue comme le noyau minimum à encarter. Elle assurerait alors le support d'applications issues de différents langages.

D'un autre coté, les Exo-noyaux proposent de fournir une interface qui donne une représentation directement associée au matériel (*c.f.* section 2.2 page 32). Dans le contexte

3. mémoire segmentée en unités de taille variables mais disposant toutes d'une marque qui définit le type de l'objet. On parle de *boxed units* par opposition à *unboxed units* lorsqu'il est impossible de retrouver des informations sur la nature d'un bloc mémoire à partir de sa seule référence.

de la carte cette interface permettrait d'allouer et d'utiliser des pages de mémoires Flash par exemple. L'avantage de cette approche est de permettre l'implantation dynamique de composants système plus performants car ils sont directement basés sur les possibilités élémentaires du matériel. La contrepartie est que l'implantation d'abstractions pour des langages de haut niveau sur cette base est une tâche plus coûteuse.

La stratégie retenue pour la conception du système de base encarté est, pour l'essentiel, celle d'un micro-noyau organisé autour d'une fonction principale : le chargement de code. Ce micro-noyau fournie une image fidèle mais synthétique du matériel. Plusieurs raisons justifient ce choix. Trois sont prépondérantes :

- le but du noyau encarté est de permettre à des bibliothèques chargées dynamiquement de proposer toutes les abstractions possibles du matériel. Pour ne pas limiter les abstractions qui peuvent être imaginées, ni nuire à leur implantation efficace dans la carte, il nous est apparu judicieux de concevoir le noyau du système encarté de telle sorte qu'il présente des interfaces de programmation les plus proches possibles de ce qu'offre physiquement le matériel. Ainsi les contraintes du matériel sont les seules à être répercutées sur les programmes qui en proposent des abstractions ;
- les principales ressources matérielles sont en fait fonctionnellement très proches d'une carte à l'autre. La stratégie d'allocation des supports physiques de mémoire persistante se résume à une stratégie basée sur l'estimation de l'usure des pages en fonction du nombre d'écritures qu'elles ont subit (une page de mémoire Flash ou EEPROM perd sa capacité de réécriture au fur et à mesure de son utilisation). La stratégie d'allocation des pages (à l'unité) est donc dépendante du matériel, ce sont les abstractions logicielles qui utilisent, chacune selon leurs besoins, les pages fournies par le matériel. Les procédures système qui prennent en charge ce type de gestion du matériel constituent véritablement un niveau élémentaire commun à tous les systèmes encartés. En définissant une vue synthétique du matériel, notre micro-noyau définit une base minimale, mais portable d'une carte à l'autre ;
- différentes recherches se sont proposées d'intégrer langages et systèmes. Nous avons retenu pour probante celle qui portait sur Oberon [Fra93] et qui est finalement la plus proche de notre démarche. Les résultats obtenus, bien que anciens, étaient encourageants.

Les autres propriétés que le noyau encarté doit apporter (interopérabilité et sécurité-innocuité des programmes) nécessitent une stratégie particulière. L'inférence de type utilisée en tant qu'outil de sécurité apporte des avantages en termes d'efficacité (le contrôle est fait au chargement) d'interopérabilité (le mécanisme définit précisément l'utilisation des données) et en termes de flexibilité (en utilisant des systèmes de type extensibles comme les systèmes de classes).

Le noyau ainsi réalisé doit assurer l'efficacité d'exécution des traitements. L'autre aspect, à savoir l'hétérogénéité des langages, est apparu trop complexe pour être entièrement traité par la carte. Notre stratégie a donc été de déplacer hors de la carte la gestion de ce

problème. Dans la carte le système reflète simplement le matériel disponible. Notons tout de même que le Système d'Exploitation de Base de la carte peut avantageusement inclure des routines qui donnent une sémantique par défaut (initiale) un peu plus riche que celle du matériel qui est extrêmement pauvre. Néanmoins le support des pseudo-codes associés aux différentes machines virtuelles est placé hors de la carte. L'idée est de suivre un processus en trois points :

1. le pseudo-code issu de la compilation d'un programme dans un langage intermédiaire particulier est rapatriée sur le terminal de chargement pour carte ;
2. ce terminal traite le pseudo-code et produit un code équivalent dans un autre langage intermédiaire qui est le seul à être véritablement accepté par les cartes ;
3. le terminal transmet à la carte le code produit ainsi que des bibliothèques d'extension qui implantent à partir des ressources du noyau encarté les abstractions utilisées par le pseudo-code original.

De cette approche distribuée des composant qui composent la plate-forme d'exécution est née le terme de « cartes exo-virtuelles ».

3.2.3 Carte exo-virtuelle

Les contraintes propres au contexte des cartes à microprocesseur, telles qu'elles ont été identifiées au début de ce document, sont trop réductrices pour permettre au logiciel encarté de prendre seul en charge le support de différents codes intermédiaires utilisés par les applications. De plus, dans la perspective d'une informatique omniprésente, elles doivent être capables de charger des logiciels exprimés avec des langages postérieurs à leurs mise en circulation.

Il convient alors de rappeler que la carte n'est jamais utilisée seule. Les terminaux sur lesquels elle se déploie ont une puissance de calcul bien supérieure. La base de notre approche et d'exploiter la puissance de ces machines. Notre stratégie suppose que la plate-forme dans laquelle est insérée la carte lorsqu'elle va recevoir de nouveaux traitements est suffisamment performante pour exécuter des programmes de traduction automatique des codes destinés aux cartes.

Ces traducteurs (que nous appellerons par la suite adapteurs) visent les mêmes objectifs que les VMlets(*c.f.* section 2.3 page 36). Ils assurent que des codes intermédiaires différents puissent être chargés sur un support d'exécution identique. Cependant les différences évidentes apparaissent très rapidement. Les VMlets ont pour objet d'être des descriptions (quasi-formelles) des machines virtuelles destinées à être exploitées par un support générique, alors que les adapteurs sont des programmes spécialisés qui analysent et transforment les logiciels pour qu'ils puissent être encartés. La carte recevra alors tous les logiciels sous la même forme, exprimés dans un langage commun, quelle que soit leur origine (Java, Visual Basic, ...).

Les adapteurs de code réalisent hors de la carte une part importante du traitement des différents langages intermédiaires. Sur l'exemple du bytecode Java, c'est l'adapteur qui

décode la structure d'un fichier « `.class` », c'est lui qui analyse les séquences d'instructions élémentaires qui composent la structure d'une méthode, et c'est encore lui qui produit une version encartable de ces programmes.

Toutefois le noyau de base du système encarté ne propose pas forcément d'équivalent pour des opérations particulières à un langage. La sémantique des tableaux définie par le langage Java par exemple ne correspond à aucun élément du matériel présenté par le noyau encarté. En conséquence l'adapteur de code inclut avec le code des applications des bibliothèques d'abstraction qui prennent en charge les besoins applicatifs inconnus de la carte. Il peut s'agir aussi bien d'enrichir le noyau avec un système de fichiers exotique (celui des cartes *SmartCard for Windows* est par exemple totalement différent de celui spécifié par les normes en usage [ISO94]) que d'une gestion de la mémoire qui apporte des propriétés particulières (comme des propriétés transactionnelles) ou encore d'une manière de gérer les tableaux, ...

Finalement l'organisation des cartes, que nous qualifions d'exo-virtuelle, consiste à déporter hors de la carte tous les aspects syntaxiques du support d'un nouveau type de langage intermédiaire (format des fichiers, codage des opérations, descriptions des structures, ...), et à placer dans la carte tous les aspects sémantiques du langage (gestion des types « naturels » du langage intermédiaire, intégrations des propriétés non fonctionnelles du langage, telles que les mécanismes d'atomicité, de cache d'accès à la mémoire, ...).

Il faut encore, pour que notre stratégie couvre complètement la répartition des composants du système, déterminer la place de la sécurité.

3.2.4 La place de la sécurité

Nous nous faisons l'écho d'Eva Rose lorsqu'elle constate que la sécurité intrinsèque d'une carte ne peut pas raisonnablement reposer sur des éléments qui lui sont extérieurs [RR98]. La plupart des solutions actuelles proposent de sécuriser le logiciel chargé dynamiquement dans la carte par le biais d'une signature numérique. Les producteurs d'un code destiné à être encarté doivent recevoir un certificat (signature digitale) d'un tiers de confiance. Cette signature est vérifiée par la carte qui accepte alors d'exécuter le code (aveuglément). Cette procédure a l'avantage d'être effectuée une fois pour toutes au chargement et de ne pas pénaliser outre mesure l'exécution du code avec des vérifications de confidentialité ou d'intégrité (qui seraient opérées dynamiquement, en exécutant le code). Cependant elle a l'inconvénient de placer la sécurité des cartes sur des systèmes informatiques extérieurs. Dans le contexte d'une informatique omniprésente, la carte ne pourra se présenter comme un outil de confiance que si elle parvient à se protéger elle-même. En fait, la sécurité des cartes à puce devrait être pensée comme une propriété intrinsèque de leurs composants, tant matériels que logiciels.

Le choix de la stratégie de sécurité a été motivé par deux critères essentiels : l'**efficacité** et la **granularité**. Dans le cas le plus fréquent, le matériel encarté ne propose pas de support particulier pour contrôler les droits d'accès à des informations. Or les techniques

logicielles de contrôle lors de l'exécution pénalisent lourdement l'efficacité d'exécution des programmes. Les techniques statiques, appliquées une fois pour toutes lorsque la carte reçoit le code de l'extérieur, semblent alors plus profitables que les techniques dynamiques (*c.f.* tableau 2.1 page 42 dans la section 2.4: *sécurité-innocuité des programmes*). Les deux approches possibles, dans la carte à microprocesseur sont les techniques statiques basées sur les adresses et les techniques basées sur les types. On constate que les techniques à base de types sont plus adaptées à l'interopérabilité et qu'elles ont un grain plus fin [HE98]. Grâce à cette approche il est en effet possible d'isoler les groupements d'information, non par rapport à une zone mémoire, mais par rapport à leur signification. Il est ainsi possible de définir les opérations correctes sur un type d'information, plutôt que se contenter d'un contrôle des droits d'accès en termes de lecture ou l'écriture.

Pour illustrer ce propos, prenons l'exemple d'un porte-monnaie électronique. Un système de types permet à cette application de spécifier la manière précise de réaliser un débit (enregistrement d'un historique, plafonner la somme retirée, ...). Le système de types garantit que l'objet partagé par l'application porte-monnaie électronique ne peut être utilisé autrement que par le biais des méthodes qui sont définies. Limiter le mécanisme de sécurité encarté à des contrôle de droit d'accès en lecture et en écriture rendrait la réalisation d'application interopérables plus complexe et faillible.

En conséquence nous avons choisi d'assurer la sécurité de la carte par l'intermédiaire d'un moteur d'inférence de types **au cœur** du système encarté. Cependant ce choix soulève des difficultés d'implantation délicates à dépasser. Pour apporter une solution satisfaisante à ce problème il faut que langage intermédiaire utilisé pour transmettre toutes les formes de traitements à une carte facilite la vérification qui est faite de son usage. Ce doit être l'une des propriétés d'un langage dédié aux cartes.

3.2.5 Propriétés d'un langage dédié aux cartes

Nous avons dégagé l'idée, en établissant notre stratégie, qu'un langage intermédiaire dédié doit permettre d'échanger toutes les formes de traitements entre la carte et le monde extérieur. C'est ce langage qui sert de cible unique aux différents mécanismes d'adaptation. C'est aussi ce langage qui doit permettre de supporter dans la carte un mécanisme d'inférence de types, et enfin c'est lui encore qui doit pouvoir assurer un haut niveau d'efficacité lorsque cela est souhaitable.

Les langages qui sont actuellement utilisés pour transmettre des traitements aux cartes ne peuvent répondre à aucun de ces trois critères. Le bytecode JavaCard est trop complexe pour que les faibles ressources d'une carte lui permettent de valider l'usage des types. Toutes les machines virtuelles encartées se montrent actuellement contre-performantes – nous avons constaté une moyenne de 15000 bytecodes Java exécutés chaque seconde si aucune opération d'écriture en mémoire persistante n'est sollicitée. Les autres pseudo-codes ne se montrent pas plus performants. C'est pourquoi les langages déjà utilisés aujourd'hui ont rapidement été écartés.

Certains langages intermédiaires ont été conçus comme des cibles potentielles de plusieurs sources. Nous nous sommes donc intéressés à des stratégies plus ouvertes. Nous avons notamment retenu le langage intermédiaire commun appelé FLINT [Sha97], qui se propose d'identifier le fondement des langages intermédiaires pour en proposer une abstraction générique et réutilisable. Cela correspond à notre stratégie. Néanmoins FLINT est défini en tant que support pour des langages de haut niveau (code générique, polymorphisme d'ordre supérieur,...), mais surtout il ne prend pas en compte les contraintes d'un support minimaliste comme la carte à puce. En conséquence nous avons décidé de définir un langage intermédiaire dédié à nos besoins.

Dans le couple carte terminal, la carte a généralement un désavantage évident en terme de puissance. Le langage doit prendre en compte ce déséquilibre. Pour cela nous proposons quelques principes fondamentaux:

- le langage intermédiaire doit permettre de déporter tous les aspects complexes de son traitement (analyse, optimisation, ...) hors de la carte. Il existe bien évidement un ensemble de techniques d'optimisation que l'on peut appliquer au niveau du langage intermédiaire (suppression de code mort, réutilisation de résultats partiels, ...). Cependant certaines de ces techniques, qui portent plus particulièrement sur la forme du code final généré, ne peuvent pas être appliquées sur les langages intermédiaires conventionnels. L'idée est alors de construire le code intermédiaire de telle sorte que ce type d'optimisation (basé sur la durée d'utilisation des variables, l'optimisation de l'usage des variables qui sont plus efficacement utilisées dans le cœur des boucles, ...) puisse être établi hors de la carte et exploité dans la carte;
- il doit permettre d'exprimer, de façon homogène, des traitements issus de langages très variés. L'idée est de maintenir un format binaire simple et générique pour toutes les opérations qui composent un traitement. Rappelons que si certaines seront fournies par le noyau du système d'autres proviendront d'une bibliothèque d'extension ajoutée au code applicatif par l'adapteur. Le but est finalement d'uniformiser le codage des opérations du langage intermédiaire, qu'elles soient définies par le noyau ou par des extensions;
- il doit être simple à décoder et fournir à la carte toutes les informations dont elle a besoin, au moment ou elle en a besoin. L'idéal serait de permettre à la carte de traiter le chargement d'un nouveau programme comme un flot, en une seule passe, pas par pas, sans avoir besoin de revenir sur le résultat d'un pas précédent. Les mécanismes d'inférence et d'optimisation ne permettent pas, normalement, de traiter un programme de cette façon. Des solutions doivent être trouvées;
- enfin il doit permettre à la carte de produire un code efficace ou compact selon les besoins. Cette propriété impose que le langage intermédiaire soit traité par la carte avant d'être stocké en mémoire. L'idée est alors d'envisager le chargement d'un programme comme étant la dernière étape d'une chaîne de compilation. Des langages tels que Juice reposent sur une démarche similaire [Fra97].

Finalement ce langage intermédiaire assure la liaison entre la carte et le terminal. Il est

placé au cœur de notre architecture générale.

3.3 Architecture générale

Pour respecter les éléments établis par notre stratégie, l'architecture globale qui a été retenue se découpe en deux grands ensembles : les composants encartés et les composants hors de la carte. Ces deux ensembles sont liés entre eux grâce à l'utilisation d'un langage intermédiaire typé et dédié à la carte à microprocesseur : FACADE. Le transport des commandes et des applications exprimés par ce langage est assuré par un échange d'APDU qui respectent les normes établies. Les réponses (résultat de l'exécution, détection d'erreurs lors du chargement, ...) se présentent sous la forme de données sérialisées obtenues en retour. Le plan général de notre architecture est présenté par la figure 3.2. Les outils qui fournissent une abstraction cohérente et utilisable de la carte sont placés sur le terminal hôte. Quatre outils système y figurent. Ce sont :

1. un module de chargement de code qui est utilisé pour transmettre tous les traitements et toutes les commandes à la carte ;
2. une console d'administration (sorte de console de commandes système qui permet de piloter manuellement la carte) ;
3. un module de conversion des requêtes provenant des bus logiciels et destinés à la carte ;
4. un support d'adaptateur qui est destiné à transformer le code des applications programmées dans des langages hétérogènes dans un format homogène transmissible au chargeur de code et donc « compris » par la carte.

Sur la carte trois modules logiciels constituent le système d'exploitation :

1. une hiérarchie de types donne un modèle, une représentation des ressources matérielles gérées (microprocesseur, page de mémoire et entrées-sorties) ;
2. un module de traitement (chargement, contrôle de sécurité et stockage prêt à l'exécution) des codes exprimés avec le langage FACADE tels qu'ils sont transmis à la carte ;
3. un support pour les différentes applications et les extensions système dont elles ont besoin.

Finalement seul un sous-ensemble minimal des outils nécessaires pour gérer la carte constitue véritablement le logiciel encarté. Le reste est placé hors de la carte. Il n'est pas nécessaire de déployer tous les composants du système distribués hors de la carte sur tous les terminaux. Lorsqu'une carte est utilisée pour accomplir une opération élémentaire, un retrait bancaire par exemple, le guichet automatique n'a pas besoin de posséder des composants système qui sont en fait des outils de développement, tels que, par exemple, le terminal d'administration pour carte.

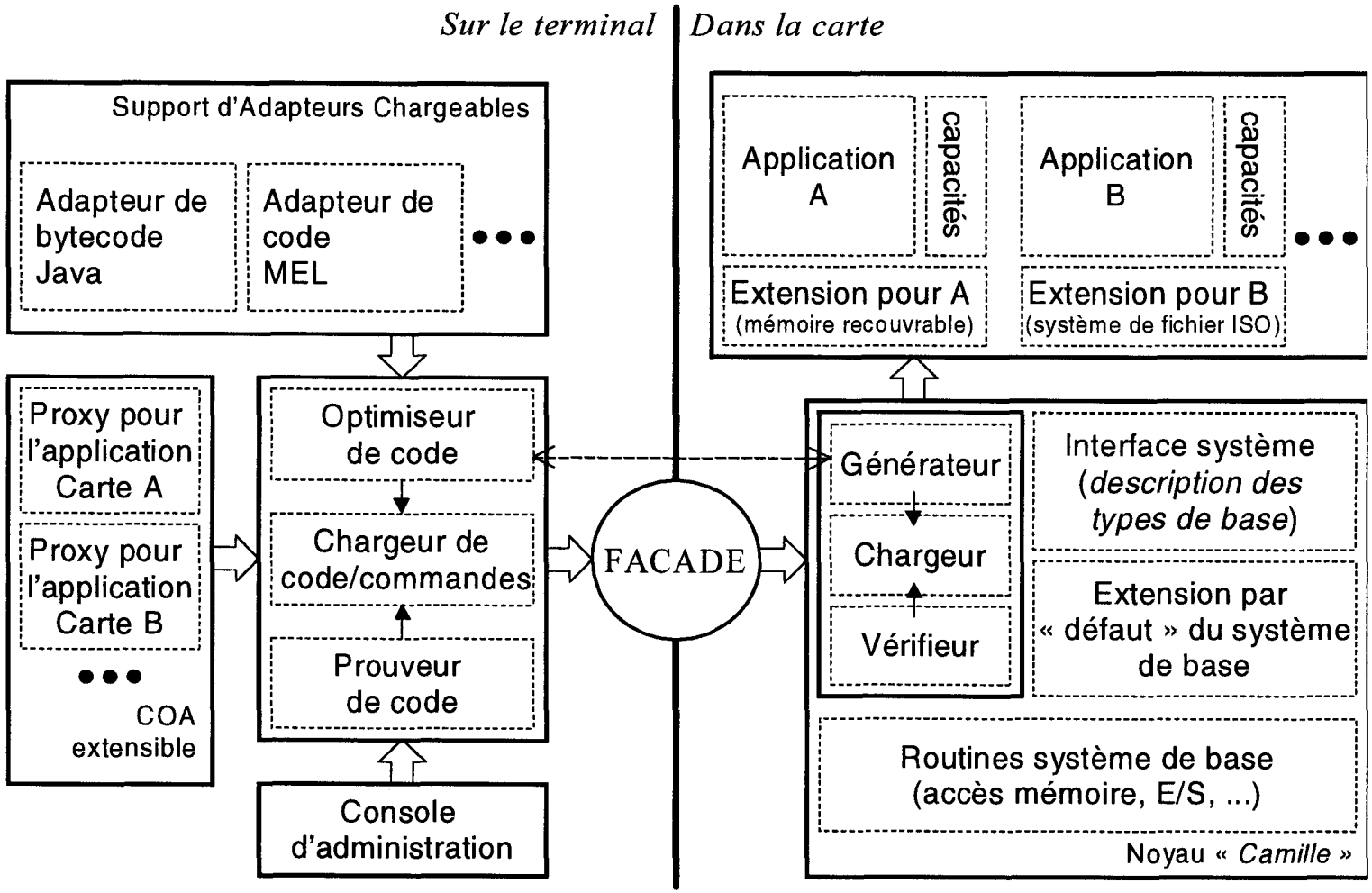


FIG. 3.2 – *Architecture globale de déploiement des composants de Camille*

3.3.1 Terminal d'Administration pour Carte

Le TAC[4] est une console d'administration essentiellement destinée à être utilisée par les concepteurs d'applications. Il permet de transmettre de manière interactive des requêtes à la carte. Ces requêtes peuvent servir à modifier la configuration de la carte ouverte. Elles peuvent aussi permettre au programmeur de tester les programmes chargés *in situ*, et donc de faciliter la détection des erreurs de programmation. Cette console s'appuie directement sur le langage FACADE en proposant une représentation facilement utilisable par le programmeur : le langage FACADEScript. De plus, parce que le logiciel de base du système d'exploitation offre une représentation utilisable de tous ses composants logiciels, la console d'administration peut être utilisée pour manipuler l'état de la carte (aussi longtemps que les droits d'accès de l'administrateur le permettent).

Le TAC permet d'agir interactivement sur le système encarté. Cependant les applications distribuées sont elles aussi amenées à inter-agir avec les logiciels encartés (surtout dans la perspective d'une carte support de l'interopérabilité des applications omniprésentes). Pour cela un autre composant système doit être placé hors de la carte, il s'agit d'un support pour les Objets d'Adaptations pour Carte.

3.3.2 Objets d'Adaptations pour Carte

Ce support de connexion réseau est destiné à permettre l'interopérabilité externe. En d'autres termes, il doit permettre la coopération entre les programmes chargés dans la carte et ceux distribués sur un réseau par-delà le terminal. Pour répondre à ce type de besoin les applications traditionnelles utilisent des modèles à objets distribués [OMG97]. Un composant d'adaptation nommé COA (Card Object Adapter) a été défini pour jouer ce rôle sur l'architecture CORBA avec des programmes encartés. Il représente les fonctions de la carte sur un bus CORBA puis transforme les requêtes reçues afin de les transmettre à la carte connectée [Van97].

La définition du COA doit être revue dans la perspective des cartes ouvertes. En effet une carte ouverte peut contenir diverses applications possédant des descriptions fonctionnelles (des interfaces) propres. Lorsqu'une carte ouverte est connectée à un terminal, des interfaces associées aux programmes contenus dans la carte doivent être déclarées disponibles par le terminal. Ainsi les programmes qu'elle contient peuvent être utilisés par d'autres logiciels distribués sur un réseau. Cette technique ne soulève aucune difficulté qui n'ait été résolue par des travaux antérieurs temps que plusieurs applications ne peuvent pas être sollicitées simultanément. Dans le cas contraire la tâche des programmes qui représentent les applications encartées hors de la carte se complique. C'est pourquoi un composant spécialisé doit prendre en charge ces nouveaux problèmes.

Cette démarche suppose que différentes applications aient pu être chargées à n'importe quel moment du cycle de vie de la carte. Pour cela nous avons proposé une solution générique pour charger toutes sortes de formats de codes dans une carte. Il s'agit d'un

4. TAC : Terminal d'Administration de Camille.

autre composant de l'infrastructure placé hors de la carte et qui est nommé le support d'adaptateurs de codes.

3.3.3 Support d'Adaptateurs de Codes

Nous avons vu que le langage FACADE servait d'intermédiaire entre les langages hétérogènes utilisés par les programmeurs et le support d'exécution unique de la carte ouverte. Pour assurer « *virtuellement* » le support de l'hétérogénéité des langages, le logiciel des plates-formes susceptibles de recevoir des cartes utilise des modules destinés à transformer le code mobile en code FACADE. La section 3.1.1 reprenait le constat de [FPR98] : les machines virtuelles se sont montrées incapables de fournir une plate-forme universelle d'exécution du code mobile.

Pour supporter toutes les variantes des machines virtuelles susceptibles d'être utilisées, nous avons défini le SAC[5]. Lorsqu'une nouvelle application est chargée dans la carte son type (la machine virtuelle susceptible de l'exécuter) est déterminé, puis le module de traduction convertit le code en traitement FACADE qui peut alors être délivré à la carte.

L'opération de conversion est une étape complexe qui est, en tant que telle, déportée hors de la carte. Lorsqu'un client souhaite acquérir un nouveau service, il consulte le prestataire et introduit sa carte dans un terminal de chargement qui lui transmet le logiciel adéquat. Les modules de conversion qui composent une partie du support de chargement des applications sont des composants logiciels eux-mêmes téléchargables. Ainsi lorsqu'un nouveau format de code doit être traduit pour être chargé dans la carte, l'adapteur de code nécessaire peut être téléchargé par le terminal.

Bien que ces modules jouent finalement le même rôle que les VMlets de l'architecture des MVV des différences apparaissent dans leur mise en œuvre. Alors que les VMLets se présentent sous la forme d'une structure de données qui décrit le jeu d'instructions d'une machine virtuelle donnée à partir du bytecode primitif de la MVV, les adapteurs de FACADE se présentent comme des programmes qui convertissent un langage intermédiaire en un autre. L'avantage des adapteurs est de pouvoir traiter des cas de conversion difficiles (*e.g.* langages intermédiaires génériques d'ordre supérieur [Sha97], langages intermédiaires à base d'arbres décorés [KF99],...). Cependant les VMlets, de par leur vocation descriptive peuvent être utilisées dans d'autres mécanismes que celui d'extension d'un support d'exécution générique. Tel est le cas par exemple du compilateur de VMLets [BP99] qui les utilisent pour automatiser la génération de nouvelles machines virtuelles (*c.f.* figure 2.4 page 39). Cette opération est bien évidement impossible à partir d'un adaptateur qui se présente comme un logiciel et non comme une description formelle exprimée dans un langage dédié.

Une fois que l'adapteur de code a terminé sa tâche, le code à charger dans la carte est exprimé en langage intermédiaire FACADE. Néanmoins, avant d'être chargé, il doit encore subir des traitements trop complexes pour être exécutés dans la carte. Il s'agit d'optimiser

5. SAC : Support d'Adapteurs de Codes.

le code et de simplifier la tâche de la preuve de programme. Ceci est réalisé par le chargeur de code FACADE.

3.3.4 Chargeur de code FACADE

Le langage intermédiaire FACADE sert à transmettre des programmes (des applications) ou simplement des commandes à exécuter immédiatement par la carte (par exemple un script d'administration). Hors de la carte on trouve (comme pour toutes les cartes ouvertes déjà existantes) un chargeur de code. Dans le contexte de l'architecture Camille, le chargeur de code se trouve augmenté de deux fonctions annexes, **optimiseur de code** et **prouveur de code**:

1. La tâche de l'optimiseur est de traiter le code intermédiaire avant sa transmission afin que les optimisations possibles à ce niveau soient réalisées. En tant que langage intermédiaire, FACADE permet d'appliquer un certain nombre de techniques d'optimisation au nombre desquelles on compte l'identification de code mort, la réutilisation d'expressions partielles déjà calculées, la propagation de copie, l'optimisation de boucles,... En résumé, l'utilisation d'un langage intermédiaire permet l'implantation de différentes techniques d'optimisation [ASU86, chapitre 10, page 585]. Les optimisations appliquées au code à charger peuvent exploiter des informations demandées à la carte au préalable. Dans ce cas, le programme d'optimisation hors carte s'informe de la stratégie de génération du code dans la carte (*e.g.* utilisation de registres pour masquer les n premières variables temporaires utilisées par le code, insertion d'instructions qui guident la gestion d'un cache d'écriture pour l'EEPROM,...);
2. le rôle du prouveur est de contrôler que le code est correct (vis-à-vis des types manipulés) et de proposer à la carte une preuve de l'innocuité du code. La vérification de la preuve est plus simple que sa génération, mais cette vérification permet de contrôler dans la carte que le programme est inoffensif. Cette technique est une adaptation à la carte des travaux proposés par Georges C. Necula [Nec97]. Elle a aussi été utilisée dans le contexte de la carte par les travaux d'Eva Rose pour proposer un vérifieur de bytecode Java léger[6] [RR98];
3. enfin la tâche du chargeur est de transmettre le code binaire FACADE à la carte. Il réalise donc les mêmes opérations que les chargeurs des cartes ouvertes du commerce, à savoir un découpage du code binaire en plusieurs paquets transmissibles, un par un, à la carte.

Finalement l'activité du module de chargement hors de la carte est de servir le code destiné à la carte. Ce code est exprimé dans un langage intermédiaire dédié à la carte: FACADE.

6. traduction du titre *Lightweight Bytecode Verifier*.

3.3.5 FACADE

FACADE est un langage typé utilisé comme support de communication pour transmettre des traitements depuis l'hôte vers la carte. Ces traitements peuvent être des bibliothèques système étendues, des applications standards, ou simplement des requêtes qui doivent être exécutées une seule fois dans la carte (immédiatement après leur chargement).

De manière informelle FACADE est à rapprocher du Langage Intermédiaire FLINT [Sha97]. Comme le montre la figure 3.3, nous proposons de redistribuer l'architecture FLINT entre la carte et ses hôtes potentiels. FACADE est défini comme un langage typé afin de respecter la stratégie présentée section 3.2. Les types ont pour objectif de permettre la vérification statique de certaines propriétés élémentaires de sécurité (comme cela est le cas de langages modernes tels que Java). Ainsi la sécurité est vérifiée lors du chargement et ne pénalise pas davantage l'efficacité d'exécution des traitements chargés. De plus, comme pour le langage FLINT, les traitements exprimés avec le langage FACADE peuvent faire l'objet de différentes optimisations classiques.

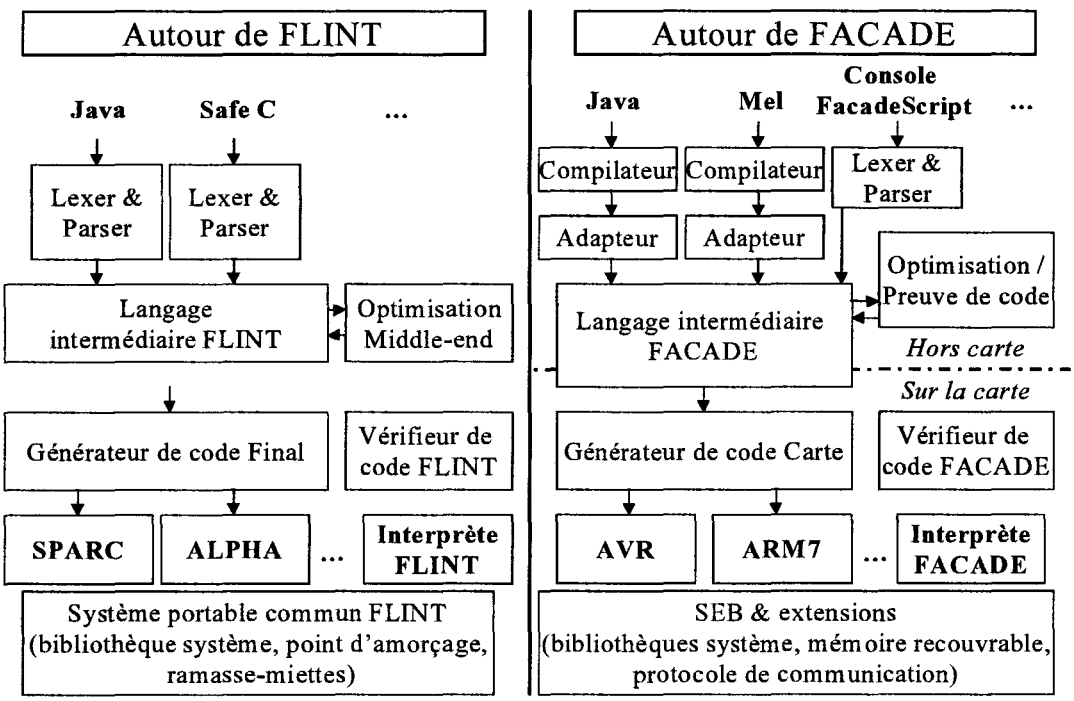


FIG. 3.3 – *Comparaison entre FLINT et FACADE*

Toutefois les comparaisons entre FACADE et FLINT s'arrêtent à la Figure 3.3. Un code FACADE est destiné à être reçu par le noyau encarté. Pour assurer le support de différents matériels encartables, il doit être *portable*. Mais contrairement à FLINT, il reste très modeste en tant que langage. Il ne supporte ni la généricité de code (*i.e.* fonction d'ordre supérieur) ni aucun système de typage polymorphique avancé. En fait la seule forme de

polymorphisme exprimée dans le langage FACADE est celle induite par l'utilisation d'une hiérarchie de classes qui s'appuie sur un modèle à héritage simple. L'introduction de programmes exprimés dans des langages de très haut niveau, tels que ceux de la génération des ML [7], présuppose donc un traitement hors de la carte, par un adaptateur de code dédié. Cet adaptateur instancie le traitement à charger en fonction des appels génériques finalement utilisés, hors de la carte, car ces techniques sont beaucoup trop consommatrices de mémoire pour être encartées et nécessitent donc d'être traitées hors du système d'exploitation de base.

Nous avons décidé de retenir une approche orienté objet pour spécifier les éléments du noyau encarté. FACADE est donc aussi un langage intermédiaire orienté objet. Les types ont donc un certain degrés de flexibilité (d'extensibilité) car se sont des classes. Cependant toutes les classes du système encarté, (les types) ne reposent pas sur une convention d'appel de méthode virtuelle. Pour le détail, notre démarche est fortement inspirée de réflexions sur l'utilisation des objets dans la conception des systèmes d'exploitation encartés [Gri97]. Dans les grandes lignes le principe est de pouvoir utiliser des conventions d'appels moins riches que l'invocation de méthode lorsque cela accroît l'efficacité du code généré. Finalement on peut comparer cette démarche à la manière dont sont implantés les appels vers des méthodes non virtuelles du langage C++, ou même pour les cas les plus critiques utilisant des mécanismes d'implantation en ligne (tels que la directive de compilation `inline` du C++).

Notons encore qu'un code FACADE décrit simplement **un** traitement (et pas une classe par exemple). Il est défini par la déclaration d'un certain nombre de variables utilisées par une séquence d'instructions élémentaires. La déclaration d'une nouvelle classe est réalisée en exécutant un traitement qui sollicite des opérations définies par le noyau pour créer une nouvelle classe, définir de nouveaux attributs, ajouter ou surcharger des méthodes (le noyau n'accepte par exemple ni covariance, ni contravariance, et ne connaît que deux niveaux de visibilité : public et privé) en leur associant d'autres traitements, et finalement terminer la construction de cette classe en la validant. Une fois validée elle devient utilisable, mais ne peut plus être modifiée (pour des raisons évidentes de sécurité).

Notons encore que le langage FACADE, en tant que support pour véhiculer des traitements est amené à définir la notion de variables et parmi elles la notion de constantes. Cependant il ne peut pas fixer le format des constantes. Dans le bytecode Java par exemple, un nombre flottant est codé en respectant la norme IEEE 754 [IEE85]. Ce choix est fixé par la description de la machine virtuelle. Imaginons qu'on souhaite réaliser un adapteur de code Java capable de supporter des nombres flottants. Cet adapteur de code devrait ajouter aux programmes qui utiliseraient des nombres flottants une bibliothèque pour gérer leurs valeurs (car les nombres flottants, très éloignés des fonctions élémentaires du matériel, ne sont pas définis dans le noyau). Encore faut-il pouvoir déclarer un nombre flottant ! Pour cela la bibliothèque d'extension qui définit la mise en œuvre des nombres flottants doit

7. ML : Meta-Langages : Génération de langage de programmation intégrant des propriétés avancées de généricité et de polymorphisme.

aussi pouvoir définir la manière dont ils sont déclarés dans le code FACADE.

Finalement la définition du langage FACADE est extrêmement simple. Tous les aspects du fonctionnement du système de la carte sont définis par des interfaces de programmation représentées par des interfaces de programmation du système d'exploitation de base.

3.3.6 Système d'Exploitation de Base

Le prototype de carte réalisé à partir de l'architecture Camille présenté ici définit un ensemble de routines de base, qui ont pour but de gérer le matériel présent dans la carte. En effet, contrairement aux ordinateurs traditionnels, le matériel (mémoires, entrées/sorties) des cartes à microprocesseur ne fournit aucune fonction *câblée* pour gérer son propre fonctionnement. Il faut que les routines système assurent une partie des tâches nécessaires à l'activité du matériel (sérialisation /désérialisation des bits transmis par le port E/S, temporisation des écritures pour le chargement des pompes de mémoire Flash, ...). Ces opérations sont intimement liées aux ressources physiques et constituent donc la couche la plus base du système d'exploitation de la carte. Elles ne définissent cependant pas un modèle d'abstraction au-dessus duquel les applications peuvent directement être chargées, car elles n'apportent aucun élément de sécurité.

A partir de ces routines de gestion du matériel nous avons défini un ensemble de composants du système qui définissent un modèle de base du matériel. Ce modèle décrit des types qui apparaissent pour le langage FACADE sous la forme de classes. Le système d'exploitation de Base (le SEB) propose donc un ensemble de classes qui permettent la manipulation des primitives de la carte. Certaines spécifient des éléments matériel tels que les blocs de mémoire persistante (pages mémoire), ou les mots machines traités par l'unité arithmétique et logique du microprocesseur. La description de ces différents composants de base, ainsi que de tout ceux qui pourraient être chargés par la suite est gérée par l'interface de gestion-système sécurisée.

3.3.7 Interface de Gestion-système Sécurisée

L' Interfaces de Gestion-système Sécurisée (l'IGS) assure quatre fonctions à l'intérieur des cartes ouvertes.

La première est de définir pour chaque classe du noyau la manière dont elle peut utilisée. Il s'agit simplement de définir les règles de types de toutes les classes du noyau. Les objets qui représentent des blocs de valeurs définissent par exemple deux méthodes (non virtuelles) qui prennent en paramètre un indice de type `CardByte` et retournent l'une un `CardByte` et l'autre un `CardShort`. Ce sont les méthodes de lecture d'un ou deux octets de données numériques. À l'aide de ces déclarations, il est possible de vérifier le code intermédiaire dans la carte. Ainsi on peut accomplir la majeure partie des contrôles de sécurité-innocuité durant le chargement des nouveaux programmes (*c.f.* section 2.4.4 page 46)...

La seconde tâche de l'IGS est de maintenir la cohérence des informations de type. Nous avons vu que la structure de classes du noyau est une base qui peut être étendue. De

nouveaux programmes peuvent définir de nouvelles classes à partir de celles déjà existantes. Pour permettre ces opérations d'extension, l'IGS déclare des classes (dans le SEB) qui définissent l'organisation des structures de données associées à la description des types (des méta-informations). Ces informations sont gérées par des classes (tel que `CardClass` et `CardCode`). Elles définissent aussi les méthodes pour permettre l'extension du système de type. Ces méthodes assurent que l'extension de la structure de classe évolue de manière cohérente.

La troisième tâche de l'IGS est de garantir la cohérence de la mémoire objet. Il s'agit principalement de gérer la libération de la mémoire de telle sorte qu'il ne soit pas possible pour une application de rendre au système un bloc de mémoire sur lequel il existe encore des références. C'est une opération délicate à réaliser lorsqu'une application est en train de s'exécuter dans la carte (à cause des références qui pourraient être placées dans le contexte d'exécution d'une méthode). Aussi l'IGS propose un mécanisme de *nettoyage* (ramasse-miettes, qui sont des pages en l'occurence, ou *garbage collector*) de la mémoire déclenché par une commande externe (APDU) spécialisée. À charge pour lui de ne libérer que ce qui n'est plus utile.

Enfin l'IGS lui-même doit être extensible. Il doit permettre à de nouvelles applications d'enrichir les contrôles sécuritaires de base du système, avec de nouveaux contrôles. L'une des premières extensions que nous avons identifiée porte sur l'intégration de capacités pour faciliter la sécurisation des applications qui coopèrent [Gri99] (ce qui est important dans un contexte d'interopérabilité). Elles ont ensuite été développées sur d'autres infrastructures cartes [HV00] (mais toujours en supposant un contrôle de type effectif au niveau de la carte). D'autres exemples de ce besoin apparaissent par exemple pour encarter des classes Java dans le noyau. Les classes Java définissent quatre niveaux de visibilité pour les méthodes (`public`,*package* implicite, `protected` et `private`). Pour réutiliser le concept de classe disponible dans le noyau pour représenter des classes FACADE, il faut pouvoir définir les niveaux de visibilité *package* implicite et `protected` que le noyau ne gère pas. Pour cela il faut que le programmeur de l'abstraction « Classes Java » puisse exprimer les conditions sous lesquelles il accepte ou refuse l'édition de lien avec les méthodes de ses propres classes. Et cela au moment même ou un programme est en cours de chargement dans la carte par le Décodeur Sécurisé de Traitements.

3.3.8 Décodeur Sécurisé de Traitements

Un module particulier du SEB gère le téléchargement de nouveaux traitements dans la carte. Ce mécanisme transforme les programmes reçus (exprimés dans le langage intermédiaire FACADE). Le DST[8] termine en fait les traitements d'une chaîne de compilation repartie entre le terminal et la carte. Toutefois, en plus de cela il contrôle que, lorsque ces traitements seront exécutés, ils ne pourront nuire aux autres applications chargées.

Concrètement la vérification exécutée assure que les applications ne peuvent interagir les

8. DST : Décodeur Sécurisé de Traitements.

unes avec les autres que par l'intermédiaire des déclarations de classes définies et partagées dans ce but. Le DST a aussi à charge de produire le code final (c'est-à-dire exécutable par la carte) équivalent. Enfin la nature des traitements effectués par le DST lors du décodage de chaque opération doit pouvoir être étendu afin de gérer les propriétés annexes de certains programmes. Ces mécanismes encartés feront l'objet de descriptions détaillées dans les chapitres 4 et 5 de cette partie du mémoire.

Les routines de gestion du matériel, avec l'IGS et le DST fournissent la base de confiance sur laquelle peuvent reposer les applications. On parle dans la littérature scientifique anglo-saxonne, de *Trusted-Computing-Base*. C'est cet ensemble qui compose le détail de l'architecture du système encarté.

