


Université des Sciences et Technologies de Lille

Thèse de Doctorat

Spécialité: **Informatique**

présentée par **Nadia Bel Hadj Aissa**

pour obtenir le titre de Docteur de l'Université des Sciences et Technologies de Lille

---

Maîtrise du temps d'exécution de logiciels déployés dans des dispositifs personnels de confiance

---

Soutenue le 29 Octobre 2008 devant le jury composé de:

Jean-Louis Lanet	Professeur	XLIM	(Président)
Isabelle Puaut	Professeur	Université de Rennes 1	(Rapporteur)
Jean-Dominique Decotignie	Professeur	EPFL/CSEM	(Rapporteur)
Jean-Jacques Vandewalle	Ingénieur de recherche	Gemalto	(Examinateur)
David Simplot-Ryl	Professeur	USTL	(Directeur)
Gilles Grimaud	Maître de conférence	USTL	(Co-encadrant)

Thèse préparée au sein du Laboratoire d'Informatique Fondamentale de Lille

Merci à tous ceux et toutes celles qui ont contribué à l'aboutissement de ce travail par un éclairage scientifique, une aide logistique, un geste amical ou simplement une parole encourageante.

Table des matières

I

Introduction générale

> *Home is behind, the world ahead,*
> *And there are many paths to tread*
> *Through shadows to the edge of night,*
> *Until the stars are all alight.*
>
> — Tolkien - The Fellowship of the Ring

> *Write everyday. Line by line, page by page, hour by hour. Do this despite fear. For above all else, beyond imagination and skill, what the world asks of you is courage...*
>
> — Robert Mckee - Story : Substance, Structure, Style and The Principles of Screenwriting

Contexte

Les constants progrès techniques de l'électronique en terme de miniaturisation ont mené à un fort développement de l'informatique ubiquitaire. Parallèlement à ce développement, le nombre et la diversité des petits systèmes informatiques sont en plein essor depuis quelques années. Nous avons, ainsi, assisté à l'apparition et la prolifération des téléphones portables, des assistants personnels (PDA [1]), étiquettes électroniques (RFID [2]), capteurs de terrains et autres cartes à puce. Ces petits systèmes informatiques, que nous désignons désormais par *Petits Objets Portables et Sécurisés*, sont les cibles principales des travaux de recherche menés au sein de l'équipe-projet POPS [3]. Ces travaux traitent conjointement les domaines des réseaux mobiles et des systèmes d'exploitation embarqués et visent à endiguer les difficultés de programmation de logiciels pour les POPS. Plus précisément, produire des logiciels sûrs destinés à ce type de cibles — plateformes non conventionnelles et souvent propriétaires — requiert un haut-niveau d'expertise de la part du développeur. Le défi de l'équipe-projet POPS consiste, dans ce cadre, à placer l'intelligence dans les outils plutôt que de s'appuyer sur la seule expertise du programmeur. De façon générale, la problématique de recherche explorée dans cette thèse repose sur le savoir-faire développé au sein de l'équipe-projet POPS. Mes travaux constituent un prolongement de ceux déjà entrepris sur la conception de systèmes d'exploitation ouverts pour carte à microprocesseur [Gri00].

Par ailleurs, mes travaux de thèse s'inscrivent dans le projet européen INSPIRED [4] [Ins04]. L'objectif de ce projet était de porter un regard sur le futur de la carte à puce à l'horizon des cinq voire dix prochaines années afin de prendre de la distance avec le carcan historique imposé par le standard ISO7816 [5] [Org99]. La notion de *dispositif personnel de confiance*, ou ce que nous appellerons TPD [6] dans le reste du document, a ainsi vu le jour avec pour objectif de remplacer dans un futur proche la carte à puce. Le projet INSPIRED s'est achevé au début de l'année 2007. Néanmoins, le consortium formé par les différents partenaires est resté actif et a contribué, en l'occurrence, à la spécification de la nouvelle version de JAVA CARD [Jav08]. Je me suis attelée, dans ce cadre, à développer une méthodologie et à fournir une chaîne d'outils permettant aux TPDs de maîtriser la consommation des ressources des logiciels et en particulier les temps d'exécutions au pire cas [7].

Orientations

Bien qu'il soit difficile de retenir une définition exacte des systèmes embarqués, la littérature nous renseigne sur leurs propriétés générales. (i) En opposition aux systèmes généralistes, ces systèmes embarqués sont dédiés à un domaine d'application spécifique. Ils sont conçus pour une utilisation très précise. Les différents composants matériels et logiciels intervenant dans leur mise au point sont choisis dans le but de répondre au mieux à cette utilisation. (ii) Dans le même but, logiciel et matériel sont fortement couplés. (iii) Le matériel et le logiciel sont utilisés de façon autonome, ou en interaction avec leur environnement. (iv) Ils sont généralement inclus dans un système plus vaste qu'ils contribuent à faire fonctionner. Il faut néanmoins faire la distinction entre les systèmes embarqués ouverts (*e.g.* TPDs) et fermés. Ces derniers sont des objets de la vie

1. PDA est l'acronyme de Personal Digital Assistant.
2. RFID est l'acronyme de Radio Frequency IDentification.
3. POPS est l'acronyme de Petits Objets Portables et Sécurisés.
4. INSPIRED est l'acronyme de INtegrated Secure Platform for Interactive tRusted pErsonal Devices.
5. ISO est l'acronyme de International Organization for Standardization.
6. TPD est l'acronyme de Trusted Personal Device.
7. Par souci de concision, nous allons utiliser, dans le reste du document, l'acronyme WCET pour Worst Case Execution Time en lieu et place du terme français "temps d'exécution au pire cas".

courante munis de capacités de calcul fournies par des composants intelligents enfouis tels qu'une carte embarquée dans une voiture pour contrôler l'injection. Cette carte ne sera pas accessible pour d'éventuelles mises à jour de son logiciel. Ces systèmes embarqués fermés souffrent ainsi, d'un manque de flexibilité dû à la limitation de l'intervention humaine a posteriori pour changer, augmenter les fonctionnalités disponibles.

A l'opposé, la prolifération de nouveaux équipements ouverts et programmables (*e.g.* TPD) a favorisé l'essor des environnements d'exécution dynamiquement adaptables ayant la possibilité de déployer à la volée de nouveaux services après leurs émissions. C'est la notion de *post-issuance* se définissant comme la capacité d'enrichir les fonctionnalités des TPDs après leurs mises en circulation (*i.e.* après qu'ils soient déployés donc délivrés à leurs porteurs). Afin de permettre cette flexibilité, il est indispensable de disposer d'un mécanisme de chargement dynamique de code, tel que les librairies dynamiques [Sma02] ou le chargement dynamique de classes dans les machines virtuelles JAVA [LY99] ou encore le moteur d'exécution CLR [8] [CLR07].

Dans ce cadre, l'utilisation des mécanismes issus de la programmation orientée objet, et notamment les notions d'héritage et de polymorphisme, s'impose. En effet, la programmation objet fournit le substrat nécessaire permettant la réutilisation mais aussi l'extensibilité. Loin du logiciel monolithique développé sur mesure d'un ancien temps, les applications doivent désormais être évolutives et correspondent souvent à un assemblage de briques logicielles fournies par des sources diverses qui ne se font pas mutuellement confiance. L'utilisation des cadres applicatifs [9] objets dans lesquels les logiciels destinés aux TPDs viennent intégrer leur code spécifique est l'un des principes clés que nous avons défendus au sein du consortium INSPIRED.

L'enjeu majeur de cette nouvelle tendance consiste principalement à trouver un compromis avantageux entre une flexibilité accrue et la nécessité d'apporter à la fois sûreté de fonctionnement et sécurité. En effet, la flexibilité permet à l'utilisateur de déployer de nouveaux services à la volée, quand le besoin s'en ressent. Néanmoins, le TPD doit être en mesure de garantir l'absence de défaillance au cours de l'exécution des logiciels — tant les nouveaux que ceux préalablement déployés. Elle ne doit pas non plus ouvrir des brèches de sécurité menaçant l'intégrité et la confidentialité des données (ou traitements) confidentielles contenues dans le TPD, ou encore la disponibilité des services offerts.

Un large pan de la littérature sur ce sujet se limite aux deux premières préoccupations. Nous avons choisi de nous intéresser à la maîtrise des temps d'exécution des logiciels en vue d'assurer la disponibilité des services offerts par le TPD.

Objectifs

A travers cette thèse, nous aspirons donc, à garantir que chaque nouveau logiciel dont le déploiement (installation) sur le TPD aboutit, est en mesure de délivrer les réponses aux requêtes qui lui sont adressées dans un délai maximal établi préalablement. Ces garanties apportées sont cruciales en terme de sûreté de fonctionnement dans la mesure où les logiciels installés sont assurés, par la plateforme hôte, d'offrir leurs services dans un laps de temps fixé. En terme de sécurité, ces garanties permettent d'assurer qu'indépendamment des nouveaux logiciels installés dans le TPD ou des évolutions du système d'exploitation, la disponibilité des services est préservée.

8. CLR est l'acronyme de Common Language Runtime
9. Le terme anglo-saxon est *framework*.

Il est nécessaire de rappeler les hypothèses fortes que nous devons prendre en compte tout au long de cette thèse.

(i) *La garantie doit être établie a priori* : L'application doit savoir à l'installation si elle est fonctionnelle ou pas. Si elle ne l'est pas, l'installation est amenée à échouer, et le service ne sera pas activé.

(ii) *Le* TPD *doit être autonome* : La sécurité du TPD ne peut pas reposer sur la sécurité d'un système tiers.

(iii) *Les* TPD *sont très hétérogènes* : La solution proposée ne peut pas reposer sur des hypothèses liées à un matériel particulier.

Enfin, la garantie apportée doit être comprise comme un système d'isolation des ressources. Les applications travaillent avec leurs ressources (ici le micro-processeur) sans souffrir d'éventuelles interférences induites par d'autres applications. Dans un premier temps, nous prendrons l'hypothèse que les applications n'interagissent pas entres elles ; l'interférence sera donc limitée à la réservation de ressources. Cependant, cette hypothèse ne peut pas toujours être consentie. Aussi, nos travaux les plus récents s'intéressent aux mécanismes nécessaires pour permettre de relâcher cette dernière hypothèse simplificatrice. Pour atteindre nos objectifs, nos travaux se sont portés sur la définition et l'implémentation d'une chaîne d'outils, certains étant intégrés dans une chaîne de compilation, lors de la production du logiciel, alors que d'autres prennent placent dans le noyau du système embarqué, pour valider les logiciels lors de leurs déploiement. Nos travaux trouvent ainsi, leur justification dans leur exploitation pratique en prise directe avec l'industrie de la carte à puce fortement représentée dans le consortium INSPIRED.

Structure du mémoire

Dans cette thèse, nos contributions visent à garantir la maîtrise des temps d'exécution pour les logiciels destinés au TPDs. Ce mémoire se décompose en cinq chapitres, outre la présente introduction.

Le **Chapitre II** sert à dessiner les contours de la problématique que nous avons adressé dans nos travaux. Nous commençerons par donner les directions de l'évolution souhaitée par le projet INSPIRED de la traditionnelle carte à puce vers le TPD. Ensuite, nous énumérerons les propriétés des logiciels embarqués en mettant en avant celles qui doivent être préservées dans le cadre d'un déploiement sécurisé. Enfin, à travers un tour d'horizon, nous survolerons les techniques et méthodes de calcul de temps de d'exécution au pire cas.

Le **Chapitre III** décrit notre approche, distribuée, permettant à un consommateur de maîtriser la complexité algorithmique du code déployé.

Dans le **Chapitre IV**, nous appliquerons ce premier résultat sur le calcul de WCET dans des conditions d'expérimentations correspondant aux profils des TPDs définis dans le projet européen INSPIRED. Ces conditions mêleront d'une part une plateforme matérielle basée sur un coeur de processeur 32-bits et d'autre part un système d'exploitation ouvert où l'on peut charger dynamiquement des logiciels destinés à s'exécuter sur des TPDs.

Dans le **Chapitre V**, nous généraliserons les résultats obtenus pour gérer les interactions ou interférences entre plusieurs applications émanant de fournisseurs de code qui ne se font pas mutuellement confiance.

Les éléments clés de mes contributions, leurs limites ainsi que les perspectives de nos travaux seront synthétisés dans le **Chapitre VI**.

II

Problématique

> *Tous les chemins mènent au même endroit. Mais choisissez le vôtre, et allez jusqu'au bout. N'essayez pas de parcourir tous les chemins.*
>
> — Paulo Coelho - *Maktub*

> *Nadia, il ne faut pas chercher à tout mettre dans une thèse, il faut simplement emmener le lecteur à travers ta vision personnelle du problème !*
>
> — Gilles Grimaud - *Le temps d'une mise au point*

Dans ce chapitre, nous commencerons par donner les directions dont nous avons discuté dans le cadre le projet INSPIRED pour l'évolution de la traditionnelle carte à puce vers le dispositif personnel de confiance ou TPD. Ensuite, nous dresserons une liste d'exigences permettant d'assurer un déploiement sécurisé des logiciels embarqués. Nous énumérerons les solutions qui y répondent et que nous jugeons pertinentes. Dans ce sens, nous étayerons nos propos par des exemples d'environnements d'exécution en mettant en avant les propriétés qui doivent être préservées dans le cadre d'un déploiement sécurisé. À travers un tour d'horizon, nous survolerons les techniques et méthodes de calcul de temps d'exécution au pire cas. Nous finirons par une synthèse qui éclairera les propositions que nous détaillerons dans le reste du document.

II.1 Contexte de l'étude

Dans cette section, nous commencerons par donner un bref aperçu de la carte à puce telle qu'elle se présente de nos jours. Nous présenterons, par la suite, la réflexion amorcée dans le projet INSPIRED en vue d'ébaucher le concept de dispositif personnel de confiance (TPD). Nous mettrons l'accent, dans cette description, sur les nouveaux défis posés par cette évolution.

II.1.1 La carte à puce

II.1.1.1 Caractéristiques matérielles

Les cartes à puce se caractérisent par des contraintes matérielles fortes imposées par l'espace que les composants intégrés peuvent occuper sur une surface de $25mm^2$ [Org99]. Elles sont pourvues de processeurs de faible puissance, et de mémoires de petite taille, réunis sur un même substrat de silicium. Typiquement, une carte à puce (*e.g.* carte bancaire) est constituée d'un coeur de processeur 8 bits cadencé à une fréquence de 4 MHz, disposant de 6 à 32 Ko de ROM [1], de 256 à 2048 octets de RAM [2] et de 1 à 32 Ko d'EEPROM [3] (Voir Table II.1). Dans ce qui suit, nous allons présenter séparément chaque élément de l'architecture encartée.

Table II.1 — *Configuration matérielle d'une carte à puce (2006)*

Composants	Caractéristiques
CPU	8-16-32 bits
RAM	256 octets à 1 Ko
ROM	jusque 32 Ko
EEPROM	256 octets à 64 Ko

1. ROM est l'acronyme de Read Only Memory.
2. RAM est l'acronyme de Random Access Memory.
3. EEPROM est l'acronyme de Electrically-Erasable Programmable Read-Only Memory.

Processeur Du produit d'entrée de gamme au circuit le plus performant, chaque génération de puces évolue vers des processeurs plus complexes (de 8 bits vers 16 bits, voire 32 bits). Le type de processeur utilisé dépend des normes ISO [Org99] concernant la résistance de la carte aux torsions et aux flexions. Par ailleurs, les besoins des utilisateurs évoluant vers des applications embarquées de plus en plus complexes, les concepteurs choisissent des circuits plus performants. Les processeurs 8-bits suffisent, par exemple, pour les cartes à puce dont le système d'exploitation n'inclut pas d'interpréteur de code portable (*e.g.* contrairement à JAVA CARD). En ce qui concerne les processeurs 32-bits, leur utilisation reste limitée par la taille du substrat de silicium (*i.e.* un processeur 32-bits occupe plus de place qu'un processeur 8-bits) ou par leur consommation énergétique. Il est souvent plus économique, en effet, d'investir dans l'optimisation du logiciel plutôt que d'utiliser un processeur occupant plus d'espace sur le silicium. Pour toutes ces raisons, les cartes à puce utilisent des processeurs dont le fonctionnement reste assez simple (*i.e.* pas de cache d'instructions et de données ou de circuit évolué de gestion de la mémoire type MMU [4] ou TLB [5]). En revanche, elles peuvent être pourvues de coprocesseurs additionnels servant à l'accélération d'algorithmes de chiffrement, tel que DES [6], ou à la génération de nombres aléatoires utile pour la génération des clés de session pour les réseaux GSM [7] ou 3G [8].

Mémoires En ce qui concerne les différents types de mémoires disponibles sur une carte à puce, elles se distinguent par leurs propriétés de modification et de persistance, leurs latences, leurs consommations énergétiques, la surface occupée sur le circuit intégré, etc... Souvent utilisée comme mémoire de travail (*i.e.* permettant de stocker des données temporaires telles que des clés de session), la mémoire RAM, bien que très rapide, n'est présente qu'en faible quantité à cause de son encombrement. La ROM non réinscriptible sert à stocker les routines du système d'exploitation, les procédures de test et le code applicatif lors du masquage initial. Son contenu est définitivement figé et ne peut en aucun cas être modifié après la fabrication de la carte. La mémoire réinscriptible EEPROM n'a pas besoin d'être alimentée pour conserver les données, occupe peu de surface de silicium, et bénéficie d'une vitesse de lecture confortable mais est relativement lente en écriture. La mémoire dite FLASH ayant des propriétés d'effacement similaires est en train de remplacer la mémoire EEPROM dans les systèmes actuels afin d'approcher le meilleur compromis entre encombrement, vitesse d'accès, consommation et endurance. D'autres types de mémoires plus exotiques (*e.g.* FRAM [9]) peuvent être présents sur une carte à puce mais nous n'en ferons pas l'inventaire exhaustif dans ce mémoire.

Entrées/Sorties La carte est accessible par ses contacts de cuivre apparents — elle est insérée dans un lecteur pour être lue et communique avec son environnement via une liaison série conforme au standard ISO7816 [Org99]. Le protocole ISO7816-3, utilisé par les cartes avec contact pour communiquer avec un lecteur, est asynchrone et half-duplex (*i.e.* les données peuvent être envoyées par le lecteur ou la carte mais pas dans les deux directions en même temps). Selon ce standard, la carte ne peut pas initier une communication avec le monde extérieur, elle agit plutôt comme un serveur traitant les requêtes venant de l'extérieur. Le protocole commande-réponse fait que la carte peut seulement envoyer de l'information vers le terminal quand il le demande. Au niveau applicatif, les commandes et les réponses envoyées sont transformées en paquets APDU [10], dont le format est normalisé par la norme ISO7186-4. Une carte à puce typique est capable de

4. MMU est l'acronyme de Memory Management Unit.
5. TLB est l'acronyme de Translation Lookaside Buffer.
6. DES est l'acronyme de Data Encryption Standard
7. GSM est l'acronyme de Global System for Mobile communications.
8. L'acronyme 3G est utilisé pour désigner la troisième génération de téléphones portables.
9. FRAM est l'acronyme de Ferro-électrique RAM
10. APDU est l'acronyme de Application Protocol Data Unit.

transmettre (recevoir) des données vers (venant de) un lecteur via une ligne série unidirectionnelle à un débit de 115,200 bit/s.

II.1.1.2 Cycle de production

Depuis la fin des années 70, le comportement des cartes à puce est défini irrémédiablement au moment de leur fabrication ; elles sont ainsi dédiées à une application spécifique (*e.g.* cartes bancaires, cartes SIM [11], cartes vitales, cartes de fidélité). Le cycle de production d'une carte à puce se décompose en plusieurs étapes et met en jeu différents acteurs de l'industrie (voir Figure II.1). Le *fondeur de silicium* sélectionne le meilleur profil matériel (micro-contrôleur, tailles des mémoires, voltage ...) correspondant à l'application à laquelle est destinée la carte, et pose la spécification du circuit intégré à utiliser. Le *fabricant de cartes*, aidé de cette spécification, fournit le système d'exploitation (*i.e.* système de gestion de fichiers et quelques algorithmes cryptographiques). Le code développé est donné au *fondeur de silicium* qui se charge d'incorporer ces données dans la mémoire ROM (processus de masquage) en parallèle avec le processus de fabrication du circuit intégré. Le *fournisseur de services* (*e.g.* banque, opérateur télécom) achète des gros volumes de cartes et les distribue à ses clients. Ces cartes sont d'abord personnalisées par le *fabricant de cartes* à la demande des fournisseurs avant d'être livrées aux *porteurs de cartes*. Une fois remise aux mains de son porteur, une carte a un usage figé jusqu'à son expiration, sa détérioration ou sa perte.

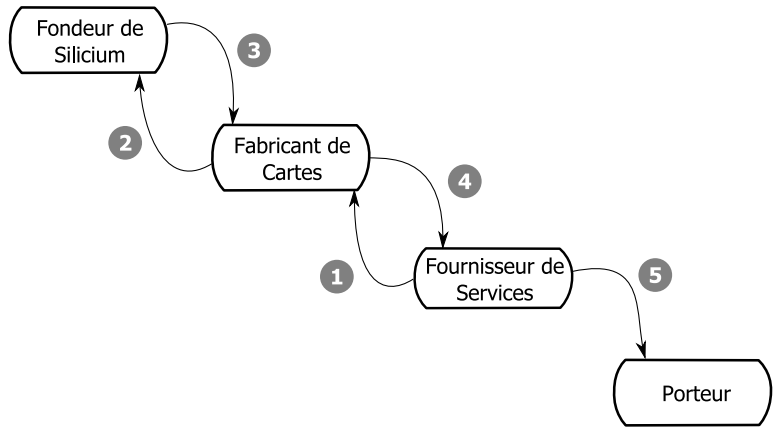


Figure II.1 — *Cycle de production d'une carte à puce : **1**. Spécification des logiciels, **2**. Image ROM du logiciel, **3**. Production du silicium, **4**. Packaging et Personalisation du logiciel en fonction des données de l'utilisateur, **5**. Distribution aux porteurs.*

II.1.1.3 Évolution vers les cartes ouvertes et multi-applicatives

Afin de réduire les coûts de développement, le besoin de munir les cartes à puce à microprocesseurs d'un système d'exploitation est apparu assez rapidement. Au départ, chaque fournisseur

11. SIM est l'acronyme de Subscriber Identity Module.

d'application était contraint de développer pour un système lui-même spécifique à un microprocesseur particulier, lui-même utilisable par un lecteur de carte précis. Cependant, la popularité grandissante des cartes à puce a mené au développement d'une population de microprocesseurs variée. Cette hétérogénéité a amplifié d'une manière dramatique les coûts de développement, car le portage d'une application d'un microprocesseur à un autre représentait un effort conséquent de la part des développeurs. Afin d'offrir plus de compatibilité et de modularité, la carte à puce embarque désormais un système d'exploitation. Dans sa première génération, le système d'exploitation se déclinait sous forme de bibliothèques stockés en ROM gérant notamment les communications de la carte avec le terminal. Plus tard, les systèmes d'exploitations sont devenus plus génériques en se développant au dessus d'une couche d'abstraction du microprocesseur (HAL [12]). Dès lors, ils offraient principalement l'infrastructure nécessaire aux applications de haut niveau pour répondre aux exigences de sécurité primordiales dans le contexte de la carte à puce.

L'utilisation d'une architecture en couches a permis, en outre, de distinguer les applications du système d'exploitation. Les applications ne sont plus écrites en mémoire morte mais en mémoire réinscriptible telle l'EEPROM. De cette façon, elles peuvent être modifiées, mises à jour, ajoutées ou supprimées à n'importe quel moment du cycle de vie de la carte. Le processus de développement a ainsi évolué vers une indépendance vis à vis de celui de la plateforme matérielle et du système d'exploitation. Cela a créé le besoin de doter les cartes déjà déployées de la capacité d'étendre et de modifier continuellement leurs fonctionnalités. Dorénavant, les applications ne sont plus fournies en tant qu'exécutables mais plutôt comme des services (*e.g.* code pré-compilé) prêts à être chargés sur des plateformes cibles où ils seront interprétés ou compilés. Notons qu'en 2006, trois milliards de puce ont été produites dont deux milliards sont des cartes SIM [Ins99] qui pour la plupart intègrent une machine virtuelle supportant les spécifications JAVA CARD [Jav07]. Par ailleurs, la création d'alliance impliquant des opérateurs provenant de secteurs économiques différents, notamment, les banques et les opérateurs de téléphonie, a créé le besoin d'apporter un support aux cartes dites multi-applicatives. C'est ainsi que, lors d'un achat, une application peut débiter un compte bancaire tandis qu'une autre crédite d'un certain nombre de points un compte fidélité sur la même carte. Aujourd'hui, les applications et les services sont en train d'occuper du terrain face à la vente de carte traditionnelles.

Dans la pratique, les cartes multi-applicatives telles qu'elles se présentent aujourd'hui ne sont pas encore ouvertes. En effet, pour l'instant, même s'il est désormais possible d'installer plusieurs applications provenant de sources différentes dans la même carte, le modèle d'utilisation est centré autour du fabricant qui a toute autorité lui permettant de contrôler le données ou les fonctionnalités présentes sur la carte. Pour changer cette tendance, la carte évolue vers un composant plus complet mais aussi plus complexe, le TPD.

II.1.2 Le dispositif personnel de confiance

Conçu par le projet européen INSPIRED, le dispositif personnel de confiance TPD se définit comme un petit objet appartenant à une seule personne et lui permettant d'établir des opérations dignes de confiance avec d'autres entités au sein d'une infrastructure informatique déjà déployée. Le rôle du TPD est plus exactement de fournir un environnement de stockage et d'exécution *sécurisé* à son porteur. Il englobe plusieurs *facteurs de formes* : carte à puce, carte SIM, carte de stockage et token cryptographique USB [13]. Grâce à son TPD, l'utilisateur doit pouvoir entreprendre des opérations telles que l'accès (à un pays, bâtiment, moyens de transport, services, ressources institutionnelles, vote, ...), le paiement (transactions financières avec des banques ou

12. HAL est l'acronyme de Hardware Abstraction Layer.
13. USB est l'acronyme de Universal Serial Bus.

des détaillants), les services commerciaux (carte de fidélité, opération de promotion), la communication (échange ou recherche d'information ou de messages). Ces cas d'utilisation définissent les différents *profils* des TPDs. Les innovations du TPD par rapport à la carte à puce traditionnelle résident principalement dans les caractéristiques énumérées ci-dessous.

II.1.2.1 Caractéristiques matérielles

La Figure II.2 met en exergue la complexité de l'architecture matérielle du TPD par rapport à celle d'une carte à puce traditionnelle.

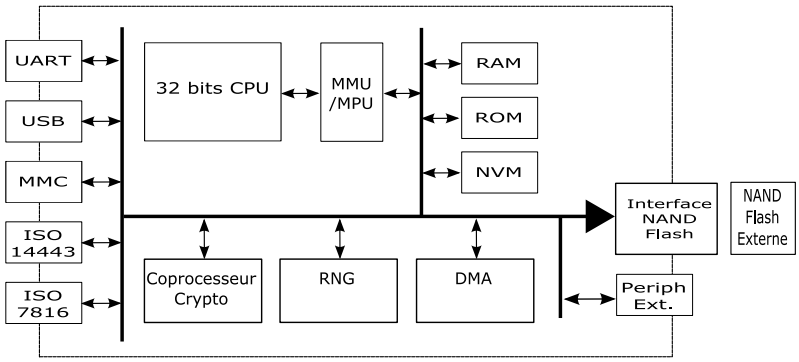


Figure II.2 — *Configuration matérielle type du* TPD.

En effet, l'architecture matérielle du TPD est centrée autour d'un unique coeur de processeur RISC [14] 32 bits et offre plus de capacité de stockage comme le montre le Tableau II.2.