3.4 Détail de l'architecture du système encarté

La partie de Camille qui est finalement placée dans la carte constitue un véritable micro-noyau organisé autour de trois fonctions fondamentales :

- permettre le chargement et l'exécution de nouveaux traitements ;
- apporter une base de confiance élémentaire pour chaque application ;
- donner une représentation fonctionnelle du matériel encarté synthétique et efficace.

Nous avons vu que la base de confiance choisie repose sur un système typé structuré en classes. Tous les éléments du système (ce qui inclut tous les éléments matériels dont le système donne une représentation) sont donc présentés comme des classes élémentaires. L'architecture globale du système encarté se présente sous la forme d'un jeu de classes qui peut ensuite être étendu. Ce jeu se compose en fait de vingt trois classes présentées par la figure 3.4. On distingue sur cette figure six regroupements de classes relatifs aux différents aspects organisationnels du système. Les aspects fonctionnels tels que l'IGS et le DST ne sont pas indiqués sur le schéma car ils représentent des structures transversales au modèle objet. Les blocs tels que la gestion des types ou encore la représentation brute de la mémoire apparaissent en pointillés sur la figure 3.4. Détaillons un peu leurs fonctions en commençant par le bloc de gestion du matériel.

3.4.1 Bloc de gestion du matériel

Le bloc nommé **Matériel** sur la figure 3.4 regroupe les opérations élémentaires liées à la manipulation logicielle du matériel. Les fondeurs hésitent en règle générale à utiliser de la logique câblée pour traiter physiquement des problèmes matériels que le microprocesseur pourrait traiter avec certaines routines (pilotage des écritures en EEPROM, gestion des temps de garde entre chaque accès, échantillonnage du port d'entré/sortie,...). C'est la principale raison d'être du bloc matériel. Aussi n'est-il composé que d'une seule classe: `CardKernel` qui regroupe toutes ces opérations dans une entité logique.

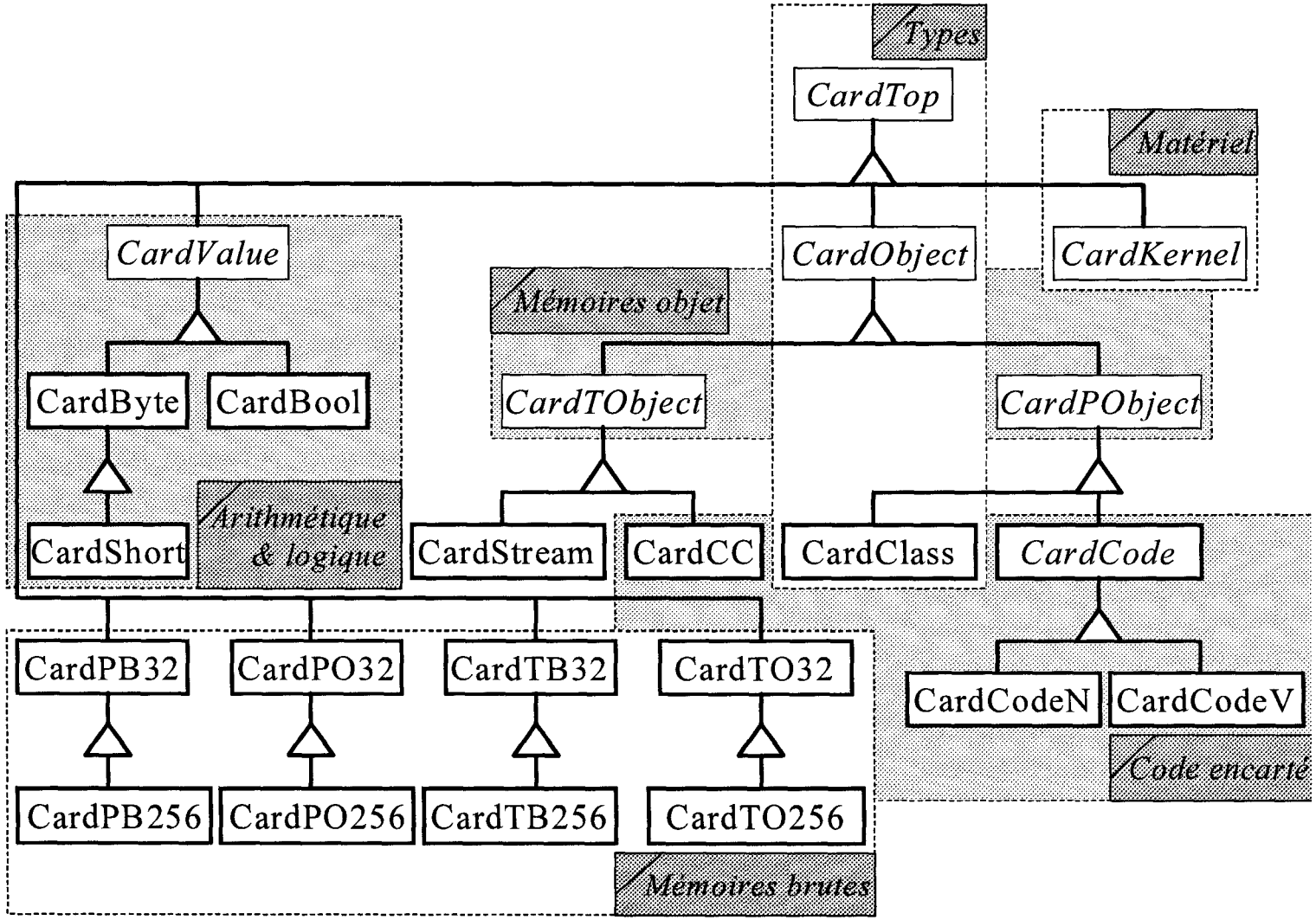


FIG. 3.4 – *Architecture des types du Système d'Exploitation de Base*

En fait la classe `CardKernel` ne peut pas être instanciée. Elle ne contient que des méthodes statiques dont l'usage est strictement réservé aux composants du noyau encarté. En effet l'accès direct, par un traitement non fiable (chargé après émission de la carte) à ces routines représenterait un singulier point de défaillance pour la sécurité de la carte. À ce niveau l'ensemble des pages mémoire par exemple peuvent être manipulées (lues et écrites) indépendamment de leur nature. C'est une autre partie du noyau qui permet d'associer à une entité physique, comme une page de mémoire ou une valeur numérique manipulable par le microprocesseur, une classe et donc un contenu sémantique. Cet acte d'association est lui même géré par le bloc des types.

3.4.2 Bloc des types

Le bloc **Types** décrit et définit la structure d'amorçage des types qui sont chargés dans la carte. Cette structure est inspirée des modèles à objets réflexifs tels que Smalltalk [GR83]. La structure de base repose sur les relations d'instance et d'héritage entre la classe `CardObject` et la classe `CardClass`. `CardObject` définit les propriétés fondamentales de toutes les instances (structures de données polymorphes) présentes dans la carte. `CardClass` hérite de `CardObject` et définit les opérations additionnelles spécifiques aux structures (objets) qui décrivent les classes. La structure de données qui décrit la classe `CardObject` ou la classe `CardClass` dans la carte est donc une instance de la classe `CardClass`.

A ce système de types de base, vient s'ajouter une troisième classe: `CardTop`. Cette classe décrit les objets dont on ne connaît rien. Dans bien des systèmes de types orientés objet le type `Top` est confondu avec la classe `Object`. Dans le langage Java par exemple, la classe `Object` est la classe de base dont héritent directement ou indirectement toutes les autres classes. Néanmoins dans ce langage, il existe des types déclarés primitifs qui échappent à la structure hiérarchique des classes. Ce sont les `int`, `float` et autres `char` ou `byte`. Pour éviter d'avoir à gérer des cas particuliers, le système encarté décrit de manière uniforme tous les types. La classe de base commune à tous, mais qui finalement ne décrit aucune manipulation sur les instances, est `CardTop`. Cette classe joue un rôle clef dans le mécanisme d'inférence de types, car elle permet de donner un type aux variables qui ne contiennent aucune valeur significative à un instant donné (et sur lesquelles on ne peut donc rien faire).

De cette classe « vide » héritent, directement ou indirectement, toutes les autres classes, aussi différentes que `CardClass` qui définit la forme d'une classe dans la carte ou `CardByte` qui est un type du bloc arithmétique et logique.

3.4.3 Bloc arithmétique et logique

Le bloc **Arithmétique et logique** donne une image des mots machines (8 bits et 16 bits) tel qu'ils sont manipulés par les différents microprocesseurs encartés. Notre hiérarchie de type peut surprendre car elle n'est pas construite selon des conventions adoptées par des

langages de haut niveau. En général un compilateur (Java, C, ...) reconnaît le type `byte` (`char`) comme une sorte de `short`, alors que nous reconnaissons un short comme une sorte de `byte`. Cependant lorsqu'une machine virtuelle Java, par exemple, accède à un byte, elle effectue une opération de conversion (en int, taille naturelle pour l'UAL de la machine virtuelle Java) qui n'est pas naturelle pour la machine réelle. Les classes du noyau (figure 3.4) reflètent d'abord le fonctionnement du matériel. En définissant `cardShort` comme une sorte de `CardByte` la hiérarchie de classes reflète l'habitude des microprocesseurs qui sont encartés (l'ARM par exemple, mais plus particulièrement l'AVR, cible de notre implantation) à lire et traiter un mot de 8 bits comme la partie la moins significative d'un mot de 16 bits. Les conversions particulières (extension/préservation du signe principalement) sont des opérations particulières de ces types que l'adapteur utilise si nécessaire pour refléter le comportement d'un code Java ou MEL.

Outre la présentation des opérations natives du microprocesseur, les classes du noyau définissent des méthodes qui réalisent des opérations arithmétiques et logiques que nous avons juger élémentaires, même si elles ne sont pas toujours câblées par le microprocesseur. Il s'agit d'opérateurs tels que la multiplication et la division entières qui ne sont pas fréquemment câblés dans une carte. La réalisation de ces opérations nous paraît judicieuse car elles sont véritablement un fond commun à tout les langages de programmation. Ce choix ne va pas à l'encontre de la stratégie générale adoptée pour concevoir le SEB qui consiste à donner une vue du matériel, mais aussi des bibliothèques par défaut « raisonnables » (*c.f* section 3.2.2 page 60).

Ce raisonnement (proposer une « minimisation raisonnable » de l'abstraction matérielle) est aussi utilisé pour concevoir les modèles liés à la représentation de la mémoire. Dans le noyau, deux blocs de classes reflètent deux vues des ressources mémoires encartés. Le bloc mémoires à objets donne une abstraction raisonnable du matériel alors que la vue minimale de cette ressource est présentée par le bloc de mémoires brutes.

3.4.4 Bloc de mémoires brutes

La mémoire des cartes à puce est décomposée en pages élémentaires. Le système de base doit veiller à gérer l'allocation de ces pages de manière à minimiser leur taux d'usure. Le bloc **Mémoires brutes** fournit une vue élémentaire, mais sécurisée des pages de mémoires des cartes.

- Le modèle distingue deux tailles de pages selon qu'elles représentent une page de 32 octets ou un groupe de 8 pages consécutives de 256 octets.
- Il distingue deux natures de pages selon qu'elles sont allouées en mémoire volatile ou en mémoire persistante.
- Enfin il reconnaît deux contenus de pages, selon qu'elles contiennent des valeurs numériques ou des références vers la mémoire.

C'est pourquoi le bloc de mémoires brutes définit $2 \times 2 \times 2 = 8$ types différents.

Les pages en mémoire volatile sont différenciées des pages en mémoires persistantes car elles ne peuvent pas être écrites de la même manière. Alors que l'accès en lecture est uniforme quel que soit le type de mémoire qui supporte le bloc, l'accès direct en écriture aux pages de mémoire persistante est impossible. Une routine particulière du système doit alors être utilisée.

Les blocs qui contiennent des valeurs numériques sont distingués des blocs qui contiennent des références vers la mémoire pour permettre à l'IGS d'implanter le mécanisme de nettoyage de la mémoire. Il est nécessaire de connaître les blocs qui stockent des références pour pouvoir garantir la deuxième propriété de gestion des pointeurs typés énoncées dans la section 2.4.4 page 46.

Enfin deux tailles de blocs ont été définies. La première correspond au grain d'usure d'une cellule de mémoire persistante[9]. La seconde correspond à un regroupement de n pages élémentaires consécutives et alignées sur une adresse qui est une puissance de huit, le tout formant un bloc large de 256 octets. Cette approche concorde avec les granularités de pages retenues par d'éventuels MMU [CCG95] embarqués, mais surtout il simplifie l'implantation du confinement de l'accès aux éléments du bloc.

Il est essentiel que l'accès à un indice d'une page mémoire soit confiné à l'espace qu'occupe réellement ce bloc. De la même manière qu'un tableau sur une machine virtuelle Java a une taille donnée (connue dynamiquement) au delà de laquelle il est impossible d'accéder, il doit être impossible de faire par exemple une opération `GetByte` (lecture de l'une des valeurs placées dans le bloc) ou `SetByte` (écriture) « en dehors » de la page fournie par le gestionnaire de **mémoires brutes**.

Cette contrepartie dynamique à la sécurité statique par les types pénalise lourdement l'accès aux tableaux Java. L'approche de l'architecture présentée ici est de minimiser ce sur-coût en utilisant des propriétés implicites des microprocesseurs et en appauvrissant la sémantique de l'opération d'accès. Il est essentiel que les temps d'accès au pages allouées en mémoire soit le plus court possible tant les bibliothèques d'extension les sollicitent. Les opérations `GetByte` ou `SetByte` par exemple ne déclenchent pas d'exception lorsqu'un programme demande l'accès à une valeur dont l'indice est hors du bloc (de la page). Elles accèdent simplement à une valeur quelconque à l'intérieur du bloc manipulé. Impossible donc, avec ces opérations de « visiter » le contenu du reste de la mémoire. Ce résultat peut être obtenu en masquant simplement les cinq premiers bits de l'indice fourni en paramètre de la méthode. Dans le cas d'une page de 256 octets, le masquage devient inutile car l'indice attendu en paramètre est codé sur un octet.

Le gestionnaire de mémoires brutes ne permet pas une définition fine du type de chaque élément stocké dans une page mémoire. Il définit simplement si les informations qu'il contient sont des valeurs numériques (sorte de `CardByte`) ou des références à des objets (sorte de `CardObject`). Il résume les blocs de mémoire à des tableaux statiquement

9. certaines technologies permettent de définir plusieurs tailles de pages ce qui limitent cette assertion à une première approximation de l'usure réelle.

bornés. Pour permettre une utilisation typées de chaque élément d'une page, le système d'exploitation de base définit un second modèle de mémoire. Ce modèle est géré par le bloc de mémoires à objets.

3.4.5 Mémoires à objets

Le bloc **Mémoires objets** propose une organisation plus souple des pages de mémoire allouées. Les pages sont présentées comme des objets au sens traditionnel. Les deux premiers octets de chaque structure forment un pointeur vers le descripteur de classe qui est associé à l'objet. Ainsi le type (c'est-à-dire la classe, et donc l'usage) peut être retrouvé dynamiquement. On lit parfois le terme de *boxed object* en opposition à *unboxed object* pour distinguer les objets dont le type est connu dynamiquement de ceux qui sont définis une fois pour toutes statiquement (par les seuls types du bloc **Mémoire brutes** dans notre cas).

Le surcoût lié à l'usage des objets pour la conception de systèmes dans la carte a déjà été évalué [Gri97]. Cette étude a montré que les cartes à microprocesseur ont atteint aujourd'hui un niveau de développement suffisant pour qu'il soit possible d'utiliser des appels de méthode à la place d'appels de procedure y compris pour la conception des systèmes d'exploitation. Bien sûr dans ce contexte, l'usage d'invocations de méthode doit être confiné aux traitements non critiques. C'est la stratégie adoptée pour le développement du SEB.

La présence d'un pointeur vers le descripteur de classe permet de retrouver l'adresse de la routine à invoquer dynamiquement et ainsi d'assurer le polymorphisme d'objets. Le descripteur de classe définit en effet une table des traitements applicables (en distinguant ceux qui sont publiques de ceux qui sont restreints, pour des raisons de sécurité). Cette table est composée de la signature de la méthode (le nombre et le type des arguments attendus par la méthode) et de l'adresse de départ de la routine associée. Dans l'implantation d'un appel à une méthode sur un bloc objet, le mécanisme d'appel retrouve l'adresse de la routine à exécuter à partir du pointeur vers le descripteur de classe associé à chaque page de la mémoire objet. Cette technique est celle utilisée par la plupart des langages à objets.

Grâce à la mémoire à objets, il est possible de définir très finement la nature d'un bloc alloué en mémoire. Cependant il ne faut pas seulement voir ce support d'objets comme un moyen de simplifier la tâche de la conversion des applications conçue avec des langages orienté objet. Il s'agit avant tout de proposer une gestion encapsulée[10] des structures de données placées en mémoire. Les modèles de mémoire objet[11] sont été proposés par des langages tels que lisp et Prolog par exemple.

L'architecture des machines virtuelles virtuelles (présentée par la figure 2.4 page 39) s'appuie entièrement sur ce seul modèle de mémoire. Nous n'avons pas retenue cette ap-

10. structure de donnée encapsulée, désigne un format de structure de donnée qui contient une marque. On lit *boxed memory* dans la littérature scientifique anglophone.

11. Le termes mémoires objet doit être lu dans le sens des *boxed object* ici.

proche car elle pénalise l'efficacité du support d'exécution lorsqu'il est utilisé pour définir des composants systèmes (un système de fichier par exemple). Aussi le SEB embarque deux modèles de gestion du typage de la mémoire, afin de concilier, selon les besoins, l'efficacité pour permettre la programmation de routines de bas niveaux et la flexibilité des mémoires objets (induit par le polymorphisme d'objet) du code encarté.

3.4.6 Code encarté

Le dernier bloc **Code encarté** concerne tous les aspects fonctionnels de la gestion des traitements encartés. Il gère le chargement de nouveaux traitements. Il assure la persistance des informations de type nécessaires aux mécanismes de sécurité (pour le contrôle de types). Enfin il implante (si nécessaire) le support d'exécution embarqué.

Les instances de la classe `CardCode` définissent les informations abstraites relatives aux codes présents dans la carte. Elles assurent la persistance des informations de signatures (arguments attendus, type de retour, type de l'instance associée, constantes utilisées, ...). Les deux classes `CardCodeN` et `CardCodeV` précisent la forme du traitement. `CardCodeN` gère les codes natifs (exprimés dans le langage machine du microprocesseur de la carte) ainsi que le point d'entrée du flot d'exécution du microprocesseur (l'adresse à laquelle l'exécution doit ce poursuivre). La classe `CardCodeV` définit en plus un interprète qui évalue le traitement qu'elle a enregistré.

Le bloc **Code encarté** supporte pour l'essentiel la fonction de vérification de Code qui est détaillée et formalisée par le chapitre 4, alors que la fonction de génération de code final (et plus particulièrement de code natif) est complètement décrite par le chapitre 5 et évaluée par le chapitre 6. Aussi nous ne commenterons pas d'avantage ce bloc. Il est l'implantation de l'essentiel des aspects novateurs de l'architecture Camille.

3.5 Aspects novateurs de l'architecture Camille

L'architecture globale que constitue Camille est une synthèse de différentes propositions formulées dans des contextes différents de la carte dans le passé. Elle est aussi un support de réflexion et d'évaluation pour une vaste palette d'expériences. Différents travaux contribuent aujourd'hui encore à l'enrichir [Dab00]. Aussi toutes les démarches entreprises autour de cette plate-forme ne seraient être détaillées par la suite dans ce document.

Dans le cadre de ce mémoire nous nous sommes plus particulièrement intéressés à deux aspects clefs du logiciel de base tel qu'il a été encarté. Il s'agit d'une part de la formalisation d'un mécanisme d'inférence de types adapté aux contraintes (de taille de mémoire de travail) de la carte à microprocesseur, et d'autre part du mécanisme de génération de code natif tel qu'il est encarté. L'inférence de types est abordée depuis peu dans le contexte des cartes ouvertes.

Le prototype du noyau Camille que nous avons réalisé est aujourd'hui le premier exemple de déploiement *in situ* des techniques qui sont proposées pour assumer les contrain-

tes réelles de la carte. Les coûts (notamment en taille de code système à encarté) pour implanter le vérifieur de code dans la carte sont détaillés dans le chapitre 6. Avant cela le chapitre 4 présente le modèle formel qui a pu être utilisé pour prouver (à l'aide du langage B) le bien-fondé de notre mécanisme d'inférence de types. Cette démarche est à rapprocher de celle entreprise par E. Rose [RR98] autour du bytecode JavaCard.

Une fois le vérifieur de code prouvé correct, nos travaux ont porté sur une première série de tests visant à valider la possibilité de générer un code natif dans la carte. Ces expériences nous ont conduit à proposer un modèle de générateur de code encartable. Notre prototype nous a permis de prouver la faisabilité d'un générateur de code natif « à la volée » mis en œuvre lors du chargement de nouveaux traitements. Une modélisation de ce générateur de code est présentée par le chapitre 5. Différentes expériences que nous avons menées sur la base de ce générateur de code natif sont présentées par le chapitre 6. Les premiers résultats obtenus sont particulièrement encourageants et valident la démarche que nous avons entreprise et qui vise à ouvrir les supports d'abstraction du système d'exploitation des cartes à microprocesseur.

Ces deux techniques (preuve de programme et génération de code natif) constituent dans le cadre de la carte à puce des innovations majeures par rapport aux cartes ouvertes existantes. Elles sont le fruit de l'architecture Camille que nous avons succinctement présentée ici mais qui a aussi engendré d'autres résultats (notamment [VG00] et [HGV00] [12]) qui ne seront pas détaillés dans ce document. Bien que nos travaux soient orientés système d'exploitation, ils sont aussi liés recherches relatives aux langages. Et je reprends les propos de Carine Baillarguet qui estime que le mariage des langages et des systèmes est aujourd'hui plus qu'une question de raison, une nécessité [Bai99]. En effet, la clef de voûte de l'architecture Camille, l'élément qui lie les différents composants du système, est FACADE : un langage intermédiaire typé pour carte.

12. Rappelons notamment que la motivation initiale qui nous a amenée à introduire la notion de capacité dans la carte est le produit de réflexions autour des limitations du modèle de sécurité induit par la vérification de type.

Chapitre 4

FACADE : Un langage intermédiaire typé pour carte

« *Reconnais la vérité pour pardonner à l'erreur* »

Traité sur la tolérance, Voltaire

Ce chapitre décrit formellement le langage intermédiaire FACADE dont le rôle de pivot a été présenté dans le chapitre 3 section 3.3.5 page 71. L'objet de ce nouveau chapitre est d'expliquer comment le langage FACADE a été construit de telle sorte qu'il permette de vérifier (à défaut de pouvoir établir) dans la carte les traitements émis. Un programme incorrect (par erreur ou par malveillance) doit pouvoir être détecté et ainsi écarté pour éviter qu'il engendre plus tard, par son activité, tout risque de dégradation des autres applications encartées.

La première section de ce chapitre a pour but de préciser les raisons qui ont motivé la conception du langage FACADE. La section suivante définit les structures clefs du langage. Elle décrit les différentes sortes de données manipulées par un traitement exprimé avec FACADE ainsi que les opérations de base qui permettent de travailler sur ces données. La troisième partie présente la notion de contrôle de type telle qu'elle est définie en général, elle détaille sa mise en œuvre sur des exemples de code FACADE. La complexité de cet algorithme ne permet pas de le placer entièrement dans la carte. Aussi la quatrième section montre comment il est possible de décomposer le contrôle de type en deux parties : *génération de preuve* et *vérification de preuve* afin que la carte n'ait plus qu'à implanter la partie *vérification de preuve* (bien plus simple qu'un contrôle de type classique). Le processus d'inférence ainsi décomposé a fait l'objet d'une formalisation afin de prouver pour qu'il soit prouvé qu'il est fiable. Les bases formelles de cette preuve sont présentées en cinquième section avant de conclure sur les difficultés que peut rencontrer un programmeur pour protéger ses programmes grâce à un système de types, en déclarant et en programmant convenablement ses classes. Car l'inférence de type est bien évidement un principe

de bas niveau qui en temps que tel constitue pour l'émergence d'outils plus pratiques aussi bien un support élémentaire qu'une source de motivation.

4.1 Motivations

FACADE est un langage intermédiaire dédié aux cartes à microprocesseur ouvertes. L'ambition de ce langage est de permettre à la carte de « voir » les traitements qu'elle charge. A ce jour, les cartes ouvertes ont une attitude a priori. Lorsqu'elles se proposent de garantir la confidentialité et l'intégrité de leur contenu, c'est en utilisant des mécanismes d'innocuité défensifs. Ces mécanismes se caractérisent par des batteries de test qui sont effectués par le microprocesseur durant l'interprétation de chaque nouvelle instruction de la machine virtuelle. Le mot défensif reflète bien l'état d'esprit de cette approche. Plutôt que de s'assurer de la bonne conduite d'un programme, le système d'exploitation de la carte à microprocesseur protège (défends) les applications des agressions qui peuvent être commises par des programmes malveillants en les confinant dans des espaces clos.

Cette stratégie pénalise indirectement, mais fortement, les capacités de flexibilité d'efficacité et d'adaptabilité des cartes à microprocesseur. La section 3.1 page 54 souligne la nécessité de pouvoir charger des traitements destinés à permettre l'intégration de propriétés non fonctionnelles. Cela implique de pouvoir charger des traitements qui sont sollicités extrêmement fréquemment par l'application (par exemple des traitements de flots, des systèmes de gestion de fichier, ou un nouveau mécanisme de recouvrement de la mémoire persistante). Pour que ces opérations particulières puissent réellement être utilisées par les applications, elles doivent être performantes.

Cependant l'efficacité de l'exécution des traitements ne peut se faire au dépend de leur portabilité. Car il pourrait sembler de judicieux que les adapteurs de code gênèrent directement des traitements natifs (hors de la carte) lorsque la contrainte d'efficacité est élevée. Toutefois dans une carte les microprocesseurs embarqués sont choisis dans une gamme de produits bien plus hétérogènes que celle de nos stations de travail. De plus de nouvelles générations de cartes qui impliquent l'utilisation de nouveaux microprocesseurs prennent place dans des infrastructures géographiquement très largement distribuées. Lorsqu'un microprocesseur différent est adopté par une nouvelle génération de cartes, il ne semble guère raisonnable de proposer une solution (pour charger des applications) qui nécessite la mise à jour de milliers de terminaux de chargements placés en des lieux distribués à l'échelle d'un pays ou même davantage. Il est apparu plus raisonnable de fournir tous les traitements encartés sous un format unique et abstrait qui puisse être compris de la carte quelque soient les détails de son microprocesseur réel.

En fait le langage FACADE se veut plus abstrait encore que le bytecode Java par exemple. Plus d'abstraction dans le langage intermédiaire utilisé pour communiquer les traitements à la carte, doit être conjugué ici avec plus de liberté dans l'interprétation des traitements par la carte. Telle est l'ambition de FACADE: Permettre aux cartes à puce d'analyser facilement et à moindre coût les codes embarqués durant leurs chargements.

Cette compréhension du code par la carte a pour but de lui permettre (1) de s'assurer de l'innocuité d'un traitement, une fois pour toute lorsqu'elle le charge, mais aussi (2) de générer un code effectivement exécuté par la suite, plus performant et plus adapté à la réalité exacte du contexte matériel de chaque carte utilisée. Après tout, si la carte a pour ambition d'être le représentant universel de son porteur, elle doit pouvoir le prouver en étant capable de se représenter elle même lorsqu'elle reçoit de nouvelles applications !

Ce qui a motivé la définition d'un nouveau langage intermédiaire minimaliste c'est avant tout que la simplicité peut être le fruit d'un plus haut niveau d'abstraction. La notion de type est à l'origine intimement associée à la notion de langage abstrait (de haut niveau). Simplifier l'implantation de l'inférence de type dans la carte pour simplifier la manipulation (le contrôle de sécurité mais aussi la génération de code) des traitements transmis, est la base de la démarche qui a guidé la conception de FACADE : Un langage intermédiaire typé dédié à la carte à microprocesseur.

4.2 FACADE : Un langage intermédiaire typé dédié à la carte à microprocesseur

Le langage intermédiaire FACADE propose une représentation abstraite des traitements. Le code FACADE est un format binaire ou les symboles et les opérations sont exprimés par des valeurs numériques (l'annexe A donne une définition complète de ce format binaire). Ce codage reste toutefois abstrait. Les valeurs qui codent les différentes opérations de base, aussi bien que les symboles ne sont pas issues de la description d'une machine (même virtuelle). Elles représentent des variables symboliques dont l'implantation réelle pourra aussi bien se trouver dans un registre que dans une zone de mémoire persistante.

Notons cependant que cette représentation abstraite serait difficilement lisible. En effet, le codage binaire d'un traitement, même abstrait reste obscur. Une version « assembleur » du langage a donc été définie. L'objet de cet assembleur est simplement de remplacer des champs de bits par des symboles lisibles. Dans les exemples de code commentés par la suite, ce sont les sources assembleur qui seront montrées. La compréhension des mécanismes expliqués ne nécessite en aucune manière la connaissance de l'encodage binaire. Il faut néanmoins garder à l'esprit que c'est bien la forme binaire que reçoit en fait la carte.

L'objectif des sections suivantes est de définir la nature des différentes données que permet de manipuler un traitement ainsi que les opérations élémentaires qui peuvent être exprimées (notamment en utilisant les données de base). Le tout forme la structure d'un traitement exprimé avec FACADE.

4.2.1 Structure d'un traitement exprimé avec FACADE

Un traitement exprimé par l'intermédiaire du langage FACADE se présente comme une chaîne binaire qui commence par le code hexadécimal `0xFACADE00`[1]. Dans les octets qui suivent, on peut distinguer trois structures distinctes:

1. La description des données manipulées par le traitement;
2. La description des points de saut (étiquettes de branchements) utilisés par le traitement;
3. La séquence des opérations élémentaires qui constituent le traitement.

La description des points de saut est probablement celle qui pourrait paraître la plus simple. Il s'agit simplement d'associer au label n°*labelId* l'opération n°n dans la séquence des opérations qui décrivent le traitement. On peut alors se demander l'intérêt même d'une telle table de conversion, car il serait possible de remplacer dans les opérations de saut les labels l directement par les opérations α qu'ils références (comme le fait MEL ou encore le bytecode JavaCard et les langages intermédiaires pour cartes). En fait cette partie d'un traitement FACADE contient d'autres informations qui ne seront complètement compréhensibles qu'avec les explications fournies par la section 4.4 page 97. Les informations additionnelles associées à chaque label sont en fait nécessaires au moteur d'inférence de type tel qu'il est mis en œuvre sur les données du langage FACADE.

4.2.2 Données du langage FACADE

Le langage intermédiaire FACADE, comme tout langage, permet d'exprimer des traitements qui manipulent des données. Traditionnellement les langages intermédiaires travaillent sur les données au moyen d'une pile. Le problème est que les microprocesseurs encartés sont en général (en fait quasi systèmatiquement) des machines à registres. Aussi la conversion vers ces microprocesseurs d'un code exprimé par le biais d'une machine à pile est une tâche délicate voir aléatoire dans le contexte minimaliste d'une carte à puce. Nous avons préféré définir les données manipulées par le langage FACADE d'une autre façon. Indépendamment du type (de la classe) de ces données nous distinguons six sortes de variables selon leur utilité (c'est à dire selon l'usage auquel elles sont destinées). Bien que les différentes sortes de données considérées par le langage FACADE soient sans doute assez intuitives il convient de les définir clairement pour qu'il n'y ait pas de confusion par la suite.

Les variables locales

Les variables locales sont des variables de travail. Elles servent à stocker « temporairement » les valeurs qu'on y place. La place en mémoire qu'elles occupent est « réservée » lors

1. Pour la petite histoire la terminaison de l'entête par l'octet `0x00` est l'abréviation de `0`rienté `0`bjet mais aussi le numéro de version du langage: `0.0` (prototype)

de l'initialisation de l'exécution du traitement. Elle est différente à chaque fois que le traitement recommence (par exemple lors d'appels récursifs). Les variables locales font partie intégrante de ce que j'appellerai par la suite: le « contexte d'exécution du traitement ».

Les arguments

Les arguments passés en paramètre lors de l'exécution d'un traitement sont traités comme des variables locales. Notons simplement que leur type est donné par l'entête du code FACADE, et ils sont déjà initialisés lorsque le traitement débute une exécution. Ils peuvent par la suite être utilisés comme des variables locales. Leur usage est, comme l'indique leur nom, de fournir des paramètres aux traitements qui sont exécutés.

le cas de `This`

Le langage intermédiaire FACADE est orienté objet. Il permet donc facilement d'exprimer des traitements définis pour des classes et donc associés à des instances: des méthodes. La variable `This` contient lors de l'exécution d'une méthode un pointeur sur une instance de la classe de la méthode. En fait il s'agit simplement d'un pointeur particulier qui associe une structure de données à l'exécution de chaque traitement, et qui en temps que tel ne peut pas être modifié par la méthode. Il faut noter, aussi, que cette variable n'est définie que si le traitement est associé à une classe, et ce n'est pas nécessairement le cas dans FACADE. Ce langage est aussi utilisé pour exprimer des requêtes ou des procédures qui sont alors appelées « traitements libres » et ne sont associés à aucune classe.

Les attributs

Si le pointeur `This` est définit c'est qu'il référence une structure de données particulière décrite par la classe dont les différents éléments sont les attributs de l'objet (l'objet en question pouvant être un descripteur de classe si la méthode est une méthode de classe). Chaque attribut est d'un type donné et ne peut pas être modifié. Les attributs servent dans FACADE comme dans tout langage orienté objet à mémoriser un état qui dépasse la « durée de vie » des variables associées au contexte d'exécution d'une méthode. Dans le noyau du système, deux sous-classes de `CardObject` définissent deux durées de vie différentes pour les attributs. Ce sont les classes `CardTObject` pour les objets éphémères (*Transient*) de la carte et `CardPObject` pour les objets persistants. Les objets éphémères sont alloués en mémoire vive et leur durée de vie est au plus d'une connexion/déconnexion de la carte, alors que les objets persistants sont alloués dans la mémoire persistante et leurs valeurs restent indéfiniment mémorisées et utilisables.

Les constantes

L'entête de déclaration d'un traitement définit aussi un ensemble de constantes. Les constantes sont vues comme des variables typées dont la valeur est initialement définie et ne peut plus être modifiée par la suite. Une constante peut être du type `CardByte` ou un

`CardBool` et de la plupart des classes du noyau, mais aussi de n'importe qu'elle autre classe qui définit la méthode (de classe) `CardObject Create(CardStream unCodeFACADE)`. Cette méthode est appelée lors du chargement du traitement pour « obtenir » la valeur de la constante à partir de sa représentation numérique exprimée dans le code FACADE. La mémoire allouée par les constantes autres que des sortes de `CardValue` n'est libérée que lorsque les traitements FACADE qui les ont créés sont effacés eux-mêmes.

Il existe encore une sorte de donnée qui peut être manipulée par un traitement FACADE. Ces données ne font cependant pas l'objet d'une déclaration dans la description d'un traitement, mais elles sont connues de la carte par le biais d'une table de persistance. Ces données sont les classes présentes dans la carte.

Les classes

Une opération peut être appliquée sur une classe comme sur n'importe qu'elle autre variable du langage FACADE. Cela permet notamment d'exprimer simplement un appel à une méthode de classe. La table de persistance des classes définit cependant des règles d'usage visant à assurer la sécurité globale des applications qui ont définit ces classes. Il s'agit simplement d'un mécanisme de contrôle d'accès qui repose sur les principes déjà bien développés et connus par ailleurs[Tra95, Gri91]. Il vise à limiter les applications ayant droit d'utilisation sur des classes (ou de leurs méthodes) associant à chacune un identifiant d'applications (dans leurs tables de persistance). L'accès en écriture à la table des classes n'est pas définit (est donc impossible à exprimer dans un traitement FACADE) mais un mécanisme d'allocation et de révocation permet de construire ou de détruire des classes. C'est par ce biais qu'il est possible de définir de nouveaux types, et donc de nouvelles utilisations des ressources des cartes à microprocesseur.

Toutes ces variables (locales, arguments, attributs, ...) permettent d'identifier des données. Cependant pour pouvoir être utilisées il faut encore définir les opérations élémentaires de FACADE.

4.2.3 Les opérations élémentaires de FACADE

Au sens strict FACADE définit seulement cinq opérations différentes qui sont utilisées pour former la séquence d'opération d'un traitement. La volonté de réduire le nombre d'opérations élémentaires qui sont utilisées pour composer un traitement a guidé la conception du langage FACADE. L'objectif est de simplifier les processus qui seront appliqués par la carte sur les traitements. Moins d'instructions signifie moins de cas différents à gérer et donc, un noyau plus compact, plus facile à encarter...

voici la liste des instructions FACADE avec une rapide description de leurs fonctions:

1. `Return` *[VarRes]*
   Cette opération termine l'exécution du traitement. Son paramètre optionnel *VarRes* est la valeur qu'elle retourne en fin d'exécution;

2. `Jump` *LabelID*
   Cette opération est un saut inconditionnel à une autre opération du programme qui est marquée avec le label *labelId*;
3. `Jumpif` *Var LabelId*
   Cette opération est un saut conditionnel. La variable *Var* est un booléen qui détermine si l'opération de saut à l'instruction marquée par le *labelId* aura lieu ou pas;
4. `Jumplist` *Var nbCase* { *LabelId1, LabelId2, ...* }
   Ce saut conditionnel branche à l'instruction qui est marquée par le label dont le numéro de séquence dans la liste des labels { *LabelId1, LabelId2 ...* } est égal à la valeur de la variable *var*. Si la valeur de la variable *var* (qui est un *CardByte*) est un nombre négatif ou si elle est plus grande que le cardinal de la séquence de label (définie par la variable *nbCase*) l'opération de saut n'aura pas lieu;
5. `Invok` *Var methodId [VarRes]* { *tabVar1, tabVar2, ...* }
   Cette opération exécute la méthode *methodId* sur la variable *Var*. Le résultat (éventuel) de l'exécution de cette opération est placé dans la variable *VarRes*. Les variables données en paramètres pour l'exécution de la méthode *methodId* sont définies par { *tabVar1, tabVar2, ...* }. En fait cette opération définie par le langage intermédiaire FACADE peut être traduite par la carte, en fonction de la nature de *methodId*, aussi bien en un appel de méthode virtuelle qu'en un ensemble d'opérations élémentaires exécutables par la machine cible.

Sur la base de cette description d'un traitement FACADE il est possible de décrire les principes de l'inférence de type.

4.3 Principes de l'inférence de type

Les chapitres précédents ont présentés les mécanismes d'inférence de type par le biais d'un des rôles pour lesquels ils sont utilisés: protéger les programmes et leurs données. Le problème de l'installation dans le milieu particulièrement contraint de la carte est resté ouvert. Pour pouvoir détailler la mise en œuvre dans la carte de ces mécanismes il convient maintenant d'en proposer une première définition.

4.3.1 Première définition

L'inférence de type est réalisée avec un interprète. De la même manière qu'une machine virtuelle qui évalue les opérations élémentaires d'un programme à partir d'une définition de la *sémantique dynamique*[2] du langage, le moteur d'inférence évalue chaque opération élémentaire en respectant une sémantique que j'appellerais par la suite la *sémantique statique*. Alors que la sémantique dynamique de l'opération `i <- 3 + 4` nous permet d'affirmer qu'après son exécution `i` vaut 7, la sémantique statique de la même opération nous permet d'affirmer que `i` est un *Entier* ou que l'opération est illégale. C'est l'évaluation

2. Parfois appelée « sémantique naturelle »[Kah74] ou « Sémantique Opérationnelle Structurée »[Plo81]

statique du programme qui détermine s'il est correctement typé. La vérification est terminée lorsque toutes les opérations du programme ont été contrôlées avec succès (quelques soient les chemins parcourus et l'état des variables). En contrôlant le bon usage des types le système de chargement vérifie que le programme chargé respecte les règles de base du système (au travers des déclarations de ces classes) et qu'il ne pourra donc pas nuire à son environnement lors de son exécution. Il devient par exemple impossible d'utiliser un *Entier* en tant que *tableau d'Entier* et ainsi de lire le contenu de n'importe quelle partie de la mémoire...

Mais pour pouvoir détailler le fonctionnement du moteur d'inférence il faut au préalable définir plus rigoureusement dans FACADE la notion de hiérarchie de types.

4.3.2 Hiérarchie de types

Dans le noyau du système encarté, comme dans le langage intermédiaire FACADE toutes les informations sont typées. Comme FACADE est un langage orienté objet, les types sont des classes dont la description est stockée en mémoire persistante par le noyau. La hiérarchie des classes est une structure de données extensible. Lorsque de nouvelles applications sont chargées, elles définissent de nouvelles classes qui peuvent hériter des classes `CardPObject` et `CardTObject`. La description de ces nouvelles classes est elle aussi enregistrée en mémoire persistante par le noyau. La hiérarchie de classe est basée sur un modèle d'arbre. Chaque nouvelle classe a une et une seule super-classe. De plus, la classe `CardTop` est au sommet de la hiérarchie. Aussi, d'une manière ou d'une autre, toutes les classes définies dans la carte étendent la classe `CardTop` qui est la base commune à tous les types manipulés dans la carte.

Pour l'analyse de type, nous nous inspirons notamment des travaux de D. Dwyer[Dwy95], et nous définissons un semi-treilli par le tuple $L = (V, \subseteq, \cap)$ ou V est l'ensemble des classes, $\subseteq$ est une relation d'ordre partiel définie sur V et $\cap$ est une opération binaire définie sur V. « $i \subseteq j$ » se lit i *est une sorte de* j. Cette relation d'ordre est définie par l'arbre d'héritage des classes encartées. Deux éléments $i,j \in V$ sont *incomparables* si ni $i \subseteq j$ ni $j \subseteq i$. De plus, l'opérateur $\cap$ est définit par la relation 4.1. J'écrirai par la suite « premier père commun de i et j » pour désigner la classe $s \in V$ tel que $s = i \cap j$. Nous avons pu constater sur le diagramme des classes présenté dans le chapitre précédent, figure 3.4 page 76 qu'il existe un plus grand élément dans l'ensemble V des classes : `CardTop` est aussi appelé le type *Top* et peut être écrit $\top$. Notons que *Top* est le seul élément de l'ensemble V des types qui ne soit jamais incomparable avec aucune autre classe (car *Top* vérifie $\forall i \in V; i \subseteq \top$). Nous ajoutons à l'ensemble V des classes du système encarté, un type, construit artificiellement (car l'héritage multiple est impossible dans le système de type définit) et noté $\bot$ qui se définit par la propriété suivante : $\forall i \in V; \bot \subseteq i$ et par le fait qu'il ne reconnaît aucune opération de ces pères. Ce type n'a pas de réalité fonctionnelle dans le langage. Il faut le concevoir comme une hypothèse de travail qui simplifie remarquablement la formalisation du mécanisme contrôle des types.

$$\forall i,j \in V;\ s = i \cap j \Leftrightarrow s \in V \text{ et } \begin{cases} i \subseteq s, j \subseteq s \\ (\forall k \in V / i \subseteq k, j \subseteq k) \Rightarrow s \subseteq k \end{cases} \quad (4.1)$$

C'est sur les bases de cette hiérarchie que peut être définit la sémantique statique du langage intermédiaire FACADE, c'est à dire les principes du contrôle de type.

4.3.3 Principe du contrôle de type

Les contrôles de programmes sont en règle générale basés sur une analyse du flot de données. Le moteur d'inférence dispose du (ou doit déterminer le) type de chaque variable, pour chaque point du programme (noté par la suite *pp*). A partir de cette information, et pour chaque point du programme il effectue des tests qui dépendent des opérations contrôlées. Par exemple, nous avons vu que `Jumpif` prend en paramètre un booléen. Il faut dont vérifier que la variable utilisée par le `Jumpif` est elle-même un booléen.

Le contrôle de type est rendu complexe par l'usage de variables de travail (principalement les variables locales dans le langage FACADE) qui sont utilisées pour stocker temporairement différentes informations au cours d'un même traitement. Dans ce cas, le type de la variable change au cours de l'exécution du traitement. Je nommerai cette catégorie de variable (qui contiennent temporairement des données de différents types) les « Variables T». Le mécanisme de contrôle de type se résume à un processus trivial lorsque le type des variables manipulées par le traitement à contrôler ne peut pas être changé. Je désignerai par la suite ces variables à « type bloqué » par le terme de « variables L ». La table 4.1 reprend les différentes catégories de variables définies par le langage est précise si elles sont manipulées en temps que variables T ou L par le langage FACADE.

Sorte de variables	Utilisables en T	Utilisables en L	Peuvent être affectées
variables locales	Oui	Oui	Oui
arguments	Oui	Oui	Oui
This	Non	Oui	Non
Constantes	Non	Oui	Non
Attributs	Non	Oui	Oui
Classes	Non	Oui	Non

TAB. 4.1 – *Sortes de variables FACADE susceptibles d'être des variables L ou T*

Se limiter à l'usage de variables L rend la mise en œuvre du contrôle de type, triviale, même dans le contexte des cartes à microprocesseur. Le système (la carte) dispose alors d'une structure de données pour définir les différentes classes des variables (attributs, classes, signature des méthodes...) ainsi que d'une description du type des constantes, des arguments et des variables locales manipulées par le traitement. Cette structure de donnée

est statique, dans le sens ou lorsqu'un type est établit il ne peut plus changer. Notons L cette structure. Vérifier un programme consiste à contrôler que ses instructions (une à une et indépendamment les unes des autres) sont correctes. Une instruction `JumpIf v,1x` est correcte si $v \in L$ et $L[v] \subseteq$ `CardBool` ; sinon l'opération est illégale vis-à-vis de l'usage des types.

L'intérêt des variables T est de permettre d'optimiser la quantité de mémoire de travail que nécessite un traitement pour s'exécuter. Une même zone du contexte d'exécution du traitement (placée en RAM, si précieuse dans le contexte de la carte à microprocesseur) pourra être utilisée pour stocker un `CardShort`, puis plus tard un `CardBuffer256`, alors qu'il fallait deux zones mémoire pour gérer ces deux types avec exclusivement des variables L. Cependant introduire des variables T complique le contrôle des types. En fait pour vérifier qu'un programme est correctement typé nous cherchons à déterminer le types des variables T pour chaque instruction du programme. La définition informelle, dont les conséquences seront discutées dans le chapitre 7 (« Limite de l'approche »), d'un programme correctement typé est donc :

Définition :	Un programme est correctement typé s'il est possible de construire une table (notée TT) qui établit un type satisfaisant les règles de types associées à chaque variable T pour chaque opération élémentaire.

Valider un programme, revient à construire TT. Si la construction échoue le programme est jugé incorrect. La construction de TT est le résultat de l'évaluation des types determinée par la sémantique statique du langage FACADE.

4.3.4 Sémantique statique du langage FACADE

Il convient maintenant de définir plus précisément la manière dont doit être interprété un traitement par le moteur d'inférence.

L'interprète analyse un programme noté P dont la déclaration (type des arguments, classe associée et type de retour) est notée $MethodDsc$. $Dom(P)$ désigne le domaine du programme, c'est à dire l'ensemble des numéros d'instructions qui le compose. $MethodDsc[arg]$ désigne le type de l'argument noté arg alors que $MethodDsc[ReturnType]$ définit le type de retour déclaré par le traitement P. L'interprète dispose d'une structure de données pour les types fixés notés L. $L[v]$ désigne le type de la variable v tel qu'il a été déclaré une fois pour toutes (dans le cas des attributs d'une classe par exemple). De plus il utilise deux structures de travail essentielles. La première est la table TT qu'il établit. Je noterais par la suite $TT[pp][v]$ le type établi pour la variable T nommée `v` associée à l'instruction du programme numéro pp. Comme la construction de cette table se fait en plusieurs étapes $TT_e[pp][v]$ désignera $TT[pp][v]$ tel qu'il est établi à l'étape e. La seconde structure de donnée est une pile notée Spp qui sert à stocker les différents chemins à évaluer. Le moteur d'inférence parcourt tous les chemins que le programme peut suivre en débutant par la première instruction (qui est le point d'entrée du programme). Lorsqu'il rencontre

une instruction `JumpIf` ou `JumpList` c'est que plusieurs chemins pourront être suivit par le programme (lorsqu'il s'exécutera réellement). Il faut donc vérifier indépendamment chacun de ces chemins. La pile de pointeur de programme est utilisée à cet effet. Je la note Spp, Spp_e désigne l'état de cette pile à l'étape e. Elle contient des couples (pp_{Src},pp_{Dst}) où pp_{Src} est le numéro de l'instruction qui a été évaluée, et pp_{Dst} est le numéro de l'instruction ou l'évaluation (dynamique) de l'instruction numéro pp_{Src} peut aboutir. Le corps de l'interprète se décompose en six pas :

1. l'analyse du programme (on pourrait dire son exécution statique) se fait avec comme hypothèse de départ que $\forall(pp,x) \in (Dom(P),Dom(TT[1]));pp > 1 \Rightarrow TT_1[pp][x] = \perp$ car aucun type n'est connu pour les variables T initialement, sauf pour la première instruction pp. En ce qui concerne la première instruction, les variables T sont soient non-initialisées (cas des variables locales T), soit affectées (dans le cas des arguments) avec une valeur initiale dont le type est déclaré par le traitement. Dans le premier cas, le type de départ est donc $\top$, et le point d'entrée du programme est la première instruction de P donc :

$$\forall x \in Dom(TT[1]);TT_1[1][x] = \begin{cases} MethodDsc[x] \text{ si } x \text{ est un argument} \\ \top \text{ sinon} \end{cases} ; \quad (4.2)$$

2. le processus d'analyse peut commencer (l'étape $e = 1$) avec la première instruction du traitement (celle par laquelle commencera l'exécution réelle). Aucun autre chemin n'est à considérer initialement donc $Spp_1 = \{\}$;
3. à ce stade il faut interpréter l'instruction courante notée pp (la sémantique statique de chaque opération sera détaillée par la suite). Si les tests sont passés avec succès en utilisant pour type des variables $TT_e[pp]$ et L, le moteur d'inférence obtient en retour la liste des numéros des instructions qui peuvent être atteintes après l'exécution de l'instruction courante. Après une opération `Invok` c'est simplement l'instruction suivante qui sera exécutée, par contre, après une instruction `Jumplist`, en plus de l'instruction suivante, c'est l'ensemble des instructions marquées par les $labelId_x$ qui est concerné ;
4. la liste des chemins à explorer depuis l'instruction numéro pp est à ajouter à la pile des chemins déjà identifiés dans Spp_e. Si l'instruction est un `Invok` par exemple, l'information ajoutée à la pile est le couple (source, destination) : $(pp,pp+1)$;
5. l'étape e se termine. Un couple (pp_{Src},pp_{Dst}) est extrait de Spp_e. Pour que le programme soit correct il faut au préalable que $pp_{Dst} \in Dom(P)$, ensuite :
   Si $\forall x \in Dom(TT_e[pp_{Src}])$; $TT_e[pp_{Src}][x] \subseteq TT_e[pp_{Dst}][x]$ il n'est pas utile de réévaluer le chemin puisqu'il a déjà été évalué avec des hypothèses sur les types des variables T comparables par d'autres itérations.
   Sinon, c'est-à-dire si l'un des types évalué sur l'instruction pp_{Dst} est plus petit ou même incomparable avec celui précédemment évalué pour pp_{Src}, la ligne doit être réévaluée. Dans ce cas, $pp_{e+1} = Spp_{Dst}$ avec de nouvelles classes utilisées pour tester l'instruction déterminée par la relation :

$$\forall x \in Dom(TT[pp_{Src}]); TT_{e+1}[pp_{Dst}][x] = TT_e[pp_{Src}][x] \cap TT_e[pp_{Dst}][x]$$

L'opérateur $\cap$ permet de trouver les plus petits ancêtres communs entre les types des variables de $TT_e[pp_{Src}]$ tels qu'ils avaient déjà été identifiés, et ceux qui proviennent du nouveau point de saut $TT_e[pp_{Dst}]$. Après cette opération, nous avons la garantie que $TT_{e+1}[pp_{Dst}][x] \subseteq TT_e[pp_{Src}][x]$ donc nous évaluons un chemin avec des types conformes à ceux qui ont été déterminés par les chemins précédents; mais aussi que $TT_{e+1}[pp_{Dst}][x] \subseteq TT_{e+1}[pp_{Dst}][x]$, ce qui implique que les types retenus soient comparables à ceux identifiés par le nouveau chemin considéré;

6. Tant qu'il existe une instruction pp_{Dst} dans Spp_e associée à des types pour les variables T qui n'ont pas été considérées jusqu'à présent le processus d'analyse boucle au point 3 de l'algorithme en passant à l'étape e à l'étape $e+1$.

Il faut maintenant définir pour chacune des cinq instructions de FACADE (dont la sémantique dynamique a été définie par la section 4.2.3 page 88) les tests que l'interprète effectue (pour le troisième point de l'algorithme) en fonction de l'instruction associée à *pp*. Cela constitue la sémantique statique du langage.

1. `Return` *[VarRes]*
   L'opération est correcte si la variable retournée VarRes est une sorte du type de retour déclaré par le traitement. Cette instruction n'est suivie d'aucune autre car elle termine l'exécution d'un traitement, aussi l'évaluation statique de cette opération ne retourne aucun couple (pp_{Src},pp_{Dst});
2. `Jump` *LabelId*
   Cette opération n'a de sens que si le point de saut *LabelId* existe et qu'il est associé à une valeur de point de saut noté $LabelDsc[LabelId]$ qui correspond bien à une instruction du traitement, c'est à dire tel que $LabelDsc[LabelId] \in Dom(P)$. L'instruction suivante à évaluer est celle pointée par *LabelId*. Le seul couple (pp_{Src},pp_{Dst}) doit donc être empilé dans la pile des sauts: $Spp_{e+1} = Spp_e \cup (pp,LabelDsc[LabelId])$;
3. `Jumpif` *Var LabelId*
   Si la variable *Var* est une sorte de `CardBool` et si le point de saut *LabelId* est correctement construit (*c.f.* voir l'opération `Jump`), cette opération est correcte. Deux instructions pourront ensuite être exécutées, selon que le programme saute à *LabelId* ou pas. Deux couples sont donc à ajouter dans Spp_{e+1}. Il s'agit de $(pp,pp+1)$ et de $(pp,LabelDsc[LabelId])$;
4. `Jumplist` *Var nbCase* { $LabelId_1$, $LabelId_2$, ... }
   Cette opération peut être exécutée si le type de la variable *Var* est une sorte de `CardByte`, et si les points de sauts $LabelId_x$ existent $\forall x \in [1 \ldots nbCase]$. Toutes les instructions pointées par des $LabelId_x$ doivent être évaluées ainsi que celle qui suit le `Jumplist`. Les couples de chemins sortants sont donc définis par l'ensemble $(pp,pp+1); (pp,LabelId_1); (pp,LabelId_2); \ldots$;
5. `Invok` *Var methodId [VarRes]* { $tabVar_1$, $tabVar_2$, ... }
   L'évaluation d'une opération `invok` est correcte si (a), la méthode *methodId* est

déclarée par la classe (le type) de la variable *var*, et si (b), les types des variables $tabVar_x$ passés en paramètres sont des sortes des types déclarés attendu en paramètre par la méthode *methodId* de la classe de *Var*, et si (c), le type de retour de *methodId* est une sorte de type de la variable *VarRes*. Le troisième test n'a de sens que si la variable *VarRes* n'est pas une variable T. Dans le cas contraire peut importe le type de la variable *VarRes*, mais l'évaluation statique de l'opération `Invok` modifie l'information de type de la variable *VarRes*. En effet, après l'exécution de cette opération la variables est affecté avec une valeur du type de retour de la méthode *methodID*. Cette opération entraîne par séquencement l'évaluation de l'instruction suivante $Spp_{e+1} = Spp_e \cup (pp,pp+1)$.

Pour appuyer cette description regardons en détail un exemple d'inférence de type sur FACADE.

4.3.5 Exemple d'inférence de type sur FACADE

La table 4.2 donne l'exemple d'un petit traitement exprimé avec la langage intermédiaire FACADE (forme déassemblée pour rester lisible) qui effectue simplement 100 itérations avant de terminer. Ce programme possède deux points de sauts: `Label0` et `Label1`. Le `Label1` est utilisé comme étiquette de boucle. Le test `i < 100` (instruction n°4) est utilisé par le `Jumpif` de la cinquième instruction pour implanter la boucle. Alors que `Label0` permet de vérifier une première fois la condition de sortie avant de « boucler » sur `Label1`. En fait, ce programme FACADE est adapté à partir d'une méthode de classe Java dont le corps du programme source est { `for(i=0;i<100;i++) ; return;` }.

pp	Label	Instruction	Commentaire
1		`v1 ← 0 asIs`	v1 prend 0
2		`Jump Label0`	branchement au teste marqué par le `Label0`
3	`Label1`	`v1 ← v1 + 1`	incrément de v1
4	`Label0`	`v2 ← v2 < 100`	test de v1 inférieur ou égal à 100
5		`Jumpif v2,Label1`	si Vrai saut en boucle marqué par le `Label1`
6		`Return Void`	sinon Fin

TAB. 4.2 – *Un exemple de code FACADE*

L'inférence des types des variables `v1` et `v2` sur cet exemple est réalisée par l'algorithme proposé précédemment en huit étapes présentées par la table 4.3:

L'analyse commence par la première instruction. Le résultat de l'affectation est de définir `v1` (une variable T) comme un `CardByte`, l'opération en elle même est correctement typée.

L'instruction suivante est un saut inconditionnel qui entraîne de moteur d'inférence à l'instruction n°4, avec comme hypothèses sur T celles qui ont été établies par la première instruction.

pas	*pp*	Label	Instruction	$T[v1,v2]$	Pile de *pp*
1	1		`v1 ← 0 asIs`	$(\top,\top)$	{}
2	2		`Jump Label0`	(`CardByte`,$\top$)	{}
3	4	`Label0`	`v2 ← v1 < 100`	(`CardByte`,$\top$)	{}
4	5		`Jumpif v2,Label1`	(`CardByte`,`CardBool`)	$\rightarrow^a${(5,3)}
5	6		`Retun Void`	(`CardByte`,`CardBool`)	$\leftarrow^b${(5,3)}
6	3	`Label1`	`v1 ← v1 + 1`	(`CardByte`,`CardBool`)	{}
7	4	`Label0`	`v2 ← v1 < 100`	(`CardByte`,`CardBool`)	$\vdash^c${}

TAB. 4.3 – *Un exemple de code FACADE*

[a] Deux chemins possibles, évaluation de l'instruction 6 et mémorisation de l'instruction 3.

[b] Chemin courant terminé, d'autres chemins sont encore dans la pile.

[c] Chemin courant terminé car l'instruction a été évaluée avec des conditions plus larges sur v (pas n°4). Fin de l'analyse, plus de chemins.

La quatrième opération charge un booléen dans `v2`. Comme `v1` est un `CardByte` elle est correcte. A ce stade trois opérations ont été évaluées.

Le programme se poursuit par un `Jumpif v2,Label1`, or `v2` est un `CardBool` ce qui est correct. Deux chemins différents peuvent être suivis selon l'interprétation dynamique de l'opération `Jumpif`. Si le booléen est `Vrai` alors le traitement continue à l'instruction n°3, sinon il continu à l'instruction n°6. L'analyseur conserve l'un des chemins dans la pile des chemins à valider (le branchement à la troisième instruction en l'occurrence) et il poursuit avec l'autre.

La sixième instruction est un `Return` correct ou pas en fonction de la déclaration du type retourné par le traitement. En l'occurrence elle est correcte. `Return` termine la séquence des instructions du chemin en cours d'évaluation. Il reste cependant à interpréter l'autre chemin, présent dans la pile, qui consistait à effectuer, depuis l'opération n°5, un branchement à l'opération n°3.

Cette opération est évaluée avec pour classes de `v1` et `v2` celles établies par l'instruction de branchement (ici l'instruction n°5). Il s'agit donc d'un `CardByte` et d'un `CardBool` pour `v1` et `v2`. L'opération n°3 (incrémentation de `v1`) est correcte car `v1` est un `CardByte`. Une fois cette instruction évaluée l'analyse se poursuit par l'opération suivante.

Celle-ci a déjà été testée précédemment (étape n°3 du processus d'analyse). Deux cas sont possibles. Soit (i) tous les types des variables sont des sortes de ceux déjà évalués; dans ce cas il est inutile de poursuivre l'analyse de ce chemin elle a déjà été réalisée. Soit (ii) les types déjà évalués sont des sortes de (voir des incompatibles avec) ceux avec lesquels l'analyse atteint maintenant ce point de programme; dans ce cas il faut recommencer le contrôle des instructions en prenant pour classes des variables T à l'étape n°7 (notés T_7) celles définies comme les premiers pères communs des différents chemins. Formellement on écrit $\forall i \in T; T_7[i \leftarrow (T_3[i] \cap T_6[i])]$. Sur cet exemple, il est évident que nous sommes dans le cas (i) : `CardByte` $\subseteq$ `CardByte` et `CardBool` $\subseteq \top$. Le chemin en cours d'analyse est terminé. Il ne reste plus aucun chemin à analyser dans la pile des chemins. Le processus

est terminé.

Nous avons déjà souligné l'importance que nous accordions au fait que la carte assure elle-même sa sécurité (*c.f.* Chapitre 3 section 3.2.4 page 63). Dans notre architecture, pour qu'il ait un sens ce processus de vérification doit être placé dans une carte.

4.3.6 Dans une carte

Il n'est guère possible de placer tel quel le contrôle de type dans la carte. La section 2.4.4 précisait que la complexité en espace et en temps était trop élevée pour cela. En fait, plus que le temps, c'est l'espace dans la mémoire de travail de la carte qui fait défaut. L'algorithme présenté par la section 4.3.3 manipule une structure de données pour déterminer le type des variables T pour chaque point du programme. Cette structure représente un espace de $Card(P) \times Card(T)$ cellules mémoires. Pour un code qui fait une centaine d'opérations élémentaires FACADE et qui manipule une dizaine de variables T, cela représente environ deux kilo-octets de mémoire qui seront accédés aussi bien en lecture qu'en écriture (lors de l'exécution des opérations $\cap$). La maquette sur laquelle a été réalisé le prototype dispose en tout de 1536 octets de mémoire de travail. Il faut y placer la pile d'exécution les structures système de gestion de la mémoire, les variables système, ... Il est donc évident que l'inférence de type devient très rapidement trop complexe pour être implantée dans la carte.

L'un des objectifs essentiels de FACADE est de parvenir à simplifier la tâche de vérification de telle sorte que l'algorithme utilisé puisse être encarté malgré les limitations drastiques de la mémoire de travail. Il convient de raisonner autrement pour que le langage FACADE permette de réduire l'inférence de type à un traitement de flot.

4.4 Réduire l'inférence de type à un traitement de flot

4.4.1 Motivations

Dans le contexte de la carte à puce, l'avantage évident d'une analyse en traitement de flot (c'est-à-dire d'une analyse qui se contente de décoder le code FACADE dans l'ordre où il est donné, opération par opération, sans avoir à revenir sur des opérations déjà analysées) est de limiter l'usage de la mémoire de travail. La solution que nous avons retenue est basée sur le principe d'un programme comprenant sa preuve de type.

4.4.2 Principes d'un programme comprenant sa preuve de type

La structure de donnée qui permet de calculer le type chaque variables T du programme à vérifier pour chaque instruction élémentaire constitue, lorsqu'elle est établie, une « preuve » que le programme est correctement typé. Comme bien souvent, établir la

preuve est plus complexe que de la vérifier. Pour établir le type d'une variable à un point de programme, il a fallu parcourir les différents chemins du programme et utiliser l'opérateur $\cap$. Vérifier un programme consiste simplement à contrôler que la valeur proposée par la preuve est correcte vis à vis des règles de type associées à l'instruction courante. formellement la génération de la preuve fait l'opération $TT_{e+1}[pp][i] = TT_e[pp][i] \cap TT_e[jp][i]$ avec jp numéro d'une instruction de saut suceptible de brancher à pp, alors que le vérifieur se contente de valider une preuve. Je note cette preuve TT_{ext}. Elle se présente sous la forme d'une table qui associe pour chaque point du programme, à chaque variable T une classe (un type). La table TT_{ext} est pour sa part calculée sur un système informatique traditionnel (pour lequel la tâche est réalisable). Le nombre d'étapes du vérifieur est donc strictement égal au nombre de pas du programme. En fait, la carte s'assure que la preuve est correcte en vérifiant simplement sa*continuité*, c'est-à-dire en vérifiant que les informations fournies par la preuve sont correctes pour les opérations qui peuvent être atteintes par séquence à partir du point courant. Formellement cette opération n'implique que l'opérateur $\subseteq$.

L'avantage de la vérification est double: (i) l'inférence de type n'a plus besoin d'accéder en écriture à la table qui représente la preuve; et (ii) il est possible de tester chaque instruction sans « suivre » les différents chemins du programme vérifié. L'intérêt du premier point est de permettre l'utilisation de la mémoire EEPROM comme mémoire de stockage temporaire pour la preuve sans pénaliser l'exécution de l'algorithme. Comme cette mémoire est présente en quantité plus importante, la taille de la preuve est de moindre importance. Le deuxième point permet de vérifier le programme en flux, c'est-à-dire instruction par instruction, sans avoir à sauter d'instructions en instructions au grès du flot de contrôle du programme. En fait il est possible de tirer un meilleur parti de ces deux propriétés tel que le montre l'implantation du vérifieur de code FACADE.

4.4.3 Vérifieur de code FACADE

La vérification des instructions FACADE, lorsque la preuve est disponible est extrêmement simple. Il suffit de vérifier que chaque instruction du programme est correctement typée avec comme celle proposée par la preuve. Ensuite il faut vérifier que les types déclarés pour les instructions immédiatement suivantes (par exemple la liste des instructions référencées par un `Jumplist`) sont cohérents avec les types de l'instruction courantes. (on vérifie que $\forall v; TT_{ext}[pp][v] \subseteq TT_{ext}[sp]$ avec $\forall sp$ dans l'ensemble des numéros d'instructions qui peuvent suivre pp). Si l'opération courante (dont le numéro est pp) engendre une modification de l'état des types des variables T les opérations (numérotées sp) doivent reporter ce changement. C'est ce qui a été appelé vérification de *continuité* de la preuve.

l'opération `i <- i + j` avec comme élément de preuve $TT_{ext}[n][i] \subseteq$ `CardByte` (ou n est le numéro de l'opération dans la séquence) est correcte si $TT_{ext}[n][j] \subseteq$ `CardByte`. Mais il faut encore qu'à la ligne suivante i soit `CardByte` $\subseteq TT_{ext}[n+1][i]$. Alors il a été vérifié que (1) l'opération était correcte vue la nature des types obtenus et (2) la preuve pour les opérations suivantes est correcte vis-à-vis de la preuve validée pour l'opération courante (je dirai par la suite qu'il faut vérifier la continuité de la preuve). Vérifier que l'opération

suivante, par exemple `Return i`, est correcte n'est pas plus difficile. Cette opération est correcte si $TT_{ext}[i] \subseteq MethodDsc[ReturnType]$. Supposons que la méthode à déclarer retournait un `CardByte` alors l'opération est correcte. Elle n'implique cependant aucune hypothèse sur les types des variables pour l'opération suivante. En effet, `Return` terminera l'exécution du traitement. S'il existe une autre opération après celle-là, elle ne sera atteinte que par un saut de programme. Donc quelque soit les types que la preuve qui suivent un `Return`, ils sont satisfaisants (par rapport à l'instruction précédente). La question que soulève cette remarque est : « Qu'est-ce qui assure que la proposition faite par la preuve à ce point du programme est correcte? ». Si ce point du programme peut être atteint, c'est qu'une opération élémentaire de saut le référence. Dans ce cas, la vérification de la continuité de la preuve sera contrôlée lorsque les opérations de saut susceptibles d'atteindre ce point seront vérifiées. Autrement dit, l'algorithme de vérification s'assure que les types des variables **à** un point de saut (une instruction référencée par une opération de saut) sont des sortes de ceux qui sont établis **depuis** l'instruction de saut. Supposons que l'instruction qui suit le `Return` déclare dans sa preuve attendre $TT_{ext}[n+2][i] \subseteq$ `CardBool` alors si une instruction numérotée m est un `JumpIf xx, n+2` elle n'est correcte que si $TT_{ext}[m][xx] \subseteq$ `CardBool` (car la variable `xx` doit être un `CardBool`) mais aussi que `CardBool` $\subseteq P[n+2][i]$ car depuis l'instruction m la variable `xx` est un `CardBool` et après le saut le contenu de cette variable ne sera pas modifiée. Dans le cas contraire, le test de continuité de la preuve échoue. Le programme est irrecevable (même si en fait il est correct) car sa preuve ne respecte pas le principe de continuité des types.

Pour clarifier ce fonctionnement prenons un exemple d'inférence assistée.

4.4.4 Exemple d'inférence assistée

La table 4.4 présente un extrait de programme et, à sa droite, sa preuve. Le processus de vérification se fait linéairement, sans tenir compte du flot de contrôle du programme en commençant par l'opération n°1 et en terminant par l'opération n°6. Détaillons l'activité du vérifieur, instruction par instruction :

pp	Label	Instruction	Preuve (avant exécution) : $TT_{ext}[v1,v2]$
1		`v1 ← 0 asIs`	($\top$,$\top$)
2		`Jump Label0`	(`CardByte`,$\top$)
3	`Label1`	`v1 ← v1 + 1`	(`CardByte`,`CardBool`)
4	`Label0`	`v2 ← v1 < 100`	(`CardByte`,$\top$)
5		`JumpIf v2,Label1`	(`CardByte`,`CardBool`)
6		`Return void`	(`CardByte`,`CardBool`)

TAB. 4.4 – *Un code FACADE avec sa Preuve*

Au point de départ du programme le type des variables de travail doit être $\top$ car elles ne sont pas initialisées (et donc aucune opération n'est possible sur elles). Ensuite l'opération

n°1 est évaluée (avec la sémantique statique encartée). Cette opération est correcte, la constante `0` est `CardByte`, l'opération `asIs` est définie, elle retourne un `CardByte`. Le résultat est que `v1` devient un `CardByte`. L'inférence vérifie que pour l'instruction suivante la variable `v1` $\subseteq$ `CardByte`. Or $TT_{ext}[2][v1] \subseteq$ `CardByte`. En fait, cette vérification doit être faite pour toutes les variables de la preuve, car si l'opération n°1 a modifié le type de `v1`, il est vrai aussi qu'elle n'a pas modifié le type de `v2` et il ne doit donc pas pouvoir être utilisé comme un objet qu'il n'est pas. Formellement le test de continuité de la preuve en ce point s'écrit ici : $\forall v$ une variable T du code ; $TT_{ext}[1][v] \subseteq TT_{ext}[2][v]$. La continuité de la preuve pour la première ligne est correcte. L'interprétation statique de l'instruction suivante peut avoir lieu.

L'instruction n°2 est un saut inconditionnel à l'instruction n°4. Il faut donc vérifier que les classes validées pour les variables `v1` et `v2` au niveau de l'instruction n°2 sont des sous-classes de celles retenues pour l'opération n°4 (test de continuité). Le test est satisfait. L'opération est correcte. Le vérifieur de code passe ensuite à la troisième opération (et non à la quatrième comme le saut l'indiquait). Cependant un `Jump` est inconditionnel, il n'implique donc aucune contrainte sur l'état de la preuve pour l'instruction suivante. La classe de chaque variable peut être prise comme vraie quelle qu'elle soit.

L'opération n°3 est une addition, elle prend deux `CardByte` (ici `v1` et `1`) d'après la preuve `v1` est un `CardByte`. L'opération est correcte, elle retourne un `CardByte` dans la variable `v1` pour l'instruction suivante. La vérification de la continuité ($\forall v$; $TT_{ext}[3][v] \subseteq TT_{ext}[4][v]$) est correcte. Cependant, la preuve déclare que `v2` considéré comme un `CardBool` dans le passé doit être vue comme un $\top$ à ce point, ce qui est correct (`CardBool` $\subseteq \top$) mais qui interdit l'usage (pas l'affectation) du booléen placé dans `v2` pour l'instruction n°4.

L'opération n°4 affecte justement la variable `v2` avec un booléen, fruit de l'opération correctement typée $<$ qui est réalisée entre deux variables `CardByte`. La preuve associée à la cinquième opération conserve $TT_{ext}[5][v2] =$ `cardBool`. Elle aurait aussi été correcte si $TT_{ext}[5][v2] =$ `cardValue` ou si $TT_{ext}[5][v2] =$ `cardTop` car ses deux classes sont des sortes de `CardBool`.

Le `JumpIf` qui utilise cette variable pour effectuer le test est donc correct à l'instruction n°5. Par séquence il peut atteindre soit la ligne marquée du label `Label1` soit la ligne n°6 (si le booléen est faux). Il faut donc vérifier que $\forall v \in TT_{ext}[1]; TT_{ext}[5][v] \subseteq TT_{ext}[6][v] \wedge TT_{ext}[5][v] \subseteq TT_{ext}[3][v]$. Les tests de continuité sont corrects. Il reste une dernière instruction.

Il s'agit de `Return` qui ne retourne rien (conformément à la déclaration du traitement...). Notons que la dernière instruction ne peut être qu'un `Return` ou un `Jump`, de telle sorte que par séquence le programme ne puisse pas sortir de son domaine de définition (c'est-à-dire exécuter des instructions au delà de celles qu'il a défini) ce qui aurait été le cas avec une opération `Invok` en ligne 6.

Ainsi se termine la vérification d'un code correct. Le mécanisme de vérification est effectivement plus simple que la preuve car il n'a pas eu besoin de l'opération $\cap$ qui est assez complexe, et il a pu analyser le code linéairement sans suivre les chemins d'exécution du code. Sa complexité est en $O(Card(P))$. Un problème reste néanmoins préoccupant. La

taille de la preuve est de $Card(P) \times Card(T)$. Cette structure doit être chargée avant le traitement car elle est utilisée non seulement au point de programme courant (la preuve de la ligne pp est nécessaire pour vérifier l'opération $P(pp)$. Mais aussi, la preuve de la ligne suivante et la preuve de tous les points de branchements pour les vérifications de continuité de preuve. Elle peut bien sûr être écrite en EEPROM car elle n'est jamais plus accédée en écriture par le vérifieur, mais ce n'est guère élégant. Il convient dont de minimiser la taille de la preuve.

4.4.5 Minimaliser la taille de la preuve

La première remarque est que dans un traitement FACADE le vérifieur accomplit pour les instructions `Invok` la même tâche que le contrôleur (générateur de preuve). Il détermine la variable affectée et si c'est une variable T il note que le type de cette variable au prochain pas de programme a changé. Ce qui est « simplifié » dans le vérifieur, par rapport au contrôleur, est le traitement des opérations de saut ainsi que l'inférence de type au point de saut recouvrant tous les chemins possibles. Si l'on résume la preuve à la seule valeur des types des variables pour les points de saut, le vérifieur de code encarté pourra établir seul les types des d'opérations séquentielles (les séquences d'`Invok`) en utilisant simplement une petite structure de donnée qui contient le type des variables T au point courant de vérification (je la noterai la table T). C'est finalement ce que nous nous sommes proposé de faire.

Reprenons l'exemple précédent (section 4.4, page 99). Initialement la structure des T est définie de la même manière que la table $TT[1]$ a été initialisée (*c.f.* équation 4.2, page 93). Notre structure de temporaire T est ensuite modifiée par la première instruction, cette opération modifie la variable `v1`. Nous modifions la table T en conséquence $T[v1 \leftarrow \texttt{CardByte}]$. La deuxième instruction est vérifiée. Elle est correcte si les types de la table T sont des sortes de ceux associés au label `Label0`. Sur notre exemple, c'est le cas. La table T est réinitialisée avec la valeur du label associée à l'instruction suivante. L'opération n°4 peut être vérifiée. Un label lui est associé. Il déclare $T[v1] \subseteq \texttt{CardByte} \wedge T[v2] \subseteq \texttt{CardTop}$, ce qui est correct vis-à-vis de l'état courant de T. Elle modifie T en chargeant un `CardBool` dans `v2` : $T[v2 \leftarrow \texttt{CardBool}]$. L'opération n°5 est ensuite vérifiée. Il s'agit du `JumpIf` susceptible de brancher à la ligne marquée `Label1`. A ce point du programme, les types de la structure T sont des sortes de ceux déclarés par la preuve pour `label1`. La vérification peut se poursuivre. Il ne reste plus que le `Return void` qui est correct aux vues des déclarations du traitement.

La vérification terminée n'a eu besoin que de l'information de deux lignes de la preuve : celles associées aux points de saut (`Label0` et `Label1`). Le bénéfice en taille de preuve est important. Lors du chargement d'un traitement, l'état des types des variables T associé à chaque label est tout d'abord donné. C'est cet élément d'information qui justifie que la déclaration de la table des labels soit fournie dans les entêtes des codes FACADE (*c.f.* section 4.2.1 page 86).

Il est encore possible de réduire la taille de la preuve. Sur des expériences basées sur des

exemples réels (extrait en l'occurrence à partir des exemples d'applets JavaCard) on peut remarquer qu'il est fréquent que les types des variables T soient identiques pour plusieurs labels. Il semble donc raisonnable de placer un niveau d'indirection entre le numéro du label et la liste des types des variables temporaires. Ainsi deux labels qui possèdent les mêmes états de pointeurs référencent la même liste de type, ce qui réduit d'autant le nombre total d'octets transmis à la carte. C'est cette structure de preuve que nous avons finalement choisie :

1. nombre d'instructions à lire (p) ;
2. nombre de points de saut (n) ;
3. nombre de descripteurs de points de saut(d) ;
4. numéro de l'instruction du premier point de saut $(< p)$;
5. numéro de descripteur associé $(< d)$;
6. numéro de l'instruction du deuxième point de saut $(< p)$;
7. numéro de descripteur associé $(< d)$;
8. $\ldots \times n$;
9. listes des types des variables T associées au premier descripteur ;
10. listes des types des variables T associées au deuxième descripteur ;
11. $\ldots \times d$;

Et c'est elle qui est utilisée pour réaliser la formalisation de l'inférence de type de FACADE.

4.5 Formalisation de l'inférence de type de FACADE

4.5.1 Un langage typé prouvé

Les systèmes de types se prêtent volontiers à une formalisation. L'intérêt de cette formalisation est de permettre de démontrer que le système de type utilisé est correct, c'est-à-dire qu'il ne reconnaît pas comme valide un programme qui utilise incorrectement les données auxquelles il a accès. La réalisation de cette démonstration sur le langage intermédiaire FACADE a été effectué à l'aide de la méthode B [Abr96]. Pour pouvoir utiliser le langage formel B pour modéliser le mécanisme d'inférence de type et ainsi prouver son bon fonctionnement, le travail a été décomposé en deux parties. Dans un premier temps une formalisation des propriétés statiques du langage FACADE a été proposée. L'article [GLV99] rédigé conjointement avec J.J. Vandewalle et J.L. Lanet détaille cette première étape. A partir de cette formalisation un modèle B et une preuve du mécanisme a pu être obtenu [RCG00]. Tout le mérite de cette deuxième étape revient à A. Requet et L. Casset. Le reste de cette section décrit la démarche que nous avons adoptée pour formaliser la sémantique statique de FACADE.

4.5.2 formaliser la sémantique statique de FACADE

Avant de donner le détail formel de la sémantique statique qui a été utilisée il est nécessaire d'introduire brièvement la notation utilisée. Les types ont déjà été formalisés dans la section 4.3.2, page 90. Tout d'abord, le système encarté maintient en permanence quelques tables: il s'agit d'une part d'une table qui définit la liste des variables notées *classDsc* et d'autre part d'une liste des méthodes utilisables *methodDsc*. La table des classes est l'image du diagramme de classe déjà présenté alors que la table des méthodes définit pour chaque classe la signature des opérations qui sont acceptées. Lorsqu'une nouvelle méthode est ajoutée (c'est-à-dire lorsqu'elle a été vérifiée par la carte) sa signature est ajoutée à l'ensemble *methodDsc*. $\forall pp \in Dom(P); P[pp]$ est la valeur de P au point de programme pp. Le modèle de notre système est donné par le tuple: $< pp,Tmp >$ où pp représente un point du programme P et Tmp les variables T. Le type des variables L qui ne change jamais, ne fait pas partie de l'état courant du vérifieur encarté. Le vecteur VT représente les informations statiques (qui ne sont jamais modifiées) relatives aux types des variables T. Le vecteur VT assigne les types pour les variables T tel que $\forall i,(T[i] : VT[i])$ qu'on lit « Quelque soit i la variable T i est du type $VT[i]$ ». VT_{pp} désigne l'état des types tel qu'il est déterminé par le vérifieur pour l'opération numérotée pp et $VT_{pp}[i \leftarrow C]$ définit la table VT dans laquelle la variable i est associée à la classe C. Enfin le type de la variable nommée i tel qu'il est défini par la preuve de programme pour le label $LabelId$ s'écrit $LabelDsc[LabelId][i]$ et $\forall pp \in LabelDsc$ définit l'ensemble des points de programmes pp associés à un point de saut.

Un programme P est vérifié « correct » si la condition initiale est satisfaite et si toutes les instructions de P sont correctes vis-à-vis de leurs définitions statiques (de vérification):

$$\frac{Dom(VT_1) = \{\} \quad \forall pp \in Dom(P),(VT,pp \vdash P)}{VT \vdash P}$$

De plus, afin de définir $VT,pp \vdash P$ en fonction de l'opération traitée, nous aurons parfois besoin d'une autre propriété: $(VT,pp \vdash LabelDsc)$. Le but est de s'assurer que lorsque par séquence, l'exécution d'une instruction n°pp entraîne l'exécution de l'instruction suivante n°$(pp+1)$(Ce qui est le cas du `Jumpif`, du `Jumplist` et du `Invok`) alors, premièrement cette instruction existe (on n'entraîne pas l'exécution du traitement hors de ce qui est définit par P), et deuxièmement, si elle est liée à un point de saut (dans $LabelDsc$) alors la définition des types associés à ce point de saut est correct vis-à-vis de ce qui a été déterminé par la ligne précédente. Dans ce dernier pas, le type des variables considéré par la suite est celui qui est définit par $LabelDsc$.

$$\frac{\begin{array}{c} pp+1 \in Dom(P) \\ (pp+1) \in Dom(LabelDsc) \\ \forall v \in VT \Rightarrow VT_{pp}[v] \subseteq LabelDsc[pp+1][v] \\ \forall v \in VT \Rightarrow VT_{pp+1}[v] = VT_{pp}[v \leftarrow LabelDsc[pp+1][v]] \end{array}}{VT,pp \vdash LabelDsc} \quad (4.3)$$

$$\frac{\begin{array}{c} pp+1 \in Dom(P) \\ \forall l \in LabelDsc\ ; pp+1 \neq LabelDsc[l] \end{array}}{VT,pp \vdash LabelDsc} \tag{4.4}$$

Il est maintenant possible d'exprimer la définition formelle de la sémantique statique des cinq instructions de FACADE tel qu'elle est traitée par la carte.