Table II.2 — *Capacités de stockage des différentes mémoires du* TPD

Mémoire	Capacité
ROM	32 à 48 Ko
RAM	64 à 512 Ko
Non volatile embarquée	jusque 2 Mo
Non volatile externe	8 Mo à 4 Go

Le choix des mémoires embarquées (ROM, RAM et non volatiles) sur le TPD et leurs capacités est différent selon les facteurs de forme et les profils d'utilisation sélectionnés. En effet, pour certains profils (*e.g.* TPDs utilisés pour stocker le dossier médical du porteur), le support d'une

14. RISC est l'acronyme de Reduced Instruction Set Computer.

mémoire de masse (de 1 Mo jusque quelques Go) est requis. Pour ce type de profil, l'abandon de la mémoire EEPROM au profit de la mémoire FLASH à haute densité est aussi fortement encouragé.

Par ailleurs, le TPD comporte, en plus de l'interface standard des cartes à puce décrite par le standard ISO7816, de nouvelles interfaces physiques de communication telles que USB, MMC [15] ou NFC [16]. Les débits de communication à travers ces nouvelles interfaces sont nettement plus élevés (voir Tableau II.3).

Table II.3 — *Débits des nouvelles interfaces de communication du* TPD

Interface	Débit
USB	1.5 à 480 Mbit/s
MMC	3 à 20 Mbit/s
NFC	106 à 6670 Kbit/s
ISO14443	106 à 847 Kbit/s

Pour renforcer la confiance de l'utilisateur, des composants matériels dédiés à la sécurité sont présents au sein du TPD. Ainsi, on pourra trouver un coprocesseur dédié à la génération de nombres aléatoires (RNG [17]) ou encore un processeur cryptographique symétrique tel que triple DES.

Pour le support d'un système d'exploitation multitâche, le processeur du TPD se voit adjoindre une unité de gestion de mémoire (MMU) permettant à plusieurs applications de partager un espace mémoire physique limité. La MMU s'occupe non seulement de la gestion de la mémoire virtuelle paginée en effectuant le partitionnement de la mémoire physique en blocs qui seront alloués aux différentes applications mais également de la translation d'adresses logiques en adresses physiques. Ainsi, chaque application exécutée par le système d'exploitation se voit attribuer une zone mémoire protégée, dans laquelle aucune autre application ne peut écrire. De même, pour la gestion des droits d'accès, la présence d'une unité de protection de mémoire (MPU) est souhaitée.

Le transfert de données entre les interfaces de communication à haut débit (*e.g.* MMC) et les périphériques internes (*e.g.* coprocesseur cryptographique symétrique) sans utiliser les ressources du CPU requiert l'utilisation d'une interface d'accès direct à la mémoire (DMA [18]). Ainsi, le processeur est libre d'entreprendre d'autres actions en parallèle. Le système d'accès direct à la mémoire est notamment utile quand le TPD reçoit un flot de données qu'il doit déchiffrer à la volée dans une application de gestion de droits numériques.

Le TPD peut aussi disposer de périphériques. Ceux-ci peuvent être directement disponibles sur le TPD ou être accessibles à travers une interface. A titre d'exemple, par le biais d'un écran, le TPD pourrait afficher le solde d'un compte bancaire ou le compte de points sur une carte de fidélité sans avoir à communiquer avec un lecteur. Un capteur biométrique permettant d'identifier le porteur pourrait également figurer parmi les périphériques externes.

15. MMC est l'acronyme de Multi Media Card
16. NFC est l'acronyme de Near Field Communication
17. RNG est l'acronyme de Random Number Generator.
18. DMA est l'acronyme de Direct Memory Access.

II.1.2.2 Caractéristiques logicielles

Les innovations réalisées au niveau du matériel sont au service d'une architecture logicielle fortement inspirée de l'architecture structurée en couches des cartes à puces actuelles. Les systèmes d'exploitation dont les fonctionnalités sont encapsulées par une machine d'exécution virtuelle et permettant d'écrire ou de charger des applications dans la mémoire non volatile sont déjà devenus un standard de facto avec l'avènement des technologies telles que JAVA CARD ou encore MULTOS [Lim06].

La Figure II.3 expose les différentes couches qui composent l'architecture logicielle du TPD.

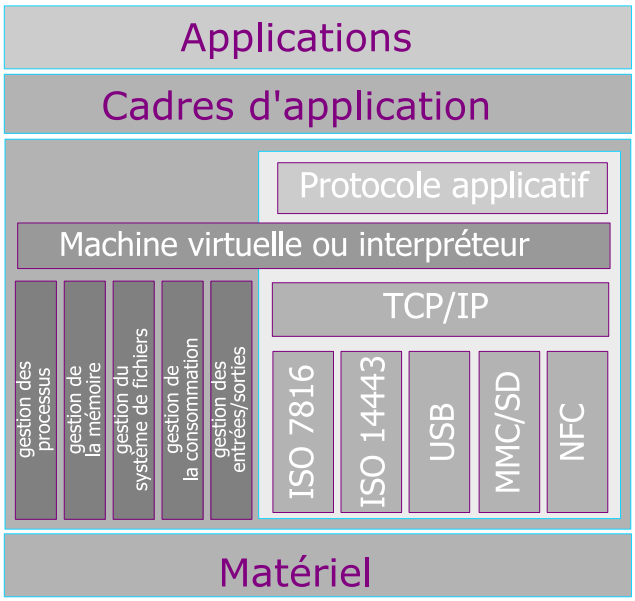


Figure II.3 — *Architecture logicielle en couches du* TPD.

Un ensemble de gestionnaires (*e.g.* gestionnaire de mémoire, de processus, d'entrées/sorties ...) permettent de gérer les interactions de l'environnement d'exécution avec les ressources matérielles du TPD. Le gestionnaire de processus doit notamment prendre en compte le fonctionnement multitâche (la création de processus concurrents, la communication entre processus et la synchronisation).

L'environnement d'exécution se compose d'une machine virtuelle et d'une bibliothèque de base. La machine virtuelle d'exécution peut être implémentée sous la forme d'une machine virtuelle JAVA/JAVA CARD ou d'un interpréteur CLI. La bibliothèque contient un ensemble de types élémentaires ainsi que quelques fonctionnalités de base utilisables par les cadres applicatifs.

Pour des raisons de sécurité, le développeur ne peut pas accéder directement au système d'exploitation. Les fonctionnalités de celui-ci sont uniquement disponibles à travers une interface de programmation offerte par la machine virtuelle.

Enfin, la couche de communication est d'une importance capitale pour répondre aux exigences de l'intelligence ambiante ou ubiquitaire où le réseau n'est plus qu'une couche neutre de transport. En effet, le TPD doit pouvoir s'intégrer dans une infrastructure de communication comme un composant communicant à part entière. En intégrant sa propre implémentation de la pile de communication TCP/IP [19], le TPD peut être considéré comme un noeud dans le réseau Internet. Cette pile est implémentée au dessus d'une couche de liaison gérant les différentes interfaces de communication. De plus, il peut jouer le rôle d'un serveur accessible à travers les protocoles standards du Web. Plus précisément, au dessus de la pile de communication, *les protocoles applicatifs* HTTP [20] ou HTTPS [21] peuvent être utilisés pour échanger de l'information. Le TPD pourra ainsi être un serveur Web personnel, sécurisé et portable avec une interface de communication sans contact et le support de l'authentification biométrique.

II.2 Déploiement sécurisé de logiciels embarqués

Comme dans toute architecture distribuée, un code mobile est associé à au moins deux entités séparées dans le temps et l'espace :

- son *producteur* : par abus de langage, nous désignerons par le terme *producteur de code*, non seulement le développeur qui implémente le logiciel, mais aussi le support permettant d'héberger les unités de logiciel prêtes à être déployées et la chaîne de compilation utilisée pour les produire.
- son *consommateur* : le consommateur est l'hôte qui exécute le logiciel. Il doit offrir l'espace de stockage nécessaire ainsi qu'un environnement d'exécution adéquat pour le logiciel qu'il va charger. Il est, de ce fait, considéré comme une plateforme de *déploiement*.

Le processus de déploiement permet de rendre le logiciel disponible et prêt à être exécuté sur les machines cibles. Une fois déployé sur le consommateur, un logiciel embarqué remplit une fonction bien déterminée. Dans le cas général, le logiciel embarqué est complet et son comportement est entièrement défini par lui-même. Toutefois, il n'est pas nécessairement monolithique et peut être composé de plusieurs modules qui interagissent entre eux.

Si le code est fourni sous sa forme exécutable, le logiciel embarqué peut être exécuté directement sur la plateforme hôte. C'est le mode de déploiement le plus simple qui correspond au mode de fonctionnement des premiers systèmes informatiques. Cependant, pour un souci de portabilité et de sécurité, les producteurs mettent leurs applications à la disposition des éventuels consommateurs sous d'autres formes qui requièrent un effort de déploiement plus important.

II.2.1 Liste des exigences

Le processus déploiement se compose alors de plusieurs activités interdépendantes [CFH+98] permettant de rendre le logiciel exécutable sur les machines cibles. En effet, une fois chargé, le logiciel subit une série d'opérations permettant de le lier dynamiquement dans l'espace d'adressage du consommateur et de lui permettre d'utiliser les ressources de ce dernier. Nous formulons ici une liste d'exigences pour un déploiement sécurisé des logiciels embarqués.

19. TCP/IP est l'acronyme de Transport Control Protocol / Internet Protocol.
20. HTTP est l'acronyme de Hyper Text Transfer Protocol
21. HTTPS est l'acronyme de Secure Hyper Text Transfer Protocol

II.2.1.1 Portabilité

Nous retenons notamment la *portabilité* qui est l'aptitude d'un logiciel à être transféré dans des environnements matériels et logiciels différents.

De nos jours, plus d'un milliard de microprocesseurs embarqués sont produits chaque année afin d'être intégrés dans des petits objets tels que les cartes à puces. Ils se distinguent par le savoir faire de leurs fabricants (Atmel, Philips, ...) et leurs architectures (CISC ou RISC, 8 bits à 32 bits). Cette hétérogénéité matérielle doit absolument rester transparente faute de quoi la distribution de nouveaux logiciels ne pourra se faire que sur des sous-ensembles de cibles.

Afin d'assurer la portabilité des logiciels, le producteur ne peut émettre au moment de la compilation aucune supposition préalable sur le consommateur (architecture matérielle, environnement d'exécution, stratégie de compilation ou d'interprétation, ...). Le code qu'il fournit doit être complètement indépendant de la plate-forme sur laquelle il va s'exécuter.

L'utilisation de langages intermédiaires (*e.g.* bytecode) suffisamment primaires pour être exécutés directement (en mode interprété) ou compilés en code natif si besoin est devenue une nécessité. Les solutions à base de code interprété, comme JAVA ou *.Net*, proposent en l'occurrence une solution à ces problèmes.

L'environnement d'exécution doit, pour sa part, fournir une couche d'abstraction, point d'entrée vers les fonctions du noyau du système d'exploitation embarqué, sous la forme de bibliothèques. Même si le code contenu dans ces bibliothèques dépend de la plateforme matérielle sous-jacente, il n'en demeure pas moins que toutes les implémentations ont la même interface. Ainsi, le logiciel embarqué n'a pas besoin de savoir la configuration matérielle de la machine cible.

II.2.1.2 Extensibilité

L'environnement d'exécution doit être en mesure de permettre le chargement tardif de briques logicielles dont la fonction serait d'étendre ou de modifier les fonctionnalités existantes. Nous présentons ici les différents mécanismes constituant le substrat nécessaire à l'extensibilité.

Paradigme objet Le paradigme objet qui a connu indéniablement un essor considérable dans le paysage du génie logiciel constitue la première brique. En effet, par le biais de ces nombreux concepts que nous décrivons ci-après, il a amené une nette amélioration de la décomposition structurelle des systèmes et de la réutilisation des modules logiciels.

Abstraction/Encapsulation L'encapsulation consiste à isoler l'interface d'un objet de son fonctionnement. Une classe encapsule ses données dans un ensemble d'attributs. Elle a également un ensemble de comportements qui sont représentés par un ensemble de méthodes qui opèrent sur ces attributs. Les attributs et les méthodes définissent un *type* à partir duquel les objets peuvent être instanciés.

Héritage Il existe plusieurs sortes de relations entre les classes (*e.g. est composé de*, *possède*, *est-un*, ...). *L'héritage*, qui traduit le principe de *généralisation/spécialisation*, se base sur l'idée qu'un objet spécialisé bénéficie ou hérite des caractéristiques de l'objet le plus général auquel il rajoute ses éléments propres. La relation d'héritage peut être perçue comme un simple partage d'attributs et de méthodes (*i.e.* réutilisation). En effet, une classe S hérite d'une classe T implique que les instances de S possèdent toutes les propriétés des instances de T (attributs et méthodes), et peuvent aussi l'étendre en redéfinissant certaines de ses méthodes ou en en définissant des nouvelles. Il est possible de redéfinir une méthode d'une classe mère par ses classes dérivées en utilisant la même signature et en donnant une implémentation différente [22].

Polymorphisme La relation de sous-typage implique aussi que partout où un objet d'un certain type est attendu, on autorise la substitution par un objet d'un sous-type. Ainsi, si S hérite de T, les objets de la classe S peuvent être utilisés partout où l'on peut utiliser des objets de la classe T. De cette dernière propriété découle la notion de *polymorphisme* qui est la faculté présentée par certains attributs ou méthodes dont le comportement est différent selon le type de l'objet auquel ils s'appliquent. À cause du polymorphisme, la résolution des appels de méthodes, autrement dit le choix dans l'arbre d'héritage d'une méthode à exécuter parmi toutes celles qui ont la même signature, s'effectue sur la base des informations de typage. Le mécanisme associé au polymorphisme et permettant de résoudre dynamiquement le traitement d'une donnée en fonction de son type est un procédé dynamique dans le sens où la méthode appelée (qui va être réellement exécutée) est déterminée le plus tard possible c'est à dire à l'exécution (par le type de l'objet référencé) et non à la compilation (par le type déclaré de la variable qui le référence).

Chargement et liaison dynamiques D'une part, nous considérons l'utilisation des concepts du paradigme orienté objet comme un pré-requis pour la réalisation de l'extensibilité. Ce premier point relève plus des préoccupations de génie logiciel. D'autre part, l'environnement d'exécution doit fournir un support au chargement ultérieur de code. En effet, le code doit être d'abord chargé dans une forme intermédiaire neutre pour assurer la portabilité. Ensuite, un mécanisme de liaison au chargement permet de raccorder les différents constituants du logiciel à l'environnement d'exécution du consommateur.

Traditionnellement, l'édition de liens est un processus *statique* prenant comme entrées les modules objets produits par les compilateurs (*e.g.* les fichiers *.o* d'UNIX) et un ensemble de bibliothèques systèmes (*e.g.* les fichiers *.a* d'UNIX) et permettant de remplacer les références symboliques (appels de fonctions) contenues dans les modules objets par des leurs adresses effectives (physiques) afin de produire des exécutables. Au final, le fichier exécutable produit contient une copie des fonctions appelées (ou des données) appartenant aux bibliothèques systèmes à chaque fois que le programme y fait référence.

La *liaison dynamique*, quant à elle, retarde le remplacement des références symboliques par leurs adresses effectives jusqu'au moment de chargement du programme. Le chargeur permet alors, au cours de l'exécution, d'amener en mémoire les sous-programmes utilisés. La seule restriction imposée stipule que le code d'un sous-programme doit être disponible en mémoire lorsque celui-ci est appelé. Afin de respecter cette restriction, le chargement peut survenir de deux manières faisant ainsi la distinction entre liaison tardive et précoce. La *liaison tardive* permet de résoudre lors du premier appel d'une méthode l'adresse effective de branchement plutôt que d'effectuer cette résolution lors du chargement de l'unité de code. Cette technique permet d'accroître les performances globale du système. En effet, seules les méthodes réellement appelées pendant

22. Le terme anglo-saxon correspondant est **overriding**.

l'exécution sont liées dans l'environnement d'exécution — plutôt que toutes les méthodes potentiellement appelées pendant le chargement. Cependant, cette technique n'est pas forcement aussi pertinente dans le contexte de l'embarqué que dans celui des environnements d'exécution généralistes[23]. À l'opposé, on parle de *liaison précoce* quand toutes les références symboliques sont résolues dès que le chargement est effectué. Toutes les classes référencées sont ainsi chargées y compris celles qui ne sont pas utilisées lors de l'exécution du code.

II.2.1.3 Sécurité

La nouvelle tendance des plateformes ouvertes et extensibles permet indéniablement de raccourcir les temps de mise sur le marché des TPDs. De cette façon, des applications peuvent être déployées plus rapidement afin de répondre à l'évolution des besoins des utilisateurs. Cependant, cette flexibilité a introduit de nouveaux problèmes de sécurité. Les aspects qui nous intéressent ici sont plus particulièrement (i) la garantie de sûreté de fonctionnement délivrée par l'environnement d'exécution aux logiciels embarqués et (ii) l'innocuité des applications, les unes par rapport aux autres.

En premier lieu, la *sûreté de fonctionnement*[24], est l'établissement d'une garantie, apportée à une application par l'environnement qui l'exécutera, lui assurant qu'elle sera en mesure de réaliser la tâche qui lui incombe le cas échéant. Aussi, la sûreté de fonctionnement repose sur l'établissement de propriétés qui permettent de certifier que le logiciel considéré sera en mesure de s'exécuter au regard des contraintes qu'il affiche. A titre d'exemple, l'environnement d'exécution pourrait donner la garantie au logiciel embarqué qu'une fois déployé il disposera des ressources mémoires et des capacités de calcul nécessaires pour rendre le service qu'il doit assurer.

En second lieu, la sécurité en terme d'*innocuité* se définit, pour sa part, dans un contexte où plusieurs logiciels sont déployés, et où certains sont potentiellement malveillants vis-à-vis des autres. Dans ce contexte de méfiance, les malveillances peuvent prendre différentes formes. On distingue alors trois sortes de garanties qu'un logiciel est susceptible de requérir, en terme de sécurité-innocuité. Il s'agit, d'une part, d'apporter aux applications la garantie de l'absence de fuite des données et traitements privés via un canal de communication non autorisé. On parle alors de *confidentialité*. Il s'agit aussi de prémunir les applications de toutes interférences produites par une application tierce visant à fausser le fonctionnement ou les résultats en cours d'établissement. On parle alors de garantie d'*intégrité*. Enfin, il s'agit d'assurer aux applications que leurs usagers seront toujours en mesure de requérir et d'obtenir leurs services. On parle dans ce cas de *disponibilité*.

La frontière entre sécurité et sûreté de fonctionnement n'est pas toujours clairement définie dans le contexte de code mobile. La sûreté est concernée par le comportement des systèmes dans la présence de dysfonctionnements. Par ailleurs, les failles en terme de sûreté peuvent être exploitées pour menacer la sécurité d'un système. Ainsi, la sûreté peut être considérée comme un pré-requis nécessaire pour apporter les garanties en sécurité aux systèmes. En d'autres mots, assurer la sûreté de fonctionnement d'un environnement d'exécution repose sur l'établissement de l'innocuité des logiciels qui vont s'y exécuter. Nous pourrons dès lors définir globalement la sécurité comme suit.

Définition II.2.1 (Sécurité) *La sécurité d'un logiciel embarqué est la garantie apportée par l'environnement d'exécution que la confidentialité de ses données (et ses traitements) ainsi que leurs intégrités et leurs disponibilités ne pourront être menacées par des applications tierces.*

23. La liaison tardive est utilisée dans la majorité des implémentations de machines virtuelles JAVA [LB98].
24. Le terme sûreté de fonctionnement peut aussi signifier la fiabilité.

Afin d'offrir un environnement d'exécution sécurisé pour le déploiement de logiciels embarqués, un consommateur de code doit établir une *base de confiance*, qu'on définit de la manière suivante :

Définition II.2.2 (Base de confiance [25]) *Une base de confiance sera constituée de tous les composants (*e.g. *matériel et système d'exploitation) qui assurent le respect des politiques de sécurité et qui y sont elles mêmes assujetties. Elle crée un cadre sûr permettant au consommateur d'exécuter du code mobile localement sur sa plateforme. Le code provenant de l'extérieur de cette base de confiance doit être vérifié avant que le consommateur ne l'exécute localement.*

Vérifier la sécurité d'un logiciel peut, dans certains cas, reposer sur la confiance accordée par le consommateur à une autorité tierce d'arbitrage. Celle-ci authentifie le producteur grâce à sa signature numérique et vérifie l'innocuité de son code afin de délivrer un certificat. Ce certificat accompagne le logiciel embarqué, et son authentification lors du chargement par le consommateur permet de valider le programme. Ce schéma élargit la base de confiance et augmente la probabilité de brèches de sécurité menaçant l'environnement d'exécution du consommateur. Pour ces raisons, il est préférable que la base de confiance n'inclut que des composants intrinsèques de l'environnement d'exécution du consommateur sans requérir une intervention extérieure.

Deux stratégies distinctes s'opposent pour l'établissement de l'innocuité au sein des environnements d'exécution embarqués. La première, défensive, consiste à stopper le logiciel lorsqu'il transgresse les règles de sécurité établies. Elle se déroule donc *pendant l'exécution*. L'autre, préventive, permet de détecter une possible défaillance *au chargement* au moment où le logiciel est déployé. Nous allons, dans ce qui suit, présenter les deux approches en étayant nos propos avec des exemples.

Approches défensives Elles consistent à *surveiller* l'exécution du programme afin d'empêcher l'occurrence d'une action potentiellement dangereuse pour le fonctionnement du système (voir Figure II.4). Les techniques défensives reposent essentiellement sur la définition de politiques de sécurité comme moyen de spécifier les règles qui ne doivent pas être transgressées par l'exécution d'un logiciel. Grâce à la mise en place de mécanismes de surveillance, il est possible de rejeter les exécutions qui violent la politique de sécurité préalablement spécifiée. Cependant, ces moyens défensifs ne doivent en aucun cas perturber les exécutions qui n'induisent pas de mauvais fonctionnement du système. L'implémentation de ces mécanismes défensifs peut se décliner sous différents formes.

Des solutions matérielles permettant d'isoler les applications les unes des autres existent. En l'occurrence, une unité de gestion de la mémoire MMU peut surveiller les zones de mémoire accédées par le microprocesseur et interdire l'accès (lecture/écriture) de l'application en dehors du segment qui lui a été attribué en déclenchant une interruption matérielle en cas d'accès illégal. Ce composant matériel protège la confidentialité et d'intégrité des données/traitements.

De même, un processeur ayant un mode protégé sur lequel le système d'exploitation peut implémenter du multitâche préemptif (*e.g.* UNIX) peut empêcher une application de dépasser le budget CPU qui lui est alloué. Dans ce cas, l'application est exécutée dans un mode utlisateur et, quand son budget temps est complètement utilisé, le système d'exploitation — en mode protégé — prend la main pour mettre fin à la tâche. Cette solution repose sur un composant (matériel ou logiciel), appelé *chien de garde* [26] [SA00a], qui est typiquement un compteur décrémenté à chaque tic d'horloge. Une fois, le décompte du chien de garde achevé, le système peut immédiatement

26. Le terme anglo-saxon correspondant est *watchdog timer*.

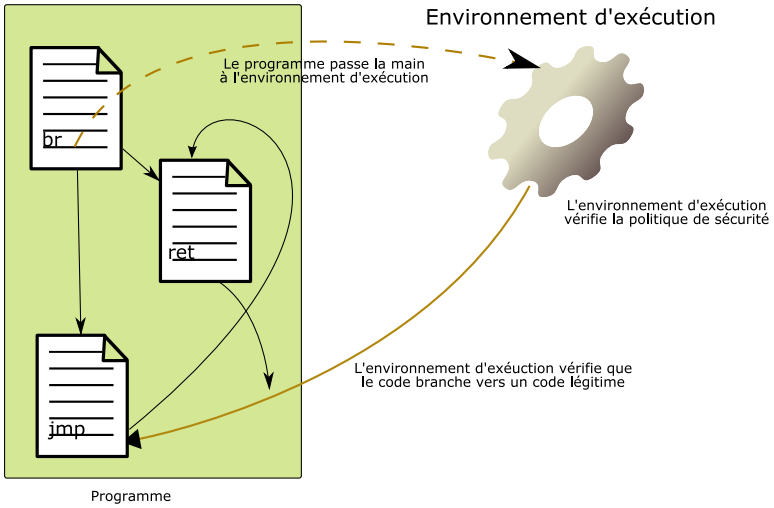


Figure II.4 — *Approches défensives*

réclamer les ressources en cours d'utilisation par le processus et ainsi l'interrompre. Sans la distinction entre le mode protégé et le mode réel, l'utilisation de chien de garde est évidemment inopérante, puisque le logiciel surveillé aurait tout loisir de désarmer le mécanisme de surveillance.

Les solutions logicielles s'avèrent plus flexibles et moins coûteuses. À titre d'exemple, l'isolation logicielle de fautes (SFI [27]) [WLAG93] permet aux processus de partager le même espace d'adressage. Toutefois, elles assurent la vérification les accès au code et aux données, ainsi que la validité des adresses mémoires par un mécanisme de surveillance lors de l'exécution. Pour ce faire, elles procèdent essentiellement par réécriture du code binaire des programmes afin d'y insérer des points de vérification lors de l'exécution du code. En outre, d'autres méthodes appelées *bac à sable* consistent en l'occurrence à exécuter le code mobile dans un environnement restreint où il ne dispose que d'un ensemble de ressources spécifiées par la politique de sécurité de l'hôte. Ainsi, le code est amené à s'exécuter dans un espace confiné (*i.e.* une couche logicielle d'isolation) délimité par des mécanismes de contrôle d'accès. Le mécanisme d'inspection de pile des plateformes JAVA et .NET en est un exemple [28].

L'inconvénient majeur des approches défensives réside dans le fait que la transgression de la politique de sécurité n'est détectée que très tardivement. Dans certains cas, il peut être nécessaire de détecter les erreurs a priori afin de ne pas gêner l'exécution d'applications critiques. En outre, l'utilisateur qui jusque là jouissait du service offert par l'application, s'en trouve soudain privé. Cette stratégie sécuritaire ne convient guère aux systèmes embarqués qui doivent assurer non seulement la sécurité des services offerts sinon leurs fiabilités. Par exemple, il ne semble guère raisonnable dans le contexte d'une carte à puce de priver l'usager de sa communication téléphonique, parce que une autre application vient d'adopter un comportement pathogène.

Approches préventives À l'opposé, les approches préventives se situent en amont afin de détecter *préalablement à l'exécution* d'éventuels dysfonctionnements (*e.g.* l'écriture dans des ré-

27. SFI est l'acronyme de Software Fault Isolation.

28. Le principe d'inspection de piles implémenté par les machines virtuelles JAVA repose sur la surveillance des piles d'appel durant l'exécution d'une application afin d'éviter que les méthodes appartenant à des applications provenant de sources inconnues (dans lesquelles on n'a pas confiance) est en train d'effectuer des actions néfastes pour le système.

gions critiques de la mémoire). Ces approches peuvent dans ce cas garantir qu'un programme vérifie certaines propriétés à la compilation ou au moment du chargement. Elles peuvent détecter par exemple si un programme respecte la confidentialité des données, ou si sa consommation en mémoire et temps n'excède pas les ressources qui lui sont attribuées. La vérification de bytecode effectuée au moment du chargement par les machines virtuelles JAVA illustre ce mécanisme [Ler03]. Les approches préventives incluent d'une part les techniques d'analyse statique incluant l'analyse de flot de données, l'interprétation abstraite ou les systèmes de types et d'autre part les approches basées sur les preuves.

Preuve de bon fonctionnement Une des démarches utilisées consiste à obtenir un logiciel correct par construction en produisant un code dont il est prouvé qu'il est conforme à sa spécification formelle. En effet, elle base le développement de systèmes (de logiciels) sur des concepts logiques (*e.g.* logique de premier ordre) et mathématiques rigoureux et bien identifiés.