`Return` ***[VarRes]***

Deux équations sont nécessaires pour décrire complètement la sémantique statique de l'opération `Return`. Car pour que l'opération `Return` soit correctement utilisé il faut soit qu'elle soit suivie d'un point de branchement $LabelId_x$ (équation 4.6) soit qu'elle soit la dernière opération de P (équation 4.5).

$$\frac{\begin{array}{c} P[pp] = \texttt{Return}\ VarRes \\ VarRes : MethodDsc[ThisMethodId][ReturnType] \\ pp+1 \notin Dom(P) \end{array}}{VT,pp \vdash P} \tag{4.5}$$

$$\frac{\begin{array}{c} P[pp] = \texttt{Return}\ VarRes \\ VarRes : MethodDsc[ThisMethodId][ReturnType] \\ \forall v \in VT \Rightarrow VT_{pp}[v] \subseteq LabelDsc[pp+1][v] \\ \forall v \in VT \Rightarrow VT_{pp+1}[v] = VT_{pp}[v \leftarrow LabelDsc[pp+1][v]] \end{array}}{VT,pp \vdash P} \tag{4.6}$$

`Jump` ***LabelId***

De la même manière que l'instruction `Return` l'instruction `Jump` entraîne une rupture du flot. L'instruction qui suit un `Jump` ne que être atteinte que si un point de saut y est définit. C'est pourquoi il y a deux équations 4.7 et 4.8 pour définir complètement sa sémantique statique.

$$\frac{\begin{array}{c} P[pp] = \texttt{Jump}\ LabelId \\ LabelId \in Dom(P) \\ \forall i \in Dom(VT_{pp}); VT_{pp}[i] \subseteq LabelDsc[LabelId][i] \\ \exists l \in Dom(LabelDsc)/pp+1 = l \Rightarrow \forall v \in VT; VT_{pp+1} = VT_{pp}[v \rightarrow LabelDsc[l][v]] \end{array}}{VT,pp \vdash P} \tag{4.7}$$

$$\frac{\begin{array}{c} P[pp] = \texttt{Jump}\ LabelId \\ LabelId \in Dom(P) \\ \forall i \in Dom(VT_{pp}); VT_{pp}[i] \subseteq LabelDsc[LabelId][i] \\ pp+1 \notin Dom(P) \end{array}}{VT,i \vdash P} \tag{4.8}$$

`JumpIf` ***Var, LabelId***

Le vérifieur encarté doit pouvoir contrôler que Var est un `CardBool` pour que l'opération soit valide. Notons encore que cette opération ne peut être la dernière du traitement $i + 1 \in Dom(P)$. Dans le cas contraire, par séquence l'interprétation du programme pourrait continuer hors de son domaine de définition. L'équation 4.9 définit la sémantique statique retenue pour cette opération.

$$\frac{\begin{array}{c} P[pp] = \texttt{JumpIf}\ Var,LabelId \\ Var : \texttt{CardBool} \\ LabelId \in Dom(P) \\ \forall i \in Dom(VT_{pp}); VT_{pp}[i] \subseteq LabelDsc[LabelId][i] \\ LabelDsc,pp \vdash P \end{array}}{VT,pp \vdash P} \tag{4.9}$$

`JumpList` ***Var,nbLabel,{LabelId$_1$,LabelId$_2$,...}***

La sémantique statique de l'opération `JumpList` est proche de celle de `Jumpif`. Le type de la variable Var pris en paramètre doit être vérifiée (ce doit être une sorte de `CardByte`) et les points de saut possibles son ceux définis par les LabelId$_x$ ainsi que l'instruction suivante. L'équation 4.10 définit la sémantique statique retenue pour cette opération.

$$\frac{\begin{array}{c} P[pp] = \texttt{JumpList}\ Var,nbLabel,\{LabelId_1,LabelId_2,\ldots\} \\ Var : \texttt{CardByte} \\ \forall i \leq nbCase, LabelId_i \in Dom(P) \\ \forall l \in [1 \ldots nbCase]; \forall i \in Dom(VT_{pp}); VT_{pp}[i] \subseteq LabelDsc[LabelId_l][i] \\ pp + 1 \in Dom(P) \\ LabelDsc,pp \vdash P \end{array}}{VT,pp \vdash P} \tag{4.10}$$

`Invok` ***Var,methodId, VarRes,{VarP_1,VarP_2,...}***

Il faut distinguer deux interprétations statiques possibles pour l'opération `Invok` selon que la variable résultat $VarRes$ soit une variable T ou une variable L. Dans le premier cas l'ancien type de $VT[VarRes]$ n'est pas utilisé, mais après l'opération le nouveau type $VT[VarRes]$ sera celui retourné par la méthode $methodId$ de la classe de Var. C'est ce que décrit l'équation 4.11. Dans le second cas (équation 4.12) le type retourné doit être une sorte de celui déclaré pour la variable $L\ VarRes$.

$$\frac{\begin{array}{c} P[pp] = \texttt{Invok}\ Var,methodId,VarRes,\{VarP_1,VarP_2,\dots\} \\ MethodId \in MethodDsc_{ClassDsc[Var]} \\ VarRes \in VT \\ VT_{pp+1} = VT_{pp}[VarRes \rightarrow methodDsc[methodId][ReturnType]] \\ \forall i \in [1 \dots methodDsc[MethodId][nbVar]] \Rightarrow VarP_i \subseteq methodDsc[MethodId][i]; \\ pp+1 \in Dom(P) \\ LabelDsc,pp \vdash P \end{array}}{VT,pp \vdash P} \tag{4.11}$$

$$\frac{\begin{array}{c} P[pp] = \texttt{Invok}\ Var,methodId,VarRes,\{VarP_1,VarP_2,\dots\} \\ MethodId \in MethodDsc_{ClassDsc[Var]} \\ VarRes \in L \\ VT_{pp+1} = VT_{pp} \\ VarRes \subseteq methodDsc[methodId][ReturnType] \\ \forall i \in [1 \dots methodDsc[MethodId][nbVar]] \Rightarrow VarP_i \subseteq methodDsc[MethodId][i]; \\ pp+1 \in Dom(P) \\ LabelDsc,pp \vdash P \end{array}}{VT,pp \vdash P} \tag{4.12}$$

Grâce à ces éléments de formalisation du mécanisme d'inférence de type tel qu'il est implanté sur notre maquette, il devient possible d'utiliser une méthode formelle, en l'occurrence c'est la méthode B qui a été choisie, pour prouver le mécanisme de vérification.

4.5.3 Prouver le mécanisme de vérification

La méthode B est une méthode formelle pour le génie logiciel développée par J.R. Abrial[Abr96]. C'est une approche orientée modèle pour la construction de logiciel. Le concept de base est la *machine abstraite* qui est utilisée pour encapsuler les données qui décrivent l'état d'un système. Des invariants peuvent être exprimés sur l'état de la machine qui ne peut être accédée que par les opérations spécifiées.

La machine abstraite est successivement raffinée pour ajouter des détails de spécification. Un certain nombre d'étapes peuvent ainsi être utilisées avant d'atteindre le niveau de description d'une implantation (dans lequel la spécification est suffisamment détaillée pour qu'il soit possible de générer du code). Les raffinements et l'implantation ont d'autres invariants qui expriment des relations entre les états des différents niveaux de raffinement.

Le processus de preuve est un moyen de contrôler la cohérence entre le modèle mathématique, la machine abstraite et les raffinements. De cette manière, il est possible de prouver qu'une implantation est correcte vis-à-vis de ses spécifications. Le logiciel « Atelier B » génère des obligations de preuve pour la spécification en fonction du modèle mathématique.

Un prouveur de théorème automatique résout automatiquement les obligations de preuve, mais un système interactif permet à l'utilisateur de l'outil d'intervenir dans la production de la preuve.

L'architecture retenue pour la spécification formelle de ce mécanisme (figure 4.1) ainsi que toute la démarche pour finalement définir en B le mécanisme de vérification est très largement inspirée des travaux [LR98]. Sur cette architecture nous pouvons identifier les différents niveaux de raffinements du modèle de FACADE. Le premier niveau : *verifier* définit la propriété fondamentale du vérifieur que l'on peut résumer dans la phrase : « *Un programme accepté est un programme correctement typé* ». Ensuite différents niveaux de raffinements s'enchaînent. Ils décrivent successivement les différentes étapes de la vérification d'une instruction du programme jusqu'à distinguer les différentes règles associées à chaque instruction (et formalisées par les équations (4.5)(4.6) (4.8)(4.7) (4.9) (4.10) (4.11)(4.12)).

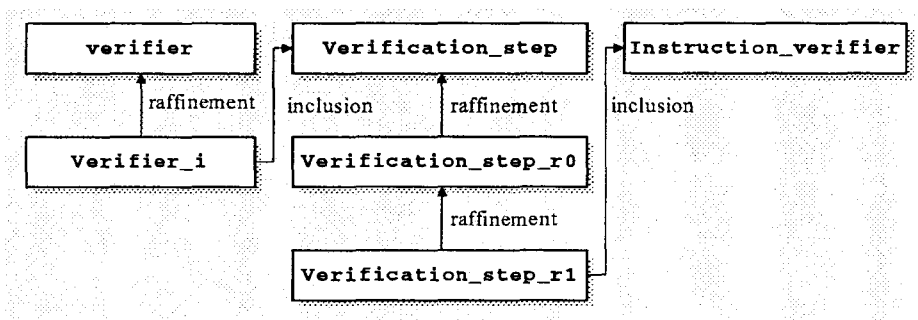


FIG. 4.1 – *Architecture de la spécification formelle en B de l'algorithme de vérification encarté de FACADE*

Au plus bas niveau, l'équation de l'opération `Jumpif` numéroté (4.9) a été transcrite par la spécification en B montrée par la figure 4.2. Sans avoir besoin de véritablement connaître le langage B on retrouve dans cette spécification les différents éléments de l'équation (4.9). Le fait, par exemple, que l'instruction `JumpIf` ne peut pas être la dernière instruction du programme est symbolisée par la ligne :

`vpp+1 ∈ dom(opcodes)`

La lecture de la spécification B nous montre que la gestion des descripteurs de point de saut (*LabelDsc*) a été intégrée, dans cette modélisation, à chaque opération du langage. C'est ce qu'indique les lignes :

```
JumpTarget(vpp) ∈ dom(labels) ∧
labels(jumpTarget(vpp)) ∈ compatibles(Tc) ...
```

4.6 Conclusion

La preuve obtenue grâce aux modèle B de la validité de l'algorithme placé dans la carte (preuve « assistée par ordinateur », dont la construction est présentée dans l'article[RCG00])

```
res → check_jumpif = PRE
    vpp ∈ dom(opcodes) ∧ opcodes(vpp) = JumpIf
  THEN
    SELECT JumpTarget(vpp) ∈ dom(labels) ∧
      labels(jumpTarget(vpp)) ∈ compatibles(Tc) ∧
      vpp + 1 ∈ dom(opcodes) ∧
      Bool ∈ ge(VarType(JumpVar(vpp),Tc)) ∧
      vpp + 1 ∈ dom(labels) ∧
      labels(vpp + 1) ∈ compatibles(Tc)
    THEN Tc := labels(vpp+1) ||
      res := TRUE || vpp := vpp + 1
    WHEN JumpTarget(vpp) ∈ dom(labels) ∧
      labels(jumpTarget)  ∈ compatibles(Tc) ∧
      vpp + 1  ∈ dom(opcodes) ∧
      Bool ∈ ge(VarType(JumpVar(vpp),Tc)) ∧
      vpp + 1 ∉ dom(labels)
    THEN res := TRUE || vpp := vpp + 1
    ELSE res := FALSE
    END
  END;
```

FIG. 4.2 – *Spécification B de l'opération de vérification de* Jumpif

permet d'affirmer que (i) l'algorithme termine, soit en ayant vérifié avec succès toutes les instructions, soit en ayant détecté une erreur, et (ii) l'algorithme retenu n'accepte pas un programme incorrect, même si les descriptions associées aux points de saut (*LabelDsc*) ont été délibérément faussées à dessin. Ainsi il est possible d'affirmer que le moteur d'inférence utilisé pour la sécurité-innocuité de notre maquette ne peut être mis en défaut.

Encore faut-il en comprendre le fonctionnement et les usages. Le système de type garantit au programmeur que ses objets, ne sont pas accessibles aux autres applications, sauf s'il leur donne explicitement accès (par le biais de méthodes de classes publiques). Une fois qu'un objet est donné à une autre application, toutes les opérations (publiques) qui sont définies sur cet objet pourront être sollicitées par l'application ou par une autre à laquelle elle aura retransmit l'objet. L'ordre des appels ne peut pas être garanti. Si l'objet partagé est une instance de la classe `CardMyFiles` par exemple, il est impossible de garantir que les méthodes `open` et `close` seront correctement utilisé. En supposant que le fichier partagé par ce moyen ne soit en fait transmit que pour que des lectures y soient réalisées, c'est à l'implantation des méthodes de `CardMyFiles` de vérifier que le fichier n'est pas utilisé autrement. Pour que la conception de la sécurité des applications ne soit pas trop difficile il faut que des outils de plus haut niveau soient proposés. L'utilisation de la notion de Capacité prend alors tout son sens dans la carte[HGV00, HV00].

Le langage FACADE ne définit pas en lui-même d'opérations élémentaires telle que des opérations arithmétiques et logiques ou des opérations d'accès à la mémoire. Toutes ces opérations, même celles qui pourraient apparaître les plus élémentaires, sont définies par l'intermédiaire d'une bibliothèque du noyau du système, une sorte de bibliothèque intime du matériel. L'avantage est de pouvoir faire évoluer facilement le cœur du langage. Il est plus simple de définir de nouvelles API que de redéfinir un jeu d'instructions élémentaires. Il est plus simple de proposer des programmes qui s'appuient sur de nouvelles bibliothèques que de changer les outils de base des programmeurs tel quel leurs compilateurs ou le chargeur encarté de code.

Chapitre 5

Chargeur encarté de code

« Il ne faut pas apprendre à écrire mais à voir. Écrire est une conséquence. [...]
Et les objets naissent de leur réaction en vous, Ils sont décrits profondément. »

Lettres à Rinette, in Œuvre complètes, A. de Saint-Exupery

Un prototype du micro-noyau encarté, tel qu'il est décrit dans le chapitre 3, a été réalisé sur un microprocesseur AVR[Atm00] qui est utilisé dans certaines cartes à microprocesseur. Ce chapitre explique comment, à partir d'un traitement exprimé dans le langage intermédiaire FACADE, le système encarté construit au cours du chargement dans la carte, un code machine performant. Le chapitre précédent a entièrement définit la sémantique statique du langage. L'objet de ce chapitre est d'expliquer le processus qui conduit à la production d'un code natif à partir du code FACADE. Bien sûr, dans le contexte de la carte, la plupart des techniques modernes d'optimisations du code machine ne peuvent pas être appliquées directement (les optimisation par fenêtres, les recherches d'antialiasing de pointeurs[Mit00], et même le cache de registres. ...). Néanmoins, la projection d'un programme FACADE vers du code natif est possible. Les résultats expérimentaux obtenus sur un premier prototype seront présentés dans le chapitre suivant. Le but, ici, est de comprendre la stratégie adoptée pour terminer l'acte de compilation dans la carte. Le détail de la génération de code est intimement lié au microprocesseur cible qui est comme nous l'avons dit, pour le prototype réalisé, un microprocesseur AVR. Cependant, pour l'essentiel, les problèmes rencontrés et résolus ici peuvent l'être de la même manière sur une large gamme de microprocesseur encartés « milieu de gamme ».

La première section rappelle les motivations qui nous ont amené à réaliser un compilateur à la volée dans une carte à microprocesseur. La suivante présente les différents aspects de la problématique de génération de code natif à partir d'un code FACADE. Ensuite l'architecture logicielle qui assure la génération de code est décrite en détail au travers de son implantation à partir d'un modèle objet du système. La quatrième partie détaille les techniques d'optimisations qui sont réalisables dans la carte, celles qui peuvent l'être avec

l'aide de l'extérieur, et les limites de la qualité du code généré. Enfin le chapitre se conclue en montrant comment l'architecture système peut être « reutilisée » par les composants d'extension des applications pour adapter les fonctions du générateur de code situé au cœur du système.

5.1 Générateur de code placé au cœur du système

L'introduction d'un moteur d'inférence dans la carte ne constitue, en somme, qu'un premier pas. Certes, la sécurité par les types, surtout lorsque ces types sont des classes, facilite l'interopérabilité des applications. Les programmeurs ont des garanties sur la manière dont les objets qu'ils partagent peuvent être utilisés. Il est vrai aussi que supprimer le contrôle de types dynamique améliore sensiblement l'efficacité du support d'exécution. Cependant le bénéfice de ce type d'optimisation de la machine virtuelle nous maintient encore bien loin de l'objectif d'efficacité qui nous permettra d'attendre les ambitions présentées par la section 3.1.

Selon les besoins le code transmis à la carte peut y être stocké, puis exécuté, de deux manières différentes.

En règle générale le code des applications n'a pas besoin d'être particulièrement performant. L'essentiel est qu'il soit compact afin de ne pas trop consommer de mémoire persistante (ressource mémoire la plus abondante mais malgré tout limitée dans ce milieu fortement contraint). Pour cela il est évident que l'interprétation d'un pseudo-code choisi de tel sorte qu'il soit compact est la plus attrayante.

Certaines routines systèmes par contre seront amenées à être évaluées très fréquemment. Il s'agit par exemple des routines d'accès à un système de gestion des fichiers spécialisé (qui gérerait par exemples les problèmes de mémoire recouvrable pour carte [DGL98]) ou encore d'un système de génération de nombres aléatoires avec des propriétés particulières pour l'implantation d'algorithmes cryptographiques [Fon98]. Dans ces cas, les traitements (les méthodes des classes systèmes) doivent s'exécuter très rapidement pour ne pas pénaliser l'application. Cet impératif prime alors sur la taille des programmes dans la mémoire. Ce type de besoin est malheureusement aujourd'hui complètement ignoré des systèmes d'exploitation et des machines virtuelles conçut pour les cartes à microprocesseur.

Pour pouvoir prendre en compte ces deux formes de code dans une même carte il faut gérer deux supports d'exécution différents : le microprocesseur et une machine virtuelle. L'objectif du chargeur de code est de pouvoir produire du code pour ces deux supports.

Cependant charger puis exécuter un code virtuel dans une carte à microprocesseur est déjà une opération courante aujourd'hui. L'aspect novateur de l'architecture Camille est sa capacité à générer un code natif performant qui est destiné à être exécuté directement par le microprocesseur. En conséquence nous nous sommes concentrés sur ce dernier point et il ne sera plus question dans la suite de ce chapitre que de génération de code natif. Quelques remarques devraient toutefois éclairer le lecteur qui s'intéresserait à l'utilisation de notre mécanisme pour générer du code destiné au support d'exécution virtuel.

Pour faciliter la compréhension du parallèle qui est établi entre génération de code natif et génération de code pour une machine virtuelle, il peut être utile de consulter quelques ouvrages sur des machines virtuelles encartées tel que celle retenue dans les travaux de thèse de J.J. Vandewalle[Van97], celle des cartes Java[Sun96].

Détaillons maintenant le chemin qui mène (à l'intérieur de la carte, dans le mécanisme de chargement) du code FACADE au code exécuté.

5.2 Du code FACADE au code exécuté

Le chargeur de code est, dans les architectures classiques des systèmes d'exploitation de cartes à microprocesseur ouvertes, un composant de moindre importance qui ne soulève aucune difficulté particulière lors de son implantation. Il s'agit simplement de recevoir par le biais d'APDU spécialisées des blocs de code qui sont écrits dans la mémoire persistante. Toute la complexité du traitement de ce code est reportée dans le processus d'exécution.

En définissant l'architecture globale du projet Camille (chapitre 3), il est apparu évident que le chargeur pouvait jouer un rôle bien plus important. Tout d'abord, le processus de chargement, grâce à l'utilisation d'un mécanisme d'inférence de type, peut vérifier l'innocuité du traitement reçu. Ensuite le langage FACADE se définit comme un **langage intermédiaire**. En tant que tel, les traitements qui sont exprimés avec FACADE ne sont pas destinés à être exécutés directement, mais bien à subir une dernière phase de compilation. Selon le choix fait par les concepteurs du traitement, ils doivent pouvoir être transformés en instructions élémentaires du microprocesseur embarqué ou en un pseudo-code compact qui sera interprété.

Pour réaliser cette ultime étape de la chaîne de compilation – qui à débutée avec le code source, puis qui c'est prolongée par une étape d'adaptation en code FACADE avant d'être transmis à la carte – il convient de définir comment les éléments fondamentaux du langage sont transcrits sur le support d'exécution réel. Par éléments fondamentaux je désigne les opérations élémentaires du langage qui doivent être traduites en opérations connues du support d'exécution par exemple, mais il s'agit aussi de définir la transposition des variables FACADE dans la mémoire physique.

5.2.1 Variables FACADE dans la mémoire physique

Les différentes sortes de variables du langage FACADE (variables locales, arguments, attributs, constantes, ...) constituent le contexte d'exécution du traitement. Pour projeter le langage sur un micro-module particulier, il convient d'associer les différentes sortes de variables FACADE aux supports d'informations proposés par le matériel. Les différentes variables sont implantées dans le code machine en utilisant différents modes d'adressage et de manipulation des données gérées par le microprocesseur (registres, valeurs immédiates, adressage indirect, pile du microprocesseur, ...).

Nous proposons dans les lignes suivante le détail de cette transposition vers le microprocesseur AVR. Des démarches comparables peuvent facilement être proposées pour la plupart des autres microprocesseurs encartables (nous avons aussi étudié ce type de projection vers les microprocesseurs ARM). Notons tout de même que les toutes premières générations de microprocesseur encartés (notamment le Motorola 6805), du fait de leurs caractéristiques très anciennes (absence de registres de pointage, l'absence de pile d'exécution, un seul registre « accumulateur », ...) rendraient cette tâche délicate.

Pour les variables locales

Les variables locales représentent les variables de travail de la procédure. Le chapitre 4 nous à amené à en distinguer deux sortes selon qu'elles peuvent contenir des données de différents types ou pas au cours de l'exécution d'un même traitement. J'écrirais par la suite variables `Temp` pour désigner les premières et variables `Local` pour parler de celles qui ont un type fixé ;

Les variables `Temp` représentent typiquement des supports de stockage pour des valeurs intermédiaires lors de l'évaluation d'expressions complexes. Elles sont finalement l'image (« abstraite ») des registres d'un microprocesseur. Bien sur le nombre de variables temporaires n'est pas limité dans FACADE. Chaque traitement déclare simplement combien il en utilise. Pourtant le nombre de registres est, lui, fixé par le microprocesseur qui exécutera le traitement. Les autres sont placées dans la pile d'exécution en RAM.

Sur le prototype réalisé avec un microprocesseur AVR, les 7 premières variables `Temp` représentent des couples de registres 8 bits du microprocesseur : (`T0` → (`R2:R3`), `T1` → (`R4:R5`), `T2` ...). Le couple de registre (`R0:R1`) n'est pas utilisé car il est réservé au stockage temporaire des valeurs de retour des blocs de code. Les autres variables (Temp et Loc) sont placées dans le sommet de la pile d'appel du microprocesseur. Les variables loc (dont le type est déclaré une fois pour toute et qui peuvent donc être utilisé immédiatement) sont initialisées à `0/NULL` (pour des raisons évidentes de sécurité).

Conformément aux affirmations des concepteurs du microprocesseur AVR [Myk00] nous avons pu constater que la plupart des traitements que nous avons testés se contentent du vaste jeu de registres disponibles et que l'efficacité d'exécution des traitements en bénéficie pleinement (*c.f.* Chapitre 6 : Résultats expérimentaux).

La stratégie des adaptateurs de code, hors de la carte, est de réutiliser de préférence les premières variables temporaires. Ainsi, les traitements produits par le générateur de code encarté, utilisent plus intensément les registres que la pile, notamment au cœur des boucles. Notons que ce genre d'optimisation *a priori* est plus difficile à réaliser sur un code intermédiaire pour machine à pile. Pourtant cela est préférable afin que les traitements produits soient efficaces.

Pour les Arguments

Les cinq premiers arguments sont passés dans des registres du microprocesseur. Ces registres sont arbitrairement désignés: (A0 → (R16:R17), A1 → (R18:R19), A2 → (R20:R21), A3 ...). Les suivants sont placés dans la pile d'exécution du microprocesseur (avant les variables locales). La table 4.1 page 91 du chapitre 4 montrait que les arguments peuvent être utilisés en tant que variables Temp. Cela permet de pratiquer hors de la cartes des optimisations qui visent à réduire le contexte d'exécution (nombre de registres utilisés et nombre d'octets pris dans la pile courante). Le fait que les arguments soient placés dans des registres permet de réduire le coût des appels de méthodes, mais comme ils peuvent être réutilisés il devient possible de minimiser hors de la carte la taille d'un contexte d'exécution.

Pour les constantes

Les constantes sont pour le microprocesseur AVR traitées en tant que valeurs immédiates. Lorsqu'une constante de type CardByte qui vaut 13 doit être additionnée à l'argument A20 par exemple, c'est une opération ADI R20, 13 qui est générée conformément aux possibilités du microprocesseur. Dans le cas ou l'adressage par valeur immédiate n'est pas possible, la valeur est chargée dans un registre temporaire par une instruction LDI Rxx, valeur.

Pour la variable This

La variable This, si elle est définie (c'est-à-dire si le traitement est lié à une classe), est elle aussi placée dans un couple de registres: (R28:R29). Cette variable est un pointeur vers une structure de donnée privilégiée du traitement. Or l'AVR dispose de trois couples de registres susceptibles d'être utilisés en temps que pointeur. Le couple de registres (R26:R27) désigne le registre de pointage X, (R28:R29) désigne le registre de pointage Y et (R30:R31) désigne le registre de pointage Z. Nous avons décidé que Y serait l'« image microprocesseur » de la variable This. Les deux autres registres de pointages restent disponibles pour toute utilisation temporaire. Ils ne sont associés à aucune variable FACADE, mais servent de tampon pour des calculs internes comme ce sera montré par la suite.

Pour les attributs

Si la variable This « est » le pointeur Y, les attributs de l'objet associés à la méthode courante sont accessibles en lecture par des opérations du type LDD Rx,Y+Dep_Attrib; ou la constante Dep_Attrib représente le déplacement (en octets) à ajouter au début de la structure pour accéder à la valeur associée à l'attribut. (L'accès à un attribut de deux octets, par exemple s'il s'agit d'un CardShort est réalisé sur l'AVR en deux opérations LDD Rx,Y+Dep_Attrib et LDD Rx,Y+(Dep_Attrib+1)). L'accès à des attributs associés à des objets qui font plus de 32 octets [1] change légèrement le procédé, mais ne le pénalise pratiquement pas.

L'acte d'écriture dans un attribut diffère selon que le traitement est associé à une classe qui est une sorte de CardTObjet ou à une sorte de CardPObject. Dans le

1. Notons que la taille maximale d'un objet du noyau est la taille d'un regroupement de 8 pages soit 256 octets. Les adapteurs de code doivent définir de nouvelles classes qui par agrégation gèrent des structures applicatives plus grandes (fichiers, tableaux Java, ...).

premier cas (écriture en RAM) une opération `STD Rx,Y+Dep_Attrib` suffit alors que dans le second (écriture en EEPROM) il faut déclencher un traitement système de la classe `CardKernel`.

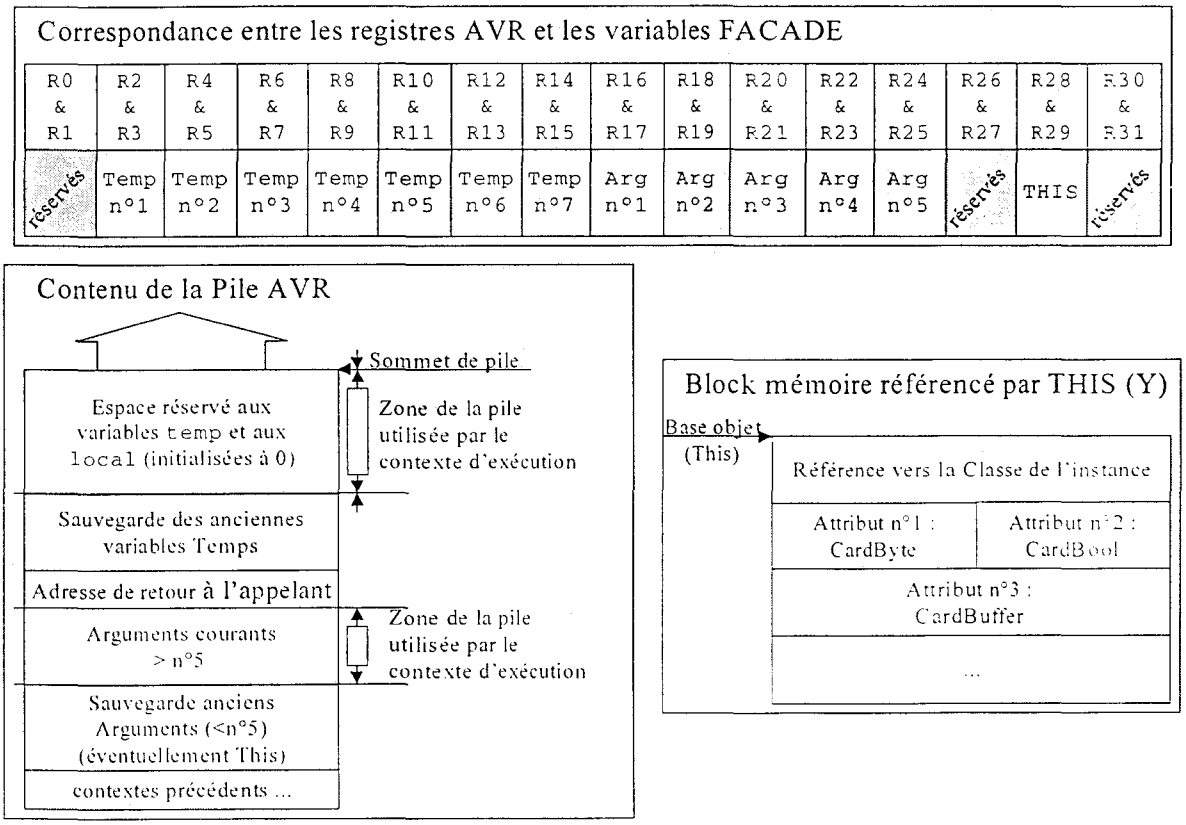


FIG. 5.1 – *Les différents composants du contexte d'exécution sur l'AVR*

La figure 5.1 reprend et résume les trois grands groupes de variables qui constituent un contexte d'exécution. Les registres sont utilisés pour masquer les sept premières variables temporaires et les cinq premiers arguments du contexte courant d'exécution. Le registre de pointage Y désigne le bloc mémoire/objet associé à This. En temps que tel il contient les attributs définis par la classe associée au traitement. On voit que la construction de la pile d'exécution du microprocesseur est structurée en différentes parties. En fait cette structure dépend aussi bien du contexte du traitement appelé que du contexte du traitement appelant.

Une fois la politique de gestion des variables FACADE définit il reste à proposer une traduction des opérations élémentaires FACADE. Elles peuvent être classées en deux catégories. L'opération `invok` d'une part et les quatre autres opérations qui sont les instructions de flots.

5.2.2 Les instructions de flots

Le langage FACADE définit quatre opérations qui structurent le flot de contrôle d'un programme: `Return`, `Jump`, `JumpIf` et `JumpList`. Ces opérations sont assez proches des instructions machines de l'AVR qui contrôlent le flot. Reprenons les une à une.

L'opération `Return`

Rappelons que cette opération élémentaire de FACADE termine le traitement en cours. Elle supprime son contexte d'exécution et elle restaure le contexte du traitement appelant. Ensuite c'est l'activité du traitement appelant qui reprend. `Return` peut être associée à une variable de travail qui est alors la valeur retournée par le traitement.

Le couple de registre (R0:R1) est réservé par le générateur de code pour transmettre un résultat du traitement appelé au traitement appelant. Si une variable résultat est prise en paramètre elle doit être chargée dans le couple de registres (R0:R1). Ensuite il faut que le traitement en cours termine et « passe la main » au traitement appelant.

Une instruction machine de l'AVR (`ret`) dépile « l'adresse de retour » à laquelle elle branche. Cependant nous avons vu sur la figure 5.1 que certaines variables ont été placées dans la pile. Il convient donc de les libérer (en les dépliants), et de restaurer les anciennes valeurs (notamment la sauvegarde des registres). Comme cette tâche est toujours la même, mais qu'elle est composée de plusieurs instructions machine (dépiler chaque registres puis `Ret`), le code des opérations `Return` est donc implanté avec un saut à une routine système spécialisée. Notons encore que le programme appelant doit restaurer les registres associés à ces arguments (qui ont été empilés avant l'appel).

L'opération `Jump`

L'opération `Jump` correspond strictement à l'instruction AVR `jmp`. Elles réalisent toutes deux un saut inconditionnel à une autre point du programme. La difficulté est de transformer le point de saut symbolique associé au `Jump` en une valeur absolu dans la mémoire de programme associée au `jmp`. Le fonctionnement d'un *éditeur de liens* qui réalise cette opération est déjà bien connu[Kra89, Session 6.3.3 : « Fonctionnement d'un éditeur de liens » page 225]. La principale difficulté dans le contexte de la carte à microprocesseur tient aux contraintes d'espace mémoire de la carte. Une solution assez simple, qui ne nécessite pas d'analyse supplémentaire du code, basée sur le principe de la liaison par substitution (présentée par la référence précédente) et sur le brevet [BGL+96] a permis l'implantation par le système de cette fonction. Elle nécessite néanmoins l'usage de $2 \times Card(Label)$ octets en mémoire de travail. En conséquence elle limite encore à quelques centaines de points de sauts différents les codes natifs qui peuvent être générés. Chaque valeur de la structure n'est en effet écrite qu'une seule fois.

L'opération `Jumpif`

L'opération `Jumpif` réalise un saut conditionnel. Cette tâche peut être implantée de différentes manières en quelques instructions machines sur le microprocesseur AVR.

La génération de code machine réalisant un `Jumpif` se décompose en deux étapes: (1) charger le booléen dans le registre d'état du microprocesseur (si nécessaire), (2) implanter l'instruction de saut conditionnel.

Dans la pratique la génération de code est un peu plus complexe. Le problème est que les instructions de sauts conditionnels du microprocesseur sur lequel notre maquette à été réalisée ne permet pas de branchements « éloignés ». En fait le point de branchement doit se situer à moins de 64 instructions (128 octets) de l'opération de saut conditionnel. Cette contrainte implique que la réalisation du point (2) se décompose en deux cas. Si le point de branchement est déjà déterminé (s'il est situé sur une instruction qui précède l'instruction courante) et s'il est à moins de 64 octets alors l'opération de saut conditionnel de l'AVR peut être utilisée simplement : sinon un branchement conditionnel (sur la condition inverse de celle testée) est encodée dans le programme généré de tel sorte qu'il « saute » l'instruction suivante. Dans ce cas c'est l'instruction suivante (un `jmp`), qui réalise un saut inconditionnel vers le point de branchement du `Jumpif`.

L'opération `Jumplist`

Une opération FACADE `Jumplist` est un branchement indexé. Le point de saut est dynamiquement établit à partir d'une table de branchement possible et d'une variable entière. Pour implanter cette opération sur l'AVR il faut utiliser table de saut. nous avons choisi de construire cette table sous la forme d'une liste d'instruction `jump` dans laquelle le microprocesseur saute en fonction de la variable qui sert d'indexe. De cette manière l'édition de lien du `Jumplist` est réalisée de la même manière que pour les opérations `Jump`.

Toutes ces opérations permettent de définir des *ruptures de flot* dans l'exécution d'un programme. Il ne reste dans le langage FACADE plus qu'une opération élémentaire qui est `Invok`. Elle donne notamment une représentation uniforme de toutes les opérations de traitement des données.

5.2.3 Les opérations de traitement des données

L'opération `invok` qui décrit toutes les opérations de traitement des données s'écrit : `[VarRes]` ← `VarSrc` *op* {`VarParam1, VarParam2, ...`}.

Cette représentation est uniforme quelque soit l'opération réalisée. L'opération *op* peut être aussi bien + qui exprime l'addition entre deux octets, que `Create()` qui est une méthode de classe qui alloue une nouvelle page mémoire par exemple. A l'origine cette uniformité des instructions élémentaires du langage FACADE à été motivée par la volonté de simplifier à l'extrême le mécanisme d'inférence de type. Il est évident cependant que l'extrême diversité des opérations qui sont exprimées par ce biais ne peuvent être implantées toutes de la même façon.

Le terme *convention d'appel* est utilisé en compilation pour préciser le code machine qu'un traitement doit effectivement implanter pour déclencher l'exécution d'une autre rou-

tine. Chaque langage définit en règle générale son protocole d'appel. L'approche descendante d'un langage vers une machine (physique ou virtuelle) introduit le plus souvent l'usage d'une pile d'exécution pour enregistrer les informations d'appels, c'est à dire le contexte d'exécution de la routine appelée (variables locales et arguments). Notre pile d'appel a été présentée sur la figure 5.1 page 116. La volonté d'optimiser le code machine généré donne un rôle central à la manipulation des registres (surtout dans les technologies RISC) pour réduire la complexité (en temps) des appels [Lev83].

Dans le langage intermédiaire FACADE, les différentes conventions d'appels utilisées par le noyau sont complètement masquées. C'est principalement en ce sens que FACADE est un langage intermédiaire plus abstrait que le bytecode JAVA par exemple. Tout les appels sont décrits avec la même opération : `Invok`. Et pourtant il est évident que l'opération `T0 ← AByte + AnotherByte` ne peut pas être réalisée de la même manière que l'opération `T0 ← MyPObject myMethod MyParam1 MyParam2` si l'on souhaite préserver l'efficacité du système. Je vais dans un premier temps détailler les différents modèles d'appel utilisés par l'architecture encartée de Camille en commençant par un « non-appel » : l'expansion de code[2].

Expansion de code

Ce modèle d'appel n'en est pas véritablement un car il consiste à ne pas appeler un traitement mais plutôt à utiliser un (ou une séquence de) code machine pour dupliquer le traitement là où on l'utilise (dans le code qui est en train d'être généré). C'est la technique utilisée par la majorité des classes systèmes qui représentent des *objets naturels* du micro-module. Pour prendre un exemple simple, considérons l'opération d'addition entre deux valeurs numériques sur 16 bits : `t0 ← t1 + t2` où `t0`, `t1` et `t2` sont des Temp qui sont placées dans les couples de registres (8 bits) R2:R3, R4:R5 et R6:R7. Le code machine le plus efficace pour implanter cette opération sur le microprocesseur AVR consiste à copier `t1` dans `t0` puis à additionner `t0` avec `t2` (contrairement à la plupart des microprocesseurs RISC, l'AVR ne dispose pas d'opérations arithmétiques et logiques à trois opérandes). Le code obtenu est présenté par la figure 5.2.

En fait, plus qu'une expansion de code, il s'agit ici d'une micro-compilation. Selon la nature des variables utilisées (variables locales, attributs, arguments, ...) le code produit diffère légèrement. Cependant le processus de génération du code reste toujours le même :

1. Charger la variable source dans des registres (si possible ceux de la destination) ;
2. Charger la variable paramètre dans des registres (s'ils n'y sont pas déjà) ;
3. additionner les registres (avec report de retenues par les parties hautes) ;
4. recopier le résultat obtenu en registre dans la variable destination (à moins qu'elle ne correspond au registre en question).

2. l'acte d'expansion de code correspond ici à ce qu'accompli un compilateur C pour implanter une fonction marquée du modifieur `inline`

`MOV R2,R4` `MOV R3,R5`	*Copie de la source dans les registres (de la destination puisque c'est possible)*
	Charger le paramètre dans des registres (inutile ici)
`ADD R2,R6` `ADC R3,R7`	*Implanter l'addition « naturelle ».*
	Écrire la variable destination (inutile ici car la destination est le couple de registres R2:R3)

FIG. 5.2 – *Implantation de l'opération '+' sur des valeurs 16 bits*

Cette technique d'implantation des opérations FACADE `invok` n'a de sens que si elle peut être réalisée avec une instruction du microprocesseur (ou, à la limite avec une séquence réduite d'instructions).

Appel procédural classique

Le générateur de code natif est parfois amené à déclencher l'exécution d'une routine système pour implanter des opérations élémentaires qui ne sont pas câblées. Normalement ces opérations sont rares car le noyau de base du système encarté de Camille reflète simplement le matériel disponible. Cependant il existe un certains nombre d'opérations qui sont communes à tout les langages de programmation. Elle paraissent véritablement élémentaires, mais ne sont pas toujours disponibles sur les microprocesseurs des cartes à puce. Dans le cas de l'AVR, l'une des opérations les plus caractéristiques de cette remarque est le décalage logique. La seule opération disponible est un décalage logique de une unité à droite. L'équivalent à gauche est obtenue par l'opération `ADD Rx,Rx`. Le noyau propose donc une procédure rapide qui permet de décaler n fois un registre, avec n qui est une constante ou une variable. Car ce type de décalage logique est extrêmement utile pour optimiser des opérations de multiplication. Aussi cette opération figure, de fait, dans les interfaces du système d'exploitation pour éviter d'avoir à les charger systématiquement avec chaque nouvelle application. Pour le microprocesseur AVR un exemple typique est la multiplication entre deux registres: `i` ← `j << k`) (qui est rarement câblée dans les micromodules). La figure 5.3 montre l'implantation d'un tel appel sur le langage machine AVR.

Il existe un certain nombre d'autres procédures dans ce cas, c'est-à-dire qui masquent des opérations jugées élémentaires mais absentes de la carte à microprocesseur. Les conventions d'appel de ces procédures ne sont pas, pour des raisons évidentes d'optimisation, implantées comme des appels de méthode.

Appel de méthode

Nous avons dit que FACADE est un langage objet. En temps que tel il exprime des traitements. Nous ne reviendrons pas ici sur ce qu'est un appel de méthode, mais plutôt sur

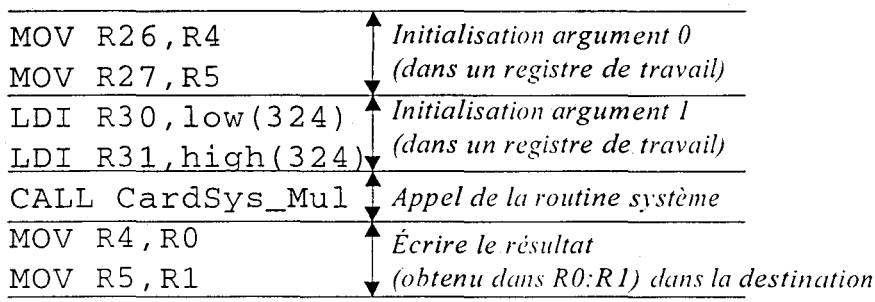


FIG. 5.3 – *Appel d'un traitement libre (déjà chargé et connu de la carte)*

la manière de le réaliser sur le microprocesseur AVR d'une carte à puce. Rappelons simplement qu'un appel de méthode contrairement à un appel procédural, ne peut déterminer statiquement (une fois pour toutes lorsque le code est chargé par exemple) quel traitement sera exécuté. C'est en fonction de l'objet receveur (la variable [*Varsource*] d'un `Invok` dans FACADE) qu'il détermine pour chaque appel le point de saut. Ainsi lorsqu'un traitement appelle une méthode m sur une variable a dont le type est Cl, il ne déclenche pas l'exécution du même code selon que l'objet placé dans a et réellement Cl ou bien qu'il appartient à une sous-classe de Cl qui surcharge la méthode m. C'est le fondement même du *polymorphisme* de classe.

L'utilisation des techniques objets pour concevoir les systèmes d'exploitations dédiés aux cartes à microprocesseur et notamment les différentes techniques d'appel de méthodes, ont déjà été évaluées sur les microprocesseurs 6805, PICO-RISC et ARM7TI [Gri97]. Le microprocesseur AVR est assez comparable au microprocesseur PICO-RISC (qui y est traité) et la convention d'appel présentée ici est largement inspirée de ces études préliminaires.

La solution la plus répandue pour réaliser un appel de méthode sur un objet et de placer dans la structure de donnée de l'objet un pointeur vers une description (plus ou moins détaillée) de sa classe. Cette description contient notamment (au minimum) une table de pointeur vers les différents programmes associés aux méthodes définies par la classe. Dans le cas des classes Cl et $Cl2$ précédemment décrites la table des méthodes donne pour m deux pointeurs différents. L'appel de méthode sur l'argument a est toujours implanté de la même manière : trouver la description de la classe associée à a, et ainsi le pointeur de la méthode m qui lui est associé. Mais selon que a est effectivement une instance de Cl ou une instance de Cl_2 le pointeur de méthode effectif ne sera pas le même. La figure 5.4 montre l'implantation de ce type d'appel par le chargeur de code pour l'opération : `attrib ← aCardStream getByte`.

Finalement le choix de la techniques d'appel utilisée est fonction des possibilités du matériel (dans le cas de l'implantation tel que le décalage logique proposé par le noyau sur l'AVR par exemple), mais aussi les classes d'extension qui peuvent définir leurs propre politique d'appel. Cette propriété du micro-noyau encarté qui assure la flexibilité et l'efficacité de la portabilité sur différentes micro-modules, est permise par l'architecture système du

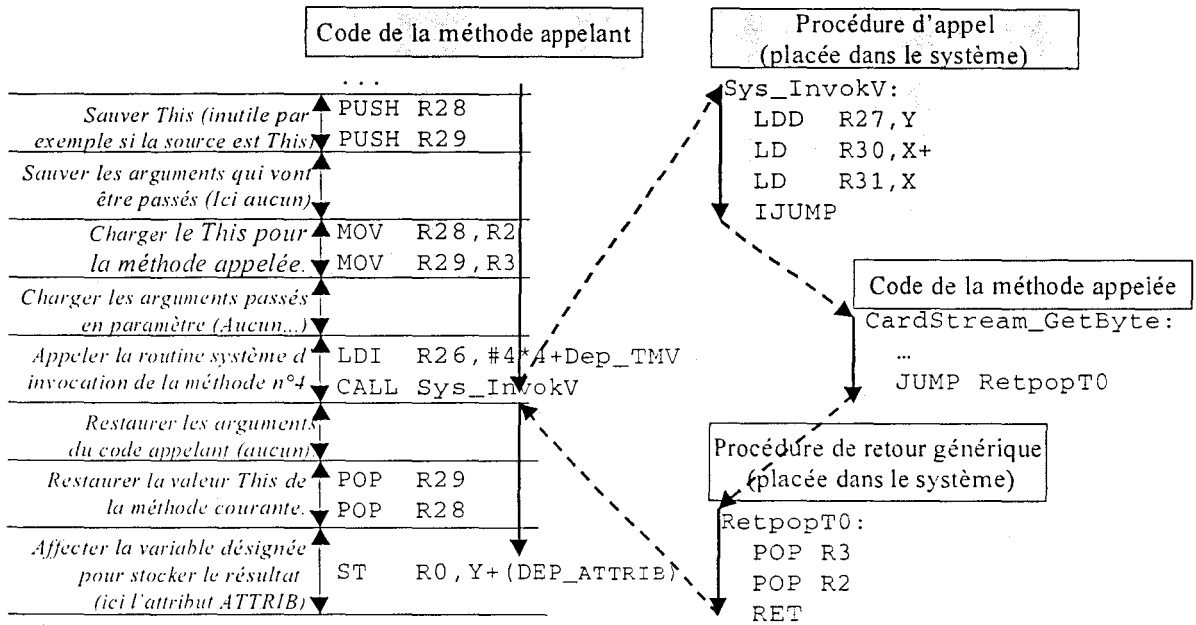


FIG. 5.4 – *Appel d'une méthode virtuelle(déjà connue de la carte)*

chargeur.

5.3 Architecture système du chargeur

L'architecture du système d'exploitation encarté de Camille présenté dans le chapitre 3 donne une représentation objet des traitements encartés sous la forme d'une classe abstraite `CardCode` qui est implantée par deux classes `CardCodeN` et `CardCodeV`. Cette architecture objet représente les différentes entités en présence. Il faut encore préciser sur cette structure de base comment s'organise la répartition des différents aspects du chargement.

5.3.1 Répartition des différents aspects du chargement

Le processus d'analyse (aussi bien l'analyse lexicale que l'inférence de type) du code FACADE est une tâche commune à `CardCodeV` et `CardCodeN` ainsi qu'à tout code conçut comme une instance produite à partir d'un code FACADE. Cette tâche est placée dans une méthode de `CardCode`. Ainsi les constructeurs de `CardCodeN` et `CardCodeV` réutilisent la même méthode en l'appelant depuis leur constructeur respectif. L'analyseur parcourt le code FACADE séquenciellement, instruction par instruction.

Une fois l'instruction lue et vérifiée, elle doit être transposée en code exécutable (natif ou virtuel) stocké dans la mémoire de la carte. Pour cela l'implantation d'une opération FACADE est décomposée en une ou plusieurs opérations machines que le générateur de

code implante en appelant un jeu de méthodes privées (c'est-à-dire dont l'usage est limité au noyau du système) déclarées dans `CardCode`. Le corps de ces méthodes (par polymorphisme) n'est définit que dans `CardCodeN` et `CardCodeV`. Selon la nature exacte de l'« objet code » construit, l'implantation du traitement est donc différente, alors que le processus de chargement du code est le même.

Les classes `CardCodeN` et `CardCodeV` définissent donc finalement quelle forme prend le code généré. Un autre intervenant contribue cependant aussi à la forme que prend le code implanté. Il s'agit du Type de la variable qui reçoit l'appel d'une opération `invok`. Nous avons en effet vue que selon la nature exacte de l'opération invoquée le code généré pourra être un appel de méthode, aussi bien qu'une addition entre deux registres. Aussi convient-il de déléguer le choix des conventions d'appels aux classes encartées.

L'objectif est de permettre aux classes de décider comment un appel à l'une de leurs méthodes doit être implanté dans le code qui les utilise. Ainsi les objets systèmes qui représentent des entités primitives du micromodule (`CardByte`, `CardBuffer`,...) sont à même de spécifier l'utilisation d'une opération machine spécifique alors que les objets plus conventionnels seront appelés avec un appel de méthode classique.

A partir de ces différentes considérations et en utilisant la sémantique statique des instructions FACADE comme un moteur pour la génération de code natif, il est possible d'organiser le processus de chargement.

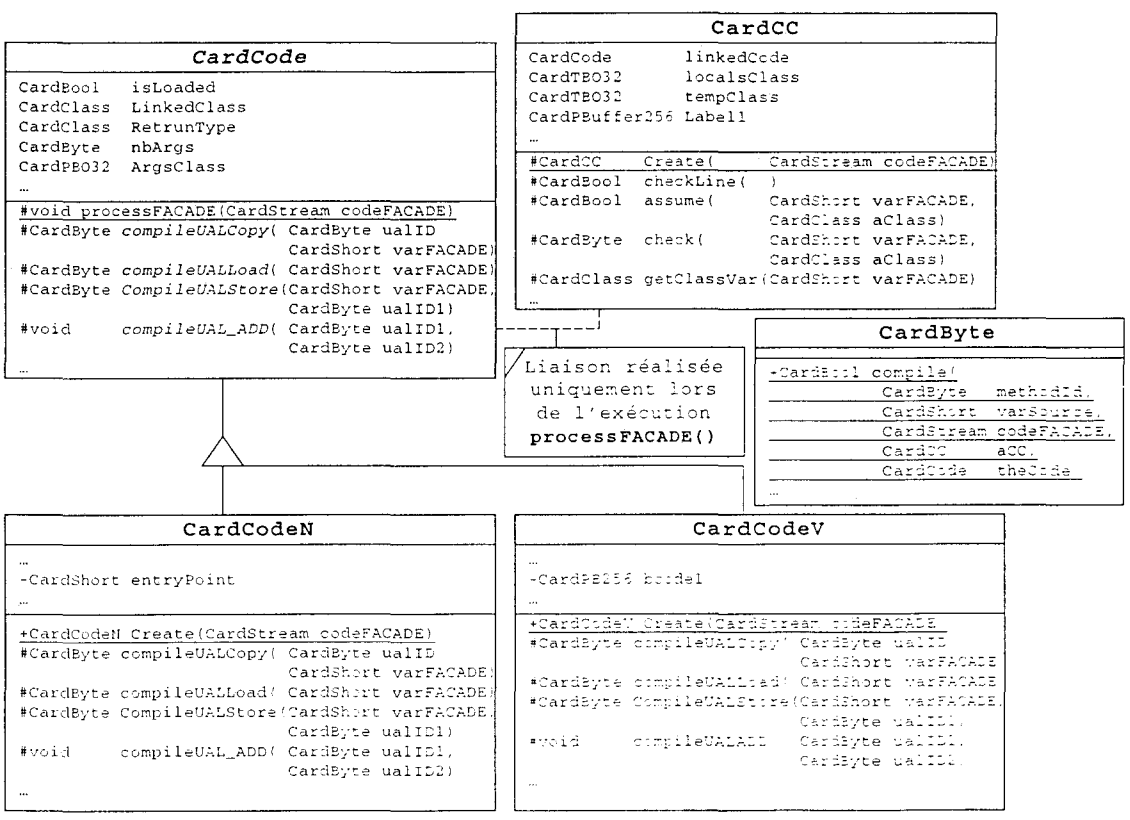
5.3.2 Organiser le processus de chargement

Rappelons tout d'abord que la structure interne du noyau encarté défini dans le bloc de code quatre classes. En plus de `CardCode`, `CardCodeN`, `CardCodeV`, une classe `CardCC` est déclarée. Les instances de la classe `CardCC` sont utilisées pour stocker les structures de données temporaires associées au moteur d'inférence et à l'édition de lien. Par exemple nous avons vue dans le chapitre 4 que la vérification de type nécessite une table qui associe une classe à chaque variable `Temp` déclarée (c'est la table notée ΓT dans le chapite 4). C'est la classe `CardCC` qui gère cette structure.

Le schéma de classe 5.5 précise quelques unes des méthodes utilisées par la fonction de génération de code. Trois catégories de méthodes sont à distinguer :

- lecture du code FACADE ;
- inférence des types ;
- génération du code cible.

Les méthodes de classe `Create()` de `CardCodeN` et `CardCodeV` (qui sont en fait des constructeurs) ainsi que la méthode `Create()` contribuent au décodage d'un traitement FACADE. Plus précisément elles renseignent les structures de données utilisées par la suite pour valider l'utilisation des types par le traitement. Il s'agit notamment de décoder le type des arguments du traitement, le type de retour, le nombre de variables locales et Temp, ainsi que la liste des labels et les types des variables `Temp` proposés par la preuve (l'inférence



La marque # désigne ici une visibilité limitée aux programmes du noyau.

FIG. 5.5 – *Architecture des classes du bloc de gestion du code encarté. La classe* `CardByte` *illustre la déclaration de la méthode* `compile`.

de type réaliser hors de la carte). La méthode `ProcessFACADE()` effectue aussi la lecture du code instruction par instruction. Elle détermine la nature de l'opération courante (`invok`, `Return`, `Jump`, `Jumpif` ou `Jumplist`), puis elle poursuit la lecture du code en fonction de l'opération (voir annexe A page 171 pour le détail du codage binaire de FACADE).

L'inférence de type, qui est l'outil de base du système pour garantir l'innocuité des programmes, est assurée par différentes méthodes du contexte de compilation. Il s'agit notamment des méthodes `checkLine()`, `check()` et `assume()`. Ces méthodes sont toutes placées dans la classe `CardCC` car elles sont intimement liées aux structures de données utilisées pour charger le traitement (table des labels, types des variables locales, ...).

Enfin l'opération de génération de code est décomposée en deux parties. La méthode `compile()` est redéfinie par la plupart des classes du noyau. Ainsi dans la classe `CardByte()`, cette méthode reconnaît l'opération + comme « native » et l'implante de manière spécifique. Pour générer le code machine associé à l'opération + la méthode `compile()` appelle sur le traitement (un `CardCodeN` ou un `CardCodeV`) d'autres méthodes. Sur la figure 5.5 nous pouvons voir `compileUALLoad()` et `compileUALADD()` par exemple. Finalement ce sont ces méthodes, déclarées dans `CardCode` (mais implantée différemment dans `CardCodeN` et `CardCodeV`) qui gênèrent le code exécutable.

Pour plus de clarté, précisons l'articulation du processus de chargement au travers des différents éléments du système encarté grâce au diagramme de séquence d'un extrait de compilation.

5.3.3 diagramme de séquence d'un extrait de compilation

La figure 5.6 montre la trace que le processus de compilation suit pour traiter le chargement de la première instruction d'un code qui débute par :

`T0` ← `A0` + `A1`.

(avec `A0` et `A1` des arguments déclarés `CardByte`. et `T0` une variable `Temp`).

Le chargement d'un traitement consiste à créer une instance de `CardCodeN` en utilisant la méthode statique `CardCodeN Create(Stream aFACADEStream)`.

Le constructeur de cette classe commence par préparer les structures de données utiles à la compilation des opérations élémentaires, notamment en construisant un contexte de compilation (instance de `CardCC`). Cet objet utilise les déclarations fournit par le traitement FACADE pour initialiser en autres les tables de variables `Temp` (qui sont le cœur de l'inférence de type). Puis elle définit la méthode `processFACADE()` qui va initiée le chargement instruction par instruction du traitement FACADE placé dans un `CardStream`.

Ensuite l'inférence de type (et donc indirectement la génération de code) sont assurées par la méthode `processFACADE()`. Cette méthode réalise ce qui a été nommé dans le chapitre 4 « l'interprétation statique » du traitement. Elle commence par décoder le début de l'opération et obtient ainsi la nature de l'opération (ici un `invok` sur la méthode +). Une opération `invok` définit au minimum un receveur, qui est pour nous

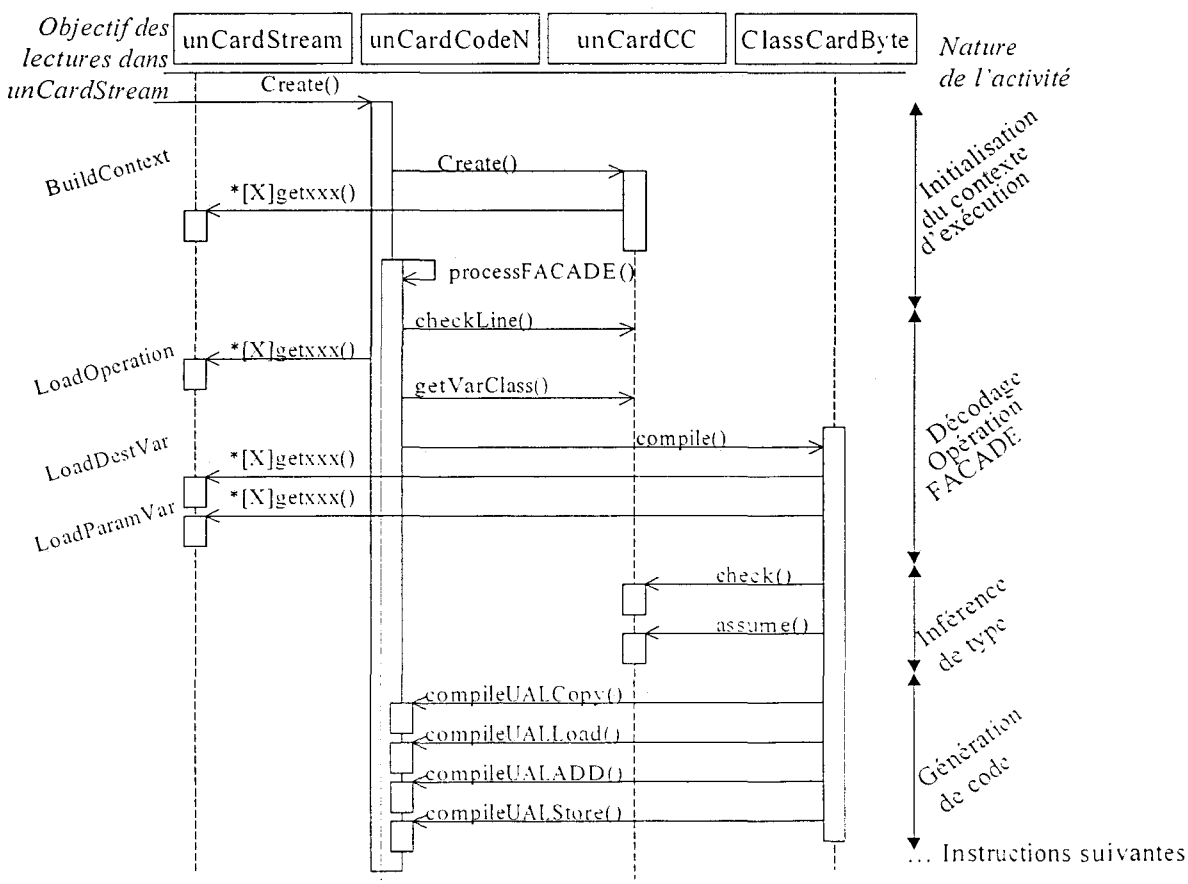


FIG. 5.6 – *diagramme de séquence de la compilation d'une addition*

la variable A0. Cette variable est un argument, dont le type (enregistré lors de l'initialisation du traitement) est CardByte. A ce stade le type CardByte est identifié comme étant la classe qui gère l'inférence de type, et l'implantation de l'opération +. La méthode processFACADE() déclenche donc l'appel de la méthode (de classe) Compile() sur la classe identifiée : CardByte. Ainsi c'est CardByte qui poursuit l'opération de chargement de l'instruction FACADE. CardByte définit une méthode + qui retourne un CardByte en résultat et qui prend en paramètre une autre variable qui doit être une sorte de CardByte. Elle charge depuis le CardStream le code de la variable qui recevra le résultat (T0), et le paramètre de l'addition (A1). A ce stade il convient de vérifier si l'opération est correctement typée.

Pour que l'opération soit correcte il faut que A1 soit explicitement du type CardByte. La méthode Check() du contexte de compilation est utilisée pour vérifier cette propriété. Comme A1 est un CardByte le paramètre de l'opération + est correct. Ensuite il faut vérifier que T0 peut être utilisé pour écrire un CardByte. La méthode utilisée est Assume() plutôt que Check(). Assume() fonctionne de la même façon que Check() à moins de la variable soit un Temp. Dans ce cas l'opération est toujours validée mais le type de la variable est modifié. Sur notre exemple le type de T0 devient CardByte. Il ne reste plus qu'a générer le code.

Nous avons vue dans la figure 5.2 que la génération d'une opération d'addition se décompose en quatre étapes :

1. tous d'abord la variable source (A0) doit être transférée dans l'UAL (pour un code natif AVR il s'agit d'un registre). Le choix du registre de destination est de préférence celui qui est associé à la destination si elle est déjà placée dans un registre. La méthode CompileCopyUAL() de la classe CardCode réalise cette tâche ;
2. puis il faut charger la variable en paramètre A1 dans l'UAL, si elle n'y est pas déjà. C'est la méthode CompileUALLoad() qui est alors appelée ;
3. ensuite il faut générer l'addition à proprement parler, entre des données accessibles depuis l'UAL, avec la méthode CompileADD() ;
4. enfin, il faut (éventuellement) copier le résultat depuis les zones utilisables par l'UAL vers la variable destination grâce à la méthode CompileUALStore().

Après cela, la méthode compile() de la classe CardByte se termine et le processus de chargement peut se poursuivre en décodant l'opération suivante...

Ce processus est répété pour chaque opérations, le code est produit au fur et à mesure. Pour éclairer ce propos il convient de commenter un premier exemple de code compilé.

5.3.4 Premier exemple de code compilé

Nous reprenons ici, en guise de programme de test pour la compilation, celui qui à été utilisé dans le chapitre 4 pour expliquer le fonctionnement du moteur d'inférence. Il s'agit d'une simple boucle de 0 à 100 qui fut présentée par la table 4.2 page 95. La table

5.1 reprend cette séquence de six opérations et montre comment elles sont traduites sur le microprocesseur AVR.

Code FACADE	Code AVR généré	Note sur l'activité du générateur
initialisation du code	0x4000 PUSH R2 0x4001 PUSH R3 0x4002 PUSH R4 0x4003 PUSH R5	Sauvegarder le contenu des registres qui vont servir à stocker les variables Temp v1 et v2
v1← 0 asIs	0x4004 LDI R26,0 0x4005 MOV R2,R26	L'instruction LDI ne peut porter sur R2 aussi le registre réservé R26 est utilisé.
Jump Label0	0x4006 RJMP 0x4009	Le Jump est implanté avec un RJMP de l'AVR
label1: v1←v1+1	0x4007 LDI R26,1 0x4008 ADD R2,R26	L'addition est réalisée comme vue précédemment.
label0: v2←v2<100	0x4009 LDI R30,100 0x400A CP R2,R30 0x400B BLD R4,1	La comparaison est une opération qui engendre un diagramme de séquence proche de celui de l'addition.
Jumpif v2,Label1	0x400C TST R4 0x400D BREQ 0x4007	le Jumpif est dirigé vers un point de saut déjà vu à moins de 64 mots...
Return	0x400E JMP 0x000C	branchement à une routine système qui restaure 4 registres et revient au programme appelant.

TAB. 5.1 – *Premier exemple de compilation encarté*

Le code obtenu est une image fidèle du traitement tel qu'il est exprimé en FACADE.

Tout d'abord un certain nombre de registres sont sauvés avant de pouvoir d'être utilisés par le traitement.

Ensuite l'image de la variable v1 (qui est dans notre cas le registre R2) est affecté avec la valeur (immédiate) 0. Cette opération est effectuée en deux étapes car l'opération LDI ne peut être utilisée qu'avec les registres de R16 à R31.

L'implantation du Jump ne pose pas de problème particulier. Il faut cependant noter qu'elle a été générée en deux étapes. La première partie de l'opération à été générée lorsque le Jump du code FACADE a été rencontré dans la séquence des opérations. Cependant le point de saut dans le code machine, associé au label0 dans le code FACADE, n'est pas encore connu. Ce n'est qu'en arrivant sur l'implantation de la ligne marquée par le label0 que le point de saut dans le code machine à pu être déterminé. à cet instant, l'adresse à associer au RJMP est connue (le générateur de code en est à 0x4009) et l'écriture de l'opération peut être finalisée.

L'opération d'addition à largement été détaillée dans la section précédente. notons simplement ici que ni `compileUALCopyVar("v1","v1")` ni `compileUALStoreVar("idR2","v1")` ne gênèrent d'opérations.

La comparaison avec la constante `100` implique que cette valeur soit chargée dans un registre réservée. `R30` est utilisé pour recevoir « dans l'UAL » les éventuelles valeurs du second argument d'une expansion de code en ligne. Le résultat de la comparaison est copier dans R4 qui correspond à `v2`.

Le saut conditionnel a pu être établit immédiatement dans l'or du décodage de l'opération `Jumpif v2,Label1` car le point de saut `label1` est déjà déterminé à ce stade de la génération de code.

Enfin l'opération `Return` est réaliser avec un `JMP` qui branche vers une zone du code système. à cette adresse figure les opérations :

```
POP R5
POP R4
POP R3
POP R2
RET
```

Le but ici est de factoriser les mécanismes de restauration des registres associés au variables `Temp`, au lieu de générer pour chaque méthode une séquence d'instruction semblable.

Un lecteur habitué à lire du code AVR trouvera facilement sur cet exemple différents points qui pourraient être écrits différemment (sans changer la structure algorithmique exprimée par le code FACADE) et qui ne sont ni optimum en ce qui concerne l'efficacité du code exécuté, ni en ce qui concerne la compacité. Mettre à zéro un registre quelconque de l'AVR, par exemple, peut être fait en exécutant l'opération `EOR Rx,Rx` (qui effectue un ou exclusif). L'addition pourrait être remplacée par l'opération `INC R2`. Et les cinq opérations qui implantent la comparaison et le saut conditionnel auraient pu être ramenées aux trois suivantes :

```
LDI R30,100
CP  R2,R30
BLT 0x4009
```

Pour obtenir ce type de code AVR, il faut optimiser le code généré.

5.4 Optimiser le code généré

Le but qui nous a amené à introduire un générateur de code à l'intérieur même de la carte à puce est de pouvoir embarquer des traitements qui nécessitent un certain niveau d'efficacité. La production d'un code natif efficace est une opération délicate qui implique des techniques d'analyses poussées. Ces techniques font l'objet d'études propres depuis que les premiers compilateurs existent. L'objet de cette section est de comprendre comment il est possible de réaliser la majorité de ces techniques d'optimisations dans la chaîne de compilation de Camille.

5.4.1 Optimisations dans la chaîne de compilation de Camille

Il est possible de distinguer trois sortes d'optimisations. La figure 5.7 (largement inspirée de [ASU86, Chapitre 10, page 585]) montre les possibilités d'optimisations aux différentes étapes de la production d'un code encarté.

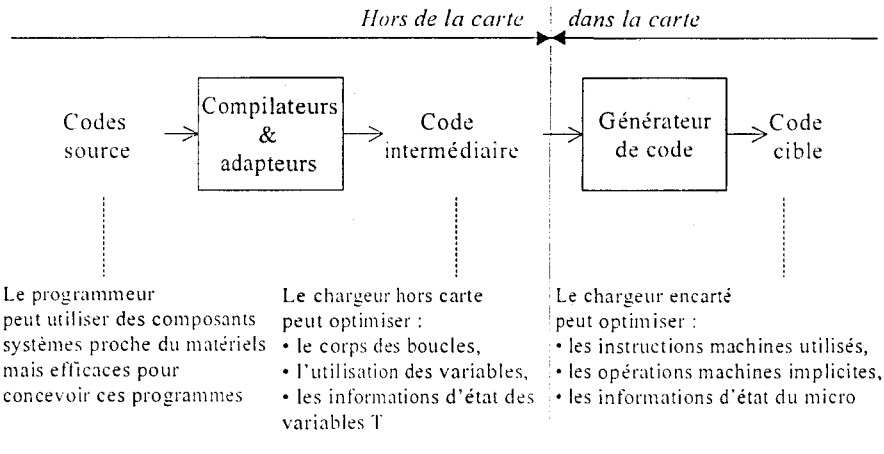


FIG. 5.7 – *Place des différentes techniques d'optimisations sur une chaîne de compilation*

La première source d'optimisation possible provient du programmeur lui-même. Dans la logique de notre architecture il existe des classes du noyau qui reflètent précisément le matériel disponible. L'utilisation de ce matériel (aux travers de ces classes) est évidement plus performante que l'utilisation de « primitives » assez éloignées des possibilités du matériel. Un exemple typique est la différence entre un tableau tel qu'il est défini par la machine virtuelle Java et un `CardTBuffer256` du noyau. Le premier est plus pratique à manipuler que le second : il ne connait pas de limitations « arbitraires » sur le nombre de ses éléments, mais en contrepartie il implique un nombre plus important d'opérations réalisées par le microprocesseur. L'architecture Camille expose les éléments fondamentaux du système. Rédiger un code critique (même dans le langage Java par exemple, en important les bibliothèques `CardKernel.*`) à l'aide de ces éléments est évidemment le premier principe d'optimisation d'un programme. Dans l'esprit de ce mémoire les programmeurs qui utilisent ces ressources systèmes élémentaires sont ceux qui conçoivent les bibliothèques nécessaires au support de codes issus de langages de hauts niveaux comme le P-code des *SmartCard for Windows* ou l'implantation des classes de `javacard.lang.*` par exemple.

Ensuite il est possible d'utiliser le code intermédiaire FACADE pour appliquer un certain nombre d'optimisations globales. Cette tâche automatisée ne peut être réalisée qu'à l'extérieur de la carte, tant les algorithmes utilisés sont complexes en temps et en espace mémoire. Il s'agit des techniques de détection et de suppression de codes morts, des techniques d'optimisation du code contenu des boucles, des techniques d'optimisation du nombre de variables temporaires... La mise en œuvre de ces techniques est cependant bien

connue, et peut aisément être appliquée au code intermédiaire FACADE. Aussi ne seront-elles pas détaillées par la suite.

Reste le problème de l'optimisation du code cible. Nous avons notamment pu voir sur l'exemple de la section 5.3.4 qu'il existe en règle générale différents moyens de traduire en code natif des opérations FACADE élémentaires. Cela tient en bonne partie au fait que les microprocesseurs encartés ne respectent pas strictement, aujourd'hui encore, les principes qui guident normalement la réalisation d'une architecture RISC. L'utilisation d'une opération EOR Rx,Rx peut être utilisée pour charger la valeur zéro dans n'importe quel registre. Elle est plus judicieuse (parfois) que la traduction générale de copie d'une valeur immédiate dans un registre qui est malheureusement limitée à un sous ensemble des registres dans l'AVR. D'autres problèmes liés à la gestion des registres, aux calculs implicites de l'unité arithmétique et logique, et à l'usage des registres de pointage constituent ce qui fait véritablement la différence entre les techniques d'optimisations conventionnelles et les techniques d'optimisations encartés.

5.4.2 Optimisations encartées

Nous avons (heureusement ?) pu constater dans la section 1.3.3. page 16. que les microprocesseurs utilisés dans les cartes ont un fonctionnement moins complexe que ceux placés dans nos stations de travail. Car l'essentiel des efforts d'optimisation du code cible des compilateurs conventionnels porte sur des aspects du fonctionnement des machines (*pipelines* d'instructions, anticipation des sauts, jeux d'instructions spécialisées, caches, ...) ardus à prendre en compte. Dans le cartes les problèmes d'optimisation du code cible portent essentiellement sur trois points :

- l'utilisation d'instructions spécialisées qui ont une fonction équivalentes à plusieurs instructions simples ;
- l'optimisation du code cible grâce à la connaissance de l'état interne du microprocesseur (par exemple, la connaissance de la valeur du registre de statut) et de ses registres ;
- l'utilisation des registres pour masquer les variables les plus souvent utilisées.

Il convient en effet de faire un bon usage des instructions élémentaires définies par le support d'exécution. L'exemple précédent montrait comment un `EOR Rx,Rx` peut éviter la séquence `LDI Ry,0` suivie de `MOV Rx,Ry`. Il existe un certain nombre d'autres opérations dans le même cas: par exemple `INC Rx`. plutôt que `LDI Ry,1` et `ADD Rx,Ry`). `TST Rx` plutôt que `LDI Ry,0` et `CP Rx,Ry` ...L'utilisation de ces cas particuliers peut être assurée en détectant dans les méthodes `compile()` du noyau les constantes qui sont privilégiées par les instructions du microprocesseur cible. La question qui se pose alors est : « un générateur de code, nanti de tous les tests de ces cas particuliers, ne devient-il pas trop volumineux pour être placé dans une carte à puce? ». Nous avons pris en compte la majorité des cas particuliers (en fait tout les cas particuliers que nous avons identifiés) nécessaires à la

génération de code de l'AVR. La taille du noyau encarté (dont le détail sera commenté dans le prochain chapitre) est resté tout à fait raisonnable.

Un problème d'optimisation bien plus délicat à prendre en charge est basé sur l'état du microprocesseur. Je pense principalement ici à l'état des *flags* du microprocesseur (c'est à dire à l'état des bits du registre de statut, *c.f.* [Atm00]) et à l'état des registres. Au fur et à mesure que l'exécution des instructions machines s'enchaîne, les registres prennent des valeurs qui, lorsqu'on en connaît la nature, permettent d'éviter la génération d'opérations machines redondantes et de réaliser, ainsi, des optimisations importantes.

Le premier problème porte sur la détermination (lors de la génération de code) de l'état interne du microprocesseur. En connaissant l'état des bits de statuts de l'AVR il est par exemple possible de déterminer s'il est utile ou non de comparer une variable donnée avec zéro. Simplement parce que cette variable a été la destination d'une soustraction qui a positionnée les bits d'état de la même manière que si elle avait été comparée avec zéro – il faut encore qu'aucune autre opération n'ait été évaluée par l'unité arithmétique et logique depuis. Inutile alors de générer une opération `TST Rx` si l'opération FACADE est `v2 ← v1 == 0` par exemple. Le problème est que pour être valide il faut prouver que l'état interne des *flags* est le même quelque soit le chemin suivit pour atteindre ce point de programme (s'il en existe plusieurs). Malheureusement les algorithmes capables de répondre à cette question sont du même ordre de complexité que ceux utilisés pour l'inférence de type (présentée par la section 4.3.3 91). La solution que nous avons retenue est de « dégrader » l'algorithme qui détermine la nature du registre d'état (ainsi que la nature des registres réservés) en se limitant à une analyse linéaire. L'idée est simple : lorsqu'une opération générée modifie l'état des registres internes du microprocesseur, cette modification est notée dans le `CardCC` associé à la génération de code. Lorsqu'une opération est générée, l'information d'état est utilisée pour déterminer si elle n'a pas implicitement été réalisée par les précédentes. Par exemple, si une opération `TST Rx` doit être générée, et si l'information associée au `CardCC` courant indique que le bit d'état est associé à `Rx`, alors aucune opération n'est générée. Cependant lorsqu'une nouvelle opération FACADE est décodée, si elle est associée à un point de saut, l'ensemble des informations d'état du microprocesseur est invalidé. Ce point de saut indique que depuis un autre point du programme (disons pp_{bis}) il est possible d'atteindre le point courant. Or on ne sait rien de l'état interne du microprocesseur pour la ligne pp_{bis}. Dans ce cas nous renonçons à établir (dans une carte à puce) l'état interne du microprocesseur.

La stratégie d'utilisation des registres est encore plus ardu à déterminer pour que le code soit optimal. Il faut déterminer quelles sont les variables placées aux cœur des boucles, factoriser leurs utilisations pour en réduire le nombre, et finalement réécrire le code machine différemment pour que les variables les plus sollicitées soit placées dans des registres. L'implantation de ce type d'algorithmes dans une carte est au moins aussi irréaliste dans une carte que l'inférence de types sans assistance. Cependant contrairement au problème précédent nous pouvons cette fois utiliser les ressources présentent hors de la carte pour nous aider. La bijection établie entre les x premières variables `Temp` de FACADE et les registres de la machine permet de réaliser une première optimisation hors de la carte. Sachant

que les variables Temp sont potentiellement associées à des registres il devient possible de réaliser hors de la carte, sur le code intermédiaire FACADE les algorithmes qui classent les variables de travail depuis les plus sollicités[3] jusqu'à celles qui sont le moins utilisées. Si le mécanisme d'optimisation ne connaît pas la valeur de x (ce qui est normalement le cas) il peut proposer une version du code FACADE "sous-optimale" par rapport à ce qui pouvait être espéré « au mieux ». Mais dans tous les cas, cette version du code sera plus efficace qu'une version non optimisée.

Pour que l'optimisation de l'utilisation des registres soit totalement satisfaisante il faut néanmoins que le générateur de code encarté puisse savoir si le contenu d'une variable Temp sera utile ou pas pour la suite des opérations. Avec cette information il peut déterminer si la variable doit être « sauvée » par exemple. Pour cela l'information de type peu être utilisée. Nous avons vu dans le chapitre 4 que les variables dont le type est CardTop ne peuvent plus être utilisées car leurs contenus est quelconque (ou même indéfini). L'idée ici est de marquer les variables TTemp que l'analyse[4], faite hors de la carte, a déterminé être terminale. Je marquerais sur les exemples suivants ces variables avec le symbole $\downarrow$ et je dirais qu'elles sont « mortes ». Cette marque signifie que le contenu de la variable (et éventuellement du registre associé) n'a plus d'importance. Le moteur d'inférence place le type associé à cette variable (v) à $\top$ (c'est à dire à CardTop) ce qui est inoffensif (car $\forall c$ type de la variable v, $c \subseteq \top$ donc on ne donne pas à la variable v le droit de faire quelque chose qu'elle ne doit pas pouvoir faire). Nous avons ainsi enrichi le mécanisme d'inférence de type d'une fonction d'optimisation répartie entre la carte et le terminal.

Pour illustrer l'impact de ces différentes techniques sur la qualité du code cible détaillons un exemple de génération de code optimisé.

5.4.3 Exemple de génération de code optimisé

L'exemple dont le code source est présenté par la table 5.2 porte sur une simple méthode Java (qui est utilisée par le noyau encarté) qui recherche une valeur dans un CardTBuffer256. Si elle la trouve elle retourne son rang, dans le cas contraire elle retourne la valeur -1.

Le code Java est ensuite compilé. Le résultat obtenu est le bytecode présenté par la table 5.3. On peut constater qu'à ce point l'utilisation de la classe « native » CardPB256 est perçue comme un appel de méthode virtuel, car le langage Java connaît les classes du noyau encarté au travers d'un *package* « comme les autres ».

Le bytecode Java est adapté en utilisant un simple algorithme de décodage et de substitution du bytecode Java. Le résultat est ensuite prouvé et optimisé (réduction du nombre des variables Temp et marquage de leurs morts). Le résultat est présenté par la table 5.4. Elle montre comment, toutes les opérations du bytecode Java sont exprimés avec des invok

3. celles qui sont manipulées au cœur des boucles principalement

4. Le principe de analyse qui détermine la durée de vie des variables a notamment été formalisé par L. Lenir [Len99]

```
/**
 * détermine le numéro du label associé à la
 * ligne line, -1 si non trouvé.
 */
byte findLabel(CardPB256 label,
               byte nbLabel,
               byte line)
{
    byte j;
    j=nbLabel;
    do{
        j--;
    }while(j>=0 && label.getByte(j)!=line);
    return j;
}
```

TAB. 5.2 – *le code source de la méthode* `findLabel()`

```
>> Method byte findLabel(CardKernel/CardPB256,byte,byte) <<
>> max_stack=2, max_locals=5 <<
 0x00- 0x1c    iload_2
 0x01- 0x36 +1 istore 4
 0x03- 0x15 +1 iload 4
 0x05- 0x04    iconst_1
 0x06- 0x64    isub
 0x07- 0x91    i2b
 0x08- 0x36 +1 istore 4
 0x0a- 0x15 +1 iload 4
 0x0c- 0x9b +2 iflt 0x19
 0x0f- 0x2b    aload_1
 0x10- 0x15 +1 iload 4
 0x12- 0xb6 +2 invokevirtual CardKernel/CardPB256.getByte
 0x15- 0x1d    iload_3
 0x16- 0xa0 +2 if_icmpne 0x3
 0x19- 0x15 +1 iload 4
 0x1b- 0xac    ireturn
```

TAB. 5.3 – *le bytecode Java de la méthode* `findLabel()`

de FACADE. On voit que la durée de vie du booléen calculés par les lignes 3 et 7 n'est utilisée que par les opérations `jumpif` qui les suivent (ligne 4 et 8).

Déclarations préalables
`.FACADE 00`
`.linked test2`
`.Arguments CardPB256 _a0`
`.Arguments CardByte _a1`
`.Arguments CardByte _a2`
`.Temp l0,_e0`
`.Return CardByte`

Corps du code			
pp	Label	Instruction	Preuve (avant exécution): P[l0_e0]
2		`l0` $\leftarrow$ `_a1 asIs`	$(\top,\top)$
1	`Label1`	`l0` $\leftarrow$ `l0` $-$ `1`	(CardByte,$\top$) ($\in$ preuve)
2		`_e0` $\leftarrow$ `l0` $<$ `0`	(CardByte,$\top$)
3		`Jumpif _e0`$\downarrow \leftarrow$ `, Label0`	(CardByte,CardBool)
4		`_e0` $\leftarrow$ `_a0 getByte l0`	(CardByte,$\top$)
5		`_e0` $\leftarrow$ `_e0` $\neq$ `_a2`	(CardByte,CardByte)
6		`Jumpif _e0`$\downarrow$ `, Label1`	(CardByte,CardBool)
7	`Label0`	`Return l0`	(CardByte,$\top$) ($\in$ preuve)

TAB. 5.4 – *le code FACADE adapté à partir du bytecode de la méthode* `findLabel()`

Le code que la carte produit à partir du code FACADE est présenté par la table 5.5. On constate d'abord qu'il est relativement compact : 15 instructions assembleurs ont suffit à implanté dans la carte un traitement qui s'exprime en 8 instructions FACADE (ou 16 bytecodes Java). La plupart des optimisations évidentes ont été réalisées (utilisation de `Dec`). Mais surtout l'instruction n°3 du code FACADE n'a donné lieu à la génération d'aucun code machine : la ligne `Label1: DEC R2` correspond au code FACADE

`l0` $\leftarrow$ `l0` $-$ `1`

et la ligne suivante `BRGE LaTmp0` correspond `Jumpif _e0`$\downarrow$ `,Label0`. Cette ligne de code FACADE n'est pas complètement optimisée. L'opération `BRGE LaTmp0` pourrait aussi être remplacée par `BRLT Label0`, ainsi le `RJUMP` qui la suit devient inutile. Le système encarté à incriminer pour cette non-optimisation est le mécanisme d'édition de lien, qui comme nous l'avons vu, pour pouvoir fonctionner en une seule passe ne cherche pas à optimiser les sauts avants (*c.f.* section 5.2.2 page 117).

Parvenir à garantir l'efficacité des traitements qui le nécessite est un impératif pour pouvoir héberger de nouveau composants systèmes. L'extensibilité du système ne peut cependant pas ce borner à cette première exigence. Aussi ne

```
        PUSH R2
        PUSH R3
        PUSH R4
        PUSH R5
        MOV  R2,R18
Label1: DEC  R2
        BRGE LaTmp0
        RJMP Label0
LaTmp0: MOV  R31,R17
        MOV  R30,R2
        LD   R4,Z
        CP   R4,R20
        BRNE Label1
Label0: MOV  R0,R2
        JMP  0x0000C
```

TAB. 5.5 – *le code machine généré dans la carte pour la méthode* `findLabel()`

5.5 Introduction à l'extensibilité du modèle de chargement

Il n'est guère possible de conclure se chapitre sans préciser la démarche qui nous a amener à rendre générique la méthode de classe `compile()`.

Nous avons expliqué dans le chapitre 4 que les mécanismes d'inférences de type sont des interprètes de code qui associent un sens différent à l'exécution d'une instruction élémentaire.

Notre interprète de code FACADE statique est cependant fortement par le matériel imposé des cartes à microprocesseur. C'est pourquoi nous avons construit un interprète qui a la propriété de parcourir le code instruction après instruction sans saut avant ou arrière durant le parcourt (ce n'est pas parce qu'une instruction `Jumpif`). Pour parvenir à ce résultat nous avons du imposer une sémantique statique particulière des opérations de contrôles (`Jump`, `Jumplist`, `Return`, ...). Il est alors devenu impossible d'envisager d'étendre la sémantique de ces opérations élémentaires sans devoir réviser le mécanisme de génération de la preuve. C'est pourquoi, à ce jour, nous y avons renoncé.

Il reste toutefois l'ensemble des instructions qui n'engendrent pas de branchement. Dans ce cas, il est envisageable de permettre l'extension de la sémantique statique de ces traitements. En FACADE ces instructions sont représentées sous le format générique `Invok`. Sur ce sous ensemble limité d'instructions élémentaire nous avons pu proposer dans le contexte de la carte à microprocesseur un certain degrés d'extensibilité de l'interprète statique.

L'interprétation statique d'une opération élémentaire `Invok` par notre chargeur de code se décompose en trois points :

1. vérifier que les paramètres de l'opération sont correct et correctement typés ;
2. assurer, pour l'évaluation des opérations suivantes, que le type de la variable modifiée par le `Invok` est/devient correct ;
3. produire le code réel qui doit être exécuté.

Chaque classe, chaque type du noyau donne un sens différent à ces trois points. Nous avons pu voir que la production de code n'était pas indépendante du type qui reçoit l'`invok` (*c.f.* section 5.2.3).

Cela permet par exemple de définir statiquement de nouveaux niveaux de visibilité pour chaque méthode d'une classe chargée dynamiquement. Telle ou telle méthode de ma classe ne peut être référencée dans un nouveau traitement (en train d'être chargé) que si une condition particulière est vérifiée (visibilité de `package` Java par exemple).