A titre d'exemple, le comportement d'un programme peut être spécifié par une sémantique axiomatique [29] plus communément appelée *logique de Hoare* [Hoa69]. La logique de Hoare s'applique à la vérification de la correction partielle des programmes séquentiels en fournissant un ensemble de règles d'inférence qui permettent de raisonner sur les triplets de Hoare ($\{P\}C\{Q\}$). Un programme C est, en effet, considéré comme une relation de correction partielle qui lie deux prédicats : une pré-condition P et une post-condition Q. La pré-condition caractérise les états possibles avant l'exécution du programme, tandis que la post-condition regroupe les états possibles après son exécution. Un programme C est *totalement correct* vis à vis de sa spécification P et Q si à partir d'un état initial vérifiant une pré-condition P, il se termine et son état final satisfait la post-condition Q. Ensuite, des formules exprimant les propriétés de correction sont dérivés dans la logique choisie. Dans le cadre de la logique de Hoare, cela consiste à extraire — automatiquement ou avec l'intervention du développeur — des *pre-* et *post*-conditions à partir du programme. Enfin, des logiciels d'aide à la preuve permettent d'automatiser les démonstrations mathématiques dans le but de garantir formellement les propriétés de sécurité et de sûreté indiquées.

D'une part, ces approches ont le mérite d'être plus expressives et plus précises puisqu'elles se basent sur des notions formelles. De plus, la preuve du bon fonctionnement du logiciel contribue à la confiance que l'on peut avoir dans le résultat du développement en raison de son caractère exhaustif. D'autre part, l'automatisation des preuves n'est pas toujours possible demandant, par conséquent, l'intervention de développeurs avec une base solide en mathématiques. Même si des solutions permettent de masquer la complexité des formalismes mathématiques [LPC+08], il n'en demeure pas moins difficile d'intégrer ces techniques dans l'environnement d'un développeur avec un profil classique.

Analyse statique L'analyse statique consiste à étudier un programme afin de prédire son comportement à l'exécution (sans pour autant l'exécuter). Elle permet ainsi de vérifier automatiquement que le programme satisfait une certaine propriété. De ce fait, il devient possible de détecter d'éventuels dysfonctionnements des logiciels très tôt dans leurs cycles de vie, c'est à dire à la compilation ou même pendant le développement. Cependant, cette approche est souvent décriée à cause de l'indécidabilité de certaines propriétés que l'on voudrait vérifier. Il convient dans ce cas d'avoir recours aux techniques d'interprétation abstraite et de typage pour considérer une approximation pertinente de la sémantique des programmes permettant d'en décrire formellement tous les comportements possibles.

29. La sémantique doit être rigoureusement définie pour pouvoir être utilisée dans le cadre de vérification des propriétés de sécurité.

Interprétation abstraite L'interprétation abstraite [CC77] permet de fournir un modèle mathématique représentant les structures créées par le programme ainsi que leur évolution au cours de toutes les exécutions possibles du programme. Pour faire face au problème d'indécidabilité, l'interprétation abstraite se restreint à des sous-ensembles pertinents des informations observées sur les comportements des programmes.

Systèmes de types À l'origine, les systèmes de type ont été mis au point afin de garantir qu'un programme utilise correctement les structures de données à l'exécution. En l'occurrence, le langage JAVA s'est distingué — par rapport aux langages C ou C++ — par son système de type qui interdit notamment de forger un pointeur ou d'effectuer des opérations arithmétiques entières sur des nombres flottants. L'analyse de type analyse statiquement le comportement dynamique d'un programme en assignant des types aux valeurs. La sémantique statique du programme est modélisée par un ensemble de règles de déduction. Ces dernières définissent le type d'une expression et les conditions, en fonction du type de ses arguments, selon lesquelles elle s'évaluera correctement. Un algorithme d'inférence est ensuite utilisé pour garantir que le programme est correctement typé.

II.2.2 Travaux existants

Assurer la sécurité du logiciel au moment du déploiement a de nombreux avantages.

L'analyse statique opère sur la forme pré-compilée (*e.g.* bytecode) du logiciel. Cela permet de réduire la taille de la base de confiance. En effet, si l'analyse se plaçait au niveau du code source, alors toute la chaîne de compilation permettant de produire le logiciel doit faire partie de la base de confiance. En revanche, seul le vérifieur au niveau du consommateur doit être validé dans le deuxième cas.

II.2.2.1 Codes porteurs de preuve

C'est à Georges Necula [Nec97] que revient le mérite d'introduire la notion de *code porteur de preuve*[30]. Cette technique permet de charger du code mobile tout en assurant le respect des politiques de sécurité définies sur la machine cible. L'idée de base consiste à faire en sorte que les producteurs de code fournissent aux consommateurs non seulement l'application mais aussi un certificat attestant qu'elle respecte leurs exigences en terme de sécurité. De l'autre coté, les consommateurs doivent être en mesure de vérifier la validité du code reçu par rapport au certificat afin d'accepter ou de refuser son exécution sur leurs plateformes. Le processus de vérification doit, en effet, être à la fois simple, automatique et fiable.

Initialement, le certificat est une preuve de certaines conditions de vérification. Cette preuve est basée sur la logique du premier ordre de Hoare [Hoa69] et permet de décrire les programmes sous forme d'axiomes. Une architecture distribuée basée sur le modèle PCC, telle que celle illustrée par la Figure II.5, inclut les composants suivants :

(i) un langage de spécification permettant d'exprimer la politique de sécurité à respecter (*e.g.* logique du premier ordre) ;

(ii) une sémantique du langage dans lequel est exprimé le code mobile ;

30. Le terme anglo-saxon est Proof Carrying Code (PCC)

(iii) un prouveur de théorème construisant une preuve transmise au consommateur avec le code mobile ;

(iv) un vérifieur de preuve qui, du coté consommateur, établit dans un premier temps la correspondance entre la preuve et le code mobile et dans un deuxième temps garantit le respect des politiques de sécurité prédéfinies.

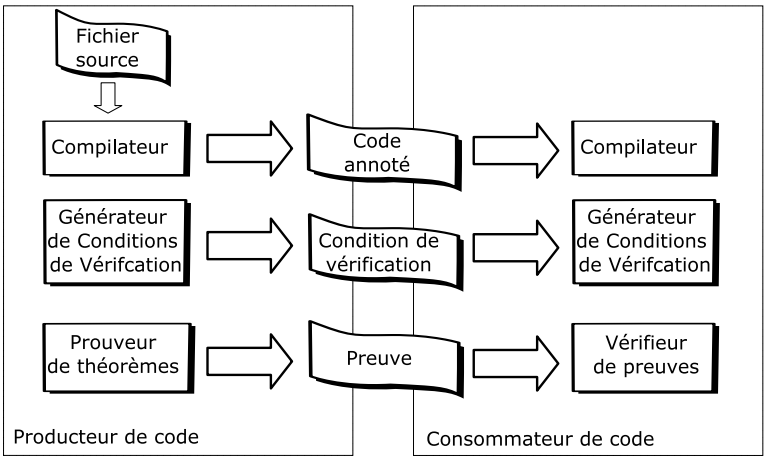


Figure II.5 — *Les consommateurs, ou une autorité qui les représente, rendent publique une politique de sécurité sous la forme d'un générateur de condition de vérification. Ce composant permet de calculer une condition de vérification qui doit être vraie pour que le consommateur accepte d'exécuter le code mobile. Afin de distribuer son programme, le producteur calcule la condition de vérification à partir du code, établit la preuve et la joint au code. Quand il reçoit le code, le consommateur recalcule la condition de vérification et vérifie que la preuve attaché établit la condition de vérification. Le producteur ajoute des annotations (*e.g. *invariant de boucles) dans le code source, soit manuellement, soit par le biais d'un compilateur certifiant (*e.g. *Touchstone [NL98]). Le code annoté est envoyé au consommateur. Le producteur et le consommateur utilisent un générateur de conditions de vérification qui se base sur les annotations fournies pour traduire le programme en formule logique.*

Il est important de souligner que les approches basées sur le modèle PCC permettent de vérifier que le code mobile transmis par le producteur respecte la politique de sécurité du consommateur. Cependant, il n'y a pas de mécanisme établissant que le code reçu est le même que celui que le producteur a envoyé. Par ailleurs, l'approche suppose une entente préalable entre producteurs et consommateurs sur la politique de sécurité que les programmes doivent respecter afin d'être acceptés sur la plateforme cible. Ainsi, la génération de conditions de vérification est-elle basée sur un protocole défini entre ces deux parties.

II.2.2.2 Langage assembleur typé

À la suite des travaux séminaux de Necula, la littérature scientifique autour de ce modèle s'est enrichie d'une multitude d'approches plus ou moins différentes. Notamment, une approche, qui s'apparente à celle des codes porteurs de preuves, ne manipule pas des preuves mathématiques mais des annotations de types. Elle enrichit les langages assembleurs traditionnellement non typés avec des annotations de type, des primitives de gestion de la mémoire, et des règles de typage. Dans [MWCG99], TAL [31] est un langage assembleur fortement typé basé sur une architecture RISC. Les annotations de type sont obtenues par extraction à partir du code source exprimé en un langage de haut niveau fortement typé, SYSTEMF. Les informations de typage sont préservées jusque dans le code assembleur. Il devient alors possible d'inférer le type des variables et leur

31. TAL est l'acronyme de Typed Assembly Language.

utilisation correcte au niveau du consommateur de code. L'inconvénient de cette solution est que les outils doivent générer un certificat spécifique à chaque code assembleur. Par contre, la taille de cette preuve est moins importante que l'approche PCC initiale.

II.2.2.3 Vérification légère de bytecode

[32] Ces travaux [Ros03] s'inspirent des codes porteurs de preuve en proposant d'effectuer la phase de vérification de bytecode JAVA sur le consommateur, de manière efficace au regard des ressources limitées dont il dispose. En effet, ce vérifieur distribué a été conçu, en premier lieu, pour les cibles contraintes en ressources, notamment l'architecture KVM [33]/CLDC [34] de Sun [Mic00]. Cependant, il est désormais intégré dans tous les JDKs [35] à partir de la version *6* moyennant quelques améliorations [Gro06a].

Contrairement au vérifieur de bytecode classique qui entreprend une analyse itérative de flot de données jusqu'à l'obtention un point fixe, la vérification légère de bytecode se passe en deux temps comme illustrée dans la Figure II.6.

(i) La première phase se déroule sur le producteur de code ; il y effectue une analyse de flot de données standard utilisant l'algorithme de Kildall [Kil73] en plusieurs itérations afin de déterminer un point fixe (*i.e.* les types des opérandes contenues dans la pile et des variables locales pour chaque point de saut). Les informations calculées sont reportées dans les fichiers *class* sous la forme d'annotations.

(ii) Quant au consommateur de code, il ne reste plus à sa charge que de confirmer que ces annotations représentent effectivement des points fixes valides. La phase de reconstruction de type et de vérification se traduit par un parcours linéaire du code en vue de valider les annotations inférées par le producteur. L'avantage de ce mécanisme est son efficacité en terme d'occupation de la mémoire — qui reste constante — et de temps d'exécution — complexité en $O(n)$.

II.2.3 Supports de déploiements extensibles et sécurisés

Nous allons présenter ici quelques environnements d'exécution qui répondent aux exigences formulées dans les sections précédentes.

II.2.3.1 JAVA CARD

La technologie JAVA CARD [Jav07] permet d'écrire des programmes dans un sous-ensemble du langage JAVA. Ces programmes sont destinés à l'exécution sur des cibles fortement contraintes telles que les cartes à puces. Le développement des applications se fait dans l'environnement JAVA classique. Un convertisseur permet en bout de chaîne de transformer le programme compilé qui consiste en un ensemble de classes (*paquetage*) en une unité de chargement appelé fichier CAP [36].

Le convertisseur vérifie que les instructions sont bien exprimées dans le sous-ensemble JAVA CARD. Puis, il effectue une série d'opérations permettant aux classes chargées d'être dans un

32. Le terme anglo-saxon correspondant est Lightweight Bytecode Verification.
33. KVM est l'acronyme de Kilo Virtual Machine.
34. CLDC est l'acronyme de Connected Limited Device Configuration.
35. JDK est l'acronyme de JAVA Developpement Kit.
36. cap est l'acronyme de Converted APplet.

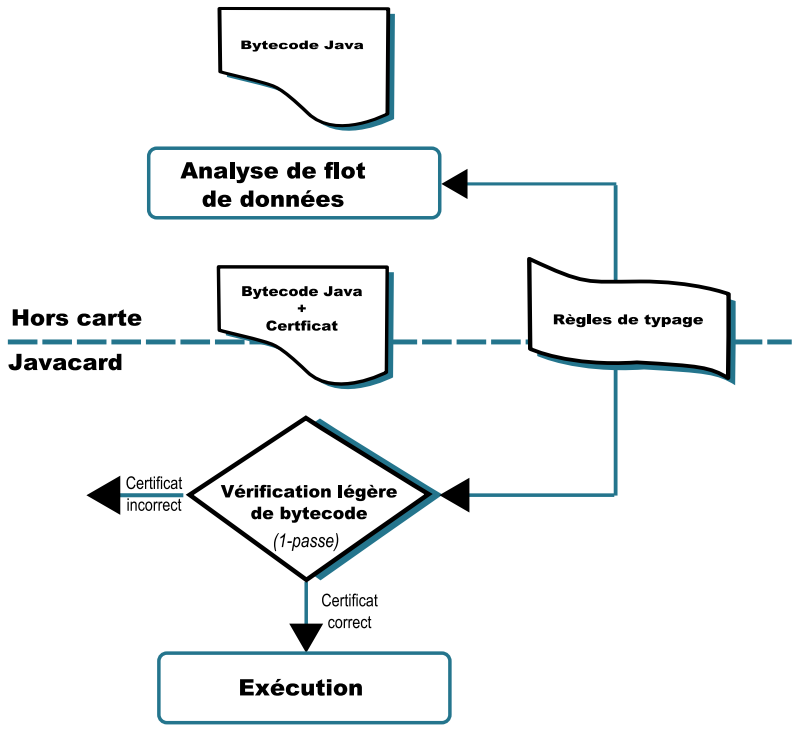


Figure II.6 — *Vérification légère de bytecode*

état très proche de leur état initial utile. Une représentation des classes chargées est d'abord créée. Un constant pool[37] global regroupe les constant pool de toutes les classes contenues dans le paquetage. Les références internes sont résolues. L'espace destiné aux champs statiques des classes est alloué. Un second fichier *export* contenant le nom et les informations nécessaires pour la liaison avec les autres classes peut ne pas être chargée sur la carte.

Les recommandations JAVA CARD stipulent qu'il faut pré-réserver/pré-allouer les objets JAVA lors du déploiement de l'application. Une fois l'application installée, elle aura réservé toutes les ressources mémoire dont elle avait besoin et elle est assurée de pouvoir s'exécuter par la suite sans risque d'échec liée à un dépassement de la capacité de mémoire. Dans le contexte de ce cycle de vie, les ramasses-miettes sont le plus souvent déclenchés sur demande par le terminal, après qu'il aie achevé l'installation d'une nouvelle application.

II.2.3.2 CAMILLE

CAMILLE est un système d'exploitation ouvert pour carte à microprocesseur issu des travaux de thèse de Gilles Grimaud [Gri00](Voir Figure II.8).

Dans CAMILLE, aucune abstraction du matériel n'est imposée au niveau du noyau. CAMILLE se base, en effet, sur les principes d'exo-noyaux énoncés dans [Eng98] qui consistent à

37. Le constant pool d'une classe est un tableau contenant toutes les valeurs statiques utilisées par cette classe.

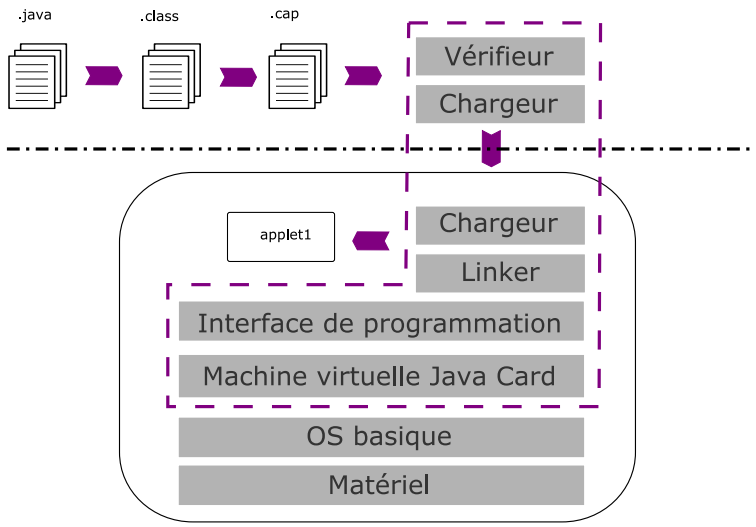


Figure II.7 — *Architecture Javacard*

utiliser la virtualisation pour implémenter le multiplexage de ressources. Une couche d'abstraction du matériel isole le noyau du système des spécificités des plate-formes matérielles.

CAMILLE propose une approche constructive particulièrement adaptée pour les cibles fortement contraintes en ressources matérielles telles les cartes à puce. Un mécanisme d'extension permet de charger des nouvelles applications ou de nouveaux composants pour le système de manière à enrichir incrémentalement ses fonctionnalités quand le besoin s'en ressent. Par défaut, CAMILLE charge un code intermédiaire écrit dans le langage FACADE [GLV99]. À partir de ce langage orienté objet, CAMILLE peut effectuer une vérification des propriétés de confidentialité et d'intégrité sur le code chargé. En effet, grâce à un modèle distribué inspiré de PCC, le système embarqué assure sans intervention extérieure (notamment en utilisant une preuve jointe au code et établie par un producteur potentiellement malveillant) le bon typage des applications qu'il charge.

Le système CAMILLE permet le déploiement de différentes applications issues de compilateurs ou de convertisseurs de bytecode. Ainsi, il est possible de déployer du code JAVA ou C au sein de la même carte à condition d'installer les bibliothèques de base requises à l'exécution [GHSR07]. Chaque application est compilée à la volée lorsqu'elle est chargée et le code FACADE est transformé en code natif pour une performance nominale.

II.3 Tour d'horizon du calcul de temps d'exécution au pire cas

Lorsque des vies humaines sont en jeu tel que dans le domaine de l'avionique ou de la santé, la prévisibilité temporelle du comportement des applications devient cruciale. Par exemple, un système de soins intensifs doit déclencher les alertes potentielles à temps en toutes circonstances.

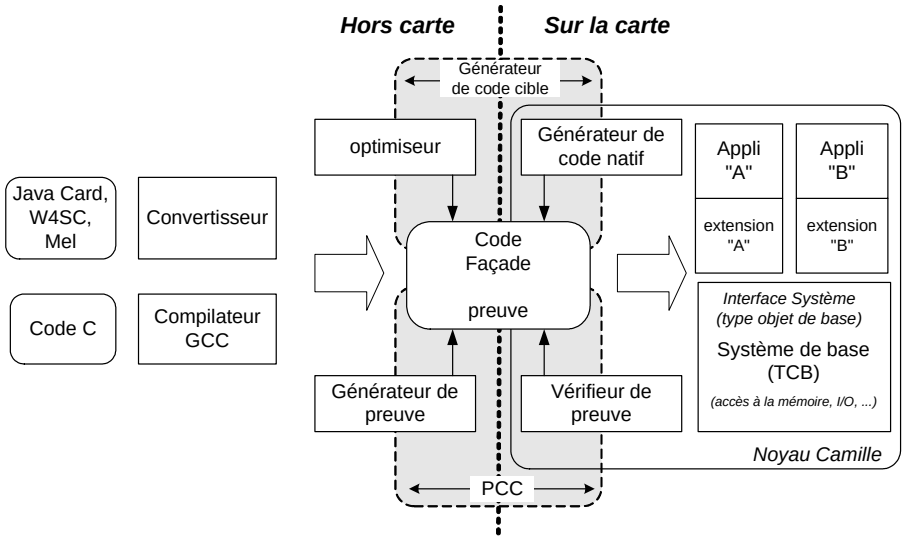


Figure II.8 — *Architecture d'un système d'exploitation ouvert pour carte à puce [Gri00]*

Ou encore, un système d'interception de missile nécessite que la riposte soit tirée dans un intervalle de temps précis. Cette nécessité de répondre dans un laps de temps donné est aussi valable pour un grand nombre d'applications quotidiennes afin de garantir une certaine qualité de service. Ce type de contraintes se rapporte aux *systèmes temps réel*.

II.3.1 Systèmes temps réel

Il est tout à fait possible de développer un logiciel ayant un comportement correct mais qui ne produit pas les résultats dans une échéance temporelle définie. A contrario, le comportement correct d'un système temps réel dépend non seulement des résultats logiques du traitement, mais également du temps auquel ces résultats sont produits [Sta88].

Les contraintes temporelles des systèmes temps-réel sont exprimées sous la forme d'échéances. Deux familles de systèmes temps réels se distinguent au regard de la flexibilité observée par rapport à leurs contraintes temporelles.

Un système temps-réel est dit *strict* s'il contient au moins une tâche temps-réel dont l'échéance est *stricte*. En d'autres mots, il est impératif de garantir la production des résultats avant la fin de l'échéance de cette tâche, faute de quoi ils seront considérés incorrects. Parmi les systèmes qui nécessitent le respect d'échéances temporelles strictes, on retrouve les systèmes critiques, typiquement les systèmes responsables de vies humaines.

En revanche, si le fonctionnement du système n'est pas remis en cause si l'échéance d'une tâche est ratée occasionnellement, celle-ci est dite souple. Dans ce cas, le non respect de l'échéance induit seulement la dégradation de la qualité du service délivré. Un système temps-réel *souple* ne contient, donc, que des tâches dont les échéances sont souples. C'est typiquement le cas des applications multimédia déployées sur un réseau. La perte ou l'arrivée de paquets en retard, engendre

la dégradation des performances du système et la qualité à la réception en sera éventuellement amoindrie.

Les applications temps réels se présentent souvent comme un ensemble de tâches ayant des besoins en ressources logiques ou matérielles (entre autres des ressources processeur). Ces tâches sont soumises à des contraintes liées aux spécifications et notamment des contraintes temporelles (*e.g.* temps d'exécution au pire cas, échéance, priorité, périodicité, ...). Le système temps-réel se charge de répartir les ressources matérielles ou logiques disponibles entre toutes les applications, tout en veillant au respect de toutes les contraintes temporelles. Un composant particulier, *l'ordonnanceur*, détermine l'ordre dans lequel celles-ci doivent être exécutées. Cet ordre, *schéma d'ordonnancement*, est arrêté en effectuant un *test d'ordonnançabilité* qui sert à évaluer la charge de travail maximale à laquelle le système peut être soumis en tenant compte des contraintes établies. Pour garantir que le schéma d'ordonnancement est faisable pour tous les chemins d'exécution possibles, l'ordonnanceur se base sur le calcul préalable du pire temps d'exécution de chaque tâche.

II.3.2 Calcul du temps d'exécution au pire cas

Le temps d'exécution au pire cas est un majorant sur tous les temps d'exécution possibles de tous les travaux de la tâche. Il dépend strictement de l'architecture matérielle sur laquelle s'exécute le programme.

Le pire chemin d'exécution observé dans un programme est défini comme suit dans [LS99] :

Définition II.3.1 (Le pire chemin d'exécution) *Le pire temps d'exécution d'un programme correspond au chemin du programme prenant le plus temps pour s'exécuter pour l'ensemble des données en entrée possibles et les états initiaux du système.*

Quelques systèmes temps-réel souples considèrent un temps d'exécution moyen, observé expérimentalement sur différentes traces d'exécution d'une application donnée.

Quatre éléments peuvent influencer le temps d'exécution d'un programme :

(i) les données en entrée : Par exemple, pour l'algorithme de tri à bulle, on constate le pire temps d'exécution quand le tableau est trié dans l'ordre inverse. Au contraire, pour l'algorithme de tri par sélection, le pire scénario consiste à ordonner un tableau préalablement trié ;

(ii) le comportement intrinsèque du programme : c'est évidemment le comportement du programme qui détermine son pire temps d'exécution ;

(iii) le compilateur : Selon la stratégie de compilation, à un programme source donné, peut correspondre un ensemble d'instructions natives différent, induisant des temps d'exécution variables ;

(iv) le matériel sous-jacent : la latence de la mémoire, la présence de cache ou de pipeline, la cadence du processeur, sont autant de critères influant sur le temps d'exécution des programmes.

Dans le processus de développement d'un système critique, la validation des propriétés temps réels joue un rôle très important car elle permet d'atteindre le niveau de confiance requis. Cependant, elle est souvent difficile et peut augmenter le temps de développement, accroissant ainsi le coût total du système. Aujourd'hui, la validation repose principalement sur des activités de

test, dont l'objectif est de découvrir les défauts présents dans le système. Cependant, il est souvent plus judicieux d'établir les propriétés des programmes en procédant par analyse statique. Un compromis entre fiabilité et précision reste à trouver entre ces deux principales approches utilisées pour le calcul de WCET.

II.3.2.1 Méthodes dynamiques

Les méthodes dynamiques procèdent en observant (mesurant) le comportement du programme en exécutant une batterie de test afin de déterminer une estimation du pire temps d'exécution en conditions réelles. Toutefois, les approches basées sur le test ne peuvent pas être exhaustives (dans le cas général) et ne garantissent pas une couverture totale de l'ensemble des entrées du programme. De ce fait, cette méthode empirique détermine souvent une estimation du WCET inférieure à la valeur réelle.

Le risque que la valeur du WCET déterminée par cette méthode ne soit optimiste est son principal inconvénient. La notion de garantie de respect des échéances devient obsolète quand il existe une probabilité que le programme prenne plus de temps que la valeur de WCET annoncée. Cette sous-estimation ne peut donc pas convenir dans le cas des systèmes temps réels durs où le respect des échéances est d'une importance capitale.

II.3.2.2 Méthodes statiques

Ces méthodes sont fondées sur une analyse statique du programme et s'affranchissent des jeux de données en entrée. Elles comportent en général trois phases.

L'analyse de haut niveau détermine tous les chemins d'exécution possibles. Elle se situe généralement au niveau du code source et se matérialise par une analyse du flot de contrôle du programme permettant d'identifier les structures conditionnelles et itératives. Les informations structurelles peuvent être extraites automatiquement à partir du programme. Cependant, d'autres informations dépendant de la sémantique du programme telles que les bornes de boucles ou les chemins infaisables sollicitent parfois le concours du développeur.

L'analyse de bas niveau, quant à elle, évalue le temps consommé — exprimé en cycle processeur — par chaque unité élémentaire d'exécution au regard des spécificités du processeur et des mémoires utilisés. Elle doit être effectuée au plus proche du matériel donc directement sur le code natif. Il existe une différenciation classique entre l'analyse locale (*i.e.* pipeline) et l'analyse globale faisant intervenir le contexte des instructions (*i.e.* cache).

Enfin, *le calcul de* WCET proprement-dit se fait en combinant les informations provenant de l'analyse de haut niveau avec celles déterminées au plus proche de l'architecture matérielle cible. On distingue trois méthodes principales de calcul de WCET : (i) celles qui exploitent l'arbre syntaxique du programme [PS91, PK89a, CP01a], (ii) celles qui se basent sur l'énumération de tous les chemins d'exécution possibles à partir du graphe de flot de contrôle [SA00b], et enfin (iii) celles qui modélisent les chemins par des contraintes arithmétiques permettant ainsi de les énumérer implicitement [LM95].

II.4 Synthèse et propositions

II.4.1 Synthèse

Comme nous l'avons expliqué précédemment, le dispositif personnel de confiance doit offrir un support d'exécution sécurisé à son utilisateur. Toutefois, cette sécurité ne doit en aucun cas reposer sur une intervention extérieure. Le TPD doit avoir les moyens de s'assurer de l'innocuité d'un logiciel qui peut potentiellement se montrer hostile à l'égard de ceux préalablement déployés. Les problèmes de sécurité de code mobile ont été largement traités en terme de confidentialité et d'intégrité des logiciels déployés, les uns vis-à-vis des autres. Les solutions existantes sont aussi bien matérielles que logicielles et opèrent tantôt lors de l'exécution (dynamiques) tantôt au moment où le logiciel est déployé dans la carte. Celles qui sont les plus probantes dans notre contexte sont statiques et logicielles.