Encore faut-il pouvoir garantir que les mécanismes d'extensions statique des opérations `Invok` sont ne mettent pas en péril la sécurité des logiciels encarté. En conséquence les méthodes `compile(...)` ne doivent pas pouvoir générer du code sans réaliser les tests de types indispensable à la sécurité. Concrètement cela signifie que les méthodes `compileUALload(...)`, `compileUALAdd(...)`, ...de la classe `CardCode` ne doivent pas pouvoir être utilisées par les extensions, faute de quoi il devient possible de produire un code qui forge des pointeurs est accède ainsi, sans restriction, à toute la mémoire persistante par exemple. C'est pourquoi la classe `CardCode` et ses deux implantations `CardCodeN` et `CardCodeV` limite la portée de ces méthodes au noyau encarté (*c.f.* figure 5.5 page 5.5).

Finalement pour étendre la sémantique statique des opérations `invok` les nouvelles méthodes `compile(...)` doivent appeler sur celles qui sont définit par le noyau. On retrouve alors un schéma d'extension classique d'un jeu d'instruction, où la définition d'une nouvelle opération est exprimée à l'aide d'instructions (pour nous de méthode `compile(...)`) « plus élémentaires ».

5.6 Limites de l'approche

Le code généré par le chargeur de code encarté est certes de bonne qualité, mais il est encore éloigné de ce que l'on est en droit d'attendre d'un véritable compilateur. Finalement notre enthousiasme (« avoir réussit à compiler du code dans une carte ») ne doit pas masquer le fait qu'une analyse linéaire, même « assistée » ne pourra jamais totalement rivaliser avec ce qu'aurait pu produire un véritable compilateur (sans les contraintes de la carte).

Sur l'exemple présenté par la table 5.5 page 136, une analyse globale du code généré aurait par exemple pu supprimé l'intermédiaire que constitue la variable `10` en utilisant directement le registre `R30` pour stocker la valeur qui est décrémentée à chaque itération.

Cette simplification permet de supprimer les deux `PUSH R2 et PUSH R3` , et surtout l'instruction `MOV R2, R30` placée au cœur de la boucle.

D'autres aspects des faiblesses des techniques d'optimisations sont plus difficiles à mettre en évidence sur des exemples simples à lire. Ils concernent tous un même problème : établir l'état interne du microprocesseur quelque soit les chemins d'exécution parcourus est impossible avec les contraintes actuelles des cartes à microprocesseur.

L'autre limite de notre démarche concerne l'extensibilité de la sémantique statique des opérations élémentaires, tel qu'elle est assurée par le chargeur de code. Nous avons expliqué dans la section précédente que nous n'avons pu proposer d'étendre la sémantique statique que pour un sous ensemble d'instructions. Ces limites sont introduites le mécanisme de preuve de programme et les contraintes matérielles de la carte.

Un premier prototype nous a néanmoins permis d'effectuer différentes mesures et de valider notre approche avec des résultats expérimentaux.

Chapitre 6

Résultats expérimentaux

« Quand on lui réclamait des solutions parfaites, qui écarteraient tous les risques : c'est l'expérience qui dégagera les lois, répondait-il, la connaissance des lois ne précède jamais l'expérience »

Vol de nuit. A. de Saint-Exupéry

Les deux aspects les plus novateurs de l'architecture Camille sont « l'inférence de type assistée » et « la compilation à la volée dans une carte ». Il nous est apparu nécessaire de réaliser un prototype qui valide ces deux aspects fondamentaux du noyau de notre architecture pour en prouver la faisabilité tout en respectant les limitations drastiques des cartes à microprocesseur. Il ne serait guère raisonnable, en effet, de proposer une solution pour le contexte spécifique de la carte sans la confronter à une évaluation en grandeur réelle. Notre maquette est finalement la seule preuve recevable de la faisabilité de nos solutions tant le milieu des cartes à microprocesseur est contraint.

Ce chapitre débute en présentant l'implantation de notre prototype. Nous précisons succinctement les différents aspects de sa réalisation et nous détaillons plus particulièrement la stratégie d'allocation des zones de mémoire. La seconde section présente quelques mesures de base sur le système réalisé. Ces mesures montrent les poids respectifs des différents éléments du système et le coût de base du noyau dans la carte. Les sections trois et quatre détaillent deux expériences qui nous ont permis de valider sur deux aspects différents l'usage du code FACADE et du générateur de code natif encarté pour introduire différents composants système à partir de notre maquette.

6.1 La maquette

Notre démarche ici a été de réaliser en « grandeur nature » le logiciel de base que nous avons précédemment proposé pour gérer le noyau de base encarté de Camille. Comme tout système d'exploitation notre prototype établit des passerelles entre un matériel et des

applications. Alors que les applications sont programmée dans des langages de haut niveau, le matériel d'une carte reste, aujourd'hui encore, minimaliste. Pour que notre expérience soit probante nous avons d'une part réalisé un adapteur de code qui convertit du bytecode Java en code FACADE, et d'autre part pour implanter le noyau nous avons respecté des contraintes qui sont celles, dans le contexte des cartes à puce, d'un support matériel réaliste.

6.1.1 Support matériel réaliste

Le prototype Camille à été réalisé sur une architecture matérielle basée sur le microprocesseur AVR (déjà été commenté section 1.3.3 page 16). Cette architecture dispose des éléments suivants :

- d'un microprocesseur AVR 8 bits restreint ;
- de 1536 octets de RAM ;
- de 32 kilos octets d'EEPROM destinés à stocker les données ;
- de 32 kilos octets d'EEPROM destinés à stocker du code AVR ;
- d'un port d'entrées/sorties non-sérialisé.

Pour le détail du système de sécurité, précisons encore que cette architecture matérielle déclenche une exception lorsque le microprocesseur adresse une donnée hors des espaces d'adressage de la RAM ou de l'EEPROM. Dans notre cas, les adresses de `0x0000` à `0x00FF` sont hors de l'espace adressable, leur accès déclenche une exception et c'est cette exception qui nous permet de gérer les « pointeurs `null` ». Nous avons utilisé le même principe pour contrôler le dépassement de pile. La pile d'exécution est placée de telle sorte que lorsqu'elle est pleine, elle engendre une écriture hors de l'espace adressable. Sur d'autres architectures matérielles une référence `null` pourrait référencer un autre point de la mémoire pour tirer le même bénéfice d'un matériel. Bien sûr il pourrait arriver que l'organisation des espaces d'adressage interdise ces techniques simples de gestion des erreurs, mais nous argumentons que leur intégration dans le matériel encarté est si triviale (aucun circuit spécialisé n'est nécessaire, il s'agit simplement d'organiser le bus d'adresses de telle sorte que l'on retrouve ces propriétés) qu'elle est une exigence envisageable sur toutes les microprocesseurs des cartes où l'on envisage l'implantation du système Camille.

6.1.2 Implantation du système Camille

La programmation d'un système d'exploitation intimement lié au matériel est toujours une tâche délicate qui nécessite une attention particulière. Les contraintes des cartes à microprocesseur et l'impérieuse nécessité de parvenir à mettre au point un système de taille réduite, nous a amené, après quelques premières expériences, à prendre la décision de programmer le noyau de Camille entièrement en assembleur AVR. Cette entreprise délicate et contestable sur un plan de génie logiciel est justifiée par la volonté d'obtenir des résultats minimalistes en taille de code du noyau tout en garantissant l'efficacité de son exécution. Aucun compilateur n'aurait pu optimiser un code machine comme nous l'avons

fait (notamment en termes de recouvrement de fonctions). En conséquence les résultats que nous avons obtenus se situent au plus près des possibilités nominales d'exploitation du matériel.

Cette approche « tout assembleur » ne nous a pas empêchés de respecter les structures orientées objet du noyau telles qu'elles sont présentées par la section 3.4 page 75. Le diagramme de séquence de la compilation finale par exemple (figure 5.6 page 126) est strictement respecté. La méthode de classe `Compile()` est appelée que la cible soit un type du noyau, ou une classe définie par la suite, par une application. Seule l'activité interne du noyau n'utilise pas les conventions d'appel de méthodes, lorsque cela n'est pas nécessaire.

6.1.3 Gestion des mémoires

Les systèmes d'exploitation traditionnels se partagent avec les langages de programmation différentes stratégies de gestion de la mémoire. Dans le système Linux par exemple, la gestion des pages physiques de mémoire allouées pour un programme est accomplie par le système alors que la gestion à grain fin des allocations (le `malloc()` du langage C) est une bibliothèque de base fournie avec le langage (`glibc` en l'occurence). Lorsqu'un nouveau processus est initialisé, un espace d'adressage virtuel initial est proposé. Si nécessaire il peut être élargi (par la fonction du noyau`sbcr()`). Mais ce n'est pas le système qui gère la fonction `malloc()`. Cette fonction gère un autre aspect du problème d'allocation de la mémoire. Elle assure que lorsqu'une nouvelle zone de mémoire est allouée c'est préférentiellement une zone déjà utilisée et libérée qui sera choisie. La stratégie de minimisation (en temps et en espace) de la révocation et la réallocation à l'intérieur de l'espace déjà utilisé par l'application est générée par le langage (avec en l'occurrence une technique dite de premier-meilleur avec fusion d'espaces libres[1]).

Il en va de même dans notre approche. Le système propose l'accès à des pages physiques, chaque application est à même d'adopter une stratégie spécifique pour les gérer à son propre compte. Cependant le matériel ne permet pas d'offrir un espace virtuel linéaire pour chaque application. Aussi les abstractions ont à coopérer dans un même espace d'adressage. Seule la gestion du matériel et la sécurisation des accès à la mémoire sont pris en charge par le noyau. La gestion de la mémoire persistante, pour être correcte vis-à-vis des contraintes matérielles doit reposer sur un suivi du « stress » des pages. Pour écrire dans une page de mémoire persistante il faut exécuter une routine dédiée de la classe `CardKernel`. Cette procédure incrémente un compteur de stress qui permet de gérer l'allocation et la réallocation de pages en fonction de ce paramètre matériel.

La gestion de la faible quantité de mémoire RAM se fait elle aussi au travers d'un découpage en deux pages. Ici les impératifs matériels ne justifient plus ce découpage, mais le bénéfice des techniques d'accès rapide (mais sécurisé) à des tableaux de données par rapport à une gestion plus souple de la mémoire sont évidents. La table 6.1 montre sur l'exemple des microprocesseurs AVR comment se justifie le bénéfice de notre modèle de

1. *best-first with coalescing* dans la littérature anglo-saxonne.

mémoire par rapport à un modèle abstrait (ici un tableau géré avec la sémantique du langage Java et compilé « au mieux »).

génération de code pour t=unCardTB256.at(i)	génération de code pour t=unCardTB32.at(i)	compilation optimale de t=TabByteJava[i]

MOV R31,R4
MOV R30,R2
LD Z,R6
	MOV R31,R4
	MOV R30,R2
	ANDI R30,3F
	ADD R30,R3
	LD Z,R6
		MOV R30,R4
		MOV R31,R5
		LD Z,R0
		LD Z+1,R1
		CP R2,R0
		CPC R3,R1
		BLE ok
		LDI R16,EX_OUTOFBOUNDS
		CALL THROWS_EXCEPTION
		ok: ADD R31,R3
		ADC R30,R2
		LDD Z, R6

TAB. 6.1 – *Comparaison de l'accès à une page mémoire avec l'accès (optimal) à un tableau Java*

Il ne s'agit pas, par le biais de cet exemple, de contester les choix du langage Java. Le bénéfice d'un contrôle stricte de la taille des tableaux a prouvé son intérêt à tous ceux qui ont déjà utilisé ce langage. Et dans la pratique l'usage (au niveau applicatif) de pages de taille fixe pour représenter la mémoire n'est absolument pas satisfaisant. Mais les objectifs des classes du noyau de Camille ne sont pas de supporter des logiciels applicatifs. L'objet de ces classes est de permettre l'accès le plus immédiat possible au matériel. Cette représentation de la mémoire de travail aussi bien que de la mémoire persistante permet, dans des conditions de sécurité nominales, un accès bien fondé et efficace au matériel. Nous étudierions dans la section 6.4 comment cette représentation de base peut être exploitée par des bibliothèques d'extension pour gérer un exemple de structures de données.

6.1.4 Structures de données

Le noyau gère un certain nombre de structures de données de base. Il est possible de les regrouper en quatre catégories :

Données associées à la gestion matérielle des mémoires

Chaque type de mémoire est géré en fonction de sa nature. La mémoire persistante utilise deux structures de données distinctes. La MAT[2] est l'équivalent d'une FAT[3]

2. MAT : Memory Allocation Table
3. FAT : File Allocation Table

utilisée sur un disque dur, elle définit pour chaque page de mémoire si elle est allouée ou pas. Cette seule information ne suffit pas pour gérer correctement la mémoire persistante. Une seconde table est utilisée pour mémoriser le taux d'accès en écriture de chaque cellule de mémoire. Cela permet de choisir la page la moins « stressée » et donc d'augmenter la longévité de la ressource mémoire. Pour réduire taille de cette table nous avons regroupé les pages par paquets de huit, en gérant globalement le stress pour ces huit pages. Pour réduire encore d'avantage la taille des informations de stress, il serait possible d'utiliser un générateur de nombre aléatoire pour n'incrémenter (statistiquement) qu'une fois sur 2^x le compteur de stress. Cela permet de réduire de x bits la taille de chaque valeur de la table. Comme une page est stressée après 10000 à 100000 écritures en n'incrémentant statistiquement qu'une fois sur 256 l'indicateur de stress on peut utiliser un seul octet (au lieu de deux) par information de stress pour huit pages. Au total ces structures de données représentent pour la mémoire persistante 128 octets de MAT et 256 octets d'information de stress. Pour la RAM seul une micro-MAT de 6 octets est utilisée.

Données associées à la gestion matérielle des entrées/sorties

La gestion des entrées/sorties qui est assurée par le noyau nécessite en fait simplement un tampon (aussi bien pour la lecture que pour l'écriture). Cette structure de données allouée en RAM est de 256 octets.

Données associées à la description des classes du noyau

Les informations de type des différentes classes du noyau, telles que la nature de leurs attributs, la description des méthodes (valeur de retour, nombre d'arguments,...) sont stockées sous la forme d'objet dans un ensemble de structures de données placées en mémoire persistante. A ce jour l'ensemble de ces informations de type représente 6,4 kilo-octets de données pour représenter les 23 classes du noyau. Cette information constitue un premier élément de mesure du micro-noyau encarté.

6.2 Mesures du micro-noyau encarté

Notre première implantation nous permet d'effectuer différentes mesures sur la viabilité du système de base tel qu'il est encarté. Ces mesures sont essentielles pour que nous puissions valider notre approche. Dans le contexte de la carte à microprocesseur, le premier critère mesurable, généralement totalement ignoré dans les systèmes classiques, est la taille de code du noyau encarté.

6.2.1 Taille de code du noyau encarté

Pour être viable un système d'exploitation encarté doit être compact. Un système ouvert traditionnel (JavaCard par exemple) « pèse » entre 16 et 64 kilo-octets (davantage lorsqu'on compte toutes les bibliothèques annexes, comme celles des cartes Java par exemple). Dans le contexte que nous avons défini le noyau que nous proposons n'est qu'une souche, au dessus de laquelle il est possible de charger de nouvelles routines qui construisent véritablement

un système d'exploitation. Il faut donc que le système d'exploitation de base ne soit pas trop volumineux pour qu'il puisse supporter ces extensions.

La table 6.2 reprend l'ensemble des classes que nous avons présentées dans le chapitre 3 et précise pour chacune la taille de code qu'elle représente. Cependant, la taille de chaque classe, bien que riche d'informations, ne fait pas apparaître le poids respectif des deux aspects fondamentaux du mécanisme de chargement de code, à savoir : le coût de l'inférence de types assistées et le coût de la génération de code encartée. C'est pourquoi nous distinguons dans la taille de code associé à chaque classe la quantité de code associée à chacun de ces deux aspects du système. Il faut noter que le code du noyau sert encore d'autres objectifs parmi lesquels on trouve la gestion du matériel (allocation mémoire, transmission E/S,...) et la gestion de l'abstraction minimale du système (routines de multiplication algébrique, routine de création de nouvelles classes, réception décodage et exécution de commandes FACADE pour la console, ...). Le coût d'implantation de ces fonctionnalités est donné part la colonne *Autres*. Notons enfin que le symbole ε associé à certains poids signifie que l'implantation mesurée ici n'est pas complètement finalisée et qu'elle pourrait donc légèrement évoluer dans les prochaines versions de cette maquette.

Ces chiffres sont éloquents.

Tout d'abord, la taille actuelle du noyau est satisfaisante (un peu plus de 17 kilo-octets). Elle correspond à un minimum par rapport au noyau des cartes ouvertes classiques. A titre d'information nous pouvons dire ici que le système d'exploitation du prototype de la Javacard *GemXpresso* 2.0 représente 16 Kilo-octets de code natif et 8 Kilo-Octets de bytecode additionnel pour le système. L'approche retenue est donc correcte vis-à-vis des critères de taille de code dans le contexte des cartes à puce.

L'intégration de la fonction de vérification de code à la volée représente environ 3.5 Kilo-octets de code natif. Cela parait plus que raisonnable, même pour une carte à puce. La comparaison de cette taille avec la taille du vérifieur de bytecode Java permettra (ou pas) de valider le bénéfice de l'usage de FACADE en tant que langage « simple à contrôler ». Malheureusement à ce jour il n'existe aucune publication relative au coût de l'implantation de l'algorithme proposé par E. Rose.

Pour la majorité le code encarté dans le noyau de Camille sert la fonction de génération de code natif. En effet, on voit sur le tableau 6.2 que cette fonction représente 8.8 kilo-octets de code, soit plus de la moitié du code du noyau. Cela était prévisible car la complexité des algorithmes mis en œuvre dépasse de loin la complexité du reste du système. Il n'en reste pas moins vrai que cette taille permet de valider l'idée de générer du code natif à partir du langage intermédiaire FACADE dans une carte à puce.

Si nous regardons plus en détails les chiffres présentés par la table 6.2 nous pouvons lire le bénéfice des optimisations strictes en assembleur. On voit par exemple que la taille de la classe `CardShort` est sans commune mesure avec la taille de la classe `CardByte`. En effet l'implantation des méthodes de `CardShort` exploite directement le code déjà réalisé pour les routines associées à `CardByte` alors qu'elle redéfinit quasiment entièrement l'implantation de chaque méthode. Il en va de même pour les différentes classes qui donnent une image de la mémoire « brute ». Il est extrêmement difficile de faire un découpage stricte entre

Classes du noyau Camille	*Aspect contrôle de types*	*Aspect génération de code natif*	*Autres*	*Total*
CardTop	19			19
CardKernel			1335+ε	1335
CardValue	*hérité*			0
CardByte	270	450	10	730
CardShort	35	23		58
CardBool	35	31+ε		66
CardPB32 CardPB256 CardPB032 CardPB0256 CardTB32 CardTB256 CardTB032 CardTB0256	*gestion globale* 150	*gestion globale* 400		*gestion globale* 550
CardObject	127	125	17	269
CardPObject	*hérité*	*hérité*	40	40
CardTObject	*hérité*	*hérité*	73	73
CardClass	*hérité*	*hérité*	800+ε	800
CardCode	1209	*hérité*		1209
CardCodeN	*hérité*	2575		2575
CardCodeV	*hérité*	*non implanté*		
CardCC	111	806		917
CardStream	*hérité*	*hérité*	233	233
Total	1829	4537	2508	8874

TAB. 6.2 – *Récapitulatif de la taille (en mots de 16 bits) de chaque élément du noyau.*

elles tant chacune exploite intimement le code des autres. Par contre, le bénéfice apparaît sur les chiffres. La contrepartie est évidente, le code est difficilement accessible à un lecteur extérieur. Ce choix (entre lisibilité et compacité) a été fait car en l'absence de mesure, il était à redouter que le code du noyau soit trop important. Au vu des résultats (rassurants) qui sont obtenus, une réalisation moins compacte, mais plus lisible, pourrait être envisagée.

Finalement les structures de données dont le traitement est le plus délicat sont celles qui représentent le Code natif `CardCodeN` et les structures de base du noyau `CardKernel`. La classe `CardKernel` regroupe l'essentiel des techniques d'allocation de la mémoire (notamment en gérant l'usure des pages d'EEPROM). Elle gère aussi les entrées/sorties ou encore les mécanismes de gestion des « erreurs système ». En conséquence, elle joue un rôle fondamental dans le système d'exploitation et sa taille, 2,5 kilo-octets, reste modeste. La gestion, et notamment la génération de code natif, représente, elle, 5 kilo-octets dans le système. La plus volumineuse des classes du noyau : `CardCodeN` est justifiée par notre volonté d'intégrer en son sein l'essentiel des techniques d'optimisation que nous avons présentées dans le chapitre précédent. Aussi les 5 kilo-octets de code représente-t'il une taille acceptable pour notre système.

Une fois les contraintes de compacité du noyau dépassées, les aspects plus classiques de mesure des performances du noyau peuvent être abordés. Nous nous sommes plus particulièrement intéressés aux deux aspects qui caractérisent le système Camille, en commençant d'abord par une évaluation de la fonction de vérification des types.

6.2.2 Évaluation de la fonction de vérification des types

Le mécanisme d'inférence de types que nous avons proposé dans le langage FACADE doit d'être simple à réaliser et surtout de ne nécessite qu'une faible quantité de mémoire de travail.

En fait pour évaluer le coût du mécanisme de vérification des types plusieurs aspects doivent être pris en compte. Nous exposons dans les lignes suivantes les résultats que nous avons pu constater sur les principaux points de notre prototype.

La taille des programmes prouvés

Nous avons vu, dans le chapitre 4, que le mécanisme de vérification du code repose sur l'adjonction d'une « preuve » au code FACADE qui est transmis à la carte. La taille du code fournis à la carte n'est pas de prime abord un paramètre essentiel, car les taux de transfert entre la carte et le terminal peuvent atteindre des vitesses importantes, jusqu'à 192000 BPS en respectant les normes actuelles. Le problème est que cette preuve, pour être validée par la carte doit être entièrement stockée et accédée fréquemment en lecture. Dans ces conditions, bien que l'EEPROM puisse être utilisée, elle risque de faire cruellement défaut. G.C. Necula annonce dans [Nec97] que la taille de la preuve est une fonction exponentielle de la taille du code prouvé. Il travaille sur un modèle plus général que celui que nous avons « dérivé »

pour simplifier l'inférence de type, cependant si cette constatation devait se confirmer cela soulèverait véritablement des difficultés dans le contexte d'une carte.

Le chapitre 4 précise notre démarche pour « minimiser la taille de la preuve » (section 4.4.5 page 101). Nous avons pu constater sur notre prototype qu'elle avait été payante. Les traitements exprimés dans le code FACADE complet (code + preuve) est 1,8 fois moins compact que le bytecode JavaCard (avec un écart type de ce ratio de 0,6). Ces mesures portent sur des programmes qui était en fait des extension systèmes comme celles qui seront présentées par les section 6.3 et 6.4. La taille de la preuve représente 30% (avec un écart type de 3%) de la taille du code FACADE tel qu'il est transmit à la carte.

En fait l'encodage des programmes FACADE (sans compter la preuve) est 10% moins compact que le bytecode JavaCard. Toutefois notre principale préoccupation n'a pas été de proposer un codage intermédiaire compact. Il est connu que dans ce domaine les codages organiser autour d'une machine à pile sont les plus performants. Si nous n'avons pas choisi ce type de modèle c'est simplement que les garanties de sécurité et d'efficacité des programmes encartés nous sont apparues être des critères plus importants.

Le chargement d'une application est une opération spécifique dont le porteur de la carte à conscience. Un délais important lors du chargement (résultat du traitement d'un code moins compact) sera accepté, alors qu'il est reconnu qu'un délais d'attente de plus de cinq secondes devant un guichet bancaire automatique est mal toléré par les porteurs de carte. Une fois l'application chargée son utilisation doit être rapide. En conséquence la durée totale de chargement d'un nouveau service n'est pas (dans une certaine mesure) un critère essentiel au même titre que la consommation de mémoire de travail par exemple.

Consommation de mémoire de travail

La vérification d'un traitement par la carte doit pouvoir s'exécuter avec très peu de mémoire. C'est l'un des objectifs essentiels du langage FACADE. La consommation de mémoire vive (RAM) par le processus d'inférence de type est extrêmement modeste. Pour la plupart des traitements que nous avons chargés dans la carte un seul bloc de 32 octets de RAM a été allouée par le mécanisme de vérification des types. Cela tient au fait que le nombre de variables locales T ($Card(T)$) est le plus souvent inférieur à 16. La quantité de RAM allouée par le processus de vérification de code est formellement donnée par la formule :

$$taille = 32 \times Int\left(1 + \frac{Card(T)}{32} \right)$$

Notons que pour totalement maîtriser le coût de mémoire de travail lors du processus de vérification nous avons défini deux sortes de variables locales. Les `Temp` sont des variables locales T (c'est à dire des variables locales dont le type peut varier au cours de l'exécution du programme) alors que les `Local`, bien qu'étant aussi des variables L, ont un type fixé une fois pour toutes. Cela permet de maîtriser la taille de la preuve et la taille de RAM nécessaire au processus de vérification en remplaçant certaines variables T par des

variables L. Les variables locales L déclarent un type une fois pour toutes au début du traitement, elles n'apparaissent donc pas dans les éléments de preuve. L'inconvénient et que l'utilisation systématique des `Local` augmente inutilement la quantité de mémoire de travail consommée par le traitement lors de son exécution. En effet, alors qu'une variable T peut être utilisée tantôt pour contenir des informations d'un type et tantôt celles d'un autre, il faudra deux variables `Local` pour exprimer le même traitement. Cependant si le nombre de variables T devient trop important il est possible de les remplacer par des variables locales L et ainsi permettre à la carte de vérifier le traitement.

Lors de nos expériences nous n'avons jamais eu besoin de recourir à cette solution extrême pour deux raisons. D'une part l'adaptateur de bytecode Java que nous avons mis au point minimise le nombre de variables T et les remplace par des variables Locales L lorsque leur type ne change pas du début à la fin du traitement – ce qu'un algorithme d'optimisation peut facilement entraîner, en regroupant les variables d'un même type qui ne sont pas utilisées au même moment). D'autre part au vue des quantités de mémoire de travail disponibles nous aurions pu gérer jusqu'a 256 `Temp` (cette limite est aussi imposée par le codage binaire actuel du langage FACADE). Or les programmes que nous avons expérimentés (notamment ceux présentés par les sections 6.3 et 6.4) n'ont que rarement nécessité plus d'une quinzaine de variables T (locales et arguments) par code chargé.

Finalement le processus de vérification des types du code ne constitue en lui-même qu'une partie du processus de chargement et les coûts en termes de temps d'utilisation du microprocesseur sont indissociables des autres aspects du chargeur de code. Pour véritablement mesurer l'ensemble du mécanisme il nous faut considérer aussi la partie terminale du processus en proposant une évaluation du générateur de code.

6.2.3 Évaluation du générateur de code

La fonction de génération de code est la partie du noyau de notre système d'exploitation la plus complexe à mettre en œuvre. Nous avons vu qu'elle représente la moitié du code encarté. Deux aspects critiques sont à considérer dans la génération de code. Le premier porte sur la quantité de mémoire de travail utilisée et le second est relatif au temps nécessaire pour charger des traitements efficaces.

Le générateur de code natif n'a en fait besoin que de deux structures de données (allouées en mémoire de travail) pour accomplir sa tâche.

La première lui permet de réaliser des optimisations locales dans le code qu'il génère. Il s'agit en fait de prendre en compte les informations placées dans les registres internes de l'AVR (et principalement les informations implicitement chargées dans le registre d'état). Par exemple, après avoir généré une opération de décrémentation (`Dec Rx`), il est inutile de comparer le registre décrémenté avec zéro. Le registre d'état de l'AVR est implicitement renseigné. Les quelques octets nécessaires à ce type d'optimisation sont placés dans des champs réservés de l'instance de `CardCC` qui est utilisée par le générateur de code.

La seconde structure de données utilisée par le générateur de code natif est plus importante. Elle permet de gérer le délicat problème de l'édition de liens. En effet, comme le code

généré n'est pas au même format que le code FACADE, la position des points de sauts dans le second n'ont pas de lien direct avec ceux du premier. L'opération dite d'édition de lien est difficile à réaliser dans une carte car elle nécessite de mémoriser l'adresse physique associée à chaque point de saut déclaré dans le code FACADE. Nous nous sommes contenté dans notre maquette d'implanter une technique de « chainage arrière » des instructions de saut. En conséquence seule une table qui associe à chaque point de saut une adresse physique est nécessaire. Cette table est donc d'une taille en mémoire de travail de $2 \times l$ (avec l le nombre de points de saut déclarés par le code FACADE reçu). C'est un volume de données de travail malgré tout important qui limite sur notre maquette à 256 le nombre de points de saut différents déclarés par un code FACADE (une méthode par exemple). Cette limite est restée théorique puisqu'elle n'a jamais été atteinte sur les expériences que nous avons menées. Notons encore qu'il est possible de proposer des algorithmes qui n'utilisent pas de table d'édition de liens. Notre choix est un compromis entre complexité du code entrant accepté, efficacité du chargement, et, qualité du code généré.

La seconde évaluation relative à la fonction de chargement de code caractérise ses performances dynamiques. La table 6.3 présente un ensemble de mesures que nous avons obtenues sur un échantillon de codes FACADE qui assurent des fonctions de type systèmes (recherche d'information, traitements d'entrés/sorties, calculs numériques, gestion de mémoire orientée base de données, ...). Il faut donc préciser que sur des traitements purement applicatifs, l'utilisation des classes du noyau serait sensiblement moins importante et en conséquence les résultats présentés ici pourraient varier. Cependant il n'est guère pertinent, pour ce type de logiciel, de solliciter la génération de code natif.

nature de la mesure	valeur moyenne	écart type
Nombre moyen de cycles machine pour traiter une instruction FACADE	80727.5	26296.1
Nombre de cycles (hors écriture en mémoire) pour traiter une instruction FACADE	37858.5	4596.3
Nombre de cycles machine moyen pour générer une instruction AVR	26416.5	24221.5
Nombre d'instruction AVR générée pour une instruction FACADE	3.05	1,04

TAB. 6.3 - *Mesure statistique des caractéristiques du générateur de code natif.*

La première ligne du tableau 6.3 montre à quel point l'opération de génération de code est une tâche délicate. Les 80727,5 cycles en moyenne nécessaires à la génération d'une opération représentent environ 11 millisecondes (avec un microprocesseur à 4,7 MHz). Une routine qui dépasse la centaine d'opérations FACADE prend plus d'une seconde pour être traitée par la carte. Bien que les routines système « critiques » soient rares, il faut bien reconnaître que le temps de chargement devient vite un obstacle à la génération de code natif dans la carte.

La deuxième ligne de la table 6.3 nous a amenée à envisager des solutions nouvelles au problème du délais qu'implique la génération de code virtuel. Le constat qui est fait est que plus de la moitié (nos mesures donnent 54% avec un écart type de 11%) du temps machine utilisé pour générer du code natif est occupée par des écritures en mémoire persistante. Or ces accès en écriture sont pour l'essentiel des accès séquentiels (sauf lorsqu'il s'agit de gérer l'édition de liens). L'idée est alors de placer un tampon (un cache) de mémoire rapide (RAM) entre le processus de génération de code et les mécanismes d'écriture en mémoire persistante.

Enfin le nombre d'instructions AVR générées en moyenne pour une opération FACADE est à peine de 3,05. Cela montre la proximité des opérations élémentaires du langage intermédiaire et les interfaces du noyau par rapport au matériel visé. En cela l'architecture du micro-noyau de Camille est inspirée des Exo-noyaux. Un nombre élevé d'instructions machines générées pour compiler une seule instruction FACADE montrerait que les interfaces qui représentent le matériel en sont éloignées. Cela aurait pu être le cas si par exemple la fonction de sécurité qui est assurée par le système n'avait pas été pensée de façon à être économique (en temps machine). Le contrôle des types, opération entièrement statique, couplé avec des techniques de confinement dynamique simplifié (tel que le confinement des blocs de mémoire) évite ce sur-coût inutile. Cependant ce chiffre particulièrement modeste montre aussi le bénéfice d'un microprocesseur disposant d'un large jeu de registres. La grande majorité des opérations élémentaires portent sur des variables de travail placées dans des registres du microprocesseur. En conséquence, les opérations élémentaires du noyau sont traduites le plus souvent avec des instructions AVR simples qui portent sur des registres. Il apparaît évident sur les codes machines générés que le nombre de registres disponibles dans le microprocesseur apporte un important bénéfice.

6.2.4 Bénéfices

Nous avons montré qu'un mécanisme d'inférence de type classique (c'est-à-dire non distribué entre plusieurs supports de calcul) repose sur une structure de données qui a une taille de $n \times t$, où n est le nombre de lignes de programme et t le nombre de variables T. La valeur notée t est par exemple, pour du bytecode Java la taille maximale de la *Frame* prise dans la pile d'exécution. En décomposant cette tâche entre la carte et « l'extérieur » nous n'avons plus eu besoin que d'une quantité de mémoire bornée par $l + t$ (avec $l \leq n$). Et encore cette quantité de mémoire sert non seulement au processus de vérification des types, mais aussi au processus de génération de code machine.

Ces résultats ne représentent cependant rien en eux-mêmes. L'objectif pour lequel l'architecture Camille a été construite est de permettre l'introduction, à n'importe quel moment de leur cycle de vie, dans les cartes à puce des traitements relatifs au système d'exploitation qui exigent en temps que tel un certain niveau de performance. Pour comprendre ce qu'apporte notre approche, les sections suivantes rapportent les résultats obtenus sur deux expériences particulièrement démonstratives. Le bénéfice de la génération de code natif est tout à fait compréhensible sur la première expérience: un générateur de nombre

pseudo-aléatoire.

6.3 Première expérience : un générateur de nombre pseudo-aléatoire

6.3.1 Motivation

Le bénéfice d'un code efficace n'est pas aussi évident dans le contexte des applications encartées que dans le contexte des applications de nos stations de travail. Le plus souvent, une carte est utilisée en tant que support de données mobile et sécurisé. Les traitements ne constituent alors qu'un « plus » de moindre importance. De plus, les temps d'écritures en mémoire persistante sont considérables. Dans ces conditions même si le temps d'exécution du code est considérablement amélioré, le résultat final est médiocre. Supposons par exemple qu'un programme prenne 20 milli-secondes à s'exécuter pendant lequel il écrivent une page de mémoire FlashRAM. L'écriture d'une page étant de 10ms, le temps de calcul est donc aussi de 10 milli-secondes. Si l'on propose un support d'exécution 10 fois plus performant pour exécuter le programme, le temps de calcul sera ramené à 1 milli-seconde, mais le temps d'écriture restant inchangé, on constatera un bénéfice réel de seulement 45%. L'idéal serait donc de changer de support de mémoire persistante.

Toutefois dès aujourd'hui, avec les technologies de mémoire largement utilisées, il arrive que l'inefficacité latente des machines virtuelles encartées soit dommageable. Le besoin de vitesse est manifeste par exemple dans la génération de valeurs aléatoires utilisées pour déterminer des *clefs de session* utilisées avec des algorithmes tels que RSA ou DES. Il est souvent plus pertinent, pour attaquer ce type d'algorithmes de chercher à déterminer la fonction du générateur de nombres pseudo-aléatoires plutôt que de factoriser de très grands nombres par exemple. Les techniques de génération de nombres aléatoires évoluent sans cesse et les techniques pour les attaquer aussi.

Permettre à une application de changer sa fonction de génération de nombres aléatoires pour générer les clefs qui lui servent à chiffrer ses messages est alors de première utilité. Le problème est que le code interprété par une machine virtuelle va être sollicité extrêmement souvent avant qu'une nouvelle clef ne soit générée. Aussi, la portion de code qui engendre des nombres aléatoires doit être exécutée rapidement.

6.3.2 Réalisation

De récents travaux proposent de nouvelles fonctions de générations de nombres aléatoires [Fon98]. Ce sont des fonctions basées sur l'utilisation de registres de L bits à décalage à rétroaction linéaire. Le principe de ces registres est présenté par la figure 6.1. A l'étape numéro i le nouveau bit pseudo-aléatoire généré par le registre est noté S_i. Les autres bits du registre (de S_{i+1} à S_{i+L-1}) sont utilisés pour générer un nouveau bit S_{i+L}, qui « rentre » à droite du registre. La fonction de génération de ce nouveau bit entrant est définie par la relation :

$$s_L = c_1 \times S_{i+L-1} + c_2 S_{i+L-2} + ... + c_{L-1} \times S_{i+1} + c_L \times S_i$$

où les coefficients de la fonction de décalage c_i sont binaires.

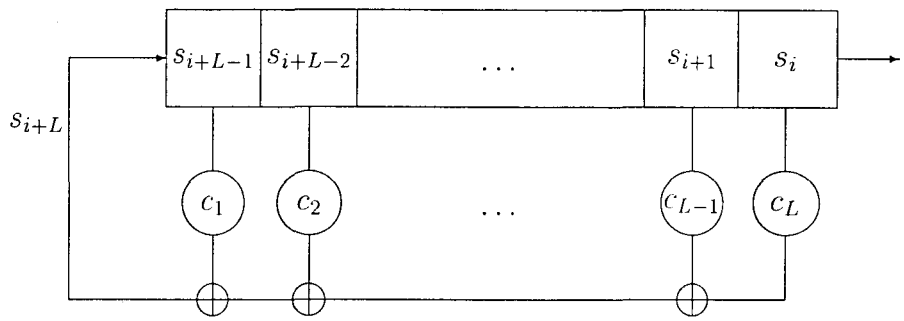


FIG. 6.1 – *Registre à décalage à rétroaction linéaire de longueur L*

Ce type de mécanisme engendre une suite de chiffres binaires qui finit par se répéter. Une fois la longueur de la suite connue il est facile de retrouver la fonction de rétroaction. Pour que ce type de calcul engendre des suites de valeurs pseudo-aléatoires suffisamment longues, il est possible de combiner par une fonction booléenne les bits sortants de plusieurs registres.

L'implantation logicielle de ce type de fonction est extrêmement simple. Le programme Java ainsi que l'ensemble des étapes de conversion de code hors et dans la carte sont présentées par l'annexe B page 179. La méthode statique `short retroaction(short etat, short f, byte L)` calcule l'état $i+1$ d'un registre en utilisant en argument l'état i, la fonction de rétroaction f et la longueur L du registre.

Un concepteur d'applications ne gère pas, en général, lui-même ce type de problème. Il utilise plutôt des bibliothèques qui lui fournissent ce service. L'objectif de notre expérience est de montrer qu'il est possible à une nouvelle application d'inclure cette fonction de génération de nombres aléatoires (via une nouvelle bibliothèque) pour qu'elle soit utilisée par le système de chiffrement de la carte. Pour déterminer la qualité (en termes d'efficacité) de notre générateur de code natif, nous devons définir un étalon de mesure.

6.3.3 Étalon de mesure

De prime abord cet algorithme ne soulève aucune difficulté pour être introduit dans une carte Java par exemple. En fait pour il suffit de définir de nouvelles classes qui héritent de la classe `javacard.security.RandomData`, définies par la référence [Sun98], afin de pouvoir surcharger la méthode `generateData(...)`. En fait il suffit de pouvoir définir au moins une souche aléatoire dans la classe `javacardx.crypto.Cipher` comme cela est le cas dans un code Java conventionnel avec le constructeur

```
public BigInteger( int bitLength, int certainty, Random rnd )
```

qui peut être utilisé pour obtenir des clefs de chiffrement de type RSA. Ce constructeur, lorsqu'il est utilisé pour produire un couple (clef privé , clef publique), demande 1536 bits au générateur de nombres aléatoires `rnd`. En supposant qu'on trouve dans la carte le même genre d'interface, la question qui se pose est : peut-on générer dans une carte les 1536 bits utilisés par le constructeur des clefs?

En fait cela ne peut pas raisonnablement être envisagé dans une carte Java. La carte GemXpresso 2.1 que nous avons utilisée comme premier étalon de performance prend 27 secondes pour générer 1536 états différents (et donc bits aléatoires) à partir de l'algorithme présenté dans l'annexe B page 179. Et encore, pour pouvoir être réellement exploitables il faudrait que plusieurs de ces registres soient couplés en cascade. Pour gérer cinq registres, il faudrait plus de 135 secondes ...

Le langage Java n'est absolument pas adapté au support de ce type de traitement (ou le temps d'exécution est essentiel, mais ou la complexité est réduite). Un second étalon est donc choisi. Il s'agit d'un compilateur C qui a pour microprocesseur cible l'AVR. Dans ce cas le langage ne prend plus en compte les problèmes de sécurité. Cependant il est bien plus adapté à la conception de composants système. Les résultats sur ce type d'algorithme sont évidement incomparables. C'est ce que montre nos résultats expérimentaux.

6.3.4 Résultats expérimentaux

Pour tester notre système encarté nous avons adapté de bytecode issu de la compilation du source Java (*c.f.* annexe B) précédent et nous avons généré le code FACADE associé. A ce stade du processus, il est possible d'appliquer (hors de la carte) différents algorithmes d'optimisation. Nous nous sommes contentés de deux types d'optimisation. La première génère l'information de durée de vie des variables, et la seconde optimise la structure des boucles (équivalent à une compilation `javac -o`). L'efficacité de chacun de ces trois programmes a été évaluée. Une fois le resultat transmis à la maquette un code machine de 62 instructions AVR est généré. Les résultats de cette évaluation sont présentés dans le tableau 6.4 qui reprend aussi les évaluations des valeurs de comparaison bytecode JavaCard interprété et code C compilé pour AVR.

La version optimisée est exécutée 85 fois plus rapidement sur notre prototype que sur une JavaCard. Cela représente une différence de plusieurs ordres d'échelle. Même par rapport à la version compilée à partir d'un source en C, le code FACADE optimisé hors carte puis converti en code natif (AVR) par la carte est encore 7% plus efficace. Ce dernier élément de résultat (inattendu il faut bien le reconnaître) est justifié par le fait que les compilateurs C pour des microprocesseurs tels que l'AVR n'implante pas des techniques d'optimisation très poussées. Plus généralement, les résultats obtenus par cette première expérience méritent quelques commentaires.

code évalué	délais d'exécution (pour 1536 bits)
Applet java encartée (GemXpesso 2.1)	26 250 ms ±0,22%
Applet java encartée (GemXpesso 2.0)	22 130 ms ±1,33%
Code FACADE non optimisé	359 ms (1 689 600 cycles machine)
Code FACADE et durées de vie	352 ms (1 658 880 cycles machine)
Code FACADE optimisé	310 ms (1 456 128 cycles machine)
Code C compilé	331 ms (1 554 432 cycles machine)

TAB. 6.4 – *temps de calcul pour générer 1536 bits avec un registre à décalage à rétroaction*

6.3.5 Commentaires

Le bénéfice évident de la génération de code natif est particulièrement mis en évidence sur cet algorithme de gestion de registres à décalage à rétroaction. Rappelons que l'approche de l'architecture Camille est de fournir dans la carte une interface la plus proche possible du matériel (ici l'unité arithmétique et logique du microprocesseur) pour gommer le surcoût d'une abstraction, quelle qu'elle soit. Notre algorithme n'exploite qu'un ensemble d'opérations arithmétiques et logiques élémentaires pour le matériel embarqué. Dans ce cas l'implantation dans la carte du code FACADE tire tout le bénéfice de la proximité permise par les types de base du système (en l'occurrence avec les classes `CardShort` et `CardByte`). Notons encore que s'il est devenu possible de générer un code natif performant c'est aussi grâce à l'inférence de types pratiquée lors du chargement. Elle apporte lors du chargement un ensemble de garanties de sécurité qui permettent de réduire autant que possible la distance entre le matériel et les logiciels qui l'utilisent.