L'intérêt de ces techniques réside dans le fait qu'elles acceptent ou refusent le logiciel au moment où il est installé. Ainsi, si un utilisateur déploie un nouveau service dans sa carte, il sait s'il aura accès au service ou pas. Les technique dynamiques sont moins satisfaisantes, parce que le programme pourra s'installer et marcher convenablement parfois. D'autres fois, lorsqu'il aura un comportement erroné ou dangereux pour les autres applications installées, il sera stoppé par l'environnement d'exécution (ou le matériel) donc il ne nuira pas au TPD mais il privera l'utilisateur d'un service qu'il a pourtant acquis (voire acheté). C'est ici tout l'intérêt de l'établissement de la sécurité par analyse statique au déploiement. Il faut noter cependant que si différentes solutions existent pour traiter la sécurité en terme de confidentialité et d'intégrité au déploiement. Il n'existe cependant des solutions convaincantes pour garantir a priori la disponibilité des ressources nécessaires à l'exécution d'une application. Aussi, l'installation d'une nouvelle application (malveillante ou simplement gourmande) peut nuire au bon fonctionnement des autres applications en consommant les ressources dont elles avaient besoin. Ainsi, dans le contexte des TPD, une nouvelle application déployée a posteriori peut non seulement se trouver dans l'incapacité de fonctionner, mais elle peut aussi rendre impossible l'utilisation d'autres applications déployées et qui jusque là fonctionnaient correctement. Notre principale préoccupation consiste donc à n'exécuter que les programmes pour lesquels le temps d'exécution au pire cas est statiquement décidable. Rendre l'exécution déterministe en terme de consommation CPU, mémoire et énergie permettra de garantir la disponibilité des services.

L'analyse dynamique du pire temps d'exécution consiste à observer le comportement du programme en exécutant une batterie de test afin de déterminer une estimation du pire temps d'exécution. Cette méthode empirique peut déterminer une estimation du WCET plus optimiste que la valeur réelle. La valeur du WCET déterminée par cette méthode étant inférieure à la valeur réelle, la notion de garantie de respect des échéances devient obsolète. En effet, il existe une probabilité que le programme prenne plus de temps que la valeur annoncée. Cette sous-estimation ne peut donc pas convenir dans le cas des systèmes temps réels durs où le respect des échéances est d'une importance capitale. En revanche, les méthodes statiques permettent d'obtenir une borne majorante fiable mais malheureusement souvent plus pessimiste. La construction de l'arbre syntaxique requiert l'extraction de la structure logique du programme. Appliquée au contexte de code mobile où le consommateur ne dispose que d'une forme intermédiaire du code à exécuter, cette analyse ne peut pas être aussi riche. En effet, les langages intermédiaires sont plus souvent proches des langages machines, proposent un jeu d'instruction limité et ne prennent pas en charge des instructions de haut-niveau telles que *if-then-else* ou *loop*. Ainsi, l'application de cette méthode dans notre contexte est compromise. Des algorithmes de résolution de contraintes ou de programmation linéaire, notamment de type simplex, sont utilisés pour résoudre ce problème de maximisation. Le calcul des n_i se fera idéalement sur le producteur alors que le calcul des

w_i dépendant de la plate-forme physique doit se faire directement au moment du déploiement du code sur le consommateur. De ce fait, le calcul du WCET global ne pourra se faire que sur le consommateur. Hors, le consommateur, qui ne dispose que de ressources limitées, ne pourrait pas assumer la charge de travail qui incombe à un algorithme de programmation linéaire. Les algorithmes de recherche du plus long chemin du type Dijikstra [Dij59a] sont appliqués aux graphes. Pour des programmes complexes, les ressources mémoire nécessaires à la recherche du plus long chemin sont largement supérieures à celles disponibles sur les consommateurs ciblés. Cette recherche ne peut pas être exécutée sur le producteur car les coûts des différents arcs ne peuvent pas être exportés à l'extérieur.

II.4.2 Propositions

Il est possible, une fois connue l'architecture matérielle sous-jacente, de connaître les ressources consommées par une section de code séquentielle. Cependant, un flot de contrôle plus complexe rend plus difficile cette détermination. En effet, l'usage de structures de contrôle plus complexes telles que les boucles, la récursivité ou encore les appels de fonction, peuvent être source d'imprévisibilité. Plusieurs solutions ont été envisagées pour répondre à ce problème :

Réduire l'expressivité du langage de programmation pour ne laisser que les structures que l'on peut borner dans tous les cas [HKM+99]. La boucle *For* du langage ADA est un exemple typique d'une structure itérative dont les bornes sont toujours décidables car le compteur de boucle ne peut être modifié dans le corps de celle-ci sans générer d'erreur à la compilation. De plus, certains travaux considèrent par exemple qu'il est interdit d'utiliser la récursivité et le chargement dynamique [Gus02]. Ces contraintes sont d'autant plus difficiles à tenir que pour les cibles adressées, l'approche constructive, par chargement de briques logicielles *a posteriori*, est une nécessité.

Demander l'intervention du programmeur pour fournir des bornes constantes à ses boucles. L'intervention peut être requise a priori au moment du développement du code sous la forme d'annotations, par exemple. Cette approche manuelle peut induire des erreurs et demande, en outre, un temps de développement plus important.

Définir un budget temps limite (arbitraire ou basé sur des tests) après lequel l'exécution de la boucle est arrêtée. Dans ce but, le développeur peut définir des assertions. Un mécanisme de surveillance doit être mis en place afin de pouvoir contrôler à l'exécution le nombre maximum d'itération des boucles. Cette approche souffre des mêmes inconvénients que la précédente mais en plus elle induit un sur-coût à l'exécution.

Les méthodes de calcul de temps d'exécution au pire cas basées sur l'analyse statique ne prennent pas en compte les spécificités du code mobile décrites précédemment. On se propose, donc, de définir un schéma de calcul de WCET distribué qui définit deux phases qui se déroulent sur deux supports d'exécution distincts. Une première phase se déroulera au moment de la compilation sur le producteur et permettra d'alléger la charge de travail qui sera effectuée, dans une seconde phase, sur un consommateur. Le défi consiste à réaliser une distribution efficace du calcul permettant de profiter de la puissance de calcul et des ressources mémoires disponibles sur un producteur de code afin de ne plus avoir qu'à finaliser le calcul au moment de son déploiement.

La distribution du calcul du temps d'exécution au pire cas est établie par analogie au schéma de fonctionnement de la compilation distribuée pour le code mobile. Comme le montre la figure II.9, le code source est d'abord transformé en code intermédiaire par la partie frontale du

compilateur qui réside sur le producteur de code. De la même manière, une première partie du calcul de WCET sera effectuée sur ce dernier. Dans cette phase où l'on dispose du source, il est plus facile d'analyser le code et de dégager les propriétés qui interviennent directement dans le calcul du pire temps d'exécution. Le code intermédiaire généré est ensuite mis à disposition du consommateur. Ce dernier, avec la partie arrière du compilateur, lit le code envoyé et produit le code natif adapté à la plate-forme sur lequel il tourne. Parallèlement, le calcul de WCET est finalisé et une valeur globale, qui se traduit en nombre de cycles de processeur consommés, est déterminée.

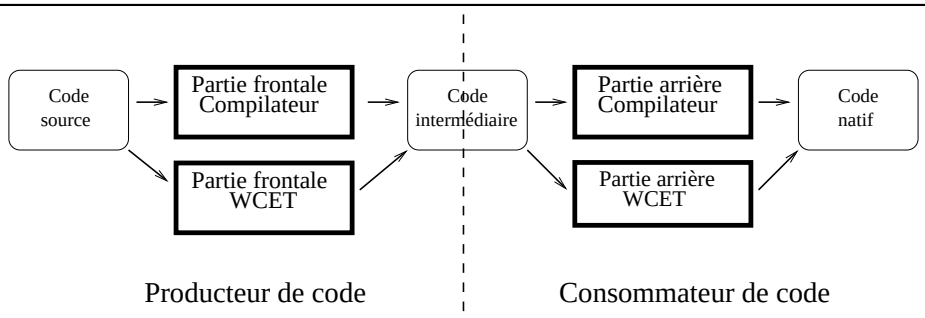


Figure II.9 — *Schéma distribué de calcul de* WCET

III

Algorithme distribué de détection des bornes des boucles

> *Quand on ne sait pas, on ne se pose pas trop de questions, mais quand on commence à disposer d'un début d'explication, on veut à tout prix tout savoir, tout comprendre.*
>
> — Bernard Werber - Les Thanatonautes

> *Je laisse intentionnellement de côté la plus ou moins grande longueur pratique des opérations ; l'essentiel est que chacune de ces opérations soit exécutable en un temps fini, par une méthode sûre et sans ambiguïté.*
>
> — Émile Borel, Le calcul des intégrales définies

On s'intéresse dans ce chapitre à la détection et la vérification de bornes sur les boucles. Cette question a déjà été étudiée dans des domaines différents, et s'appuie sur certaines bases théoriques que l'on va maintenant détailler.

III.1 Préliminaires théoriques

Le lecteur trouvera ici les pré-requis lui permettant de comprendre les bases théoriques de l'approche que nous proposons pour la détection des bornes des boucles.

Nous allons nous intéresser plus particulièrement aux approches basées sur l'analyse statique de code. Celle-ci consiste à étudier le comportement à l'exécution d'un programme (sans pour autant l'exécuter) et à utiliser les informations collectées pour vérifier que le programme satisfait une certaine propriété (ou encore l'optimiser). Cependant, l'analyse statique se heurte au problème d'indécidabilité[1] dans le cas général comme l'a montré Alan Turing dans [Tur37]. Le théorème de Rice qui a été initialement énoncé dans [Ric53] stipule, en effet, que :

Théorème 1 (Rice) *Toute propriété non triviale*[2] *qui dépend de la sémantique d'un programme est indécidable.*

L'analyse statique peut prendre place à différentes granularités de code comme le montre la Figure III.1. En effet, les programmes sont structurés comme un ensemble de sous-programmes (respectivement *procédures* ou *fonctions* pour les langages procéduraux et *méthodes* pour les langages orientés objets). Le flot de contrôle d'un sous-programme se définit par les instructions de branchement qui permettent de le découper en blocs de base (*i.e.* une section séquentielle de code ayant un seul point d'entrée, la première instruction, et un seul point de sortie, la dernière instruction). Au niveau du bloc de base, on parle d'analyse *locale*. Si l'on s'intéresse aux instructions de branchements au sein de la même méthode, on parle d'analyse *intra-méthode*. Les instructions d'appel de méthodes donnent lieu à l'analyse *inter-méthodes*.

III.1.1 Analyse de flot d contrôle

Dans le cas général, une analyse statique se base sur la connaissance du flot de contrôle. L'analyse de flot de contrôle repose essentiellement sur la construction de graphes de flot de contrôle[3][ASU86]. Construire le CFG à partir d'un programme donné revient à le découper en blocs de base. Les noeuds représentent les blocs de base. Les instructions de branchement permettent de construire les arcs orientés du CFG.

Un CFG peut être décrit formellement par $G =< \mathcal{N}, \mathcal{A}, \text{Entrée} >$ avec $\mathcal{N}$ l'ensemble des noeuds, $\mathcal{A}$ l'ensemble des arcs (*i.e.* $\mathcal{A} \subseteq \mathcal{N} \times \mathcal{N}$) et Entrée un noeud qui n'a pas de prédécesseurs et qui permet d'atteindre tous les autres noeuds du graphe.

1. Dans la théorie de calculabilité, une propriété est décidable s'il existe un algorithme permettant de décider si cette propriété est vraie ou fausse dans un temps fini.

2. Par propriété *non triviale*, il désigne celles qui peuvent être tour à tour vraies ou fausses pour différents programmes écrits dans le même langage de programmation.

3. Dans le reste du document, nous utiliserons l'acronyme CFG[4] pour faire référence à un graphe de flot de contrôle.

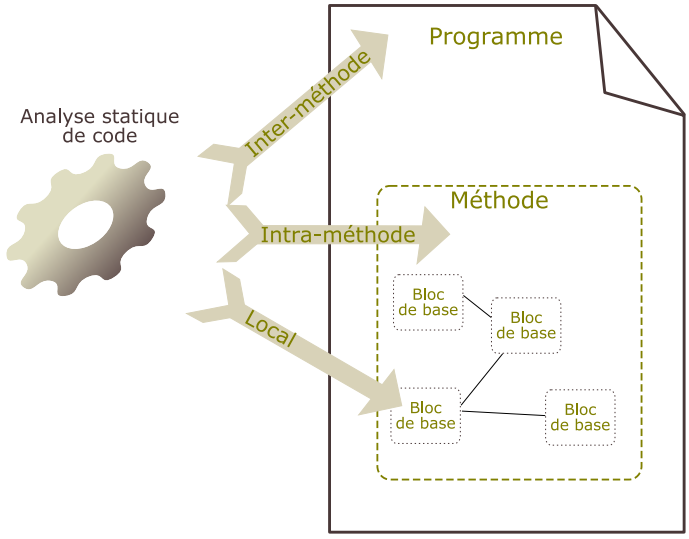


Figure III.1 — *Analyse statique de code à différentes granularités*

Un flot de contrôle séquentiel peut être interrompu par des instructions de branchement formant ainsi des structures itératives, telles les boucles. Un CFG est dit *réductible* si toutes les boucles qu'il contient sont *naturelles* (Voir Figure III.2a). Une boucle est dite *naturelle* si elle vérifie deux propriétés de base [ASU86] :

(i) Elle doit avoir un seul point d'entrée appelé *entête* qui est exécuté à chaque itération ;

(ii) Elle a au moins un *arc de retour* vers l'entête ce qui lui permet de se répéter.

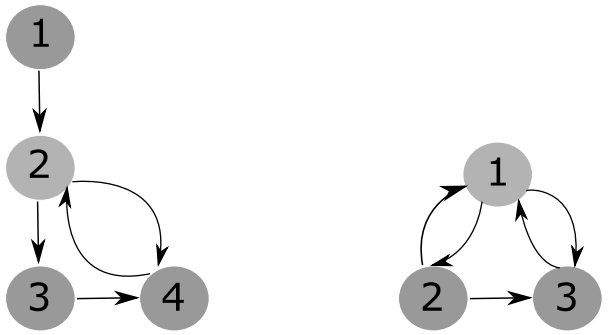
L'identification des boucles naturelles dans un CFG est effectuée en détectant les arcs représentant des branchements en arrière dans le flot de contrôle. Un arc $a = (x, y)$ de $\mathcal{A}$ est dit arrière si et seulement y domine x. Dans ce cas là, y est la tête de la boucle naturelle et tous les noeuds situés sur les chemins entre y et x forment le corps de la boucle.

Définition III.1.1 (Relation de dominance) *$\forall x, y \in \mathcal{N}$, x domine y si et seulement si tout chemin de partant de l'entrée du graphe et menant à y passe par x. Cela signifie que si le bloc y est exécuté, x a aussi été exécuté.*

Il faut noter qu'il existe des cas où des boucles peuvent avoir en commun la même entête ou être mêlées sans qu'il ne soit possible de déterminer une structure de boucle imbriquée comme le montre la figure III.2b.

Dans ce cas, le CFG est dit *irréductible*. Il n'est plus possible alors d'utiliser la relation de dominance pour identifier les boucles. Dans ce cas, le recours au tri topologique des noeuds du CFG est nécessaire. Trier topologiquement les noeuds d'un graphe revient à donner une énumération de ses noeuds telle que s'il y a un arc de v_i vers v_j, alors v_i précède v_j dans l'énumération.

Définition III.1.2 (Ordre topologique) *On appelle tri topologique de l'ensemble des sommets $\mathcal{N}$ d'un graphe G une bijection γ de $\mathcal{N}$ dans $\{1, \ldots, n\}$ tel que si (n_i, n_j) appartient à l'ensemble des arcs $\mathcal{A}$ alors $\gamma(n_i) < \gamma(n_j)$. L'ordre topologique pourra s'écrire alors $n_1 \prec n_2 \prec \ldots \prec n_n$.*



(a) Boucle naturelle

(b) Boucle irréductible

Figure III.2 — *Exemples de graphes de flot de contrôle.*

Une fois les noeuds triés topologiquement, il faut parcourir le CFG en visitant chaque noeud dans l'ordre topologique inverse et en déterminant s'il y a un branchement arrière vers ce noeud afin de situer les boucles.

III.1.2 Analyse de flot de données

L'analyse de flot de données [Flo67] sert à collecter des informations pour chaque instruction en propageant itérativement les informations de flot de données calculées localement à travers le CFG. Les techniques de propagation de constantes, par exemple, suivent le CFG et examinent pour chaque instruction l'ensemble des variables susceptibles d'être modifiées.

Dans le cas général, une analyse de flot de données se déroule en plusieurs étapes. D'abord, il s'agit de générer un système d'équations exprimant une propriété locale pour chaque bloc de base. En effet, à chaque bloc de base BB_i, sont associées deux variables $IN(BB_i)$ et $OUT(BB_i)$ reliées par les équations suivantes :

$$\begin{aligned} IN(BB_i) &= \bigcup_{n \in Pred(BB_i)} OUT(n) \\ OUT(BB_i) &= F(BB_i, IN(BB_i)) \end{aligned}$$

F est une fonction de transfert reliant les variables $IN(BB_i)$ et $OUT(BB_i)$ pour chaque bloc de base et $Pred(BB_i)$ désigne l'ensemble des prédécesseurs de BB_i dans le CFG.

Les propriétés collectées sont ensuite exprimées sous la forme d'un treillis. Ces derniers sont des objets mathématiques qui peuvent être manipulés en tant qu'ensembles partiellement ordonnés et qui ont été amplement étudiés dans la littérature [5]

Définition III.1.3 (treillis) *Un treillis est défini par* $(\mathcal{L}, \subseteq, \cup, \cap, \top, \bot)$ *où :*

5. Nous renvoyons le lecteur aux travaux [Pic05] dont nous nous sommes inspirés pour donner ces définitions.

(i) $(\mathcal{L}, \subseteq)$ *est un ensemble partiellement ordonné (voir la Définition III.1.4) ;*

(ii) $\top$ *est le plus grand élément défini sur l'ensemble* $\mathcal{L}$ *tel que* $a \subseteq \top$ *pour tout* a *appartenant à* $\mathcal{L}$*. De même,* $\bot$ *est le plus petit élément ;*

(iii) $\cap, \cup$ *sont des relations binaires entre les éléments de l'ensemble. La relation* $\cap$ *est définie telle que pour toute paire d'éléments* $a, b \in \mathcal{L}$*, il existe une borne supérieure, notée* $a \cap b$*, qui est le plus petit élément vérifiant* $a \subseteq a \cap b$ *et* $b \subseteq a \cap b$*. Il en est de même pour* $\cup$*.*

Définition III.1.4 (Ensemble partiellement ordonné) *Un ensemble partiellement ordonné est un ensemble* $\mathcal{L}$ *muni d'une relation d'ordre partiel* $\subseteq$ *qui est :*

(i) réflexive $(\forall a \in \mathcal{L}, a \subseteq a)$*,*

(ii) anti-symétrique $(\forall a, b \in \mathcal{L}, a \subseteq b, b \subseteq a \rightarrow a = b)$*,*

(iii) et transitive $(\forall a, b, c \in \mathcal{L}, a \subseteq b, b \subseteq c \rightarrow a \subseteq c)$*.*

L'ensemble partiellement ordonné $(\mathcal{L}, \subseteq)$ *pourra être représenté par un graphe appelé diagramme de Hasse*[6]*.*

Enfin, le système d'équations calculées à tout point du programme est résolu dans le treillis en propageant la propriété que l'on souhaite exprimer à travers le CFG du programme. Les équations sont calculées itérativement pour chaque bloc de base jusqu'à ce qu'un point fixe soit atteint. L'algorithme itératif général est présenté dans la Figure III.3.

```
1. OUT(s) := NULL
2. Pour tout n ∈ N − {s}
        3. OUT(n) := 1
4. Faire
        5. stable := vrai
        6. Pour tout n ∈ N − {s}
                7. IN(n) := ⋃ OUT(Pred(n))
                8. NEW := f_n(IN(n))
                9. Si NEW ≠ OUT(n) alors
                        10. OUT(n) := NEW
                        11. stable = faux
                12. fin Si
        13. fin Pour
14. Tant que ( !stable)
```

Figure III.3 — *Algorithme itératif d'analyse de flot de données*

L'existence d'une solution du système d'équations n'est assurée que si les opérateurs $\cap$ et $\cup$ qui apparaissent dans les équations sont monotones.

6. Chaque élément de $\mathcal{L}$ y est représenté par un sommet. Un arc reliant deux sommets du diagramme signifie que les éléments représentés par ces sommets sont comparables. Par convention, l'ordre est croissant dans le sens du bas vers le haut du diagramme.

Définition III.1.5 (Fonction monotone) *Une fonction* $f : \mathcal{L} \mapsto \mathcal{L}$ *est dite monotone* si $\forall a, b \in \mathcal{L}, a \subseteq b \Rightarrow f(a) \subseteq f(b)$

À partir de ces bases théoriques, différents travaux ont déjà traité de la détection de bornes de boucles.

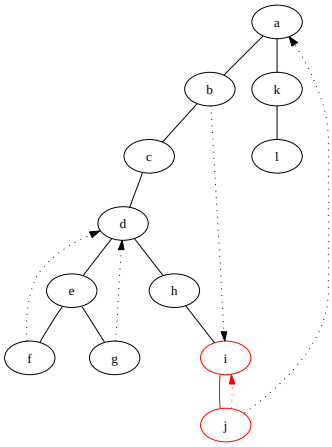
III.2 Travaux connexes

Dans ce qui suit, nous allons nous faire l'écho de quelques travaux académiques qui se préoccupent de la détermination du nombre d'itérations des boucles. Nous allons regrouper ces travaux en terme de méthode d'extraction du flot du programme.

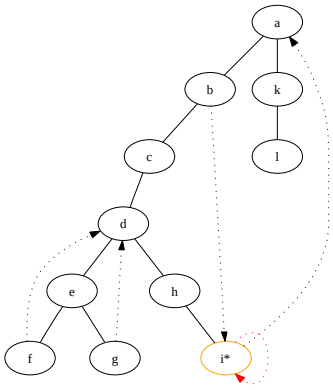
III.2.1 Patrons structurels prédéfinis

En se basant sur la méthode décrite dans [Sha80], quelques travaux se servent des patrons prédéfinis pour déterminer les différentes structures syntaxiques du programme. En d'autres mots, le CFG est décomposé en un ensemble de régions correspondant aux structures sémantiques de base (boucles, structures conditionnelles, ...).

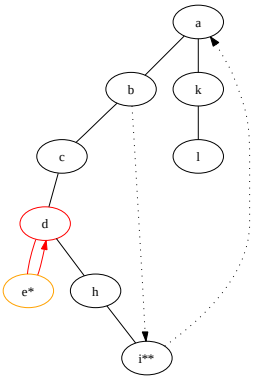
L'algorithme utilisé effectue un parcours post-fixé en profondeur d'abord du graphe de l'arbre couvrant du CFG. Pour chaque noeud traversé, un outil permet de reconnaître la structure sémantique en cours en effectuant des comparaisons successives avec les modèles prédéfinis. Dans le cas où une structure sémantique est reconnue, elle est remplacée par un noeud structurel. L'algorithme continue à itérer en remontant à partir des feuilles vers la racine de l'arbre jusqu'à converger vers une structure inconnue. La Figure III.4 montre le fonctionnement de la méthode sur un exemple.



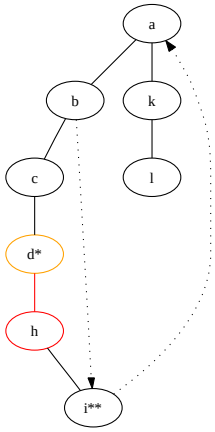
(a) i,j forment un bloc i*



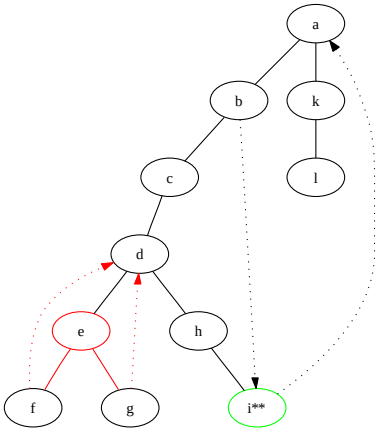
(b) i* forme une boucle i**



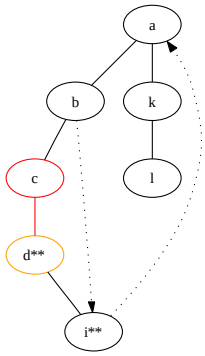
(c) e, f, g forment une structure if-then-else e*



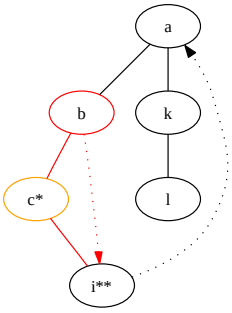
(d) d, e* forment une boucle while d*



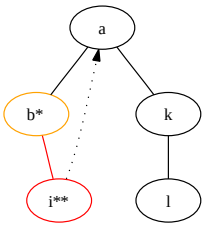
(e) d*, h forment un bloc d**



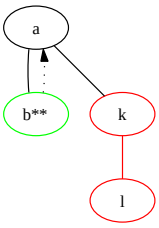
(f) c, d** forment un bloc c*



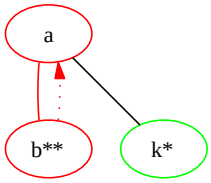
(g) b, c* forment une structure if-then bb*



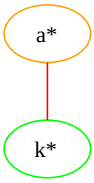
(h) b*, i** forment un bloc b*



(i) k, l forment un bloc k*



(j) a, b** forment une boucle while a*



(k) a*, k* forment un block a**

(l) plus de réductions possibles

Figure III.4 — *Analyse structurelle permettant de détecter les boucles.*

Cette méthode est utilisée dans les travaux de Corti et al. [CG04] qui l'ont appliqué sur du bytecode JAVA. Elle est aussi utilisée dans les premières versions de l'outil commercial **aiT**. Ces patrons dépendent du compilateur utilisé et même des niveaux d'optimisation du code.

III.2.2 Logique de Presburger

Dans l'outil Bound-T [Bou07], la détection automatique de boucles permet d'identifier celles qui définissent une variable compteur qui serait incrémentée ou décrémentée d'un pas constant à chaque itération et qui se terminent lorsque le compteur devient supérieur ou inférieur à une valeur limite constante. Cet outil exprime les relations entre les variables du programme en utilisant la logique de Presburger [Pre29] [7] décidable pour des valeurs appartenant aux entiers relatifs. Un outil de résolution automatique appelé Omega [Pug91], permet d'éliminer les quantificateurs des formules de Presburger afin de déterminer le nombre d'itérations de la boucle. La détection se passe au niveau du code objet.

III.2.3 Analyse de flot de données

Dans [HSR+00], les auteurs se basent sur les principes de l'analyse de flot de données dans leur technique de détection automatique de boucles. Ils définissent la notion de *branchement d'itération* qui se matérialise par un branchement conditionnel à la fin d'un bloc de base et qui implique le transfert du flot de contrôle vers deux points de saut qui peuvent affecter directement ou indirectement le nombre d'itérations d'une boucle. Une fois les blocs contenant les branchements d'itération identifiés grâce à la relation de dominance, ils sont isolés et reliés entre eux pour former un graphe de précédence. Ce graphe dirigé acyclique renseigne sur l'ordre dans lequel les blocs de base concernés s'exécuteraient lors d'une itération de la boucle. L'évolution de la variable d'induction de la boucle est suivie pour chaque branchement d'itération en suivant l'ordre d'exécution indiqué par le graphe de précédence.