Cependant il ne faut pas faire de contresens sur la signification de ce coefficient d'accélération (× 85). Les applications traditionnelles ne bénéficient que très partiellement de cet accroissement de vitesse, car elles reposent sur l'utilisation d'abstractions logicielles telles que les interfaces du langage Java, ou les systèmes de fichiers, qui sont éloignées du matériel. Tout l'intérêt de l'efficacité de la génération de code natif est justement de permettre de construire ces abstractions à partir du matériel disponible. Pour véritablement mesurer le bénéfice de notre approche il faut mesurer l'évaluation. C'est pour ces raisons que nous nous proposons maintenant de commenter une deuxième expérience : un système de fichiers.

6.4 Deuxième expérience : un système de fichiers

Nous nous proposons maintenant de définir un système de fichiers qui pourra être dynamiquement importé dans notre prototype en même temps que les applications qui les utilisent.

6.4.1 Motivation

Le but de cette expérience est de montrer qu'il est possible d'utiliser les bibliothèques de base fournies dans le noyau pour construire des abstractions de plus haut niveau. L'expérience est concluante si l'abstraction peut être chargée dynamiquement dans la carte sans ralentir le fonctionnement de l'application qui l'utilise.

La gestion des fichiers est un problème bien connu des systèmes d'exploitation conventionnels [Kra89, Tan89] et même des systèmes d'exploitation encartés [ISO94, Mic98]. Cependant la sémantique associée à la notion de fichier est très variable d'une carte à l'autre. Notre but ici est d'évaluer le coût, en terme de performance, d'un système de fichier, (qui est simplement perçu comme un composant système qui peut être chargé après que les cartes aient été émises) par rapport à un système de fichiers spécialement conçu pour un produit carte avant son émission.

Cette expérience doit permettre de mesurer non seulement l'efficacité d'une démarche de proximité des bibliothèques de base vis-à-vis du matériel, mais aussi, la capacité de bâtir de nouvelles abstractions véritablement utilisables. Pour cela, il faut que les bibliothèques de base qui proposent un support minimal du matériel en permettent la réalisation.

6.4.2 Réalisation

La mémoire persistante sur laquelle repose la maquette présente un ensemble de types qui permettent d'utiliser des pages de mémoire persistante. Ce sont les classes `CardPB32`, `CardPB256` et `CardPObject`. La construction d'un nouveau système de gestion de fichiers repose forcément sur l'utilisation de ces classes de base, tout simplement parce que la construction d'une structure de données persistante repose sur l'existence d'une mémoire persistante. Cependant les classes du noyau, citées précédemment ne sont pas un stricte reflet du matériel. Elles assurent deux fonctionnalités de base : la gestion du matériel et la sécurisation de son utilisation. Concrètement, cela signifie entre autres que la politique d'allocation des pages n'est pas définie par le module d'extension (ce qui distingue notre architecture d'un exo-noyau [Can98]. Le noyau de base choisit la page la plus adaptée en fonction de critères propres au matériel (usure des pages d'EEPROM dans notre cas). De plus le noyau de base sécurise l'accès à la mémoire.

L'abstraction que fournit notre système de fichiers est celle définie par les normes ISO 7816-4 [ISO94]. Il faut rappeler que ce type de fichiers permet la création de fichiers dont la taille est fixée au moment de la création. Pour le reste, on accède en lecture et en écriture

de la même manière que pour un fichier classique (avec des commandes pour identifier le fichier, pour l'ouvrir, pour le lire, pour l'écrire et pour le fermer).

La figure 2.1 page 30 présentait la structure interne du logiciel qui gère traditionnellement ce type de fichier dans une carte. L'objectif de cette expérience est de montrer qu'il est possible de construire la couche « gestion logique de la mémoire et de la sécurité » en utilisant les éléments du noyaux qui représentent la couche « Gestion physique de la mémoire ».

Notre implantation repose sur le diagramme de classes présenté par la figure 6.2 qui reprend les différentes « couches » logicielles et qui leur associe des classes. La gestion physique de la mémoire étant le propre du noyau, seuls les niveaux logiques nécessitent la création de nouvelles classes.

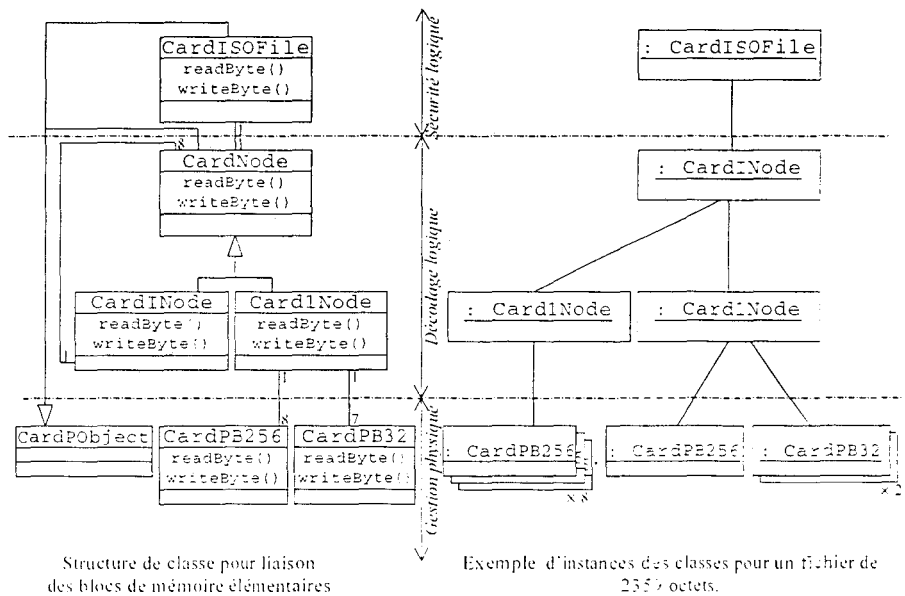


FIG. 6.2 – *Architecture d'une bibliothèque d'extension qui gère des fichiers.*

On voit sur ce diagramme que nous avons défini trois classes pour la gestion logique des fichiers (`CardNode`, `CardINode` et `Card1Node`) et une troisième pour implanter la sécurité logique (`CardISOFile`). Le rôle de la classe `Card1Node` est de définir un format de page qui permet de structurer entre eux les blocs de base qui contiennent l'information stockée dans le fichier (et qui sont des instances de `CardPB256` et `CardPB32`). Les instances de `Card1Node` sont exactement de la taille d'une page de mémoire persistante (de 32 octets). Cela limite le nombre de liens vers des blocs de base à 8 blocs de 256 octets et 7 de 32 octets. Pour pouvoir gérer des fichiers plus grands une seconde classe `CardINode` est définie. Ces instances (de la taille d'une page de mémoire) regroupent elles aussi des pages. Cependant au lieu que les pages concernées soit des `CardPBxxx` ce sont des pages de liaison directes (c'est-à-dire des instances de `Card1Node`) ou indirectes (`CardINode`).

Ainsi un fichier construit selon ce principe ne connaît pas de taille maximale, avec un niveau d'indirection il gère des fichiers pouvant aller jusqu'à de 2048 octets, avec deux niveaux 16384 octets, avec trois niveaux 131072 octets... Cette architecture à un coût en terme d'efficacité, mais avant de l'évaluer plus précisément il faut préciser ce qui doit être l'objet des mesures.

6.4.3 Objet des mesures

La réalisation d'un système de fichiers à partir des structures de base du système d'exploitation ne pose aucun problème de conception particulier. En fait notre architecture est une image aussi fidèle que possible (dans le contexte bien particulier de la carte à puce) de l'implantation d'un système de fichiers traditionnel.

Ce qui reste à montrer ici, c'est que ce composant peut être chargé dynamiquement au dessus du noyau Camille et ensuite être utilisé par l'application qui en a besoin. Cette démarche soulève deux interrogations essentielles. La première porte sur la quantité de mémoire que nécessite l'installation dynamique des classes esquissées par la figure 6.2. La seconde porte sur l'efficacité de l'implantation dans la carte.

L'acte d'écriture en mémoire persistante reste une opération très coûteuse en temps. En conséquence l'optimisation d'un code qui gère les écritures n'apporte qu'un bénéfice difficilement appréciable (à moins qu'elle ne réduise le nombre d'écritures à accomplir). C'est pourquoi seules les opérations de lecture sont relatées dans la présentation de nos résultats expérimentaux.

6.4.4 Résultats expérimentaux

La table 6.5 reprend les différentes classes que nous avons réalisées pour implanter notre système de fichiers. Pour chaque classe la taille de code ainsi que la taille des signatures est donnée. Tout le code transmis à la carte a été converti en code natif par le système d'exploitation de base.

Classe	Taille du code généré	Taille du descripteur de classe	Temps de génération
`CardNode`	112 octets	352 octets	96 ms
`Card1Node`	1 308 octets	352 octets	2 518 ms
`CardINode`	1 092 octets	352 octets	1 181 ms
`CardISOFile`	418 octets	480 octets	399 ms

TAB. 6.5 – *Taille du système de fichiers chargé sur le noyau Camille.*

Nous pouvons faire trois remarques relatives aux chiffres présentés.

1. Le temps total de chargement de l'extension « Fichier ISO » est de 4,19 secondes sur un microprocesseur AVR cadencé à 4,7 MHz. Cette durée reste importante même si

l'on retire le temps d'écriture en mémoire persistante (qui représente sur le code de ces classes 33% du temps de génération). Elle est toutefois acceptable dans le cadre d'une opération particulière (et supposée peu fréquente) d'extension du système.

2. La taille du code généré (entièrement en version AVR pour obtenir un système de fichiers performant) est assez importante. Elle ne représente néanmoins que 9% (2930 octets) de l'espace total disponible pour le stockage de code AVR. Une douzaine de composants système du même niveau de complexité peuvent donc être chargés. Tous heureusement ne sont pas aussi volumineux (le registre à décalage à rétroaction linéaire ne représente que 126 octets, soit 0,3% de l'espace total disponible). De plus, l'évolution des capacités des mémoires persistantes nous laisse penser que des cartes avec 64 kilo-octets de code seront bientôt monnaie courante.
3. La taille des descripteurs de classes (de types) d'environ 1,5 kilo-octets de mémoire de données (différenciée de la mémoire de code sur l'architecture matérielle que nous considérons) est assez volumineuse. Cette expérience nous a fait prendre conscience qu'il est essentiel de proposer une représentation compacte des informations de type. En fait ce résultat est facilement perfectible en réutilisant les déclarations héritées – cela réduirait considérablement la taille du descripteur des classes `Card1Node` et `CardINode` qui héritent entièrement leurs déclarations de `CardNode` sur notre exemple – et en supprimant les informations devenues inutiles après chargement de la classe (les déclarations des champs et les méthodes `private` par exemple).

Au vu de ces chiffres, il est tout à fait raisonnable d'encarter à la demande un système de fichiers dans la carte. L'efficacité des temps d'accès au système de fichier ainsi obtenu reste encore à évaluer. Pour établir une échelle de mesures sur ce dernier point nous avons utilisé un système de référence. Il s'agit du système de fichier de la carte Java GemXpresso 2.0 (la GemXpresso 2.1 ne propose plus ce support de fichier). La table 6.6 reprend nos résultats.

Taille du fichier	Notre implantation	GemXpresso 2.0
$\leq$ 2048 octets	25,9μs (écart type 6,4μs)	587μs (écart type 5,7μs)
$>$ 2048 octets	52,3μs (écart type 30,4μs)	587μs (écart type 5,7μs)

TAB. 6.6 – *Temps d'accès pour lire un octet dans un fichier.*

On voit sur ces chiffres que notre implantation est de dix à vingt fois plus efficace que celle des premières versions de GemXpresso. On est loin ici du rapport de 84 que nous avions constaté sur notre première expérience. Cela mérite quelques commentaires.

6.4.5 Commentaires

La structure de base (Notamment les classes `CardNode`, `Card1Node` et `CardINode` qui a été définie ici a été réutilisée telle qu'elle pour assurer l'implantation de la notion de tableau

(persistant) du langage Java. Lorsque l'on compare les temps d'accès d'un tableau dans une carte Java avec le temps d'accès des notres, on trouve un rapport comparable. Cette remarque mets en valeur un résultat fondamental de l'architecture Camille. Cette nouvelle manière de gérer le logiciel encarté ne permet pas de réaliser des supports d'exécution pour les applications particulièrement plus performants que ceux qui existent déjà. Elle permet simplement de charger lorsque le besoin s'en fait sentir de nouveaux éléments du support d'exécution, avec des propriétés différentes. C'est l'objectif principal qui était visé. Camille n'est pas une maquette de carte ouverte au même titre que les *SmartCard for Windows*, par exemple. L'architecture Camille propose une organisation telle que le système d'exploitation soit ouvert, c'est-à-dire extensible et flexible pour véritablement permettre à des programmes conçus avec des abstractions très différentes d'être placés sur le même support physique. C'est un premier pas qui est nécessaire pour permettre l'interopérabilité de la plus large palette d'applications possible dans une même carte. Ainsi la carte pourra véritablement jouer le rôle de pivot tel que nous le percevons (*c.f.* section 1.5.2 : « Pour que demain soit enfin *la nuit des temps* » page 25).

C'est sur cette remarque que se termine le commentaire de nos expériences. Bien d'autres sont encore à envisager, cependant après trois années de travaux il est possible de prendre un certain recul sur ce le travail réalisé. Il ne s'agit plus alors de mesures et de preuves, mais plutôt de l'avancement d'une réflexion personnelle autour de Camille et des systèmes d'exploitation encartés qui constitue pour ce document un ensemble de conclusions et perspectives.

Chapitre 7

Conclusions et perspectives

« Mais je comprends aussi que rien de ce qui concerne l'homme ne se compte ni ne se mesure. »

Pilote de guerre, A. de Saint-Exupéry

Ce chapitre conclut le mémoire sur des aspects moins formels que le chapitre 4 et moins chiffrés que le chapitre 6. Il s'agit ici de souligner certains des sentiments que je partage avec ceux qui y ont participé et qui nous ont été inspirés par deux années de recherche sur ce projet. Il ne faut cependant pas voir dans ces remarques une conclusion définitive à notre démarche car la rédaction de ce document ne coïncide pas avec la fin des travaux réalisés autour de l'architecture Camille.

Pour commencer nous détaillerons deux aspects des contraintes qui semblent régir la construction de notre architecture. Il s'agit des rapports latents entre la sécurité, l'efficacité, et l'extensibilité au sein de notre approche. Ensuite je présenterai les enseignements personnels que j'estime avoir tirés de ces travaux. Pour finir, le chapitre se termine sur les perspectives à venir de la recherche autour de l'architecture Camille et plus généralement autour des systèmes d'exploitation dédiés aux cartes à microprocesseur.

Limites de l'approche

Notre architecture bien que résolument ouverte et extensible reste enfermée dans un double rapport, entre d'une part *efficacité et sécurité*, et d'autre part *efficacité et extensibilité*, le modèle de sécurité que nous avons choisi en définissant un langage intermédiaire typé assimilable par la carte à microprocesseur introduit néanmoins certaines contraintes qui constituent les limites du mécanisme d'inférence de type.

Limite du mécanisme d'inférence de type

La manière dont sont gérés les types dans le langage intermédiaire FACADE entraîne des limites dans l'optimisation des programmes (hors de la carte) qu'il convient de préciser. Ces limites sont induites par l'utilisation de descripteurs de point de saut. Considérons le programme FACADE présenté par la figure 7.1. Ce programme recherche dans un fichier la première valeur supérieure à une valeur `max` passée en paramètre. S'il ne la trouve pas il retourne 0.

pp	Label	Instruction	Commentaire
		`.Return CardByte`	le traitement retourne un `CardByte`
		`.Arguments CardShort fileId`	il prend en argument n°1 un `CardShort`
		`.Arguments CardByte max`	il prend en argument n°2 un `CardByte`
		`.Temp f,i,b`	il utilise trois variables temporaires
1		`f <- CardFile open fileId`	f est un fichier de l'application
2	`bcl:`	`b <- f eof`	f en fin de fichier? (rangé dans `b`)
3		`JUMPIF b,fin`	branchement à la fin
4		`i <- f getByte`	sinon lecture d'un octet dans le fichier
5		`b <- i <= max`	valeur inférieure à max? (rangé dans `b`)
6		`JUMPIF b, bcl`	alors branchement en boucle à `bcl`
7	`fin:`	`f! close`	ici on a terminé, on ferme le fichier
8		`JUMPIF b!, fin2`	si `b` est vrai, rien trouvé, saut à `fin2`
9		`RETURN i`	sinon c'est que `i` convient
10	`fin2:`	`RETURN 0`	ici on a rien trouver on retourne 0

TAB. 7.1 – *Un exemple de code FACADE correct, efficace mais invérifiable*

Cet exemple soulève un problème d'identification de type pour la variable `i`. En effet, le point de saut marqué du label `fin` qui termine la boucle peut être provenir des lignes 3 et 6. Depuis la ligne 3 indique que la fin de fichier est atteinte, et que, éventuellement, `i` n'a jamais été affecté, son type est donc $\top$. Si le flot d'exécution provient de la ligne 6, alors la variable `i` contient la valeur recherchée. `i` est donc un `CardByte`. Finalement, le type de `i` pour le point de saut marqué `fin:` est $TT[i] = \top \cap$ `CardByte`. Le résultat ne fait pas de doute, car $\top \subseteq$ `CardByte`. La preuve déclarera donc $TT[i] = \top$. Toutefois, après que le fichier ait été fermé, ce qui doit être retourné c'est la valeur de `i` s'il a été affecté. Comme le moteur d'inférence dans la carte a établi à partir de la preuve que $TT[i] = \top$ l'opération 9 est illégale. Ce code, bien qu'utilisant convenablement ces variables ne peut pas être validé par notre mécanisme d'inférence de type.

Cet exemple a été rédigé intentionnellement, les compilateurs n'engendrent pas naturellement de programmes de cette forme. Néanmoins, il montre que si nous avons pu prouver qu'un programme incorrect n'est pas accepté par le vérifieur de type encarté, un programme correct pourra être rejeté.

Limite du rapport entre extensibilité et efficacité

Tout d'abord il est bon de rappeler qu'un système de type (ou plutôt de classe pour préciser la nature extensible du système utilisé par FACADE) n'est pas explicitement une garantie de sécurité.

Il ne suffit pas d'avoir la garantie qu'un objet partagé ne peut être manipulé qu'au travers des méthodes qu'il définit pour avoir la garantie que l'information qu'il renferme est inattaquable. Il faut encore que les méthodes qu'il propose soient correctement conçues pour apporter la sécurité de l'application.

L'inférence de type n'est qu'un outil à la disposition des architectes de sécurité logicielle. Nous nous sommes efforcés de garantir sa fiabilité (en la prouvant) et de le rendre extensible pour qu'il s'adapte à d'autres besoins. L'innovation majeure dans ce contexte est l'extensibilité de la sémantique statique, par le biais de la méthode `compile()`).

Cependant la manipulation d'un système de type peut s'avérer complexe. Des outils de plus haut niveau doivent être proposés pour simplifier la tâche des concepteurs d'applications. Il peut notamment s'agir de solutions à base de capacité [HGV00, HV00].

L'inférence de type ne dit rien a priori sur la rétrocession de droits. Une application A qui a obtenu « légalement » un objet délivré par une application B peut alors le partager avec une application C, bien que B n'ait jamais souhaité donner cet objet à C. Bien sûr le code qui est exécuté sur les données de l'objet partagé a été écrit par l'application B. Les méthodes de l'objet partagé peuvent vérifier statiquement (grâce à un identifiant de contexte d'exécution) que l'appelant à le droit de les solliciter. Néanmoins cette approche a l'inconvénient de consommer du temps machine lors de l'exécution des traitements (ce que nous essayons justement d'éviter en introduisant un système de sécurité statique dans la carte ...).

Enfin notre mécanisme d'inférence de type, bien qu'il ait été conçu de façon à permettre son extensibilité, reste fermé à certaines vérifications. La gestion de la notion d'interface, telle qu'elle est définie par le langage Java par exemple, pose des problèmes de transcription dans le système Camille.

Pourtant le principe est simple. Chaque classe contient une table des interfaces qui lui sont associées. Chaque interface est elle-même une table d'entier qui associe à un numéro de méthode d'interface un numéro de méthode de la classe. Lorsqu'une méthode `m` de l'interface `itf` est invoquée sur l'objet `o` il faut déterminer à l'aide de la table de l'interface associée à la classe de `o` quelle est la méthode réelle à invoquer. Finalement il suffit d'implanter l'appel virtuel au travers de la TMV[1] avec un code de la forme `o.TMV[itf[m]](...)` au lieu de `o.TMV[m](...)`.

Le problème est que le moteur d'inférence de notre micro-noyau ne peut pas déterminer statiquement la signature de la méthode `TMV[itf[m]]`. A partir du noyau que nous proposons, les différentes solutions que nous avons expérimentées reviennent toutes à la même

1. TMV : Table des Méthodes Virtuelles

technique. Lorsqu'un objet est utilisé au travers d'une interface Java, c'est qu'il est une sorte de `CardJavaObject`. Cette classe est ajoutée aux applications Java pour gérer les particularités des objets de ce langage. Pour invoquer une méthode d'interface sur des objets de cette sorte il faut :

1. demander à la classe de l'objet `o` de retourner le `CardCode` associé à la méthode `m` d'interface `itf` ;
2. ensuite il est possible d'utiliser la méthode `exec(...)` sur l'instance de `CardCode` qui est retournée pour déclencher l'exécution du code associé à la méthode `m` d'interface `itf`.

L'inconvénient de cette technique est simplement que la méthode `exec(...)` définie par le noyau sur les objets `CardCode` effectue une vérification de type dynamiquement, pour chaque appel. Finalement, sur cet exemple nous constatons que pour maintenir la sécurité, l'extensibilité se fait au détriment de l'efficacité. C'est un moindre mal car la propriété d'efficacité du système encarté a principalement été souhaitée pour assurer la propriété d'extensibilité. Cependant des systèmes de type plus complexes et plus performants pourraient peut-être être adaptés à la carte afin d'assurer une plus large extensibilité statique du mécanisme de type.

Enseignements

Ce mémoire ne serait pas complet si je ne donnais pas ici un résumé des leçons que j'ai tirées des différents aspects et implantations du système Camille. Il ne s'agit pas toujours ici de remarques quantifiables ou démontrables. Ce sont plutôt des convictions intimes et les conclusions personnelles que je tire maintenant de trois années de travail. Les quelques points suivants reprennent l'essentiel de mes impressions. Soulignons encore qu'il s'agit d'enseignements fondés sur mon expérience personnelle.

1. **Les informations de type sont volumineuses.** Le chapitre 6 annonce que l'ensemble des déclarations de type des classes du noyau représente 6.4 Kilo-octets de donnée (en mémoire persistante). Ce volume de mémoire (considérable dans le contexte des cartes à puce) soulève un problème que nous n'avions pas anticipé. Il est nécessaire d'organiser les structures de méta-données selon des schémas compacts (ce qui n'a pas été suffisamment le cas dans notre maquette).
2. **Définir un modèle élémentaire et sécurisé mais générique des ressources de la carte ne pose pas de problème particulier.** La construction des bibliothèques de base du noyau que nous avons encarté ne pose pas véritablement de problème. L'identification des ressources est facilitée par leur petit nombre, et elles ne changent finalement guère d'une carte à l'autre. Une fois leurs caractéristiques communes identifiées il est aisé de définir des bibliothèques qui font la synthèse de leurs utilisations. Cela est lié au fait qu'il n'existe finalement qu'un ensemble beaucoup plus modeste de ressource dans une carte à microprocesseur que dans l'informatique conventionnelle.

La difficulté porte pour l'essentiel sur les mécanismes de sécurisation de l'accès à ces ressources. Des solutions simples peuvent être trouvées pour les problèmes de confidentialité et d'intégrité une fois qu'un moteur d'inférence de type est encarté. Les attaques en terme de disponibilité sont plus délicates à traiter et semblent nécessiter un enrichissement du matériel (système d'interruption basé sur le temps par exemple).

3. **L'absence de la prise en compte de cas particuliers par le langage intermédiaire simplifie la génération de code natif spécialisé pour un microprocesseur.** Cette remarque n'était qu'une intuition forte lorsque les premières implantations du noyau ont été réalisées. Nous avons vu qu'en FACADE toutes les constantes sont traitées de la même façon. Elles sont déclarées une fois dans le code, puis utilisées comme des variables du langage. Cette uniformité du langage source simplifie la tâche du générateur de code qui doit déjà supporter les particularités du langage cible. L'introduction de disparités au niveau des opérations aussi bien qu'au niveau des variables du langage intermédiaire ne fait qu'augmenter la taille de code du vérifieur de type et du générateur de code.
4. **La définition des droits d'utilisation des méthodes de base du système est une opération délicate.** Les classes du noyau définissent des méthodes qui, si elles sont utilisées à des fins malveillantes, peuvent nuire à l'intégrité et à la confidentialité de la carte. Le noyau ne connaît à la base que deux niveaux de visibilité pour les méthodes d'une classe : `public` et `private`. L'attribution d'un niveau de visibilité aux méthodes du système devient rapidement un véritable casse-tête. L'inférence de type ne sonnera pas le glas des architectes de sécurité des systèmes encartés : elle se présente plutôt comme un outil de base, puissant – surtout lorsqu'il est couplé avec un système extensif, par le biais des méthodes `compile()` dans l'architecture Camille – mais délicat à manipuler. En tout état de cause, l'inférence de type est un mécanisme de base qui doit être enrichi avec des outils plus pratiques pour que les programmeurs d'applications maîtrisent véritablement la sécurité de leurs programmes.
5. **Il est plus simple de réaliser des bibliothèques d'extension au dessus du noyau de base du système encarté qu'en s'appuyant directement sur le matériel.** J'ai réalisé, avant de débuter les travaux commentés ici une machine virtuelle Java pour une carte à puce. Aussi ai-je pu constater, en réalisant notamment le système de fichiers (deuxième expérience commentée dans la section 6.4), qu'il est bien plus simple de réaliser un composant d'abstraction au dessus d'interfaces système sécurisées que directement au dessus du matériel (ce qui était le cas pour les composants du système comportant la machine virtuelle JavaCard).
6. **Malgré les efforts entrepris pour rendre extensible le modèle de vérification du programme, il n'est toujours pas possible de valider statiquement des opérations particulières à certains langages.** Nous avons en effet rencontré des difficultés pour implanter efficacement certaines abstractions au dessus de notre noyau. C'est le cas, nous l'avons vu, de la notion d'interface définie dans le langage Java. Ce type de problème montre comment notre stratégie de sécurité (inférence de type simplifiée) peut pénaliser l'efficacité des composants d'abstraction.

7. **La généralisation des mécanismes d'exception des différents langages intermédiaires est un problème particulièrement ardu.** L'architecture que nous avons présentée dans ce mémoire ne traite pas cet aspect difficile des langages de programmation. Le problème est duel. Il est difficile de faire une synthèse des différents mécanismes de traitement des erreurs proposés par les langages. La différence de sémantique entre les mots clefs `try {...} catch(...){...}` du langage java et l'opération `On error Gosub ...` du langage Visual Basic, qui sont pourtant tous les deux encartés actuellement révèle la nature du problème. Cependant le plus difficile dans le contexte de notre architecture est de déterminer la forme que doit prendre la notion d'exception dans le noyau, afin qu'elle puisse être sécurisée, efficace, et extensible pour supporter aussi bien un programme Java qu'un programme pour *SmartCard for Windows*. Gageons que les travaux de ceux qui cherchent à généraliser la notion de machine virtuelle contribueront dans un avenir proche à trouver une solution à la première partie de la difficulté.
8. **Le niveau d'abstraction du langage intermédiaire utilisé n'est pas un critère décisif dans la carte à puce.** Nous avons vu que FACADE est d'un niveau d'abstraction plus élevé que le bytecode Java par exemple. Cela n'a pourtant pas compliqué la tâche de gestion de ce langage. Les travaux d'E. Rose devraient nous permettre prochainement de comparer la taille d'un vérifieur de bytecode Java avec la taille de notre vérifieur de code FACADE. Cependant d'ores et déjà nous pouvons annoncer sans trop de doute que les 3658 octets qui implantent sur un microprocesseur AVR notre algorithme d'inférence de type sont un gage de la simplicité d'exploitation par la carte d'un langage de haut niveau. Il m'apparaît évident aujourd'hui que l'adaptation d'un langage intermédiaire au contexte de la carte est décorrélé de son niveau d'abstraction.
9. **Plus généralement les contraintes spécifiques au contexte des cartes nécessitent certes une démarche particulière, mais elles ne justifient pas de renoncer à y implanter des techniques informatique de pointe.** L'inférence de type et la génération de code natif dans la carte, qui sont les résultats les plus marquants de ce mémoire, s'ajoutent à une longue liste de réalisations destinées aux cartes (gestion d'une base de donnée, notion de machine virtuelle, implantation d'un gestionnaire de mémoire transactionnelle, ...) pour montrer qu'il est possible d'embarquer sur de très petits composants des algorithmes et des processus réputés complexes. Toutefois cela nécessite des aménagements qui ne sont généralement pas envisagés par les recherches et les chercheurs qui proposent des innovations dans le domaine de l'informatique.

Finalement, l'architecture présentée ici est un compromis entre trois principes fondateurs: (1) proposer une représentation simple et fiable du matériel. (2) supporter la diversité des sources de programmations. (3) assurer l'efficacité des programmes chargés. L'action principale entreprise ici a été de modifier les points de chaque critère jusqu'à ce qu'une nouvelle position d'équilibre soit atteinte. Bien que cette recherche soit délicate, il ne semble pas impossible d'en trouver d'autres pour dépasser les verrous technologiques

qui aujourd'hui encore freinent l'expansion de ce domaine de l'informatique. Ce n'est qu'au prix d'une recherche minutieuse que l'on pourra établir quel est le meilleur point d'équilibre dans le contexte des cartes à puces.

Perspectives

Évolution des système d'exploitation pour les cartes

S. Lecomte concluait en 1998 son mémoire de thèse [Lec98] par ces mots: « [...] ce mémoire démontre que l'on ne pourra plus parler de système d'exploitation générique pour carte à microprocesseur ». Cette déclaration s'appuie sur le constat que les applications de la carte à puce se sont tellement diversifiées qu'il n'est plus guère raisonnable de vouloir proposer une carte offrant une abstraction unique, et capable de répondre aux besoins de toutes les applications existantes et imaginables. Doit-on renoncer dans ces conditions à héberger sur un même support des applications trop différentes?

Deux ans plus tard, ce document propose une première ébauche de réponse. Tous les modèles de mémoires recouvrables imaginables ont une caractéristique commune. ils s'appuient sur l'existence d'une mémoire persistante. Quelles que soient les abstractions souhaitées par une application, elles s'appuient nécessairement sur le matériel. Le noyau de notre système, propose une représentation simple, efficace, sécurisée et synthétique du matériel. Il permet ainsi à chaque application de gérer « à sa façon » les ressources qu'elle souhaite exploiter.

Cette démarche est un retour aux sources. On renonce à fournir un haut niveau d'abstraction dans le noyau de notre système. Il ne s'agit pas, cependant, de dire au programmeur de renoncer au confort des langages et des outils de programmation modernes. Le but est de permettre à ceux qui conçoivent ces outils de le faire dans les meilleurs conditions. Ainsi les outils qu'utilisent les applications peuvent eux aussi être dynamiquement chargés. Pour cela le noyau de base ne se contente pas de gérer le matériel, il en donne une représentation synthétique, efficace et sécurisée, mais sans le « dénaturer ». Les outils systèmes peuvent alors être exprimés sur cette base.

De nouveaux composants matériels, tels qu'un support d'interruption, une unité de gestion de la mémoire, pourront permettre d'offrir un noyau de base plus riche. L'unité de gestion de la mémoire (MMU) permettrait par exemple de définir un système de droit d'accès sur les instances des classes `CardPB32` et `CardPB256`. Rappelons que notre prototype garantit qu'un bloc de mémoire alloué par un programme est inaccessible aux autres tant qu'il ne l'a pas partagé (il est impossible de forger des référence à partir de FACADE et des interfaces proposées dans le noyau). Mais si ce bloc est confié à un autre, plus aucune garantie n'est fournie. L'intégration d'une MMU dans le système de base permettrait de définir des politiques de contrôle d'accès sur les blocs partagés. Cependant quel est l'intérêt de ce type de partage alors que la définition d'un nouveau type dans la carte permet un contrôle beaucoup plus fin? La question mérite d'être posée.

Ces évolutions du matériel (qui sont attendues depuis longtemps, et qui finiront probablement par voir le jour) permettront de valider un autre aspect du noyau : sa capacité à s'adapter à l'évolution du matériel. L'intégration de pages de mémoire gérées par une MMU se ferait simplement en proposant un type dérivé de celui qui définit les blocs `CardPB32`. Mais l'utilisation de ce type de blocs rendraient les programmes moins portables, à moins qu'une version « tout logiciel » des blocs, avec contrôle d'accès, puisse être proposée. La capacité du noyau à s'adapter aux évolutions du matériel fait partie des perspectives de recherche avenir. Finalement le niveau de portabilité de bas niveau que nous avons retenu contraint-t'il l'évolution du matériel ?

En fait la fin des systèmes d'exploitation génériques pour carte à microprocesseur annonce l'émergence des systèmes d'exploitation extensibles dans laquelle notre approche ne constitue qu'une solution parmi d'autres.

Recherches à venir

L'architecture générale de distribution de notre système Camille, telle qu'elle a été présentée dans le chapitre 3, reste aujourd'hui encore un vaste champ de recherche pour de nouvelles expériences. Certaines sont déjà initiées, d'autres méritent de retenir notre attention.

Actuellement les cartes à puce connaissent un modèle d'exécution (lancement d'un programme, réponse à une exception, durée de vie des données, ...) extrêmement simple : le terminal émet une requête, la carte exécute le traitement associé et retourne le résultat. L'accroissement des tâches confiées à la carte ne lui permettra bientôt plus de se cantonner à un fonctionnement aussi rudimentaire. Plusieurs contextes d'exécution devront pouvoir être supportés au sein de la carte. Cela pose de nouveaux problèmes :

- Comment et où ordonnancer les communications entre la carte et différents logiciels qui la sollicitent simultanément ?
- Comment gérer dans la carte à puce les contextes d'exécution, et les problèmes de cohérence qu'ils soulèvent ?
- Comment représenter dans notre micro-noyau le matériel (la pile d'exécution et les mécanismes d'interruption) qui permettraient de supporter simultanément différentes applications utilisant leur propre représentation du partage du temps machine ?

D'autres problèmes sont soulevés par le fait que certaines routines système, que nous envisageons dans cette architecture de pouvoir charger dynamiquement, devront avoir une propriété dite de « temps constant ». Il s'agit de pouvoir garantir que, quel que soit le chemin d'exécution emprunté lors de l'exécution de la routine, le temps d'exécution reste le même. Concevoir ce type de routine n'est aujourd'hui envisageable qu'en les programmant en assembleur, et en vérifiant le temps de chaque opération dans chaque branche du programme. Il n'est pas impossible d'assurer cette propriété au niveau du langage intermédiaire.

Le programme de gestion des registres de décalage à rétroaction linéaire qui est généré par notre chargeur de code (*c.f.* Annexe B, 179) présente cette propriété (en supposant que la routine de décalage logique appelée par le `call` soit elle-même temps constante). Toutefois cette caractéristique doit pouvoir être garantie au moment-même où le programme est chargé, et quelque soit l'architecture matérielle de la carte qui reçoit le traitement.

Deux questions restent ouvertes. (1) Comment exprimer, ce type de propriété au niveau du langage intermédiaire? Et (2) comment le chargeur de code peut l'assurer dans la carte?

Une troisième problématique est mise en perspective dans notre architecture. Il s'agit de l'intégration d'éléments d'extensibilité dans le mécanisme d'inférence de type. Notre modèle permet à un programme A de définir un comportement sécuritaire (statique) propre à chacune des classes qu'il partage lorsque un nouveau programme (B) en train de se charger fait référence à des classes de A. C'est tout l'intérêt de la méthode `compile(...)`. Il est simple de concevoir, en surchargeant cette méthode appelée par le système lors du chargement, un mécanisme d'inférence de type qui puisse être étendu pour vérifier statiquement que l'accès `package` (implicite) à toutes les méthodes de la classe n'est pas transgressé. Cela permet en fait à chaque programme, à chaque classe, d'enrichir avec n'importe quel test le contrôle de type effectué statiquement. Toutefois des catégories entières de modèles de sécurité restent impossibles à valider statiquement. La gestion de la notion d'interface (tel qu'elle est définie par le langage Java), n'a pu être réalisée qu'au prix d'une vérification dynamique du type de chaque argument défini par l'objet `CardCode` associé à la méthode finalement appelée. Cette vérification est normalement faite statiquement par le langage Java. Peut-on concevoir un système d'inférence de type ouvert à des enrichissements plus forts que ceux que nous proposons avec notre système de base? Des éléments de réponse existent déjà dans la littérature associée aux méta-langages: ces techniques peuvent-elles être adaptées à la carte? Quels sont les éléments qui peuvent être distribués sur le terminal? Ces questions restent aujourd'hui encore ouvertes.

La pérennité la problématique carte

Les travaux présentés ici, comme par le passé ceux de l'équipe RD2P (dans lesquels ils s'inscrivent), ont visé à dépasser les contraintes matérielles des supports informatiques aussi exotiques que la carte à puce. Cependant la progression exponentielle des caractéristiques des composants électroniques rapprochera bientôt le haut de gamme des cartes à puce d'une « puissance critique » au-delà de laquelle il ne sera plus utile d'entreprendre des réflexions spécifiques pour adapter les progrès informatiques à des contraintes matérielles fortes.

Dans ce contexte n'est-il pas légitime de s'interroger sur le bien-fondé de recherches qui puisent leur originalité dans les contraintes qu'elles cherchent à dépasser? Ce type de recherche est-il anecdotique et voué à disparaître prochainement?

On compare parfois le comportement des informaticiens à celui des gaz parfaits. Lorsqu'une nouvelle génération de matériel offre plus de puissance, ce surplus est immédiatement

occupé par les nouvelles générations de logiciels. Cette « propriété » est probablement l'un des moteurs du progrès technologique en électronique. Il n'en reste pas moins vrai qu'elle est parfois une nuisance au développement d'objets novateurs.

Une nouvelle sorte de composant électronique trouve aujourd'hui des débouchés prometteurs : il s'agit des étiquettes électroniques. Pour caricaturer cette nouvelle technologie, on peut dire qu'elle est aux objets ce que la carte est à l'individu. Ses contraintes sont encore plus extrêmes que celles des cartes à puce. Ces composants électroniques doivent tenir sur une surface de silicium inférieure à 2 mm^2 (27 mm^2 pour les cartes). Une première expérience sur ce type de support [LSK+00] m'amène penser que ces objets pourront progresser sur les chemins tracés par la carte à puce. Dans quelques années ce qui est accompli aujourd'hui pour la carte pourra être utile à l'étiquette électronique.

En fait ce type de recherche est liée à des problèmes de « changement d'échelle ». Ce qui est démontré pour une unité l'est-il encore pour cent mille? Et pour un cent millième? L'informatique devient un support de réflexion incroyablement diversifié. Il me semble que l'attrait pour les systèmes toujours plus puissants et omniprésents, tant ils sont largement distribués, ne devrait pas effacer l'intérêt des micro-technologies, mêmes si elles ont des contraintes diamétralement opposées. La carte à puce a montré jusqu'à présent que le progrès des plus grands systèmes est lié au progrès des plus petits.

Annexe A

Codage binaire de FACADE

La structure binaire du code FACADE reconnue par le prototype réalisé est la suivante :

```
CardCodeFile {
  q8          magic;

  q2          Constant_count;
  ConstStruct Constant_pool[0..Constant_count-1];

  q2          Temp_count;

  q2          Local_count;
  q2          localClassID[0..Local_Count-1];

  q4          Instruction_count;
  q2          Label_count;
  q2          LabelDsc_count;
  LabelStruct LabelDscId[0..LabelDsc_count-1];
  q2          ProofClassId[0..Instruction_count-1][0..LabelDsc_count-1];

  q2          LinkedClassId;

  q2          ReturnClassId;

  q2          Argument_Count;
  q2          ArgumentClassId[0..Argument_Count-1];

  InstStruct  Instruction[0..Instruction_count-1];
}
```

cette notation distingue les différents « champs » du code. Le flot binaire est traité quartet par quartet de gauche à droite. qx désigne un nombre (non-signé) codé sur x

quartets.

`magic`

Cette valeur permet d'identifier le flot binaire. Elle a la valeur `0xFACADE00`.

`Constant_count`

La valeur de `Constant_count` est non-signée. Elle définit le nombre de constantes déclarées pour le traitement.

`Constant_pool[]`

Le `Constant_pool[]` est une table de structures de taille variables. Le nombre d'entrées de cette table a été fixée par `Constant_count`.

Le premier octet de chaque entrée définie la classe de la constante déclarée par ce moyen. C'est la classe identifier par ce premier octet qui définie la structure des octets suivants. En fait le chargeur de code appelle la méthode `CardTop CreateStream(CardStream aCardCodeFile)` pour que la classe construise un objet à partir du flot d'entré. La classe `CardByte` du noyau par exemple, lit un octet dans la `CardStream` pour connaître la valeur de la constante ainsi déclarée. la valeur (ou référence) retournée par la méthode `CreateStream` est ensuite utilisée par le chargeur de code.

Grâce à ce mécanisme il est notamment possible de définir une constante `CardCode` dans un code. Cette constante sera alors pré-compilée et pourra ensuite être utilisée comme un sous-programme (par exemple).

`Instruction_count`

`Instruction_count` Définit le nombre d'instructions qui composent le corps du traitement. Cette valeur est non signée. Elle peut être nulle.

`Label_count`

`Label_count` définit le nombre de point de saut dans le corps du programme. Cette valeur est non signée. Elle peut être nulle.