III.3 Proposition d'un système de types pour la détection de boucles

Puisque le calcul de toutes les propriétés intéressantes d'un programme n'est pas décidable, l'usage des approximations fiables est souvent nécessaire. L'interprétation abstraite associe à chaque instruction du programme un élément d'un treillis de propriétés. La correction est établie en reliant cette sémantique abstraite à une autre plus concrète. Une autre façon d'atteindre cette objectif est de donner une logique ou un système de types dans lequel on peut dériver des jugements affirmant que chaque instruction du programme satisfait un prédicat particulier. C'est la méthode que nous avons choisi.

Un système d'inférence de type classique se compose d'une algèbre de types, d'un ensemble de règles logiques affectant un type à chaque expression du langage étudié et d'un algorithme de reconstruction de types calculant le type le plus général d'une expression quelconque.

7. La logique de Presburger correspond à toutes les formules linéaires que l'on peut écrire à partir de contraintes linéaires $\sum_n^1 \alpha_i.x_i = c$ ou $\sum_n^1 \alpha_i.x_i \leq c$ où α_i et c sont des constantes entières, des conjonctions et disjonctions de formules, des quantifications existentielles $\exists p$ et universelles $\forall p$.

III.3.1 Jeu d'instructions

Un programme P est une séquence d'instructions I chacune identifiée par un compteur pc qui représente sa position dans le programme. Nous nous intéressons à un ensemble fini d'instructions. Des opérations arithmétiques élémentaires, nous ne retiendrons que les opérations d'affectation, d'addition et de comparaison. Les instructions de flot de contrôle *Jump* et *Jumpif* sont utilisées respectivement pour le branchement et le branchement conditionnel. Pour prendre en compte l'appel de fonctions, les instructions d'invocation de méthodes et des instructions de retour à l'appelant sont aussi prises en compte. Le jeu d'instructions complet est illustré par la Figure III.5.

Instructions	I : :=	v ← c	(1)
		v ← v + c	(2)
		b ← v *cmp* c	(3)
		vd ← vs op tabArgs	(4)
		Jump L_i	(5)
		Jumpif v, L_i	(6)
		Return v	(7)
Programme	P : :=	PI | I	

Figure III.5 — *Ensemble d'instructions*

III.3.2 Définition de la relation de précédence

Pour détecter les bornes des boucles, il est nécessaire d'identifier chaque étape dans les modèles d'exécution cités dans [BGSR06]. Je définis la relation de précédence comme l'ordre dans lequel les blocs de base s'exécutent à l'intérieur d'une boucle. Soient x et y deux noeuds du **CFG** . $x \subseteq y$ si et seulement si x s'exécute toujours avant y dans tous les chemins d'exécution possibles. Nous utiliserons une fonction *ordre* qui attribue un numéro à chaque noeud tel que : $\forall x, y \in V / x \subseteq y \rightarrow ordre(x) < ordre(y)$. La fonction *ordre* peut être assimilée à un ordre topologique (*i.e.* effectué en parcourant en profondeur d'abord le **CFG**). Le parcours en profondeur visite tous les noeuds d'un graphe en les marquant au fur et à mesure.

La figure III.6 illustre la relation de précédence dans nos deux modèles de boucles ; les numéros indiqués sont les numéros d'ordre correspondant au tri topologique.

III.3.3 Système de type

Une variable d'un programme donné peut avoir plusieurs types tel qu'illustré dans la figure III.7. Si une variable a un type inutile pour la détection des itérateurs, son type est considéré $\top$. Une variable peut être par ailleurs une valeur constante, un intervalle ouvert d'entiers (où α est la borne inférieure de l'intervalle et σ le pas d'incrémentation), une condition (où *var* est comparée à une valeur constante ψ en utilisant l'opérateur *cmp*) ou un itérateur de boucle (où α est la borne inférieure de l'intervalle, ψ la borne supérieure et σ le pas d'incrémentation).

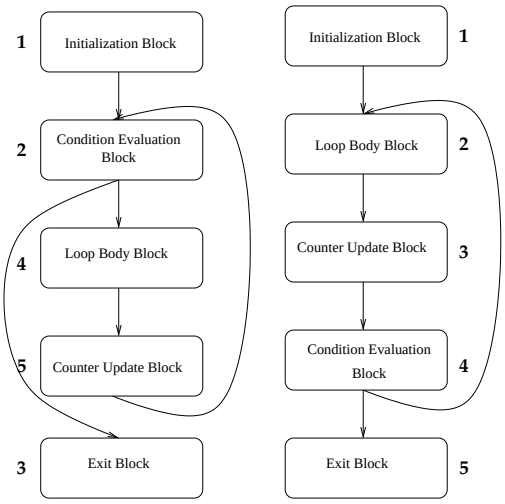


Figure III.6 — *Dans le cas où le test de sortie se trouve à l'entrée de la boucle, on distingue les étapes suivantes dans l'exécution d'un e boucle : initialisation (1), condition d'évaluation (2), mise à jour du compteur (5).*

Types	T : :=	$\top$	Inutile
		$\{\alpha\}_e$	Valeur constante
		$[\alpha \stackrel{\sigma}{:} [_e$	Intervalle ouvert
		$(var, cmp, \psi)_e$	Condition
		$[\alpha \stackrel{\sigma}{:} \psi]_e^?$	Iterateur possible
		$[\alpha \stackrel{\sigma}{:} \psi]_e^!$	Iterateur confirmé

Figure III.7 — *Le système de type*

Nous faisons une distinction entre un itérateur possible et un itérateur confirmé grâce aux étiquettes ? et !. Un itérateur possible est obtenu quand une variable a été successivement initialisée, incrémentée et comparée à une valeur constante en suivant un seul chemin d'exécution. Un itérateur confirmé est obtenu quand une variable a été successivement initialisée, incrémentée et comparée à une valeur constante en suivant tous les chemins d'exécution.

Les types sauf $\top$ portent une estampille e qui renseigne sur l'étape à laquelle le type a été créé dans le modèle d'exécution de la boucle. Cette estampille permet de s'assurer de la relation de précédence entre les blocs de base spécifiques identifié à la figure III.6. Une fonction χ sert à attribuer un numéro d'ordre à chaque instruction selon le bloc de base auquel elle appartient. Quand une nouveau type T est créé à l'instruction i, la fonction $\chi(i) = e$ permet d'estampille le type nouvellement crée par le numéro d'ordre du bloc de base en cours (T_e).

III.3.3.1 Sémantique opérationnelle

La sémantique du langage est donnée dans un style opérationnel.

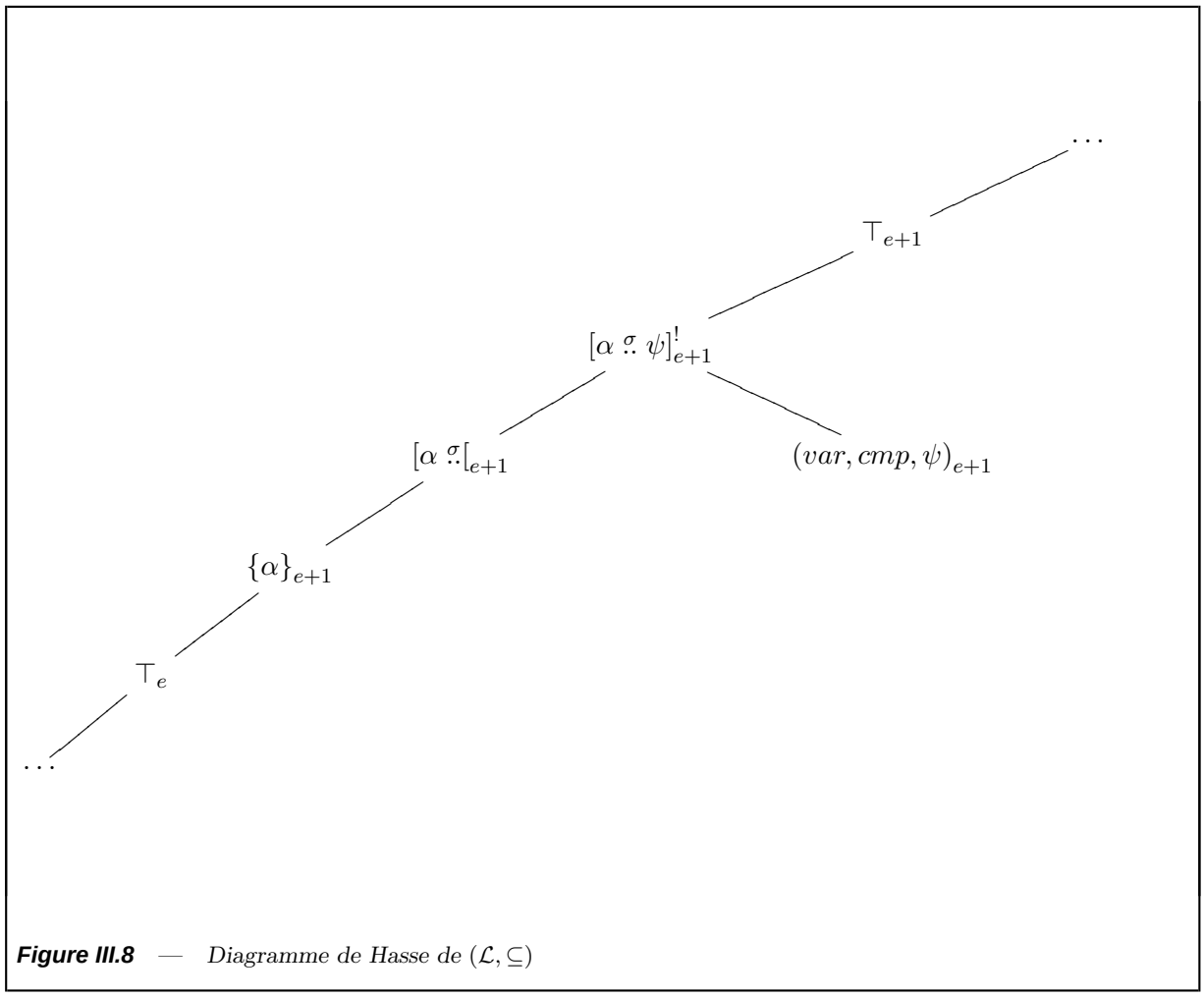


Figure III.8 — *Diagramme de Hasse de* $(\mathcal{L}, \subseteq)$

On modélise l'état du système par un tuple $\langle pp, T\rangle$, où pp est le compteur d'instructions, T est l'état courant des variables du programme qui met en correspondance un ensemble de variables avec un ensemble de valeurs. L'exécution de chaque instruction, sauf *return*, change l'état du programme P de $\langle pp, T\rangle$ à un état $\langle pp', T'\rangle$. Les règles de sémantique opérationnelle décrivent formellement le changement d'état dû à l'évaluation des instructions et leur impact sur le système :

Règle 1 : L'instruction évaluée est une affectation à une valeur constante. Le type de v change et devient $\{\alpha\}_e$ avec e le numéro du bloc de base courant. La prochaine instruction à être évaluée correspond à $pp + 1$.

$$\frac{\begin{array}{c} P[pp] = v \leftarrow \alpha \\ \alpha \in Cst \\ (pp+1) \in Dom(P) \end{array}}{\langle pp, T\rangle \mapsto \langle pp + 1, T'\rangle \ \text{où}\ T'[v] = \{\alpha\}_e}$$

Règle 2 : L'instruction est le résultat de l'évaluation d'une comparaison entre une variable v et une constante ψ. Le type de la variable b devient $(var, cmp, \psi)_e$ avec e le numéro du bloc de base courant. La prochaine instruction à être évaluée correspond à $pp + 1$.

$$\frac{\begin{array}{c}P[pp] = b \leftarrow v\ cmp\ \sigma \\ cmp \in \{<, \leq, >, \geq\},\ \sigma \in Cst \\ (pp+1) \in Dom(P)\end{array}}{\langle pp, T\rangle \mapsto \langle pp+1, T'\rangle\ \ où\ T'[b] = (var, cmp, \psi)_e}$$

Règle 3 : L'instruction évaluée correspond à l'incrémentation de la variable v par un pas constant σ. Si la variable v a déjà été initialisée(*i.e.* son type étant $\{\alpha\}_e$), le nouveau type qu'elle prend est $[\alpha \stackrel{\sigma}{..}[_{e'}$ avec e' le numéro du bloc de base courant. La prochaine instruction à être évaluée correspond à $pp+1$.

$$\frac{\begin{array}{c}P[pp] = v \leftarrow v + \sigma \\ \sigma \in Cst \\ T[v] = \{\alpha\}_e \\ (pp+1) \in Dom(P) \\ \chi(pp) = e'; e' \geq e\end{array}}{\langle pp, T\rangle \mapsto \langle pp+1, T'\rangle\ \ où\ T'[v] = [\alpha \stackrel{\sigma}{..}[_{e'}}$$

Règle 4 : Si la variable v a déjà été initialisée, incrémentée et comparée à une borne supérieure (*i.e.* son type courant est $[\alpha \stackrel{\sigma}{..} \psi]^{?}_{e}$), la ré-évaluation de l'instruction d'incrémentation garantit que la variable v est un itérateur confirmé dans ce chemin d'exécution et son type devient $[\alpha \stackrel{\sigma}{..} \psi]^{!}_{e'}$. e' correspond au numéro du bloc de base courant. La prochaine instruction à être évaluée correspond à $pp+1$.

$$\frac{\begin{array}{c}P[pp] = v \leftarrow v + \sigma \\ \sigma \in Cst \\ T[v] = [\alpha \stackrel{\sigma}{..} \psi]^{?}_{e} \\ (pp+1) \in Dom(P) \\ \chi(pp) = e'; e' \geq e\end{array}}{\langle pp, T\rangle \mapsto \langle pp+1, T'\rangle\ \ où\ T'[v] = [\alpha \stackrel{\sigma}{..} \psi]^{!}_{e'}}$$

Règle 5 : Dans le cas d'une boucle du type post-test, la variable b contient l'évaluation de la comparaison entre la variable v et une valeur constante par un opérateur de comparaison $\{<, \leq\}$. La variable v est un intervalle ouvert $[\alpha \stackrel{\sigma}{..}[_{e'}$. L'évaluation du branchement conditionnel implique un changement à deux points du programme. Le premier correspond à un saut au label $LabelId$. Si la variable v est initialisée, incrémentée et comparée à une borne supérieure, alors son type devient $[\alpha \stackrel{\sigma}{..} \psi]^{?}_{e''}$ avec e'' le numéro du bloc de base portant le label $LabelId$. Le deuxième point de saut correspond au branchement implicite à l'instruction suivante $pp+1$. Cela a pour effet de propager des informations inutiles pour la détection d'itérateurs et le type de toutes les variables devient $\top$ à ce point du programme.

$$\frac{\begin{array}{c}P[pp] = Jumpif\ b,\ LabelId \\ T[b] = (var, cmp, \psi)_e,\ \text{cmp} \in \{<, \leq\} \\ T[v] = [\alpha \stackrel{\sigma}{..}[_{e'} \\ (pp+1),\ LabelId \in Dom(P) \\ \chi(LabelId) = e''; e'' \geq e' \geq e\end{array}}{\begin{array}{l}\langle pp, T\rangle \mapsto \langle pp+1, T'\rangle\ \ où\ \forall\ v \in Var,\ T'[v] = \top \\ \langle LabelId, T'\rangle\ \ où\ T'[v] = [\alpha \stackrel{\sigma}{..} \psi]^{?}_{e''}\end{array}}$$

Règle 5' : Dans le cas d'une boucle du type pre-test, la variable b contient l'évaluation de la comparaison entre la variable v et une valeur constante par un opérateur de comparaison $\{<, \leq\}$. La variable v est un intervalle ouvert $[\alpha \; \sigma[_{e'}$. L'évaluation du branchement conditionnel implique un changement à deux points du programme. Le premier correspond à un saut au label $LabelId$. La variable v dépasse la borne supérieure et le type de toutes les variables devient $\top$ à ce point du programme. Au deuxième point de saut, le type de la variable devient $[\alpha \; \sigma \; \psi]^{?}_{e''}$ avec e'' le numéro du bloc de base contenant l'instruction $pp + 1$.

$$\frac{\begin{array}{c} P[pp] = Jumpif\ b,\ LabelId \\ T[b] = (var, cmp, \psi)_e,\ \text{cmp} \in \{>, \geq\} \\ T[v] = [\alpha \; \sigma[_{e'} \\ (pp{+}1),\ LabelId \in Dom(P) \\ \chi(pp+1) = e''; e'' \geq e' \geq e \end{array}}{\begin{array}{c} \langle pp, T\rangle \mapsto \langle LabelId, T'\rangle \ \text{où}\ \forall\ v \in\ Var,\ T'[v] = \top \\ \langle pp+1, T'\rangle \ \text{où}\ T'[v] = [\alpha \; \sigma \; \psi]^{?}_{e''} \end{array}}$$

Règle 6 : Cette règle correspond à l'évaluation d'une invocation de méthode quelconque. Cette évaluation produit des informations impertinente pour la détection d'itérateurs, donc le type de toutes les variables destination devient $\top$ à ce point du programme.

$$\frac{P[pp] = vd \leftarrow\ vs\ op\ tabVar}{\langle pp, T\rangle \mapsto \langle pp+1, T'\rangle \ \text{où}\ T'[vd] = \top}$$

III.3.3.2 Règles d'unification

III.3.4 Description à travers un exemple

III.3.4.1 Description du traitement du coté du producteur de code

Nous allons nous restreindre dans ce qui suit à déterminer le nombre d'itération de chaque boucle naturelle (*i.e.* déterminer les poids des arcs de retour). Concrètement, l'objectif de l'algorithme est d'interpréter abstraitement le programme de façon à en connaître le pire chemin d'exécution.

La table III.1 donne l'exemple d'une boucle vide qui s'exécute dix fois avant de se terminer. Dans ce programme, il y a deux points de saut correspondant aux labels L1 et L2. La valeur initiale du compteur est affectée à la variable i. L1 permet de vérifier une première fois la condition de sortie avant de boucler sur L2 en comparant la variable temporaire t à la valeur maximale du compteur de la boucle.

La recherche d'un point fixe nous amène à garder trace des différents états abstraits que peuvent prendre les variables du programme sur tous les chemins d'exécution possibles. Sur le chemin d'exécution BB0 -> BB1 -> BB2 -> BB1 -> BB3 par exemple, l'analyseur commence par initialiser les deux variables i et t à un état $\top$. À ce stade, aucune information sur les deux variables ne peut être déduite. À la fin du bloc BB0, la variable i est initialisée par une constante. L'algorithme garde trace de l'état abstrait *Cst(0)*. Quand BB1 se termine, on détecte que la variable t est interprétée comme une condition sur la variable i. L'algorithme doit garder

a \ b	$\top$	$\{\alpha\}_b$	$[\alpha \stackrel{\sigma}{\cdot\cdot} [_b$	$[\alpha \stackrel{\sigma}{\cdot\cdot} \psi]^?_b$	$[\alpha \stackrel{\sigma}{\cdot\cdot} \psi]^!_b$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
$\{\beta\}_a$	$\top$	$\{\min(\alpha,\beta)\}_b$	$[\min(\alpha,\beta) \stackrel{\sigma}{\cdot\cdot} [_b$	$[\alpha \stackrel{\sigma}{\cdot\cdot} \psi]^?_b$	$[\min(\alpha,\beta) \stackrel{\sigma}{\cdot\cdot} \psi]^!_b$
$\left[\beta \stackrel{\phi}{\cdot\cdot} \right[_a$	$\top$	$\{\alpha\}_b$	$\left[\min(\alpha,\beta) \stackrel{\min(\sigma,\phi)}{\cdot\cdot} \right[_b$	$\left[\min(\alpha,\beta) \stackrel{\min(\sigma,\phi)}{\cdot\cdot} \psi\right]^?_b$	$\left[\min(\alpha,\beta) \stackrel{\min(\sigma,\phi)}{\cdot\cdot} \psi\right]^!_b$
$\left[\beta \stackrel{\phi}{\cdot\cdot} \gamma\right]^?_a$	$\top$	$\{\alpha\}_b$	$\top$	$\left[\min(\alpha,\beta) \stackrel{\min(\sigma,\phi)}{\cdot\cdot} Max(\psi,\gamma)\right]^?_b$	$\left[\min(\alpha,\beta) \stackrel{\min(\sigma,\phi)}{\cdot\cdot} Max(\psi,\gamma)\right]^?_b$
$\left[\beta \stackrel{\phi}{\cdot\cdot} \gamma\right]^!_a$	$\top$	$\top$	$\top$	$\top$	$\left[\min(\alpha,\beta) \stackrel{\min(\sigma,\phi)}{\cdot\cdot} Max(\psi,\gamma)\right]^!_b$

Table III.1 — *Code intermédiaire d'une boucle avec pré-test*

Bloc de base	Label	Instruction	Commentaires
BB0		i := 0	Initialisation du compteur de boucle
BB1	L1 :	t := i>9	Condition de sortie de la boucle
		jumpif t, L2	Si vrai brancher au label L2 (Fin du programme)
BB2		i := i + 1	Incrémentation de i
		jump L1	Branchement au test de sortie
BB3	L2 :	return t	Fin du programme

trace de l'opérateur de comparaison ainsi que la borne supérieure de la boucle. Dans BB2, la variable est incrémentée de un. La variable i appartient donc à l'intervalle ouvert $[0; \infty[$ avec un pas de un. À ce point, quand l'analyseur rencontre l'instruction de branchement conditionnel à la fin de BB1, l'algorithme joint les deux informations récoltées jusqu'à présent pour inférer que la variable i appartient à l'intervalle $[0; 9]$ avec un pas de un. À ce stade, on peut définir la boucle $<(BB2- > BB1$, i, 0, 1, >, 9>.

Table III.2 — *Trace d'exécution sur le chemin BB0 -> BB1 -> BB2 -> BB1 -> BB3*

	T(i,t)
init.	$(\top, \top)$
BB0	$(Cst(0), \top)$
BB1	$(Cst(0), cond(>, i, 9))$
BB2	$([0; \infty[_1, cond(>, i, 9))$
BB1	$([0; 9]_{1,>}, cond(>, i, 9))$
BB3	$([0; 9]_{1,>}, cond(>, i, 9))$

III.3.4.2 Description du traitement du coté du consommateur de code

Le producteur fournit un binaire contenant le code intermédiaire que la partie frontale du compilateur a généré ainsi que l'arbre annoté. A partir de ces éléments, le consommateur devrait être capable de vérifier les variables d'itérations annoncées par l'extérieur. Lorsque cette étape de vérification est validée, le code peut être transformé en code natif et les performances du système sont désormais connues (***i.e*** *nombre de cycles du processeur consommés par instruction*). Les WCET des blocs de base peuvent alors être calculés et un algorithme de recherche du chemin le plus coûteux sur l'arbre annoté permettra de déterminer le WCET global.

Vérification des itérateurs Les points fixes représentant les itérateurs des boucles sont déterminés à la compilation sur le producteur de code. L'interprétation abstraite étant une technique lourde gourmande en ressources processeur et mémoire, son implémentation sur le producteur est moins contraignante. Le consommateur n'a plus qu'à valider les points fixes annoncés par une vérification linéaire. La complexité algorithmique de ce dernier est en $O(n)$ avec n nombre de

lignes du programme, alors que l'interpréteur abstrait sur le producteur induit une complexité algorithmique qui croît non linéairement avec le nombre de chemins d'exécutions possibles.

Supposons que le producteur annonce que la boucle L est définie par $<(BB2- > BB1$, i, 0, 1, >, 9>. Au fur et à mesure que la partie arrière de compilation qui réside sur le consommateur reçoit les instructions intermédiaires, le vérifieur doit valider que la variable i a bien été initialisée à 0, comparée à 9 puis incrémentée comme le montre le Tableau III.3.

Table III.3 — *Vérification des itérateurs.*

Bloc de base	Label	Instruction	Vérification
BB0		i := 0	$<?,?,?,?,?,? >$
BB1	L1 :	t := i>9	$<?,?,0,?,?,? >$
		jumpif t, L2	$<?,?,0,?,>,9 >$
BB2		i := i + 1	$< BB2- > BB1,?,0,?,>,9 >$
		jump L1	$< BB2- > BB1,?,0,1,>,9 >$
BB3	L2 :	Return	$< BB2- > BB1,i,0,1,>,9 >$

III.3.5 Traiter le cas des boucles annotées

Dans le cas général, les bornes des boucles ne peuvent pas être déterminées automatiquement. Pour les boucles dont le nombre d'itérations dépend des entrées du programme (***e.g*** *saisie clavier, relevé capteurs ...*), par exemple, il est nécessaire que les bornes soient fournies au préalable afin de pouvoir calculer le pire temps d'exécution du programme. Pour un grand nombre de structures algorithmiques dont la terminaison est indécidable, il est aussi critique pour le calcul du WCETde spécifier a priori des conditions d'arrêt.

Il est difficile avec un algorithme tel que celui décrit précédemment de reconnaître une structure cyclique avec plusieurs arcs de retour comme une seule boucle. Quand les boucles ont le même entête comme illustré dans la figure III.10, l'algorithme cité ci-dessus détectera deux variables d'itérations distinctes et déterminera le poids des arcs de retour $BB2- > BB1$ et $BB3- > BB1$. Dans ce cas, les deux boucles ne peuvent pas être considérées comme imbriquées et le calcul des bornes des boucles sera incohérent.

Pour traiter le cas des boucles dont on ne saurait pas déterminer les bornes automatiquement, ces dernières devraient être données sous forme d'annotations constantes [PK89b, CBW96] en commentaires au niveau du code source [Col01], dans un fichier attaché [HBW02b] ou demandées interactivement au programmeur à chaque fois qu'une boucle est détectée. L'outil Heptane [Col01] nécessite, par exemple, que le code source soit annoté pour fournir le nombre maximum d'itérations pour chaque boucle et introduit une nouvelle approche d'annotations symboliques adaptées aux boucles non rectangulaires.

Dans le contexte de code mobile, les annotations du programmeur peuvent cependant introduire des erreurs et entraîner un mauvais fonctionnement de l'hôte. Pour se protéger contre un éventuel dysfonctionnement du consommateur, nous proposons que les annotations introduites par le programmeur se traduisent par une injection de code de contrôle au niveau du code intermédiaire. Par exemple, si une annotation déclare qu'une boucle s'exécutera 16 fois, le

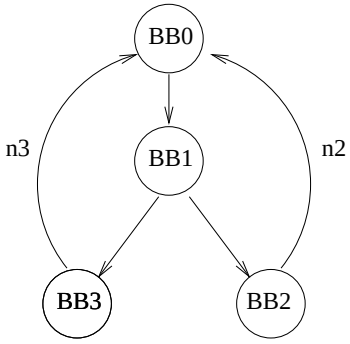


Figure III.9 — *Boucles avec la même entête.*

compilateur doit produire un code qui permettrait de sortir effectivement de la boucle après 16 itérations. Par le biais de l'injection de ce code de contrôle, le consommateur se prémunit contre la sous-réservation de ressource qui pourrait induire une attaque de type déni de service. Dans le cas de sur-réservation, l'ordonnanceur du système cible respectera une politique de quotas pour passer la main à un autre processus. L'injection de code de contrôle a néanmoins un contre-coût parce qu'elle engendre des opérations supplémentaires qui pourrait dégrader les performances du système. Ainsi, Le WCET calculé sera d'autant plus pessimiste. Une autre approche décrite dans [CBW96] utilise les techniques de preuves formelles pour vérifier les annotations fournies par l'utilisateur.

III.4 Modélisation du flot de contrôle

Par pondérer le graphe de flot de contrôle, on entend déterminer les poids des différents arcs qui le constituent, comme illustré par la figure III.9. Un arc orienté allant du bloc de base BB1 à BB2 ayant un poids n signifie que l'arc orienté BB1 vers BB2 sera emprunté au maximum n fois lors de l'exécution du programme. Pondérer le graphe de flot de contrôle est nécessaire à l'algorithme de calcul de WCET qui doit savoir le nombre maximum de fois qu'un bloc de base s'exécute pour pouvoir donner l'estimation globale du WCET.

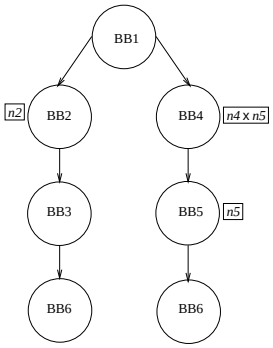


Figure III.10 — *Arbre de flot de contrôle décoré*

Pour ce faire, on doit être capable d'extraire automatiquement cette information à partir du langage intermédiaire, plus facile à analyser. Nous avons choisi de baser notre approche d'analyse sur l'interprétation abstraite[CC77]. L'idée principale de cette technique est d'inférer des informations provenant des programmes en les interprétant par le biais de valeurs abstraites au lieu de valeurs concrètes. Cette interprétation permettra d'obtenir des approximations fiables du comportement du programme et plus précisément dans notre cas une estimation des poids des arcs du graphe de flot de contrôle.

On appelle *arbre de flot de contrôle décoré* un graphe de flot de contrôle dont on a retiré les cycles. La valeur de la fréquence d'exécution des blocs de base étiquette sur ces noeuds

Une fois les boucles identifiées, les poids reportés sur les arcs orientés du graphe de flot de contrôle nous servent à déterminer le nombre d'occurrences de l'exécution de chaque bloc de base. Le poids de l'arc de retour allant de BB2 à BB1 est propagé sur les noeuds BB2 et BB1 ainsi que sur tous les noeuds qui se trouvent sur un chemin menant de BB2 jusqu'à BB1.

Dans le cas de boucles imbriquées comme dans la figure III.11, la boucle correspondant à l'arc de retour $BB4- > BB5$ englobe la boucle correspondant à l'arc de retour $BB4- > BB4$. Dans ce cas, le nombre d'exécutions de BB4 est le résultat de la multiplication des poids des arcs de retour de la boucle englobée et englobante.

Une fois le nombre d'exécutions de chaque bloc de base calculé, tous les arcs de retour sont supprimés. On construit un arbre à partir du graphe obtenu en ajoutant un nouvelle branche d'exécution à chaque fois qu'une alternative est rencontrée. Cette transformation permet une recherche du plus long chemin plus rapide.

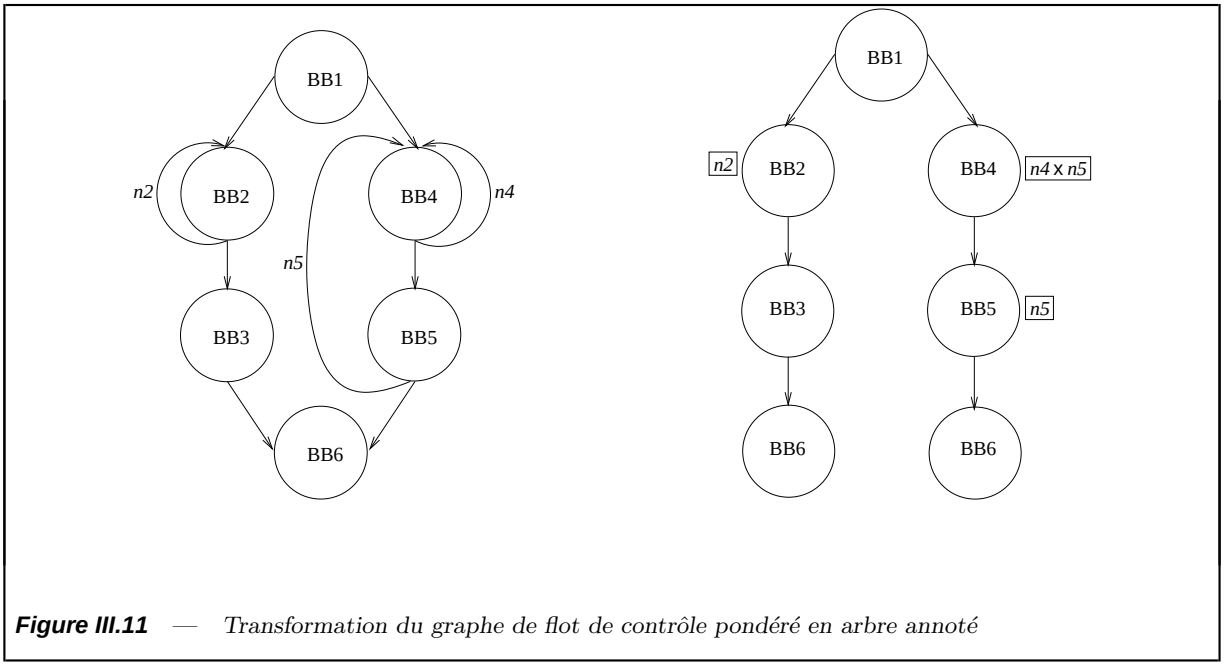


Figure III.11 — *Transformation du graphe de flot de contrôle pondéré en arbre annoté*

Encodage de la preuve des boucles La preuve des boucles se présente comme des annotations du programme transmis au consommateur. Ces annotations sont elle-mêmes transmises, en même temps que le programme associé, dans un format bien particulier. En effet, le but principal à ce niveau est de minimiser leur taille. D'une part, cela réduit le temps de transmission, de l'autre cela minimise le temps de lecture de ces indications. Par exemple, la preuve de typage

dans CAMILLE représente 30 % de la taille totale des données transmises au consommateur. Il est important de noter que dans les applications actuelles de l'approche PCC originelle, la taille de la preuve atteint trois fois la taille du code applicatif. La différence est due aux efforts de compression et d'encodage réalisés.

III.5 Vérification

Une fois qu'il a reçu l'application et la preuve de typage, le consommateur vérifie cette preuve de typage selon un mécanisme lié à l'installation du programme.

III.5.1 Mécanisme de vérification

L'outil de détection de bornes des boucles — du côté producteur — effectue un processus itératif d'inférence de types. Le résultat de cette analyse est la preuve fournie avec le code mobile. Le consommateur doit ensuite vérifier que les codes mobiles qu'il reçoit satisfont les deux critères suivants :

(i) Tout bloc de base doit avoir une estampille de précédence plus petite que tous ses successeurs. Dans le cas contraire, cela signifie que la vérification linéaire est impossible.

(ii) Les bornes des boucles doivent être dans l'intervalle de valeurs annoncé par le producteur. Dans le cas où cette propriété n'est pas vérifiée, le consommateur se trouve vraisemblablement en présence d'une fausse preuve, provenant potentiellement d'un producteur mal intentionné.

(iii) Tout programme dont le WCET reste non décidable statiquement ne doit pas être accepté, même si le programme et la preuve ont été altérés.

La vérification qu'on veut établir a les caractéristiques suivantes :

(i) il est crucial que la vérification consomme le moins de ressources possibles. On souhaite en fait qu'elle soit faisable en une seule passe linéaire sur le code ;

(ii) à la volée en même temps que le code est chargé. De cette façon, on ne stocke pas la forme intermédiaire du code reçu ;

(iii) en conséquence, la vérification est basée sur la connaissance de l'état passé mais pas de l'état à venir.

La vérification de ces propriétés est effectuée de la manière suivante :

(i) à l'entrée de chaque bloc de base, les types calculés des variables locales sont initialisés aux types donnés par la preuve ;

(ii) faire, en une seule passe, une interprétation abstraite linéaire sur la séquence d'instructions du bloc, en appliquant les règles d'inférence expliquées précédemment aux types calculés ;

(iii) à la fin de chaque bloc de base, vérifier que les types calculés sont cohérents, au vu du treillis de type, avec les types inférés au début de chacun de ses successeurs. Si les annotations ne vérifient pas cette propriété, le code est rejeté.

III.5.2 Exemples

Exemple 1 La preuve d'annotation de type jointe au code reçu par le consommateur est représentée ci-dessous :

Label	i	tmp
L1	$[0 \stackrel{1}{..} 9]^{!}_{2}$	$\top$

La table III.4 montre les étapes du mécanisme de vérification de l'exemple 1. À l'étape 1, en fonction des règles de sémantique opérationnelle détaillées précédemment, les types calculés des variables i et tmp correspondent à $\{0\}_1, \top$. À l'étape 2, l'instruction en cours d'évaluation est la cible d'un branchement. L'état courant indiqué par $(\{0\}_1, \top)$ doit correspondre aux annotations fournies par le producteur$([0 \stackrel{1}{..} 9]^{!}_{2}, \top)$. Informellement, on doit vérifier que l'étape d'initialisation de la boucle doit avoir eu lieu avant la création du type itérateur. Étant donné que l'estampille de l'état courant est plus petite que celle du type donné dans les annotations, la conformité de la preuve est vérifiée et l'état courant devient $([0 \stackrel{1}{..} 9]^{!}_{2}, \top)$. Pour les instructions suivantes, les types sont calculés en utilisant les mêmes règles d'inférence que celles utilisées sur le producteur.

Table III.4 — *Processus de vérification de l'exemple 1 sur le consommateur*

Étape	pp	Instruction	$T_a[i, tmp]$	$T_d[i, tmp]$
1	1	i $\leftarrow$ 0	$\emptyset$	$\{0\}_1, \top$
2	2	i $\leftarrow$ i + 1	$[0 \stackrel{1}{..} 9]^{!}_{2}, \top$	$[0 \stackrel{1}{..} 9]^{!}_{2}, \top$
3	3	tmp $\leftarrow$ i $\leq$ 9	$\emptyset$	$[0 \stackrel{1}{..} 9]^{!}_{2}, \top$
4	4	jumpif tmp L1	$\emptyset$	$[0 \stackrel{1}{..} 9]^{!}_{2}, (i, >, 9)_2$
5	5	return i	$\emptyset$	$[0 \stackrel{1}{..} 9]^{!}_{2}, (i, >, 9)_2$

Exemple 2 La preuve reçue par le consommateur en même temps que l'application est la suivante :

Label	i	tmp
L1	$[0 \stackrel{1}{..} 9]^{!}_{4}$	$\top$
L2	$\top$	$\top$

La table III.5 montre les étapes du processus de vérification pour l'exemple 2. Le fonctionnement est sensiblement le même que pour l'exemple précédent, mais illustre un cas où le test de sortie est effectué après le corps de la boucle. Ainsi, à l'étape 1, les variables i et tmp ont les types $\{0\}_1, \top$, et à l'étape 2, l'instruction évaluée est la comparaison correspondant au point de saut.

Exemple 3

Label	i	tmp
L1	$[0 \stackrel{1}{..} 9]^{!}_{2}$	$\top$
L2	$\top$	$\top$

Table III.5 — *Processus de vérification de l'exemple 2 sur le consommateur*

Étape	**pp**	**Instruction**	$T_a[i, tmp]$	$T_d[i, tmp]$
1	1	i ← 0	$\emptyset$	$\{0\}_1, \top$
2	2	tmp ← i > 9	$\left[0 \doteq 9\right]^!_4, \top$	$\left[0 \doteq 9\right]^!_4, \top$
3	3	jumpif tmp L2	$\emptyset$	$\left[0 \doteq 9\right]^!_4, \top$
4	4	i ← i + 1	$\emptyset$	$\left[0 \doteq 9\right]^!_4, \top$
5	5	jump L1	$\emptyset$	$\left[0 \doteq 9\right]^!_4, \top$
6	6	return i	$\emptyset$	$\top, \top$

Concernant ce troisième exemple, notre outil de détection de boucles ne détecte pas d'itérateur. En effet, la section de code ne correspond pas aux modèles de boucle reconnus par notre analyseur. Le programme est donc rejeté. La table III.6 illustre cette inconsistance. L'étape d'initialisation ne change pas, et à l'étape 2, l'instruction évaluée correspond à la cible du branchement. L'outil de vérification s'assure que l'état courant indiqué par $(\{0\}_1, \top)$ est cohérent avec les annotations faites sur le producteur : $(\left[0 \doteq 9\right]^!_2, \top)$. Ensuite, la vérification est faite que l'étape d'initialisation est effectuée avant la création d'un type itérateur. Étant donné que l'ordre de précédence de l'état courant est plus petit que le type annoté, la conformité de la preuve est vérifiée et l'état courant est mis à $(\left[0 \doteq 9\right]^!_2, \top)$. Puis, à l'étape 4, la valeur constante 4 est assignée à i. Cette affectation a lieu dans un bloc de base que l'on trouve *après* la création de l'itérateur. Le vérifieur rejette donc la preuve et le code associé à celle-ci, car preuve et code mobile ne correspondent pas.

Table III.6 — *Processus de vérification de l'exemple 3 sur le consommateur*

Étape	**pp**	**Instruction**	$T_a[i, tmp]$	$T_d[i, tmp]$
1	1	i ← 0	$\emptyset$	$\{0\}_1, \top$
2	2	i ← i + 1	$\left[0 \doteq 9\right]^!_2, \top$	$\left[0 \doteq 9\right]^!_2, \top$
3	3	jumpif tmp L2	$\emptyset$	$\left[0 \doteq 9\right]^!_2, \top$
4	4	i ← 4	$\emptyset$	Preuve Inconsistente
5	5	tmp ← i ≤ 9	$\emptyset$	Preuve Inconsistente
6	6	jumpif tmp L1	$\emptyset$	Preuve Inconsistente
6	6	return i	$\emptyset$	$\top, \top$

III.6 Conclusion

Nous avons détaillé dans ce chapitre un mécanisme de détection/vérification de bornes des boucles. L'originalité de celui-ci réside dans le fait qu'il est distribué entre producteurs et consommateurs de code. De cette façon, les propositions faites permettent d'assurer le respect des contraintes de sécurité de code mobile tout en restant en adéquation avec les contraintes

matérielles des consommateurs. En effet, ces derniers disposent de ressources (puissance de calcul, mémoire, énergie) limitées. La solution proposée repose sur deux principes. Premièrement, elle utilise une infrastructure de type PCC, laquelle permet de prouver et vérifier une propriété donnée. En l'occurrence, cette propriété est que l'on peut trouver une borne supérieure à toute boucle du programme analysé. Le processus de vérification se sert des annotations de typage — transmise avec l'application — pour s'assurer que les boucles sont bornables. En second lieu, la définition d'un treillis de type adéquat permet d'effectuer cette vérification linéairement. Nous avons publié ces résultats dans [BGSR06], mais aussi partiellement dans [BGB07].

IV

Application du calcul distribué de temps d'exécution au pire cas

> *Les grandes personnes aiment les chiffres. Quand vous leur parlez d'un nouvel ami, elles ne vous questionnent jamais sur l'essentiel. Elles ne vous disent jamais : Quel est le son de sa voix ? Quels sont les jeux qu'il préfère ? Est-ce qu'il collectionne les papillons ? Elles vous demandent : Quel âge a-t-il ? Combien a-t-il de frères ? Combien pèse-t-il ? Combien gagne son père ? Alors seulement elles croient le connaître.*
>
> — Le Petit Prince - Antoine de Saint-Exupéry

IV.1 Introduction

Le calcul de WCET est réputé gourmand en ressources matérielles (consommation mémoire, CPU), on veut prouver dans ce chapitre qu'en utilisant l'approche distribuée que nous avons décrit précédemment le calcul de WCET est possible sur une petite cible telle qu'une carte à puce.

L'architecture matérielle cible est composée d'un processeur ARM7 et de mémoires diverses. Le système d'exploitation, CAMILLE, est un exo-noyau adapté pour les cibles fortement contraintes en ressources matérielles. Il virtualise les ressources comme le processeur, la mémoire, l'interface réseau pour les partager entre les différentes extensions à travers une interface bas-niveau. Les applications choisies sont des applications courantes du domaine de la carte à puce.

La Figure IV.1 représente l'architecture sur laquelle nous allons baser nos expérimentations.

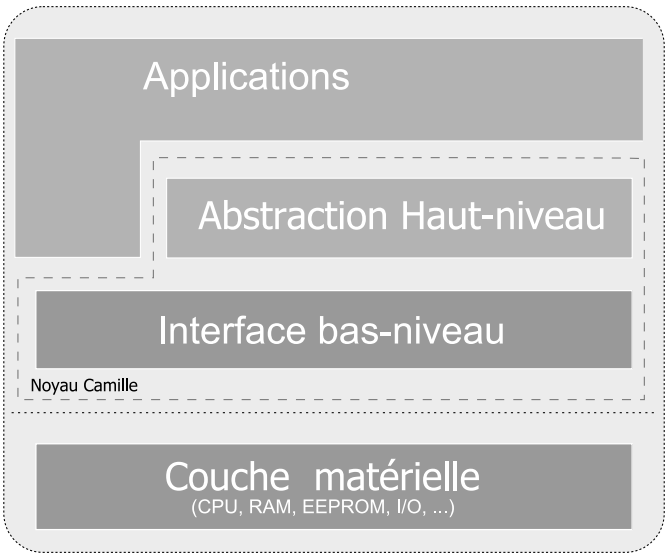


Figure IV.1 — *Schéma en couches ou guide de lecture du chapitre.*

IV.2 Configuration matérielle

De nos jours, la complexité des applications embarquées ne cesse de croître (*e.g.* les offres de contenu multimédia en haute définition sur téléphone mobile) ainsi que leurs besoins en terme de performance. De ce fait, la tendance est, actuellement, à remplacer les processeurs 8 bits par ceux à 32 bits.

IV.2.1 Le processeur ARM7TDMI

L'ARM7TDMI est le processeur RISC le plus simple appartenant à la famille de processeurs ARM7 commercialisés par *Advanced Risc Machines Ltd.*[ARM04]. Il existe quatre variantes du processeur ARM7 dont un seul intègre un cache unifié de 8 KO et une unité de gestion mémoire (MMU). Développé en 1995, l'ARM7TDMI reste toutefois l'un des processeurs les plus utilisés de nos jours dans le domaine de l'embarqué. Il tire, en effet, un grand parti de sa taille réduite, de sa faible consommation d'énergie et d'un code de densité élevée. Ces caractéristiques techniques sont représentées dans la table IV.1.

Table IV.1 — *Caractéristiques techniques de l'*ARM7TDMI

TECHNOLOGIE CMOS	$0.35\mu m$	**TRANSISTORS**	74K	**MIPS**	60
COUCHES MÉTALLIQUES	3	**SURFACE**	2.1 mm^2	**CPI**	1.9
VDD	3.3 V	**HORLOGE**	0-66 MHz	**MIPS/W**	690

Les instructions ARM7 ont une longueur fixe égale à 32 bits. Cependant, l'ARM7TDMI supporte un jeu d'instructions supplémentaire, appelé THUMB, où les instructions sont de longueur 16 bits.

L'architecture ARM7TDMI offre un mécanisme général permettant d'étendre les ressources du processeur par l'adjonction de un ou plusieurs co-processeurs. Par exemple, l'ARM7TDMI peut intégrer une unité arithmétique flottante, une unité de gestion de mémoire, ou une mémoire cache comme co-processeurs.

Le codage des instructions sur un seul mot mémoire permet d'assurer un flux régulier des instructions vers le processeur.

Le jeu d'instruction de l'ARM7TDMI peut être décomposé selon leurs fonctionnalités en cinq familles d'instructions :

(i) Les instructions de traitement de données servant à effectuer des opérations logiques ou arithmétiques sur deux opérandes (dont l'une est obligatoirement un registre) ;

(ii) Les instructions de chargement/sauvegarde impliquant un seul registre ;

(iii) Les instructions de chargement/sauvegarde impliquant un ensemble arbitraire de registres ;

(iv) Les instructions de branchement permettant d'interrompre le flot séquentiel des instructions d'un programme ;

(v) Les instructions de multiplication.

Pipeline à trois étages Un pipeline se compose d'un certain nombre d'étages (*i.e. profondeur*). A un instant donné, plusieurs instructions occupent des étages différents. Chaque instruction progresse dans le pipeline à chaque cycle du processeur. La *latence* d'une instruction est le nombre de cycles processeur nécessaires pour traverser tous les étages du pipeline. Idéalement, la latence est égale à la profondeur. Cependant, quelques aléas rompent le pipeline des instructions et introduisent des cycles de latence dans le temps global d'exécution d'un programme.

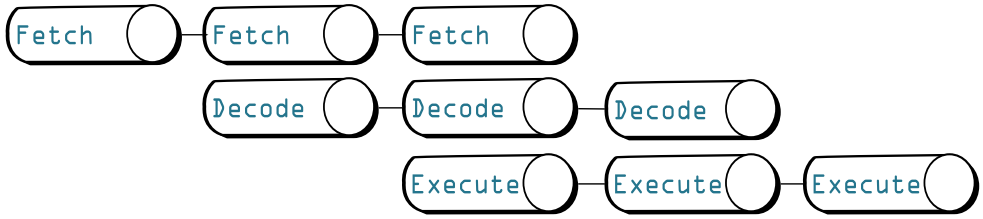


Figure IV.2 — *Exemple de pipeline à trois étages*

L'ARM7TDMI utilise un pipeline à trois étages tel qu'illustré par la figure IV.3. Le premier étage du pipeline lit l'instruction dont l'adresse se trouve dans le registre PC à partir de la mémoire. La valeur contenue dans le registre d'adresse est incrémentée pour pointer vers l'adresse de la prochaine instruction à rechercher. L'étage suivant décode l'instruction préchargée et prépare les signaux de contrôle nécessaires pour son exécution. Le troisième étage fait le plus gros du travail, il lit les opérandes dans les registres, effectue les opérations arithmétiques et logiques nécessaires, lit ou écrit dans la mémoire pour écrire finalement dans les registres destination les valeurs modifiées.

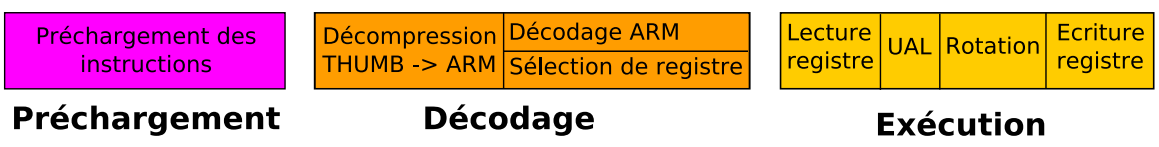


Figure IV.3 — *Exécution pipelinée sur L'ARM7TDMI : Dans le premier étage du pipeline, l'instruction est préchargée à partir de la mémoire. Puis, dans le deuxième étage, elle est décompressée si elle est au format THUMB. L'instruction ARM obtenue est décodée et le registre de travail sélectionné. Enfin, dans la dernière étape, la banque de registres est lue, les opérations de rotation sur les opérandes sont effectuées, le résultat des opérations arithmétiques et logiques généré est stocké dans le registre destination.*

La plupart des instructions ARM sont exécutées en un seul cycle. Typiquement, à un instant donné, trois instructions cohabitent dans le bloc de traitement de données. Quelques instructions consomment, néanmoins, plusieurs cycles perdant ainsi l'avantage de l'exécution pipelinée. Le nombre de cycles moyen mis par le processeur ARM pour exécuter une instruction dépend de la latence des cycles d'accès au bus, de la longueur des instructions (ARM ou THUMB), de la quantité de donnée échangée et de l'espace d'adressage accédé. Nous renvoyons le lecteur à la Section IV.4 pour plus de détails concernant le calcul de l'utilisation du processeur en terme de cycles.

Accès à la mémoire L'ARM7TDMI intègre une architecture mémoire de type *Von Neuman* permettant d'utiliser le même bus pour véhiculer aussi bien les instructions que les données. De ce fait, la bande passante de la mémoire est moindre mais en contre-partie cela permet de réduire la surface occupée sur le silicone ainsi que son coût de production.

En outre, l'ARM7TDMI est une architecture du type chargement/sauvegarde[1]. Les instructions ne portent donc que sur des opérandes situées dans des registres. Les accès à la mémoire

1. **Load/Store** est le terme anglo-saxon correspondant.

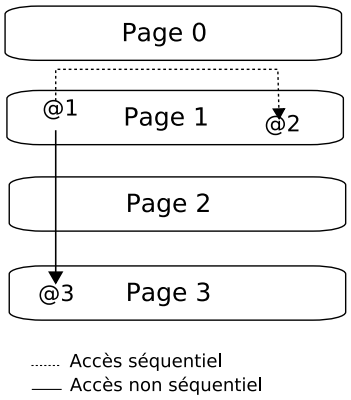


Figure IV.4 — *Accès à la mémoire.*

sont explicitement réalisés par des instructions dédiées qui permettent de transférer une valeur depuis la mémoire vers un registre ou inversement.

Les accès à la mémoire peuvent être séquentiels ou non séquentiels (voir Table IV.2). Un accès séquentiel `S` à la mémoire se produit quand le processeur demande un transfert à partir de (ou vers) la même adresse que celle utilisée dans le cycle précédent ou à une adresse qui est située un mot plus loin (voir Table IV.4). Un accès non séquentiel `N` consomme plus de cycles puisqu'il requiert le transfert à partir de (ou vers) une adresse qui n'est pas reliée à l'adresse utilisée dans le cycle précédent (voir Table IV.4). Un cycle interne `I` est consommé quand le processeur ne fait pas de transfert et exécute une fonction interne. Un cycle du coprocesseur `C` ne met en jeu que les registres de ce dernier, aucune action n'est demandée à la mémoire.

Table IV.2 -- *Caractéristiques des cycles du bus*

Type	Description
I	Aucune activité extérieure sur le bus mais indication qu'aucun chargement de code ne peut être entrepris.
N	Utilise une adresse sans lien avec l'adresse du cycle précédent.
S	Utilise une adresse qui est l'adresse suivante ou l'adresse précédente par rapport à l'adresse utilisée dans le cycle précédent.
C	Transfert de (ou vers) les registres d'un coprocesseur.

Prédiction de branchement L'ARM7TDMI utilise un mécanisme de prédiction de branchement statique élémentaire en considérant que le branchement est toujours non pris. En effet, le processeur continue à précharger les instructions séquentiellement après une instruction de branchement. Si effectivement le branchement n'est pas effectué, le pipeline n'est pas rompu et il n'y a dans ce cas aucune pénalité. Dans le cas contraire, l'avantage du pipeline est annulé puisque les instructions préchargées ne seront pas exécutées. Cette heuristique simple de prédiction de branchement n'engendre pas d'effets négatifs majeurs dans les pipelines de profondeur modérément

élevée. Ce choix a été, de ce fait, retenu sur l'ARM7TDMI qui ne comporte pas de mécanisme de prédiction de branchement plus complexe.

Multiplication L'ARM7TDMI permet d'effectuer des opérations arithmétiques de type multiplication accumulation. Cette architecture utilise un multiplieur combinatoire 8 bits de Booth [Boo51] qui permet d'accélérer efficacement la vitesse de la multiplication binaire. L'opération de multiplication s'exécute en plusieurs cycles en fonction de la valeur contenue dans le multiplicande. Le multiplieur partitionne le multiplicande en groupe de 8 bits. Si à une des étapes les bits restants du multiplicande sont à zéro dans le cas des nombres positifs ou à un dans le cas des nombres négatifs, la multiplication se termine.

IV.2.2 Les mémoires

L'architecture sur laquelle nous avons testé notre implémentation intègre en plus du coeur de processeur ARM7TDMI des mémoires diverses : (i) 32Ko de mémoire de travail directement sur le corps du processeur (IWRAM) et (ii) 256Ko de mémoire de travail externe (EWRAM). La mémoire de travail externe servira à simuler le fonctionnement de la mémoire persistante. Tandis que la mémoire de travail est assimilée à une mémoire à accès rapide de type ScratchPad. Les mémoires ScratchPad sont des petites zones de mémoire rapide directement transposées dans l'espace d'adressage et gérées directement et explicitement au niveau logiciel.

Peu importe la technologie sous-jacente, le temps aussi bien que l'énergie requis par un accès à la mémoire dépendent de sa taille. Plus elle est grande, plus les temps d'accès et l'énergie consommée augmentent. La taille du bus est aussi déterminante. La Table IV.3 donne une vue sommaire sur les caractéristiques des différentes mémoires utilisées dans nos expériences. Nous reviendrons sur les latences des différentes mémoires dans la Section IV.4.1.2.