`LabelDsc_count`

Ce champs définit le nombre de table de description de type pour les variables `Temp` de points de branchement utilisés par le programme. Ces une valeur non signée sur deux quartets.

`LabelDscId[]`

Cette table définit autant d'entrées qu'il y a de point de branchement dans le programme. Chaque entrée de la table est composée :

1. d'un numéro d'instruction (celui auquel le point de branchement correspond) qui est codé sur 4 quartets (non signé). cette valeur ne peut pas être plus grande que `Instruction_count`. Elle désigne le numéro de ligne qui est marqué par le label ;
2. d'un numéro d'index dans la table de description `ProofClassId[][]` codé sur deux quartets. Cet indirection définie quelle ligne de la table de preuve est associée à ce point de saut.

`Instruction_count`

Cette valeur sur quatre quartets définit le nombre d'instructions qui composent le corps du traitements. Cette valeur est non signée supérieure à zéro. (un traitement vide doit au moins contenir l'instruction élémentaire `Return`.

`Label_count`

Ce champs définit le nombre de points de branchement utilisés par le programme. Ces une valeur non signée sur deux quartets.

`ProofClassId[][]`

Cette table définie un ensemble d'entrées sur des tables. Chacune de ces tables associent à chaque variable `Temp` deux quartets qui définissent la classe associée à la variable pour ce descripteur de point de branchement.

`linkedClassId`

Ces deux quartets définissent le numéro de la classe (présente dans la carte) à laquelle ce rattache ce traitement. Une valeur égale `0xFF` signifie que le code FACADE n'est attaché à aucune classe.

`ReturnClassId`

Ces deux quartets définissent le numéro de la classe (présente dans la carte) que doit retourner l'exécution du traitement. Une valeur égale `0xFF` signifie que le traitement n'a pas de valeur de retour.

`Argument_count`

`Argument_count` définit le nombre d'arguments attendu par le traitement. Cette valeur est non signée et elle peut être nulle.

`ArgumentClassId[]`

Cette table définit pour chaque argument déclaré un identifiant de classe (codé sur deux quartets). Cette information constitue avec `linkedClassId`, `ReturnClassId` et aussi `Argument_count` la signature du traitement.

`Instruction[]`

Cette table définit le corps du traitement. Pour cela elle associe à chaque entrée une instruction qui est une structure de taille variable.

La nature de l'instruction est définie par le premier quartet. L'organisation et le nombre de ceux qui suivent dépend de cette première information. Les tables A.1, A.2, A.3, A.5, A.6, A.7 ci-après reprend la signification du premier quartet et l'organisation de ceux qui suivent.

opCode	structure de l'opération

`Return` [*ReturnVar*]	`ReturnStruct {`
	`q1 ReturnId = 0x0 ;`
	`VarStruct ReturnVar`[a]`; }`

TAB. A.1 – *Table de codage binaire de l'instruction* `Return`

[a] champs définit seulement si `ReturnClassId`≠`0xFF`

opCode	structure de l'opération

`Jump` *LabelId*	`JumpStruct {`
	`q1 JumpId = 0x1 ;`
	`q2 LabelId`[a]`; }`

TAB. A.2 – *Table de codage binaire de l'instruction* `Jump`

[a] index valide dans la table `LabelDscId[]`

Les structures de données des différentes instructions utilisent parfois une autre structure qui représente les variables. Le format de cette structure dépend du premier quartet qui la catégorie de la variable. La table A.8 reprend ces différentes structures.

opCode	structure de l'opération

`Jumpif` *BoolVar,LabelId*	`JumpifStruct {`
	`q1 JumpifId = 0x2 ;`
	`VarStruct BoolVar ; }`
	`q2 LabelId`[a]`; }`

TAB. A.3 – *Table de codage binaire de l'instruction* `Jumpif`

[a] index valide dans la table `LabelDscId[]`

opCode	structure de l'opération

`Jumplist` $ByteVar,\{LabelId_i\}^n$	`JumplistStruct {`
	`q1 ReturnId = 0x3 ;`
	`VarStruct ByteVar ;`
	`q2 Label_count , (`n`)`
	`q2 LabelId[0..Label_count-1]`[a]`; }`

TAB. A.4 – *Table de codage binaire de l'instruction* `Jumplist`

[a] chacune des entrées de la table est un index valide dans la table `LabelDscId[]`

opCode	structure de l'opération

$[d \leftarrow]\ s\ m\ \{p_i\}^{f(s,m)}$	`Invok256Struct {`
	`q1 Invok256Id = 0x4 ;`
	`q2 methodId ;` (m)
	`VarStruct SourceVar ;` (s)
	`VarStruct DestVar`[a]`;`
	`VarStruct ParamVar[0..`$f(s,m)$`]`[b]`; }`
	`q2 LabelId[]`[c]`; }`

TAB. A.5 – *Table de codage binaire de l'instruction* `Invok` *(forme1)*

[a] éventuellement absent si méthode invoquée ne retourne pas de valeur

[b] le nombre d'entrées (éventuellement nulle) de cette table est fonction de la déclaration de la méthode `methodId` dans la classe associée à la variable `SourceVar`.

[c] chacune des entrées de la table est un index valide dans la table `LabelDscId[]`

opCode	structure de l'opération
$[d \leftarrow]\ s\ m\ \{p_i\}^{f(s,m)}$	IInvoktruct { q1 Invok16Id = 0x5 ; q1 methodId ; $(= m - 11)$ VarStruct SourceVar ; } (s) VarStruct DestVar[a]; } VarStruct ParamVar[0..$f(s,m)$][b]; }

TAB. A.6 – *Table de codage binaire de l'instruction* `Invok` *(forme2)*

[a] éventuellement absent si méthode invoquée ne retourne pas de valeur

[b] le nombre d'entrées (éventuellement nulle) de cette table est un fonction de la déclaration de la méthode `methodId` et de la classe associée à la variable `SourceVar`.

opCode	structure de l'opération
InvokIm	InvokImStruct { q1 InvokIm ; $\in$ [0x6..0xF] $(= m - 6)$ VarStruct SourceVar ; (s) VarStruct DestVar[a]; VarStruct ParamVar[0..$f(s.m)$][b]; }

TAB. A.7 – *Table de codage binaire de l'instruction* `Invok` *(forme3)*

[a] éventuellement absent si méthode invoquée ne retourne pas de valeur

[b] le nombre d'entrées (éventuellement nulle) de cette table est fonction de la déclaration de la méthode `methodId` dans la classe associée à la variable `SourceVar`.

entête	sens	structure effective de varStruct
0x0	*R.U.F.*	
0x1	*R.U.F.*	
0x2	une Temp	TempStruct { q1 TempId = 0x2 ; q1 Temp_number ; }
0x3	une Temp ↓	TempDStruct { q1 TempDId = 0x3 ; q1 Temp_number ; }
0x4	une Local	LocalStruct { q1 LocalId = 0x4 ; q1 Local_number; }
0x5	une Local	LargeLocalStruct { q1 LargeLocalId = 0x4 ; q2 Local_number; }
0x6	un argument	ArgStruct { q1 ArgId = 0x6 ; q1 Arg_number; }
0x7	This	0x7
0x8	*R.U.F.*	
0x9	une Classe	ClassStruct { q1 ClassId = 0x9 ; q2 Class_number; }
0xA	un attribut	AttStruct { q1 AttId = 0xA ; q1 Arg_number; }
0xB		LargeAttStruct { q1 LargeArgId = 0xB ; q2 Att_number; }
0xC	une constante	CstStruct { q1 CstId = 0xC ; q1 Cst_number; }
0xD	une constante	LargeCstStruct { q1 LargeCstId = 0xD ; q2 Cst_number; }
0xE	la constante n°0	0xE
0xF	la constante n°1	0xF

TAB. A.8 – *Table de codage binaire des variables*

Annexe B

Trace du chargement de retroaction()

Le code source exprimé en Java

```
private static short retroaction(short etat,short f,short L)
{
  short bit_retroaction;
  short x = (short)(etat & f);
  bit_retroaction = (short)0;
  short i;
  for(i = (short)0;i<L;i++)
      bit_retroaction += (short)((x >> i) & (short)1);
  bit_retroaction = (short)(bit_retroaction & (short)1);
  i = (short) ( bit_retroaction << (short)(L - 1) );
  etat = (short)((etat << 1) ^ i);
  return etat;
}
```

Le bytecode Java obtenu après compilation

```
0x00- 0x03    iconst_0
0x01- 0x3e    istore_3
0x02- 0x1a    iload_0
0x03- 0x1b    iload_1
0x04- 0x7e    iand
0x05- 0x93    i2s
0x06- 0x36 +1 istore 5
0x08- 0x03    iconst_0
```

```
0x09- 0x36 +1 istore 4
0x0b- 0xa7 +2 goto 20
0x0e- 0x1d    iload_3
0x0f- 0x15 +1 iload 5
0x11- 0x15 +1 iload 4
0x13- 0x7a    ishr
0x14- 0x04    iconst_1
0x15- 0x7e    iand
0x16- 0x60    iadd
0x17- 0x93    i2s
0x18- 0x3e    istore_3
0x19- 0x15 +1 iload 4
0x1b- 0x04    iconst_1
0x1c- 0x60    iadd
0x1d- 0x91    i2b
0x1e- 0x36 +1 istore 4
0x20- 0x15 +1 iload 4
0x22- 0x1c    iload_2
0x23- 0xa1 +2 if_icmplt 0xe
0x26- 0x1d    iload_3
0x27- 0x04    iconst_1
0x28- 0x7e    iand
0x29- 0x93    i2s
0x2a- 0x3e    istore_3
0x2b- 0x1a    iload_0
0x2c- 0x04    iconst_1
0x2d- 0x7a    ishr
0x2e- 0x1d    iload_3
0x2f- 0x1c    iload_2
0x30- 0x04    iconst_1
0x31- 0x64    isub
0x32- 0x78    ishl
0x33- 0x93    i2s
0x34- 0x82    ixor
0x35- 0x93    i2s
0x36- 0x3b    istore_0
0x37- 0x1a    iload_0
0x38- 0xac    ireturn
```

Le code FACADE fourni par l'adapteur Java

```
;  Construction de la methode :CTest_retroaction
```

```
    .FACADE 00
    .linked CTest Class
    .Arguments CardShort _CardShort0
    .Arguments CardShort _CardShort1
    .Arguments CardByte _CardByte2
    .Temp l0,l1,l2,_e0,_e1,_e2
    .Return CardShort
    l0 <- %0 asIs
    l2 <- _CardShort0 & _CardShort1
    l1 <- 0 asIs
    jump label0
label1:
    _e1 <- l2 >> l1
    _e0 <- _e1 & %1
    l0 <- l0 + _e0
    l1 <- l1 + 1
label0:
    _e0 <- l1 < _CardByte2
    jumpif _e0 , label1
    l0 <- l0 & %1
    _e0 <- _CardShort0 >> %1
    _e2 <- _CardByte2 - %1
    _e1 <- l0 << _e2
    _CardShort0 <- _e0 ^ _e1

    return  _CardShort0
```

Le code AVR obtenu dans la carte

```
4000:[0x922F] PUSH    R2
4001:[0x923F] PUSH    R3
4002:[0x924F] PUSH    R4
4003:[0x925F] PUSH    R5
4004:[0x926F] PUSH    R6
4005:[0x927F] PUSH    R7
4006:[0x928F] PUSH    R8
4007:[0x929F] PUSH    R9
4008:[0x92AF] PUSH    R10
4009:[0x92BF] PUSH    R11
400A:[0x92CF] PUSH    R12
400B:[0x92DF] PUSH    R13
400C:[0x2422] CLR     R2
400D:[0x2433] CLR     R3
```

```
400E:[0x2E60] MOV     R6, R16
400F:[0x2E71] MOV     R7, R17
4010:[0x2262] AND     R6, R18
4011:[0x2273] AND     R7, R19
4012:[0x2444] CLR     R4
4013:[0xC010] RJMP    +0x0010         ; ( Destination :4024 )
4014:[0x2DA6] MOV     R26, R6
4015:[0x2DB7] MOV     R27, R7
4016:[0x2DE4] MOV     R30, R4
4017:[0x940E] CALL    0x0027
4019:[0x2EAA] MOV     R10, R26
401A:[0x2EBB] MOV     R11, R27
401B:[0x2C8A] MOV     R8, R10
401C:[0x2C9B] MOV     R9, R11
401D:[0xE0A1] LDI     R26, 0x01       ; 1, +1b
401E:[0x27BB] CLR     R27
401F:[0x228A] AND     R8, R26
4020:[0x229B] AND     R9, R27
4021:[0x0C28] ADD     R2, R8
4022:[0x1C39] ADC     R3, R9
4023:[0x9443] INC     R4
4024:[0x1644] CP      R4, R20
4025:[0xF880] BLD     R8, 0
4026:[0xF36C] BRLT    -0x00000013     ; ( Destination : 00004014 )
4027:[0xE0A1] LDI     R26, 0x01       ; 1, +1b
4028:[0x27BB] CLR     R27
4029:[0x222A] AND     R2, R26
402A:[0x223B] AND     R3,R27
402B:[0x2E80] MOV     R8, R16
402C:[0x2E91] MOV     R9, R17
402D:[0x9496] LSR     R9
402E:[0x9487] ROR     R8
402F:[0x2EC4] MOV     R12, R20
4030:[0x94CA] DEC     R12
4031:[0x2DA2] MOV     R26, R2
4032:[0x2DB3] MOV     R27, R3
4033:[0x2DEC] MOV     R30, R12
4034:[0x940E] CALL    0x003D
4036:[0x2EAA] MOV     R10, R26
4037:[0x2EBB] MOV     R11, R27
4038:[0x2D08] MOV     R16, R8
4039:[0x2D19] MOV     R17, R9
403A:[0x250A] EOR     R16, R10
```

```
403B:[0x251B] EOR     R17, R11
403C:[0x2E00] MOV     R0, R16
403D:[0x2E11] MOV     R1, R17
403E:[0x940C] JMP     0x000004          ; (Destination 0004)
```

Annexe C

Glossaire

— A —

A.P.D.U.: *Application Protocol Data Unit*, les APDU s'appuient sur les TPDU pour définir les couches 5,6 et 7 du modèle OSI.

A.P.I.: *Application Programming Interface*, ensemble d'interfaces logicielles qui à pour but de faciliter la programmation des applications.

— B —

B.P.S.: Bits Par Secondes, unité de mesure du débit d'un port d'entrées/sorties.

Bibliothèques système désigne un ensemble de programmes, de routines, et d'interfaces, qui gèrent le matériel pour le compte des applications.

— C —

Cartes MULTOS cartes qui supporte un système d'exploitation MULTOS (*c.f.* MULTOS).

Cartes ouvertes désigne les cartes à microprocesseurs capable de recevoir dynamiquement, à chaque étape de leurs cycle de vie de nouvelles applications.

C.I.S.C.: *Complex Instruction Set Computer.*

C.M.O.S.: *Complementary Metal-Oxide Semiconductor*, désigne une famille technologique de composants électroniques.

C.O.A. : *Card Object Adapter*, Objet d'Adaptation de la Carte sur un bus IIOP.

C.Q.L. : *Card Query Language*, langage dédié à la carte qui permet de formuler des requêtes d'interrogation sur les informations organisées selon un schéma de base de donnée.

— D —

D.S.T. : Décodeur Sécurisé de Traitements. Composant encarté de l'architecture Camille, qui gère le processus de chargement d'un nouveau programme dans la carte.

— E —

E.E.P.R.O.M. : *Electric Erasable Programmable Read Only Memory.* Cela désigne une technologie de mémoire persistante.

Encarter terme propre aux fabriquants de carte. Il désigne le procédé de collage du micromodule dans le support de plastique. Par extension il est aussi utilisé pour désigner aussi le placement des programmes dans la carte.

Exo-Noyau traduction du terme *ExoKernel* dans la littérature anglo-saxonne.

Expansion de code l'acte d'expansion de code correspond à ce qu'accompli un compilateur C pour implanter une fonction marquée du modifieur `inline`. Il duplique le code de la fonction là ou elle est appelée.

Expansion de code virtuel généralement connues sous le nom de *direct-threading*. Cette technique consiste à remplacer le pseudo-code d'une machine virtuelle par une instruction native qui est un saut à la séquence d'instruction machine qui implante l'opération virtuelle.

Extension système désigne le fait d'ajouter de nouvelles bibliothèques système pour supporter des applications qui ont des exigences particulières vis-à-vis du matériel.

— F —

FlashRAM : la mémoire FlashRAM (aussi appelée Flash) est une mémoire EEPROM (*c.f. EEPROM*) dont l'architecture à été modifiée pour gérer une granularité d'écriture moins fine, mais des délais d'écriture moins importants.

FeRAM : RAM Ferro Electrique. Nouvelle technologie de mémoire persistante, plus performante que l'EEPROM et la FlashRAM.

— G —

G.I.E. : Groupement d'Intérêt Économique.

G.S.M. : *Global System for Mobil communication*, norme européenne de gestion des réseaux de téléphone mobiles.

— I —

I.I.O.P. : *Internet Inter-ORB Protocol.*

Informatique omniprésente traduction du terme anglo-saxon *Ubiquitous Computing.* Désigne une nouvelle manière de penser les technologies informatiques très largement répandues et plus discrètes que celles qui existent aujourd'hui.

Interfaces système Interface de programmation proposer par le système d'exploitation de base pour programmer des abstractions du matériel.

I.S.O. : *International Standardization Organisation*, organisme de standardisation international.

— J —

JavaCard est un nom déposé pour les cartes qui supporte l'exécution de programmes exprimés avec le langage Java.

— M —

Masquage désigne procédé d'impression du logiciel sur le support de silicium des cartes à microprocesseur commandées par les fabricants.

Mécanisme système(s) mécanisme définit par un (ou des) système(s) d'exploitation.

M.E.L. : *MULTOS Executable Language*, langage spécifiant le format des instructions exécutables par les cartes MULTOS.

Mémoire d'objet mémoire décomposée en blocs de taille variables qui représentent des objets et qui contiennent nécessairement un lien vers un descripteur de classe. On parle de *boxed units* par opposition à *unboxed units* lorsqu'il est impossible de retrouver des informations sur la nature d'un bloc mémoire à partir de sa seule référence.

M.M.U. : *Memory Management Unit*, ces mécanismes peuvent servir à accélérer matériellement la gestion des droits d'accès et/ou la gestion des mémoires virtuelles.

M.L. : *Meta Langagues*, Génération de langage de programmation intégrant des propriétés de généricités et de polymorphismes de haut niveau.

MULTOS : *MULTi-application Operating System.*

M.V.A. : Machine Virtuelle Adaptative.

M.V.R. : Machine Virtuelle Récursive.

M.V.V. : Machines Virtuelles Virtuelles.

— O —

O.S.I. : *Open System Interconnexion*, modèle de référence pour la communication entre des systèmes informatiques interconnectés de l'ISO.

— P —

P.C.C. : *Proof Carrying Code*, Code comportant sa preuve.

Point Mémoire circuit électrique qui engendre la mémorisation d'un bit (ou d'une unité élémentaire d'informations).

— R —

R.A.M. : *Random Access Memory*, mémoire de travail.

Réseaux bancaires désigne les réseaux numériques dédiés aux transaction bancaires.

R.I.S.C. : *Reduce Instruction Set Computer.*

R.O.M. : *Read Only Memory*, mémoire morte.

Routine système(s) procédure définit par un (ou des) système(s) d'exploitation.

R.S.A. : *Rivest, Shamir et Adelman*, algorithme de chiffrement à clef privée.

— S —

S.A.C. : Support d'Adapteurs de Codes. Composant logiciel qui assure la cohésion des mécanismes de conversion de code.

S.E.B. : Système d'Exploitation de Base. Micro-noyau minimaliste et proche du matériel qui est construit autour d'un mécanisme de chargement, de génération et d'exécution de code.

Sémantique dynamique désigne le sens à donner à l'exécution de chaque instruction d'un langage.

Sémantique statique désigne le sens à donner au chargement de chaque instruction d'un langage. (par opposition à la sémantique dynamique d'un langage).

S.F.I. : *Software Fault Isolation.*

S.I.M. : *Subscriber Identity Module*, Module d'identification de l'abonné sur les réseaux GSM.

S.L.K. : *Safe Language Kernel*, nom d'un noyau expérimental basé sur une machine virtuelle Java.

SmartCard for Windows sont des cartes capables de recevoir un pseudo-code défini société Microsoft.

S.Q.L. : *Structured Query Language*, Langage normalisé de manipulation de bases de données.

— T —

T.A.C. : Terminal d'Administration de Camille. Désigne une console qui d'échanger des commandes interactives avec le système d'exploitation encarté de l'architecture Camille.

T.A.L. : *Typed Assembly Language*, Langage assembleur enrichi des informations de type.

T.C.P. / I.P. : *Transmission Control Protocol / Internet Protocol.*

Technique système(s) technique ou technologie propre à un (ou des) système(s) d'exploitation.

T.P.D.U. : *Transport Protocol Data Unit*, équivalents d'une trame de TCP/IP.

— U —

U.M.L. : *Unified Modeling Language.*

— V —

Vérifieur de bytecode Java léger traduction du titre *Lightweight Bytecode Verifier.* Il s'agit d'une technique de simplification du processus de vérification du bytecode Java par adjonction de la preuve au traitement prouvé (*c.f.* PCC).

Bibliographie

Références

[Abr96] J.R. Abrial. *The B Book, assigning Programs to Meanings.* Cambridge University Press, 1996.

[AK96] R. Anderson and M. Kuhn. Tamper resistance - a cautionary note. In *he Second USENIX Workshop on Electronic Commerce Proceedings*, 1996. `http://www.cl.cam.ac.uk/users/rja14/tamper.html`.

[Amb00] Scott W. Ambler. The uml v1.1 and beyond: The techniques of object-oriented modeling, February 2000. AmbySoft Inc. White Paper `http://www.ambysoft.com/umlAndBeyond.html`.

[AMO+98] K. Asari, Y. Mitsuyama, T. Onoye, I. Shirakawa, H. Hirano, T. Honda, T. Otsuki, T. Baba, and T. Meng. FeRAM circuit technology for system on a chip. In *Proceedings of the The First NASA/DoD Workshop on Evolvable Hardware*, 1998. `http://computer.org/proceedings/eh/0256/02560193abs.htm`.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, principles, Techniques, and tools.* Addison Wesley publishing compagny, 1986. ISBN 0-201-10088-6.

[Atm00] Spécification Atmel. Avr 8-bit risc - data sheets, 2000. `http://www.atmel.com/atmel/products/prod200.htm`.

[Bai99] Carine Baillarguet. Mvv: Langage & système, plus qu'un mariage de raison. In *Journée des Jeunes Chercheurs (JCS99)*, jun 1999. `http://www-sor.inria.fr/publi/MLSPUMR_jcs99.html`.

[BGL+96] P. Biget, P. George, S. Lecomte, P. Paradinas, and J.J. Vandewalle. Procédé de chargement d'un programme d'utilisation dans un support à puce, 1996. Brevet International, numéro d'enregistrement 96-162-12. Gemplus, France.

[BP99] Carine Baillarguet and Ian Piumarta. An highly-configurable, modular system for mobility, interoperability, specialization, and reuse. In *2nd ECOOP Workshop on Object-Orientation and Operating Systems (ECOOP-OOOSWS'99)*, jun 1999. `http://www-sor.inria.fr/publi/HCMSMISR_ecoop99.html`.

[BSP+95] B.N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Beker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the spin operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain Resort, CO, USA, dec 1995.

[Can98] George Candea. *Flexible and Efficient Sharing of Protected Abstractions.* PhD thesis, MIT University,, May 1998. `http://amsterdam.lcs.mit.edu/exo/theses/candea/thesis.ps`.

[Car98] D. Carlier. *Representation permanente, coordonnée par une carte à microprocesseur, d'un utilisateur mobile.* PhD thesis, Université de Lille 1, France, 1998.

[CCG95] V. Cordonnier, C. Cormier, and G. Grimonprez. Picorisc: A r.i.s.c. approach for smartcards. In *21st IEEE EuroMicro Conference (EuroMicro'95)*, 1995.

[CD94] D. Cheriton and K. Duda. A caching model of operating system kernel functionnality. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 179–193, nov 1994.

[CLT96] D. Carlier, S. Lecomte, and P. Trane. Smartcard use to manage user's mobility. In *CARDIS'96, Second SmartCard Research and Advanced Application Conference*, 1996.

[CLT97] D. Carlier, S. Lecomte, and P. Trane. The use of a representation as a solution to wireless drawbacks: Application in the context of smart card. In *IEEE-ICPWC'97, Bombay*, 1997.

[Cor92] Vincent Cordonnier. Assessing the future of smart cards. In *CardTech Conference*, September 1992.

[Cor94] C. Cormier. Picorisc, architecture d'un microprocesseur spécifique aux cartes à accès protégés. Technical report, L.I.F.L., Univ. Lille 1, France, 1994. (mémoire de DEA de l'université de Lille 1).

[Cor96] Vincent Cordonnier. The future of smartcards: Technology and application. In *IFIP World Conference on Mobile Communication*, Camberra, September 1996.

[Dab00] Lamine Dabouz. Extension du domaine de confiance pour les programmes et les données, jun 2000. (mémoire de DEA de l'université de Lille 1).

[DGL98] D. Donsez, G. Grimaud, and S. Lecomte. Recoverable persistant memory of smartcard. In *3rd Smart Card Research and Advanced Application Conference (CARDIS'98)*, number 1820 in Lecture Note in Computer Science, pages 13–26, Louvain-la-Neuve, Belgique, sep 1998. Springer.

[DKL+98] J.F. Dhem, F. Koeune, P.A. Leroux, P. Mestré, J.J. Quisquater, and J.L. Willems. A practical implementation of the timing attack. In *Proceedings of the Third Smart Card Research and Advanced Application Conference*, Louvain-la-Neuve, sep 1998.

[Dwy95] M. Dwyer. *Data Flow Analysis for verifying Correctness Properties of Concurrent Programs*. PhD thesis, Université du Massachusetts, Etats-Unis, 1995.
`http://www.cis.ksu.edu/~dwyer/papers/thesis.ps`.

[EK95] Dawson R. Engler and M. Frans Kaashoek. Exterminate all operating system abstractions. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 78–83, Orcas Island, Washington, May 1995. IEEE Computer Society.
`http://www.pdos.lcs.mit.edu/papers/exo-abstract.ps`.

[Eng98] Dawson Engler. *The Exokernel Operating System Architecture*. PhD thesis, MIT University, Cambridge, Massachusetts, Oct 1998.
`http://amsterdam.lcs.mit.edu/exo/theses/engler/thesis.ps`.

[ESP98] Programme ESPRIT EP8670. Cascade : "chip architecture for smartcard and intelligente device", 1998. projet Européen.

[Fab74] R. S. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7):403–411, Jul 1974.

[Fon98] C. Fontaine. *Contribution à la recherche de fonctions booléennes hautement non linéaires, et au marquage d'images en vue de la protection des droits d'auteur.* PhD thesis, Université Pierre et marie curie - Paris VI, France, nov 1998.

[FPR97] Bertil Folliot, Ian Piumarta, and Fabio Riccardi. Virtual virtual machines. In *Proceedings of the 4th Cabernet Radical Workshop*, sep 1997. `http://www-sor.inria.fr/publi/VVM_radical97.html`.

[FPR98] Bertil Folliot, Ian Piumarta, and Fabio Riccardi. A dynamically configurable, multi-language execution platform. In *SIGOPS'98 Workshop*, 1998. `http://www-sor.inria.fr/publi/DCMEP_sigops98.html`.

[Fra93] M. Franz. Emulating an operating system on top of another. In *Practice and Experience*, volume 23, pages 677–692, Jun 1993. `http://www.ics.uci.edu/~franz/publications`.

[Fra97] M. Franz. Run-time code generation as a central system service. In *Proceedings HotOS-VI*, May 1997. `http://www.ics.uci.edu/~franz/publications`.

[GJ91] Paulo Guedes and Daniel Julin. Object-oriented interfaces in the mach 3.0 multi-server system. Oct 1991. `ftp://ftp.cs.cmu.edu/project/mach/doc/published/multiserver_interface.ps`.

[GJ99] G. Grimaud and S. Jean. Gum-e^2: une approche orientée carte à microprocesseur pour la gestion sécurisée de la mobilité de l'utilisateur dans les échanges électroniques. In *Trusded Electronic Tread (TET'99)*, pages pp. 415–429, Marseille, France, jun 1999.

[GJ00a] G. Grimaud and S. Jean. Carte et mobilité. In *4ème Ecole d'Informatique des Systèmes Parallèles et Répartis*, Toulouse, France, feb 2000.

[GJ00b] G. Grimaud and S. Jean. Interoperability of services in multi-applications smart cards, new approaches for security and flexibility. In *1st Eurosmart security conference*, Marseille, France, jun 2000.

[GLV99] Gilles Grimaud, Jean-Louis Lanet, and Jean-Jacques Vandewalle. Facade: A typed intermediate language dedicated to smart cards. In *Software Engineering - ESEC/FSE'99*, number 1687 in Lecture Note in Computer Science, pages 476–493, Toulouse, France, oct 1999. Springer.

[GM99] Neal Glew and Greg Morrisett. Type-safe linking and modular assembly language. In *26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Antonio, TE, USA, jan 1999. `http://simon.cs.cornell.edu/Info/People/jgm/papers/mtal.pdf`.

[Gon97] L. Gong. Java security: Present and near future. In *IEEE Micro*, volume 17:3, pages 14–19, Jun 1997.

[GR83] A. Goldberg and D. Robson. *SMALLTALK-80, the language and its implementation.* Addison-Wesley, 1983. ISBN 0-201-11371-6.

[Gri91] G. Grimonprez. *Card Query Language.* PhD thesis, Université de Lille 1, France, 1991.

[Gri97] Gilles Grimaud. Mecan'os: Evaluation des techniques de génie logiciels utilisables dans le domaine des dossiers portables, 1997. (mémoire de DEA de l'université de Lille 1).

[Gri99] Gilles Grimaud. Une architecture nouvelle pour carte à microprocesseur ouverte. In *Première Conférence Française sur les Systèmes d'Exploitation*, pages 13–26, Rennes, France, jun 1999.

[HCC+98] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing multiple protection domains in java. In *USENIX Annual Technical Conference*, New Orleans, LA, USA, June 1998.

[HE98] C. Hawblitzel and T. Von Eicken. A case for language-based protection. Technical Report TR98-1670, Cornell University, mar 1998. `http://www.cs.cornell.edu/slk/papers/TR98-1670.pdf`.

[HGV00] D. Hagimont, G. Grimaud, and J.J. Vandewalle. Contrôle d'accès par capacités pour des applications coopérantes dans le contexte de la carte à puce, 2000. Brevet International, numéro d'enregistrement 96-162-12, Gemplus, France.

[HHL+97] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of *micro*-kernel-based systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, 1997.

[HHM97] D. Hagimont, O. Huet, and J. Mossiere. A protection scheme for a corba environment. In *ECCOP'97*, Jyväskyllä, Finland, June 1997. `http://www.cis.upenn.edu/~shap/EROS/pos96.ps.gz`.

[HI97] D. Hagimont and L. Ismaïl. A protection scheme for mobile agents on java. In *Proc. Third ACM/IEEE Int. Conf on Mobile Computing and Networking (MobiCom'97)*, Budapest, September 1997. `http://sirac.imag.fr/PUB/97/97-mobicom-PUB.ps.gz`.

[Hoe97] M.P. Van Hoecke. *Contribution à la modélisation des Systèmes d'Information Communicationnels intégrant des cartes à microprocesseur.* PhD thesis, Université de Lille 1, France, 1997.

[HV00] D. Hagimont and J.J. Vandewalle. Jccap: capability-based access control for java cards. In *CARDIS 2000: Smart Card Research and Advanced Application*, Bristol, Angletterre, September 2000.

[IEE85] ANSI IEEE. IEEE Standard for Binary Floating-Point Arithmetic, 1985. Std. 754-1985, IEEE, New York.

[ISO87] International Standard Organisation: ISO. Carte d'identification - carte à circuit intégré à contacts - partie 1: Caractérisitques physiques, 1987.

[ISO88] International Standard Organisation : ISO. Carte d'identification - carte à circuit intégré à contacts - partie 2 : Dimensions et emplacements des contacts, 1988.

[ISO89] International Standard Organisation : ISO. Carte d'identification - carte à circuit intégré à contacts - partie 3 : Signaux électronique et protocole de transmission, 1989.

[ISO94] International Standard Organisation : ISO. Carte d'identification - carte à circuit intégré à contacts - partie 4 : Commandes intersectorielles pour les échanges, 1994.

[ISO95] International Standard Organisation : ISO. Carte d'identification - carte à circuit intégré à contacts - partie 5 : Systèmes de numérotation et procédures d'enregistrement d'identificateurs d'applications, 1995.

[ISO96] International Standard Organisation : ISO. Carte d'identification - carte à circuit intégré à contacts - partie 6 : Eléments de données intersectoriels, 1996.

[ISO99] International Standard Organisation : ISO. Carte d'identification - carte à circuit intégré à contacts - partie 7 : Commandes intersectorielles pour langage d'interrogation de carte structurée (scql), 1999.

[JDL00] S. Jean, D. Donsez, and S. Lecomte. Utilisation des bases de données pour la flexibilité de services coopérants dans la carte à microprocesseur. In *Actes du congrès INFORSID XVIII*, pages 117–129, Lyon. France, may 2000.

[JW75] A.K. Jones and W.A. Wulf. Towards the design of secure systems. *Software pratice and Experience*, 5(4):321–336, 1975.

[Kah74] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information processing 74*, pages 471–475, North-Holland, 1974.

[KEG+97] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 52–65, Saint-Maló, France, October 1997.

[KF99] T. Kistler and M. Franz. A tree-based alternative to java byte-codes. *International Journal of Parallel Programming*, 27(1):21–34, Feb 1999. `http://www.ics.uci.edu/~franz/publications/TreeBasedAlternativePrepub.pdf`.

[Kra89] Serge Krakowiak. *Principes des systèmes d'exploitation des ordinateurs*. Dunod, 1989.

[Lan98] J.J. Lanet. The use of formal methods for smart cards, a comparison between b and sdl to model the T=1 protocol. In *International Workshop on Comparing Systems Specification Techniques*, Nantes, France, Mar 1998.

[Lan00] J.J. Lanet. Formal methods and smart cards. In *JavaCard tutorial, ECOOP 2000, Nice*, jun 2000.

[LD97] Sylvain Lecomte and Didier Donsez. Intégration d'un gesdtionnaire de transaction dans des cartes à microprocesseur. In *Colloque international NOTERE'97*, Pau, France, November 1997.

[Lec98] S. Lecomte. *COST-STIC, Carte Orienté Service Transactionnel & Service Transactionnel Intégrant des Cartes.* PhD thesis, Université de Lille 1, France, 1998.

[Len99] Yannick Lenir. Analyse de bytecode java, 1999. (mémoire de DEA de l'université de Lille 1).

[Lev83] Bruce W. Leverett. *Register Allocation in Optimizing Compilers.* UMI Press, 1983. ISBN 0-201-10088-6.

[Lev84] H. M. Levy. Capability-based computer systems, 1984.

[LGD99] S. Lecomte, G. Grimaud, and D. Donsez. Implementation of transactional mechanisms for open smartcard. In *Gemplus Developer Conference (GDC'99)*, Paris, France, jun 1999.

[LR98] J.L. Lanet and A. Requet. Formal proof of smart card applets correctness. In *3rd Smart Card Research and Advanced Application Conference (CARDIS'98)*, Lecture Note in Computer Science, Louvain-la-Neuve, Belgique, sept 1998. Springer.

[LSK+00] M. Latteux, D. Simplot, R. Kalinowski, D. Brienne, and G. Grimaud. Procédé d'identification d'Étiquettes Électroniques par rondes adaptatives, 2000. Brevet International, numéro d'enregistrement 99 07239, Gemplus, France.

[LY96] T. Lindholm and F. Yellin. *The Java (TM) Virtual Machine Specification.* Addison-Wesley, 1996.

[Mao98] Maosco Ltd. «multos» web site, 1998. `http://www.multos.com`.

[Mic98] Microsoft Corp. «smart card for windows» web site, 1998. `http://www.microsoft.com/windowsce/smartcard/`.

[Mit00] Mark Mitchell. Type-based alias analysis: Optimization that makes c++ faster than c. Dr. Dobbs Journal, octobre 2000. `http://www.ddj.com/articles/2000/0010/0010d/0010d.htm`.

[MV96] MasterCard and Visa. Secure electronic transaction, book 3: Technical specifications, jun 1996.

[MWCG98a] Greg Morrisett. David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. In *25th Symposium on Principles of Programming Languages*, San Diego, CA, USA, jan 1998. `http://simon.cs.cornell.edu/Info/People/jgm/papers/tal.ps`.

[MWCG98b] Greg Morrisett. David Walker, Karl Crary, and Neal Glew. Stack-based typed assembly language. In *Workshop on Types in Compilation*, Kyoto, Japan, mar 1998. `http://simon.cs.cornell.edu/Info/People/jgm/papers/fullstal.ps`.

[Myk00] Gaute Myklebust. The AVR microcontroller and C compiler co-design, 2000. ATMEL Development Center, Trondheim, Norway
http://www.atmel.com/atmel/acrobat/compiler.pdf.

[Nec97] Georges C. Necula. Proof-carrying code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, jan 1997.
http://www.cs.cmu.edu/~necula/popl97.ps.gz.

[NL96] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *2nd Symposium on Operating System Design and Implementation*, oct 1996.
http://www.cs.cmu.edu/~necula/osdi96.ps.gz.

[OMG97] Object Managment Group: OMG. A discussion of the object managment architecture, 1997.
http://cgi.omg.org/library/oma1.html.

[Pel95] T. Peltier. *La Carte Blanche: un nouveau système d'exploitation pour objets nomades.* PhD thesis, Université de Lille 1, France. 1995.

[PH97] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface, Second Edition.* Morgan Kaufmann Publishers, 1997. ISBN 1-55860-428-6.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[PR98] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, jun 1998.
http://www-sor.inria.fr/publi/ODCSI_pldi98.html.

[PV94] P. Paradinas and J.J. Vandewalle. Procédé de conduite d'une transaction entre une carte à puce et un système d'information. 1994. Brevet Français, numéro d'enregistrement 2-720-848, Gemplus. France.

[RAA+92] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the chorus operating system. pages 39–69. Apr 1992.

[Rat97] Rational. *The Unified Modeling Language v1.1 Documentation Set.* Rational Software Corporation, Monterey California, 1997.

[RCG00] A. Requet, L. Casset, and G. Grimaud. Application of the b formal method to the proof of a type verification algorithm. In *The 5th IEEE High Assurance Systems Engineering Symposium (HASE 2000)*, Albuquerque, New Mexico, November 2000.

[RR98] Eva Rose and Kristoffer H. Rose. Lightweight bytecode verification. In *Formal Underpinnings of Java, OOPSLA'98 Workshop*, Vancouver, Canada, oct 1998.
http://www-dse.doc.ic.ac.uk/~sue/oopsla/rose.f.ps.

[Sch89] Bruce Scheneier. *Applied Crytography.* John Wiley & Sons, Inc., 1989. ISBN 0-471-59756-2.

[SESS96] M. Seltzer, Y. Endo, C. Small, and K Smith. Dealing with disaster : Surviving misbehaved kernel extensions. In *Proceedings of the Second Symposium on Operating Systems Design and implementation*, pages 213–228, oct 1996.

[SFS96] Jonathan S. Shapiro, David J. Farber, and Jonathan M. Smith. State caching in the eros kernel – implementing efficient orthogonal persistence in a pure capability system. In *7th International Workshop on Persistent Object Systems*, Cape May, N.J, 1996.
`http://www.cis.upenn.edu/~shap/EROS/pos96.ps.gz`.

[Sha97] Zhong Shao. Typed common intermediate format. In *USENIX Conference on Domain-Specific Languages*, Santa Barbara, CA, USA, Oct 1997.

[SS94] C. Small and M. Seltzer. Vino : an integrated platform for operating systems and database research. Technical Report TR-30-94, Harward, 1994.

[Sun96] Sun Microsystems. The javacard(tm) 2.1.1 language subset and virtual machine specification, 1996.
`http://java.sun.com/products/javacard/javacard21.html#browseDoc`.

[Sun98] Sun Microsystems. Java card 2.1 specification, 1998.
`http://java.sun.com/products/javacard/`.

[Sun00] Sun Microsystems. Jdk 1.2 security documentation, 2000.
`http://java.sun.com/products/jdk/1.2/docs/guide/security/index.html`.

[SV96] G.I.E. Sesam Vital. Cahier des charges sesam-vital, dec 1996. Version 1.0.

[Tan89] Andrew Tanenbaum. *Les Systèmes d'exploitation.* InterEditions, 1989. ISBN 2-7296-0259-2.

[Tra95] P. Trane. *Conception et réalisation d'un système de contrôle d'accès pour la carte à microprocesseur.* PhD thesis, Université de Lille 1, France, 1995.

[Van95] Jean-Jacques Vandewalle. Loading several services into multi-purpose integrated circuit card: distribute functions to users, gather data into cards! In *European Research Seminar on Advances in Distributed Systems*, Grenoble, France, April 1995.

[Van97] J.J. Vandewalle. *OSMOSE: Modélisation et Implémentation pour l'interopérabilité de services carte à microprocesseur par l'approche orientée objet.* PhD thesis, Université de Lille 1, France, 1997.

[VG00] J.J. Vandewalle and G. Grimaud. Procédé de factorisation des codes pour cartes à puces permettant des chemins de migration depuis plusieurs languages sources vers plusieurs plate-formes matérielles ou logicielles cibles, 2000. Brevet Internationnal, numéro d'enregistrement 99 07239, Gemplus, France.

[VV98] Jean-Jacques Vandewalle and Eric Vétillard. Developing smart card-based applications using java card. In *3rd Smart Card Research and Advanced Application Conference*, Louvain la Neuve, Belgique, September 1998.

[WB97] M. Weiser and J. S. Brown. *The Coming Age of Calm Technology Beyond Calculation: The Next Fifty Years of Computing.* P. Denning and R. Metcalfe Springer-Verlag, New York, 1997.

[Wei93] M. Weiser. Some computer science issues in ubiquitous computing. *Communication of the ACM*, 36(7):75–85, jul 1993.

[WGB99] M. Weiser, R. Gold, and J. S. Brown. The origins of ubiquitous computing research at parc in the late 1980s. *IBM Systems Journal - Pervasive Computing*, 38(4):693, jul 1999.
`http://www.research.ibm.com/journal/sj/384/weiser.html`.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *14th ACM Symposium on Operating System Principles*, Asheville, NC, USA, dec 1993.
`http://http.cs.berkeley.edu/~tea/sfi.ps`.

[WLH81] W. A. Wulf, R. Levin, and S. P. Harbsion. *Hydra/C.mmp: An Experimental Computer System.* McGrawHill, 1981.

Table des matières

Table des figures

Liste des tableaux