Table IV.3 — *Caractéristiques des mémoires utilisées*

Nom	Taille	Bus
ROM		16-bits
IWRAM	32Ko	32-bits
EWRAM	256Ko	16-bits

Remarque Même si les concepteurs de circuits intégrés ont vu les capacités d'intégration croître de façon considérable, il n'en reste pas moins vrai que plusieurs éléments architecturaux qui connaissent un véritable engouement dans les ordinateurs personnels, par exemple, restent difficiles à implémenter sur une puce. L'utilisation des mémoires cache dans les ordinateurs personnels a connu un véritable essor puisqu'elle a permis d'améliorer les performances dans le cas moyen et d'amoindrir le coût des accès à la mémoire. Cependant, pour les systèmes embarqués, les caches n'ont pas les mêmes attraits. En effet, de nombreux aspects tels que le coût surfacique, la consommation d'énergie, la latence d'un accès réussi CPU/cache [2], et les garanties temps réels sont de réels obstacles pour le développement des mémoires cache pour ce genre de cibles [PP07].

2. L'accès est dit réussi si la donnée est présente en cache. Le terme anglo-saxon utilisé est *(cache hit.*

IV.3 Applications

Nous avons privilégié pour notre expérimentation des applications typiques du monde de la carte à puce.

IV.3.1 CardEdge

L'applet CardEdge [3] est une applet Java Card implémentant la spécification de vérification d'identité personnelle (PIV) [4] du NIST [5].

L'applet fournit une interface permettant d'accéder aux informations confidentielles (*e.g.* clés, certficats, ...) contenues dans la carte à puce. Toutes les commandes envoyées à la carte sont redirigées par le système d'exploitation à l'applet CardEdge. Dans la terminologie ISO, les commandes correspondent aux APDU.

Un utilisateur doit s'authentifier bar le biais d'un code PIN [6] pour que la carte l'autorise à accéder aux données contenues dans la carte (*e.g.* une clé publique permettant de générer une signature). Si l'étape d'authentification basée sur le code PIN réussit, les données confidentielles sont exportées sous forme d'objets en RAM et peuvent désormais être être lues par l'application hôte.

IV.3.2 MatchOnCard

Afin de renforcer les mécanismes d'authentification par mot de passe ou code PIN, le recours aux caractéristiques biologiques est devenu de plus en plus populaire de nos jours. Un système biométrique est un système automatisé capable de :

(i) saisir un échantillon biométrique d'un utilisateur final (*e.g.* empreinte digitale) ;

(ii) extraire les données biométriques de cet échantillon ;

(iii) comparer les données biométriques avec celles contenues dans un ou plusieurs gabarits de référence ;

(iv) décider de leur degré de correspondance ; et enfin,

(v) indiquer si une identification ou une vérification d'identité a été réalisée ou non.

Les cartes à puce biométriques permettent de stocker dans l'EEPROM un échantillon biométrique précédemment traité et mémorisé correspondant à l'identité que prétend posséder l'intéressé et d'effectuer les étapes iii et iv. Typiquement, une fois l'utilisateur authentifié, l'accès à la carte est ouvert et il peut accéder aux fichiers stockées en EEPROM, permettre de générer une signature numérique ou débiter un portefeuille. Ces étapes ne requièrent pas plus d'une seconde pour s'exécuter dans le cas normal. S'il est codé sur la carte l'algorithme de match-on-card consomme 2,000 octets de RAM.

L'applet que nous utiliserons dans la suite confronte (match) un modèle (minutie) de l'image capturée pour la transaction aux modèles (minuties) des images capturées à l'enrôlement. Il s'agit essentiellement de réaliser des calculs avec le processeur embarqué, et plus particulièrement le

3. Le code source de l'applet est disponible à l'adresse http://www.identityalliance.com/downloads/PIV-II-MSU.zip.
4. PIV est l'acronyme de Personal Identification Verification.
5. NIST est l'acronyme de us National Institute of Standards and Technology.
6. PIN est l'acronyme de Personal Identification Number.

match-on-card, c'est à dire effectuer la comparaison entre les données qui viennent d'être acquises et les données biométriques enregistrées.

IV.4 Expérimentations

IV.4.1 Description de notre démarche

IV.4.1.1 Intégration dans la phase de compilation embarquée

La phase de calcul de WCET doit s'intégrer à la phase de compilation tardive effectuée sur la plateforme cible. Celle-ci repose sur des composants essentiels du noyau CAMILLE dont nous énumérons ici les principales fonctionnalités :

(i) Le processus de chargement de code dans CAMILLE peut être vu comme une simple boucle de lecture d'opérations élémentaires FAÇADE. Le flux binaire correspondant est analysé afin d'identifier l'opération en cours en déterminant sa nature afin de créer les structures de données correspondantes.

(ii) Un mécanisme d'inférence de types prend ensuite la main dans le but de vérifier l'innocuité du code chargé de façon séquentielle — instruction par instruction.

(iii) Une fois l'instruction lue et vérifiée, une dernière phase de compilation a lieu. Un composant du système — le *générateur de code natif* — est, en effet, sollicité pour compiler la forme intermédiaire (*i.e.* le code FAÇADE) en une forme exécutable par la plateforme matérielle sous-jacente. Le code exécutable ainsi généré est stocké dans la mémoire de la carte.

IV.4.1.2 Instanciation du calcul de WCET sur la cible matérielle

Chaque instruction native générée par le compilateur est traitée séparément car les dépendances entre paires d'instructions n'a pas pas d'effets néfastes sur le pipeline. Afin de compter les cycles consommés, il suffit d'identifier la classe à laquelle appartient l'instruction courante et de cumuler le nombre de cycles au pire cas qu'elle peut passer au troisième étage du pipeline (*i.e.* EXECUTE). Cette information est disponible dans la documentation fournie par le constructeur (voir Table IV.4). Pour chaque famille d'instructions, une opération de décalage augmente le temps d'exécution d'un cycle. Modifier la valeur du compteur ordinal est aussi pénalisé par deux cycles.

Dans le cas d'une instruction de traitement de données, le résultat généré par l'UAL[7] est directement écrit dans les registres correspondants et l'étage d'exécution consomme seulement un cycle (Voir Figure IV.5a).

Dans le cas d'une instruction de chargement/sauvegarde, l'adresse mémoire calculée par l'UAL est mise sur le bus d'adresse et la mémoire est accédée au cours du deuxième cycle passé dans le troisième étage du pipeline (Voir Figure IV.5c).

Quant à la multiplication, chaque étage composé de 8 bits consomme un cycle pour s'exécuter. Une instruction de multiplication peut, de ce fait, consommer de un à quatre cycles au pire des cas.

7. UAL est l'acronyme de Unité Arithmétique et Logique.

Table IV.4 -- *Coût des instructions* ARM

Instruction	Cycles	Cycles additionnels
Traitement de données	1S	+1S+1N si R15 est chargé, +1I si SHIFT(Rs)
MSR,MRS	1S	
LDR	1S+1N+1I	+1S+1N si R15 chargé
STR	2N	
LDM	nS+1N+1I	+1S+1N si R15 chargé
STM	(n-1)S+2N	
SWP	1S+2N+1I	
BL(THUMB)	3S+1N	
B,BL	2S+1N	
SWI	2S+1N	
MUL	1S+ml	
MLA	1S+(m+1)I	
MULL	1S+(m+1)I	
MLAL	1S+(m+2)I	
CDP	1S+bI	
LDC,STC	(n-1)S+2N+bI	
MCR	1N+bI+1C	
MRC	1S+(b+1)I+1C	
{*cond*} faux	1S	

IV.4.2 Résultats

Afin d'évaluer mon approche, j'ai évalué différentes parties du processus de certification. Ce processus faisant intervenir diverses étapes à la fois du côté producteur et du côté consommateur, j'ai évalué :

(i) Côté producteur :

L'efficacité de la détection des boucles

La taille de la preuve générée

(ii) Côté consommateur :

Le coût de la vérification

La détermination du WCET

IV.4.2.1 Détection des boucles

Dans le code noyau de CAMILLE programmé en langage C, nous avons dénombré 60 boucles **For** ainsi que 13 boucles **While**. Ce même code traduit en FAÇADE se compose de 120 méthodes effectuant pour l'essentiel des opérations arithmétiques et logiques ou de gestion de la mémoire. IV.6.

L'outil d'inférence des bornes de boucles — sur le producteur — a permis de borner 70% des méthodes du noyau. Les 30% restantes contiennent des boucles infinies ou dépendent des entrées

Table IV.5 -- *Latence des mémoires utilisées*

Mémoires	Type Cycles	Taille des données	Latence
ROM	N	8 bits	5
	S	8 bits	3
	N	16 bits	5
	S	16 bits	3
	N	32 bits	8
	S	32 bits	6
EWRAM	N	8 bits	3
	S	8 bits	3
	N	16 bits	3
	S	16 bits	3
	N	32 bits	6
	S	32 bits	6
IWRAM	N	8 bits	1
	S	8 bits	1
	N	16 bits	1
	S	16 bits	1
	N	32 bits	1
	S	32 bits	1

du programme. L'analyse abstraite a itéré cinq fois sur l'ensemble des classes du noyau avant de trouver un point fixe.

IV.4.2.2 Taille de la preuve

Il est important, voire crucial, de mesurer la taille de la preuve dans la mesure où celle-ci est chargée sur la carte conjointement au code sur lequel elle porte. La taille de la preuve croît proportionnellement au nombre de blocs de base (points de saut) et au nombre de variables contenues dans l'unité de code analysée. Chaque variable est codée pour l'instant sur 4 octets mais une compression reste envisageable. La taille de la preuve dépend aussi de la taille de l'unité de code analysée.

Seule une ligne de preuve est stockée en RAM au moment où l'instruction correspondante est traitée.

IV.4.2.3 Évaluation du mécanisme de vérification embarquée

La vérification des bornes de boucles s'effectue séquentiellement au fur et à mesure que les opérations FAÇADE sont décodées. Il s'agit, lors de cette passe, d'établir la concordance entre les annotations inférées contenus dans la preuve et le type calculé linéairement des variables. Ce mécanisme s'intègre comme nous l'avons préalablement expliqué (Voir Section IV.4.1.1) dans le processus de chargement de CAMILLE. Cette vérification induit un sur-coût de 2% dans le temps d'exécution global de chargement,vérification de types, et génération de code natif. Cependant, ce sur-coût est masqué par les latences des entrées/sorties à la lecture du flux binaire FAÇADE .

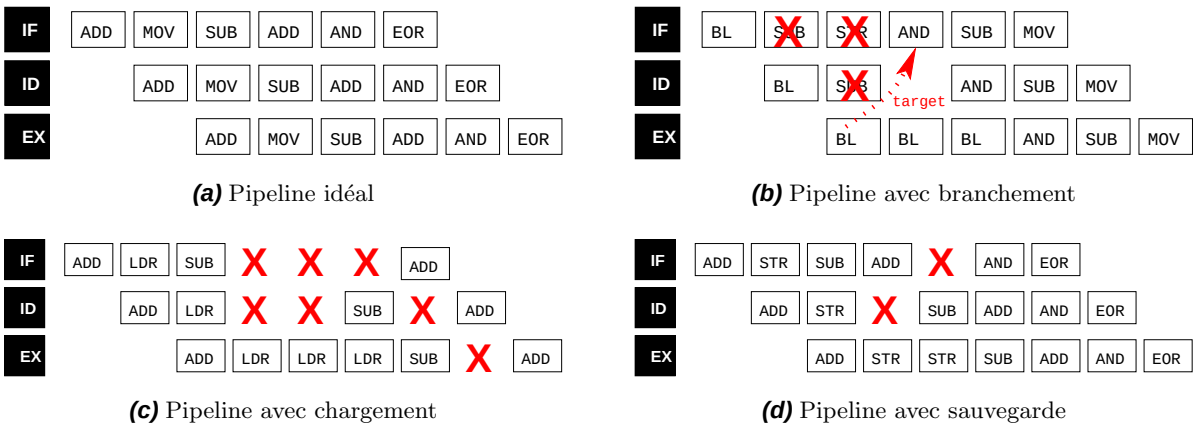


Figure IV.5 — *Différents comportements du pipeline : (a) pipeline idéal ne comportant que des instructions de traitement de données (b) pipeline avec instruction de branchement, (c) pipeline avec opération de transfert à partir de la mémoire, (d) pipeline avec opération de transfert vers la mémoire .*

Table IV.6 — *Description du code du noyau* CAMILLE

Métriques	**Code Source** C	**Code intermédiaire** FAÇADE
Taille (Ko)	195	148.5
Nombre de lignes	5962	6606

En effet, le temps de chargement croît linéairement avec la taille du code à charger. Le temps de chargement global en prenant en compte le calcul de WCET représente 150% du temps de chargement le cas échéant. Néanmoins, dans la pratique, ce surcoût est insignifiant étant donné le débit limité de la ligne série à travers laquelle le code est transmis à la carte à puce. De même, les latences d'écriture en mémoire persistante des instructions natives générées rendent acceptable le sur-coût dû au calcul de WCET. Une fois la vérification achevée, le code FAÇADE et la preuve concernant les bornes des boucles correspondantes ne sont pas conservés. L'empreinte mémoire du code chargé demeure ainsi inchangée après la vérification des bornes de boucles.

IV.4.2.4 Détermination du WCET

Afin d'exécuter l'applet CARDEDGE, il a fallu traduire en FAÇADE et charger quelques classes appartenant à l'API JAVA CARD. Elles sont au nombre de 19 et permettent le support des fonctionnalités cryptographiques incluant les fonctions asymétriques, la génération/gestion de clés, et la gestion des codes PINs. La taille de l'applet codée dans le sous-ensemble JAVA CARD est de 1500 lignes de code source.

La Table IV.7 montre le temps d'exécution mesuré dans la première colonne. La deuxième contient le pire temps d'exécution estimé par nos outils. La troisième donne une idée sur le pessimisme de notre analyse de WCET. Pour des programmes simples comme CARDEDGE, le temps d'exécution observé est de 59770 cycles du CPU. En revanche, le WCET prédit par nos outils est de 67142 cycles entraînant ainsi un pessimisme de 1.12 qui reste tout à fait convenable. En revanche, pour l'applet MATCH-ON-CARD, dont le code s'avère être plus complexe à analyser (*i.e.* contenant de multiples boucles imbriquées), nos outils de prédiction induisent une importante perte de précision.

Table IV.7 -- *Confrontation des* WCET *mesurés et estimés*

Application	Mesuré (cycles)	Estimé (cycles)	Ratio
CardEdge	59770	67142	1.12
Match-on-Card	10309321	54378867	5.27

IV.4.3 Discussion

Obtenir des estimations de WCET aussi précises sur la première applet CARDEDGE tire parti de la simplicité de l'architecture matérielle pour laquelle nous avons opté pour nos expériences. Rappelons que cette architecture est conforme, néanmoins, aux produits existants sur le marché. Les sources restantes de sur-estimation doivent être éliminées. La langage intermédiaire FAÇADE a cette particularité des plus intéressantes d'exprimer les opérations effectuées directement sur le matériel à travers de simples appels de méthodes. En effet, chaque composant du matériel (Pages mémoires, registres, ...) est décrit par une classe dans la sémantique FAÇADE. Les appels de méthodes sur ces composants sont traduits en instructions natives élémentaires (*e.g.* opérations arithmétiques et logiques sur des registres). C'est sur cette propriété du langage FAÇADE que nous allons nous baser nos améliorations en vue d'obtenir une estimation moins pessimiste du WCET.

Instructions de multiplication Le noyau CAMILLE transpose les différentes variables FAÇADE en registres du processeur ou en pages allouées dans la mémoire physique. Les variables numériques FAÇADE peuvent être représentées par des classes particulières notamment : les CARDBYTES (8 bits), les CARDSHORTS (16 bits), et enfin les CARDINTS (32 bits) comme illustré dans la Figure IV.6a. Comme nous l'avons expliqué précédemment, la longueur des données détermine le nombre de cycles consommés par une opération de multiplication sur l'ARM7TDMI . Dans le pire des cas, l'opérande est assimilée à un entier de 32 bits. Dans ce cas, la multiplication consomme quatre cycles. Le type de chaque variable FAÇADE peut être identifié à chaque point de saut par le biais de la preuve de type. La taille de l'opérande peut etre donc déterminé si le type de la variable FAÇADE est connu. Soit une variable qui prend le type CARDBYTE le long d'un bloc de base donné, Si cette même variable est impliquée dans une opération de multiplication, alors le nombre de cycles consommés par cette instruction est égal simplement à deux (un pour charger l'instruction et un dans l'étage de multiplication).

Instructions d'accès à la mémoire Le noyau CAMILLE donne aux applications l'impression d'accéder directement au matériel sans s'embarrasser d'abstraction de plus haut-niveau. En l'occurrence, les systèmes d'exploitation traditionnels représentent la mémoire physique en mémoire virtuelle et gèrent la traduction d'adresses virtuelle/physiques. En revanche, un exo-noyau multiplexe la mémoire physique directement. Ainsi, chaque page de la mémoire physique est exposé comme un composant du système comme représenté dans la Figure IV.6b. CAMILLE représente la mémoire physique comme une ensemble de pages élémentaires (32 Bytes). Les pages mémoire peuvent être allouées dans la mémoire de type SCRATCHPAD ou encore dans la mémoire de travail externe ou dans la mémoire persistante FLASH. Las pages mémoires peuvent contenir des valeurs (CARDTBMEM, CARDPBMEM) ou de références (CARDTOMEM, CARDPOMEM). Des tailles de pages différentes de 32, 64, 256 and 512 octets sont disponibles. Dans le pire des cas, une page mémoire est allouée en mémoire FLASH qui a le plus grand nombre de cycles de latence. A chaque accès à une page par le biais d'une invocation de méthode, les informations de typage à l'entrée de chaque bloc de base indiquent le type de mémoire dans laquelle elle est allouée. Si une page mémoire est du type CARDTB, il est ainsi possible de savoir que la page est allouée dans la SCRATCHPAD et donc aucun cycle de latence n'est rajouté.

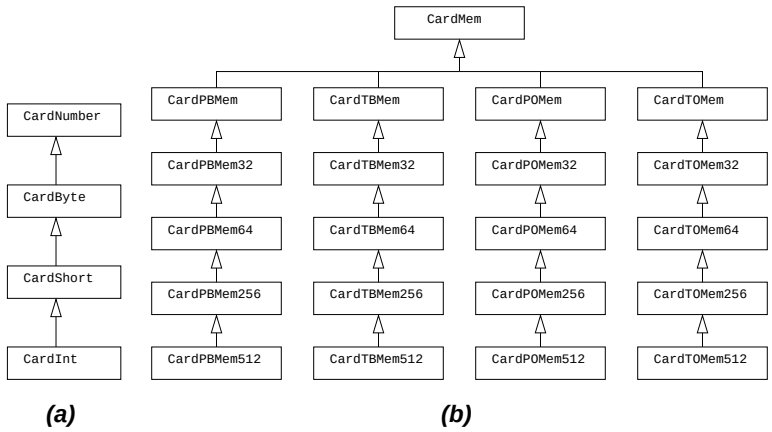


Figure IV.6 — *Sous-ensemble de la hiérarchie de types* CAMILLE

La Table IV.8 illustre les bénéfices de la connaissance des types pour diminuer le pessimisme sur le calcul de WCET. L'amélioration du pessimisme est de 81,22%.

Table IV.8 -- *Bénéfices de la connaissance des informations de typage*

Sans les types (cycles)	Avec les types (cycles)	Gain (%)
67142	65758	81,22

IV.5 Conclusion

Dans ce chapitre, nous avons présenté les résultats expérimentaux que nous avons obtenu sur des applications typiques de la carte à puce. Ces applications, mises en avant dans le cadre du projet européen InspiredD mettent en évidence la solidité de la démarche et son applicabilité dans un contexte d'utilisation réel. La plateforme cible que nous avons traité, bien que *simple* au regard d'un processeur intel, soulève de nombreux problème (prédiction du pipeline, délais de latence des mémoires, instructions à temps d'exécution variables, ...) source d'imprécision dans un calcul de WCET. Nous avons montré que ces difficultés pouvait être surmontée. Finalement, l'ensemble de ces résultats ont été présentés dans [BGB07].

V

Gestion du partage d'objets entre applications

> *How joyful am I made by this contract !*
>
> — Shakespeare - King Henry VI

> *L'important n'est pas de convaincre, mais de donner à réfléchir.*
>
> — Bernard Werber - Le Père de nos pères

V.1 Introduction

Dans les chapitres précédents, nous avons considéré implicitement qu'à chaque fois que l'analyseur — situé au niveau du consommateur de code — rencontre une instruction d'appel de méthode, son temps d'exécution calculé a priori vient s'ajouter au temps d'exécution global du programme. Nous avons donc supposé l'existence d'une table associant à chaque méthode appelée une valeur numérique — un nombre de cycles processeurs — correspondant à son WCET. Cette hypothèse simplificatrice, stipulant que l'analyse inter-procédurale se déroule d'une manière complètement transparente, ne correspond pas à ce qui se passe en pratique.

Outre les problèmes inhérents à l'utilisation d'un langage de programmation orienté objet (*e.g.* polymorphisme et appel de méthodes virtuelles), nous voulons prendre en compte dans ce chapitre les problèmes spécifiques aux environnements d'exécution ouverts, sécurisés et adaptables dynamiquement tels que ceux dont les TPD sont pourvus. En effet, traditionnellement, le calcul de WCET fait partie intégrante des processus de compilation et d'édition de liens réalisés sur un seul et même site. Les techniques d'analyse statique se basent, dans ce cas, sur la connaissance du logiciel dans son ensemble. L'analyse globale effectuée perçoit, en effet, le logiciel dans son intégralité et peut en traiter les différents constituants (*e.g.* classes, méthodes,...) simultanément. Nous nous plaçons, en revanche, dans le cas où la liaison entre les différents constituants du logiciel est réalisée par le consommateur de code faisant suite aux opérations de chargement et de vérification. En amont, les différentes unités de code constituant le logiciel sont compilées séparément par le producteur de code et fournies aux consommateurs sous une forme intermédiaire. Dans ce contexte, nous allons étudier le problème sous deux angles différents.

Nous nous intéresserons, tout d'abord, au cas où tout le logiciel applicatif est fourni par un unique producteur de code. Nous définissons, dans ce cadre, un schéma de déploiement visant, en particulier, à permettre au consommateur de code de disposer au moment du calcul du WCET d'une unité de code des coûts des unités dont elle dépend. Notre contribution réside dans l'établissement d'un ordre de déploiement permettant de calculer le WCET des nouvelles unités de code en respectant leurs dépendances respectives.

Ensuite, nous nous préoccuperons du cas où plusieurs logiciels, émanant d'entités différentes, sont amenés à cohabiter dans le même environnement d'exécution et à interagir pour amener une valeur ajoutée. Dans ce contexte, les réflexions proposées dans la première contribution sur l'ordre de chargement peuvent être incompatibles avec les choix commerciaux ou industriels des différents fournisseurs de code. Nous mettrons l'accent sur l'impact de l'interaction entre les différents logiciels — ceux déjà déployés et ceux qui seront chargés ultérieurement — sur les garanties de disponibilité préalablement apportées par le système. Nous proposerons, en perspective, une solution à base de contrats permettant d'empêcher la répudiation de ces garanties.

V.2 Préliminaires

V.2.1 Contexte

V.2.1.1 Chargement dynamique de code

Comme nous l'avons préalablement énoncé dans le Chapitre II, un TPD subit plusieurs phases allant de (i) sa création proprement dite à partir d'un circuit intégré et d'un masque logiciel

(*i.e.* contenant le système d'exploitation), (ii) une ou plusieurs étapes de personnalisation, (iii) et pour finir sa livraison à son utilisateur final. Néanmoins, il est important de souligner qu'il n'est pas toujours possible (ni même souhaitable) de charger toutes les applications au moment de la personnalisation initiale. Quelques applications sont, en effet, écrites en ROM durant les toutes premières étapes du cycle de production. Cependant, l'un des besoins forts identifiés par l'industrie au travers de projets tel qu'INSPIRED, consiste précisément à permettre à d'autres applications d'être chargées et activées en mémoire réinscriptible (*e.g.* FLASH) après que le TPD n'ait été délivré à son usager final. Nous insistons ici sur les différents bénéfices de cette approche tout en relevant les nouveaux problèmes qu'elle soulève.

Rappelons que pour les cartes à puce traditionnelles (*e.g.* carte bancaire, SIM), une mise à jour du système requiert le rappel chez le fabricant. Alors que les cartes sont déployées en quantités de plus en plus grandes, il est devenu vite impossible de procéder de cette manière. Par ailleurs, de nouvelles cartes doivent être fabriquées afin de remplacer celles qui n'ont plus les fonctionnalités désirées. Outre le coût élevé de fabrication des nouvelles cartes, il n'en demeure pas moins difficile de demander à chaque usager de rendre sa carte et d'attendre la création d'une nouvelle. La situation est d'autant plus problématique avec les cartes multi-applicatives. Ces dernières résultent de partenariats signés entre des organismes de secteurs d'activité différents et réunissent, par exemple, une carte bancaire aux prestations classiques mais aussi des services offerts par des organismes non bancaires (*e.g.* un service de téléchargement de musique, un service de rechargement de téléphone, ...). Dans ce cadre, la production d'une nouvelle carte requiert une re-négociation complète entre les différents organismes impliqués. En pratique, imaginons une carte qui délivre un service de paiement bancaire, et qui est utilisée également pour contrôler l'accès à des services de transport en commun. Le rappel chez le prestataire de service pour l'enrichissement des fonctionnalités de l'application bancaire prive l'usager, du moins momentanément, de l'accès au service des transport en commun. C'est bien sûr une situation difficilement envisageable pour les usagers. À partir de ces deux situations, il apparaît clairement que la gestion, c'est à dire le déploiement et/ou le retrait de différents modules logiciels, doit pouvoir être faite sans forcément produire un nouveau TPD.

D'autre part, il est aussi important de constater qu'une fois le TPD délivré à son utilisateur, ses fonctionnalités évoluent au gré des choix de celui-ci. Comme expliqué précédemment, le TPD étant un *objet personnel*, son mode d'utilisation est centré uniquement sur son porteur contrairement à la carte à puce traditionnelle. Désormais, toute utilisation — dans le but de le configurer ou d'y installer/désinstaller une application donnée — ou accès aux données confidentielles qu'il contient par un tiers (*e.g.* le fabricant ou un fournisseur de service) doivent être soumis au consentement du porteur du TPD.

Cette mise en contexte rappelle les contraintes que nous devons prendre en compte dans nos investigations. En premier lieu, il est crucial de permettre au TPD d'évoluer alors qu'il se trouve déjà dans la poche de son utilisateur final sans solliciter un rappel chez le fabricant pour chaque mise à jour. De plus, l'installation des applications et la souscription à de nouveaux services sont complètement tributaires des choix de l'utilisateur final. En l'occurrence, les installations/désinstallations sont espacées dans le temps et dépendent de la volonté de l'utilisateur possédant le TPD. Le comportement du TPD évolue, donc, en conséquence. En d'autres termes, puisque les fonctionnalités de chaque TPD évoluent au gré des choix de son porteur, nous assumons qu'aucune infrastructure globale, aucun producteur de code, n'est en mesure, à un moment précis, de déterminer l'ensemble de logiciels préalablement installés sur un TPD donné.

V.2.1.2 Cadres d'applications orientés-objets

Depuis quelques années, l'utilisation des cadres d'applications orientés objets s'est imposée dans le processus de développement afin de maîtriser le coût et la complexité des logiciels. Un cadre d'application peut être défini comme une conception abstraite et réutilisable dédiée à un domaine d'application spécifique [Foo88]. En effet, si la programmation orientée objet implique la réutilisation de composants logiciels, le cadre d'applications se place à un plus haut niveau d'abstraction en permettant la réutilisation de l'architecture et des solutions récurrentes dans un domaine d'application spécifique.

Concrètement, c'est un ensemble de classes abstraites et d'interfaces donnant la possibilité aux développeurs de spécialiser un comportement pour l'intégrer dans leurs logiciels. Afin de masquer la complexité des algorithmes spécifiques à un domaine particulier aux développeurs, l'utilisation de la classe *java.net.Socket* dans une application JAVA standard permet, par exemple, d'écrire des applications indépendantes du protocole de transport sur lequel les paquets TCP/IP sont acheminés. Aussi, une applet JAVA CARD doit obligatoirement hériter de la classe de base *javacard.framework.Applet*. En héritant de cette classe de base, l'applet sera automatiquement considérée comme un processeur de paquets APDU exécuté dans un mode commande-réponse asynchrone et half-duplex.

Grâce à la notion d'*inversion de contrôle*, ce n'est plus à l'application de définir elle-même son corps et à appeler les parties qu'elle désire réutiliser. A contrario, elle réutilise le corps défini dans le cadre et définit le code qui sera ensuite appelé par ce dernier. Enfin, les cadres d'applications imposent un ordre et une structure aux applications. Le cadre d'application fournit l'architecture finale. L'application précise seulement les étapes qui lui sont spécifiques.

L'application n'interagit plus avec le système d'exploitation, mais uniquement avec le cadre d'application comme le montre la Figure V.1.

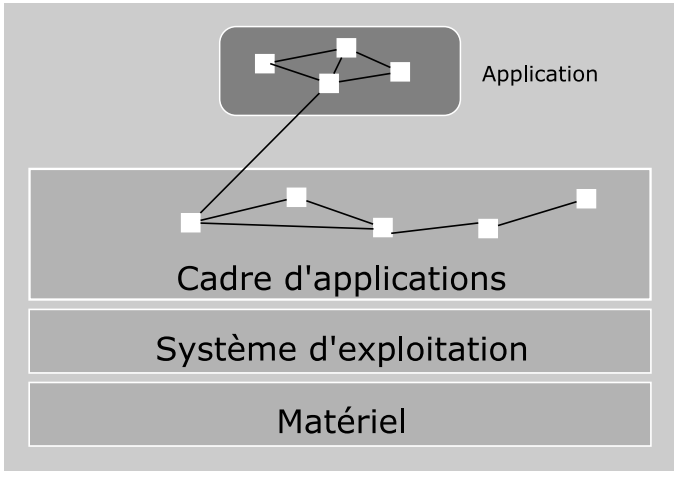


Figure V.1 — *Interaction avec le cadre d'application.*

V.2.2 Exposition du problème

Sur la base de ces constats, nous pouvons décliner les deux hypothèses que nous prendrons dans le reste de ce chapitre.

Déploiement d'unités de code autonomes Tout d'abord, nous allons nous intéresser au déploiement d'unités de code *autonomes*. Par unités de code autonomes, nous désignons celles qui ne sont pas amenées à interagir — durant tout leur cycle de vie — ni par accès aux données (*e.g.* lecture de champs) ni par échange de messages (*e.g.* appel de méthode).

Imaginons, par exemple, un TPD contenant un service de paiement classique mais servant également comme outil de vote électronique. Ces deux applications n'ont a priori aucune raison d'échanger des informations ou de partager des objets. De même, le déploiement ultérieur de l'une n'induit pas d'interférence avec le fonctionnement de l'autre. Nous pouvons ainsi considérer — analyser — l'une d'une manière complètement indépendante de l'autre.

La Figure V.2 illustre cette situation. Il n'y a visiblement aucune corrélation entre les applications A, B et C déployées respectivement aux instants t,t' et t''. Leurs comportements peuvent donc être analysés indépendamment les unes des autres.

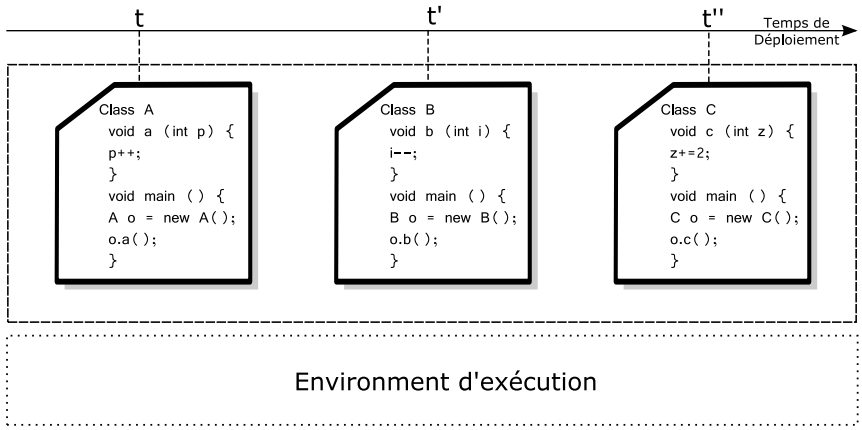


Figure V.2 — *Déploiement d'unités de code autonomes.*

Déploiement d'unités de code coopératrices Par opposition aux unités de code autonomes, nous désignons par le terme *coopératrices* les unités de code qui sont amenées à réaliser une valeur ajoutée en interaction avec des applications déjà déployées sur le TPD. Des unités de code provenant de fournisseurs de code — ne se faisant pas mutuellement confiance — peuvent coopérer afin de fournir un service commun. La coopération se matérialisera alors par tout accès à des objets partagés.

Tel est le cas pour l'application du porte-monnaie électronique [HGSC04]. Ce cas d'étude est utilisé régulièrement à des fins d'illustration dans la littérature traitant de la vérification de la sécurité des applications encartées. Nous nous en sommes inspirés afin de décrire un cas d'utilisation mettant en relief nos hypothèses et appuyant notre démarche. Le diagramme de classe de la Figure V.3 en donne une idée plus synthétique. La classe *Purse* gère les opérations de débit et crédit du porte-monnaie ainsi qu'un historique des transactions. La classe *Loyalty*

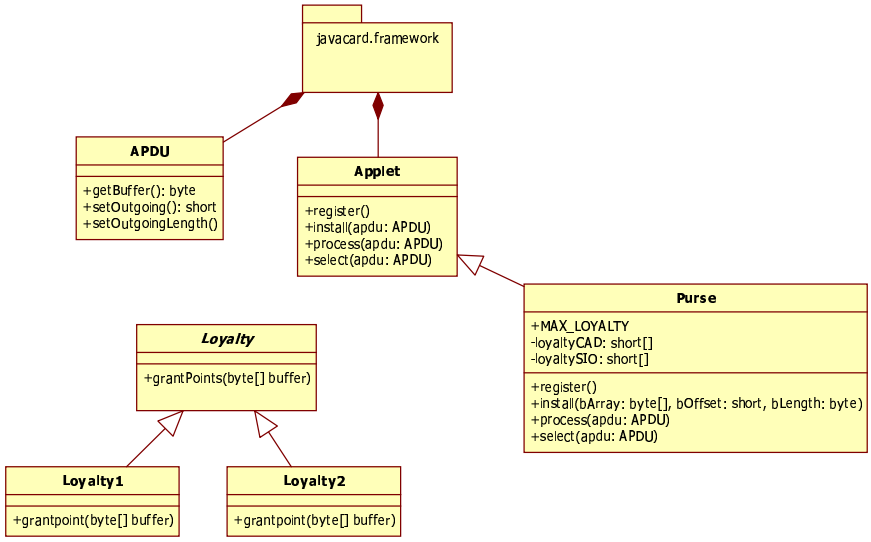


Figure V.3 — *Diagramme de classes du porte-monnaie électronique.*

octroie les points de fidélité en fonction des achats effectués avec le porte-monnaie. Quand une transaction se termine, la méthode *CompleteTransaction* de la classe *Purse* appelle la méthode *grantPoints* qui permet de mettre à jour les points de fidélité du client. Lorsque le détenteur d'un TPD souhaite participer à un programme de fidélité, il lui suffit de charger une sous-classe de la classe abstraite *Loyalty* fournie par le marchand concerné. Ces points de fidélité lui seront automatiquement crédités.

Supposons qu'un usager lambda dispose d'un TPD où une application porte-monnaie électronique est déployée. Lorsqu'il s'en sert, il débite ou crédite de l'argent dans le porte-monnaie. Lors d'un voyage, une compagnie aérienne *AirBeta* propose à l'usager de souscrire à un programme de fidélité. L'application *LoyaltyAirBeta* est alors déployée, elle vient surcharger les méthodes de la classe abstraite *Loyalty* initialement déployée avec *Purse*. L'application *AirBeta* vient ensuite s'enregistrer auprès de la classe *Purse* et lorsque l'usager débite une nouvelle fois une somme, celle-ci notifie l'instance de *AirBeta* qui s'est enregistré en observateur. Le code déployé par *AirBeta* est alors exécuté. Ainsi, grâce au programme de fidélité, l'usager gagne des miles pour tout achat effectué avec son TPD et peut accéder au bout d'un certain nombre de points cumulés à la prime de son choix : billets d'avion, nuits d'hôtels, location de voiture...

V.3 Première hypothèse : unités de code émanant d'un même producteur de code

Nous prenons, dans un premier temps, l'hypothèse simplificatrice selon laquelle l'intégralité du nouveau logiciel à déployer est disponible lors du chargement. De plus, nous considérons que le producteur de code est à même de décider l'ordre dans lequel il présente les composants logiciels et les traitements associés.

V.3.1 Calcul de WCET inter-méthode

Nous utiliserons désormais les notations définies ci-dessous dans le reste du document. A chaque classe C (notée $C ::= (\mathcal{F}(C), \mathcal{M}(C))$), on associe l'ensemble des informations suivantes :

(i) l'ensemble $\mathcal{F}(C)$ des attributs définis dans cette classe (non héritées) ainsi que leur type,

(ii) l'ensemble $\mathcal{M}(C)$ des méthodes implémentées dans cette classe (*i.e.* non héritées).

L'accès à la méthode m d'une classe C, se fait avec l'expression fonctionnelle $C.m$. Une méthode m est représentée par une signature et un corps.

$$\forall m \in \mathcal{M}(C), \quad C.m \quad ::= (signature_m, corps_m)$$

Le corps d'une méthode m d'une classe C se compose du flôt de ses instructions tel que :

$$corps(C.m) \quad ::= \quad \{[label_1]instruction_1 \ldots [label_n]instruction_n\}$$

Chaque instruction dans le corps d'une méthode m est identifiée par une étiquette *label* qui dénote une position dans le corps d'une méthode. La fonction *inst* associe une étiquette à chaque instruction dans le corps d'une méthode tel que $inst(label) = instruction$. On note $Labels(C.m)$ l'ensemble des labels et $Instructions(C.m)$ l'ensemble des instructions dans une méthode m.

La signature d'une méthode peut être définie comme suit :

$$signature(C.m) \quad ::= \quad \langle mname, type\_retour, [type\_arg_1 \ldots type\_arg_n] \rangle$$

où *mname* spécifie le nom de la méthode, *type_retour* désigne le type de retour de la méthode, et $[type\_arg_1 \ldots type\_arg_n]$ désignent les types des différents paramètres s'il y a lieu.

On représente par $\mathcal{S}(C)$ l'ensemble des signatures des méthodes définies dans cette classe. La signature d'une méthode représente un identifiant de la méthode au sein d'une même classe. En effet, il ne peut y avoir deux méthodes avec des signatures égales. Par contre, dans une autre classe, il est possible d'avoir une méthode avec la même signature. La signature d'une méthode n'est donc pas un discriminant absolu, mais relatif à la classe.

Calculer le pire temps d'exécution d'une méthode revient à évaluer le coût en terme de consommation processeur de toutes les instructions qui se trouvent dans son corps et de trouver le pire scénario d'exécution possible. Dans l'algorithme de calcul présenté ci-dessous, c'est la fonction w_{intra} qui renvoie le pire temps d'exécution connaissant le coût attribué à chaque instruction y compris les appels de méthodes. Le WCET d'une méthode m d'une classe C ne peut être déterminé qu'une fois tous les appels de méthodes effectués dans son corps sont associés à leur coûts respectifs. En effet, le calcul doit suivre les dépendances jusqu'à arriver aux méthodes dont les pires temps peuvent être directement calculés (où il n'y plus d'appels de méthodes dont le WCET est inconnu). La représentation usuelle des dépendances entre les méthodes est le graphe d'appel. L'algorithme de calcul de pire temps d'exécution d'une méthode se traduit, donc, aisément en un parcours de bas en haut du graphe d'appel partant des méthodes porteuses de l'information de WCET (*i.e.* feuilles du graphe d'appel) afin d'obtenir le pire temps d'exécution de la méthode appelante. Dans l'algorithme illustré par la figure V.4, la fonction $w!$ est appelée récursivement afin de calculer le pire temps d'exécution des feuilles du graphe d'appel avant de remonter l'information à la méthode appelante.

Le principe de calcul que nous venons d'énoncer est valable autant pour les langages procéduraux que pour les langages orientés objets. La principale spécificité des langages orientés

objets réside dans le fait que la combinaison des mécanismes d'héritage, de polymorphisme et de liaison dynamique rend délicat de déterminer avant l'exécution la méthode qui sera réellement exécutée suite à une invocation. En effet, pour prendre en compte cet aspect dynamique, il est d'usage de considérer que le pire des cas correspond au plus grand des WCET des méthodes qui seront potentiellement appelées.

On définit la fonction de calcul $w!$ qui renvoie le résultat du calcul du pire temps d'exécution de la méthode m une fois que les pires temps d'exécution de toutes les méthodes dont elle dépend sont connus selon l'algorithme de la figure V.4.

Algorithme $w!(C.m)$
{
  Si $(\exists d_i \in \mathcal{D}(C.m)$ pour qui $\nexists x$ tel que $(d_i, x) \in \Sigma)$
    $\Sigma \leftarrow \Sigma \cup (d_i, w(d_i))$;
  return $w_{intra}(C.m, \Sigma)$;
}

Figure V.4 — *Algorithme récursif de calcul du* WCET

On note Σ l'ensemble des tuples associant à chaque méthode désignée par $C.m$ une valeur numérique correspondant à son pire temps d'exécution tel que :

$$\Sigma = \{\ \sigma_i |\ \sigma_i = (C.m, w(C.m))\}$$

La fonction w permet de déterminer le pire temps d'exécution de toutes les redéfinitions d'une méthode m dans un treillis de type donné selon la définition suivante :

$$w(C.m) \quad = \quad \underset{\forall C' \subseteq C}{Max}(w!(C'.m))$$

La fonction $w_{intra}(C.m, \Sigma)$ peut calculer le pire temps d'exécution de la méthode désignée par $C.m$ si et seulement si on a un tuple qui représente chaque appel de méthode dans le corps de m dans l'ensemble Σ. Si l'ensemble Σ est vide, c'est à dire que le calcul de WCET de la méthode $C.m$ ne dépend d'aucune autre, la fonction $w_{intra}(C.m, \emptyset)$ retourne directement le pire temps d'exécution de la méthode m.

Exemple 1 (Polymorphisme) Pour illustrer ces propos, reconsidérons la figure **??** où la classe W définit une méthode publique m qui est redéfinie dans sa classes fille Z. La classe Y qui hérite de la classe X redéfinit elle aussi la méthode m. Le temps d'exécution de la méthode d'instance m de la classe W est différent dans les sous-classes Z et Y où la méthode est redéfinie. On suppose que les pires temps d'exécution respectifs de la méthode m définie dans les classes W, Z et Y sont 100, 120 et 35 ms. L'évaluation du pire temps nécessaire pour exécuter l'appel de la méthode m à travers l'instance o doit prendre en compte la nature polymorphe de cet appel et doit maximiser tous les pires temps calculés pour toutes les sous-classes où la méthode m est redéfinie. En l'occurrence, dans le pire des cas, l'appel de m coûtera 120ms. Dans l'algorithme illustré par la figure V.4, la fonction w renvoie le maximum des pires temps d'exécution calculés sur l'ensemble des méthodes qui peuvent être potentiellement exécutées suite à un appel polymorphe.

$$\begin{aligned} w(W.m) &= Max(\ w!(W.m)\ , w!(Z.m)\ , w!(Y.m)\ ) \\ &= Max(\ \overbrace{w_{intra}(W.m, \emptyset)}^{100}\ , \overbrace{w_{intra}(Z.m, \emptyset)}^{120}\ , \overbrace{w_{intra}(Y.m, \emptyset)}^{35}\ ) \\ &= 120 \end{aligned}$$

V.3.2 Évaluation

L'algorithme récursif de calcul (voir Figure V.4) se termine quand il arrive à calculer le pire temps d'exécution de la méthode appelante au début de la chaîne d'appels. En d'autres mots, le pire temps d'exécution d'une méthode m ne peut être obtenu que si le pire temps d'exécution de toutes les méthodes dont il a besoin a été préalablement calculé.

$$w_{intra}(C.m, \Sigma) \text{ est obtenu } \textbf{ssi } \forall d_i \in \mathcal{D}(C.m), \exists x \text{ tel que } (d_i, x) \in \Sigma \qquad \text{(V.3A)}$$

S'il y a un appel récursif dans la méthode analysée, l'algorithme de calcul de pire temps d'exécution que nous avons présenté dans la figure V.4 ne se termine jamais. Dans [Gus94b], l'auteur décrit les différents cas de récursivité rencontrés dans le contexte de la programmation orienté objet. Comme pour les langages procéduraux, la récursivité peut être directe (la méthode appelante est elle même appelée) ou indirecte (la suite d'appels qui part de la méthode appelante peut passer par plusieurs autres méthodes avant d'arriver de nouveau à l'appelant). A partir des spécificités introduites par la programmation orientée objet, l'auteur distingue notamment trois sortes de récursivité :

(i) **récursivité d'instance :** Quand une méthode m d'un objet o fait appel à elle-même (appel à la même instance). Il est aussi possible qu'une méthode $m1$ d'un autre objet o_1 fasse appel à la méthode m de objet o, qui elle même fait appel à la méthode $m1$ de l'objet o_1.

(ii) **récursivité de classe :** Quand dans une méthode m d'un objet o_1 (instance de la classe C) on appelle la méthode m d'un objet o, lui même instance de la même classe C.

(iii) **récursivité polymorphique :** Quand dans le corps d'une méthode m, il y a un appel vers une méthode m sur des objets dont on ne connaît pas la classe. C'est potentiellement un cas de récursivité de classe mais à cause du polymorphisme on ne saurait le dire qu'au moment de l'exécution.

Dans les deux premiers cas, l'algorithme de la figure V.4 ne se termine jamais comme l'illustre la trace d'exécution ci-dessous :

$$\begin{aligned} w!(C.m) &= w_{intra}(C.m, \{(C.m, w(C.m)\})) \\ &= w_{intra}(C.m, \{(C.m, \underset{\forall C' \subseteq C}{Max}(w!(C'.m))\})) \\ &= w_{intra}(C.m, \{(C.m, Max(\ldots, \boxed{w!(C.m)})\})) \end{aligned}$$

Le troisième cas est induit par l'impossibilité de résoudre les appels de méthodes à la compilation et peut introduire une récursivité potentielle dans le système. En l'occurence, dans le cas de la figure V.5, on veut calculer le pire temps d'exécution de $subA.m$. Pour ce faire, on doit construire l'ensemble $\mathcal{D}(subA.m)$ contenant toutes les redéfinitions de la méthode m dans les sous-classes de A. En l'occurence, l'ensemble $\mathcal{D}(subA.m)$ contient $subA.m$. Ainsi, pour évaluer le coût de l'appel $a.m()$, on doit d'abord calculer le pire temps d'exécution de $subA.m$. Cela produit un cycle et le calcul ne se termine jamais comme le montre la trace d'exécution ci-dessous :

$$\begin{aligned} w!(subA.m) &= w_{intra}(subA.m, \{(A.m, w(A.m)\})) \\ &= w_{intra}(subA.m, \{(A.m, \underset{\forall A' \subseteq A}{Max}(w!(A'.m))\})) \\ &= w_{intra}(subA.m, \{(A.m, Max(\boxed{w!(subA.m)}, w!(A.m))\})) \end{aligned}$$

```
Class A

void m (subA sa)
{return;}
```

```
Class subA

void m (A a)
{a.m();}
```

Figure V.5 — *Cas de récursivité polymorphique*

Tant que $((\exists C.m \in \Delta) \wedge (\forall d_i \in \mathcal{D}(C.m), \exists x \text{ tel que } (d_i, x) \in \Sigma))$
{
$\Sigma \leftarrow \Sigma \cup (C.m, w_{intra}(C.m))$
$\Delta \leftarrow \Delta / \{C.m\}$
}

Figure V.6 — *Algorithme mono-passe de calcul du wcet*

V.3.3 Algorithme de calcul itératif

Nous proposons un algorithme qui définit un ordre de calcul afin de permettre de fournir le pire temps d'exécution d'une méthode particulière en une seule passe sur son code. Il suffit simplement d'entamer le calcul de pire temps d'exécution de la méthode m de la classe C une fois que toutes les méthodes appartenant à l'ensemble $\mathcal{D}(C.m)$ sont associées à une valeur dans l'ensemble Σ.

On note Δ l'ensemble des méthodes auxquelles on n'a pas encore attribué un pire temps d'exécution. Au début de l'algorithme, Δ contient l'ensemble des méthodes disponibles sur le système.

S'il y a un appel récursif dans une des méthodes analysées, l'ensemble Δ sera non vide.

V.3.4 Calcul de WCET inter-méthode embarqué

Le processus de calcul du pire temps d'exécution sur du code mobile diffère du processus classique et soulève de nouveaux problèmes liés à la spécificité de ce paradigme.

Nous avons présenté dans [BRG04], un algorithme qui permet d'inférer les bornes des boucles contenues dans le code en profitant des ressources *infinies* dont dispose le producteur de code. Le calcul de WCET était par la suite instancié sur l'environnement d'exécution du consommateur en associant à chaque instruction machine un côut en terme de consommation de CPU.

On s'inscrit dans la même logique en voulant distribuer l'algorithme décrit dans la figure V.6. La figure V.7 présente les comportement attendus du producteur et du consommateur de code. Pour les besoins de l'algorithme d'envoi (voir figure V.7a), on définit Ω l'ensemble des méthodes non encore envoyées au consommateur.

Tant que $(\exists C.m \in \Omega)$
{
  Pour chaque $C'.m'$ **dans** $\mathcal{D}(C.m)$ **Faire**
  {
    $Envoyer(C'.m')$
    $\Omega \leftarrow \Omega \backslash \{C'.m'\}$
  }
}

(a) Producteur de code

Tant que (vrai)
{
  Charger$(C_{recu}.m_{recu})$
  Si $((\nexists x$ tel que $(C_{recu}.m_{recu}, x) \in \Sigma) \wedge$
  $(\forall C'.m' \in \mathcal{D}(C_{recu}.m_{recu}), \exists x$ tel que $(C'.m', x) \in \Sigma)))$
    $\Sigma \leftarrow \Sigma \cup (C_{recu}.m_{recu}, w_{intra}(C_{recu}.m_{recu}))$
}

(b) Consommateur de code

Figure V.7 — *algorithme de chargement*

V.3.5 Bénéfices et limites de l'approche proposée

Dans le même esprit que le Chapitre III, nous avons maintenu, dans cette la Section V.3, l'impératif de fournir au TPD toutes les informations dont il a besoin au moment où il en a besoin. Cette finalité est de permettre le chargement/installation d'un nouveau logiciel comme un flot en une seule passe sans avoir besoin de revenir sur un résultat établi le pas précédent ou d'attendre un résultat ultérieur. Ce mode de chargement où les traitements sont présentés au consommateur de code dans un ordre bien précis a plusieurs avantages. En premier lieu, la tâche du consommateur est nettement simplifiée puisqu'elle se réduit à traiter le code reçu en flot sans avoir à revenir sur des opérations déjà analysées. Plus important encore, sans la possibilité d'organiser l'ordre dans lequel les unités de code lui parviennent, le consommateur est obligé d'attendre la réception de tout le code pour entamer les calculs selon l'algorithme de la Figure V.4. La complexité en terme de consommation mémoire augmente ainsi que le temps d'exécution de l'algorithme qui devient alors plus perceptible pour l'utilisateur. En fait, plus que la transparence de l'exécution du calcul de WCET, c'est l'espace mémoire pour contenir l'intégralité du code qui risque de faire défaut sur un dispositif contraints comme le TPD.

Enfin, cette approche atteint, néanmoins, ses limites dans certains cas de figures : (i) le code doit pouvoir être réorganisé. Or, cela n'est pas forcément le cas, surtout lorsque cette réorganisation suppose de permuter l'ordre de présentation des différentes procédures/méthodes entre différents modules/classes. Elle est inapplicable à une JVM, par exemple, car elle suppose qu'on puise charger une méthode d'un premier *.class*, puis une seconde d'un autre *.class*, avant de charger une troisième issue du premier *.class*. Les JVM chargent effectivement les classes, une par une. (ii) En pratique, notre démarche ne supporte pas la récursivité, et cela dépasse le

seul problème de la stratégie de déploiement, puisque nous avons supposé, dès l'analyse intra-procédurale (Voir Chapitre III), que les applications traitées n'étaient pas récursives.

V.4 Deuxième hypothèse : Applications émanant de fournisseurs de code différents

Dans la Section V.3, nous avons implicitement assumé que le fournisseur de code dispose de tout le logiciel sollicité pendant le traitement de ses requêtes et qu'il était donc en mesure de réorganiser la présentation du logiciel à sa guise. La méthode de calcul que nous avons présenté supposait également l'indépendance entre les applications déployées sur le TPD. En d'autre mots, cela se traduit par l'absence de tout échange de service ou d'information. Cette hypothèse n'est cependant pas toujours possible à vérifier. Nous nous intéressons donc dans cette section au cas où les applications peuvent communiquer par le biais d'appels de méthodes. Permettre à des applications de communiquer soulève cependant de nouveaux problèmes : (i) les applications doivent s'assurer que leurs méthodes publiques ne sont pas appelées par des applications mal intentionnées, (ii) les applications manipulant des données confidentielles doivent s'assurer qu'il n'y aura pas de fuite, (iii) dans le cas qui nous intéresse les applications auxquelles le système a garanti un certain budget CPU doivent s'assurer que si elles interagissent avec d'autres elle ne font pas de la sur-consommation.

V.4.1 Nouveaux problèmes

Jusque là, il a été possible de calculer le WCET à la granularité d'une méthode grâce au schéma de déploiement incrémental. Ce schéma permet d'amener les unités de code dans un ordre respectant les dépendances, permettant ainsi à l'analyseur de disposer des coûts des méthodes appelées au moment où il en a besoin. On ne peut pas maintenir l'hypothèse de précédence quand il existe des points d'ouverture dans le code (par exemple si on déclare une interface générique telle qu'une classe abstraite). Une sous-classe peut être chargée plus tard en redéfinissant une méthode dont le WCET a été déjà calculé par le système. Dans ce cas, deux solutions se présentent.

Première solution Lorsqu'une nouvelle unité de code est amenée sur le système, tous les calculs de WCET déjà effectués doivent être refaits. En particulier, toutes les unités de code dont le WCET dépend de celui de la nouvelle sont désormais invalides et doivent être refaits.

Cette solution est relativement complexe à implémenter en pratique. (i) le temps de calcul deviendrait de plus en plus long au fur et à mesure que de nouvelles applications sont installées. En effet, il faudrait re-vérifier la garantie apporté à toutes les applications déjà installé après avoir calculer le WCET de la nouvelle application.

La première consiste à invalider tous les calculs déjà effectués. Ils doivent être refaits pour chaque nouveau logiciel chargé sur le système. Reprendre les calculs à chaque fois induit un surcoût conséquent : il faut chercher les WCET affectées et les recalculer. En outre, lors de la phase de compilation tardive effectuée sur le consommateur, le chargeur de CAMILLE notamment ne garde pas le code intermédiaire mais seulement le code natif. Il devient donc impossible de refaire les calculs puisque les informations de boucles, par exemple, portent seulement sur le code intermédiaire. De plus, pour pouvoir refaire les calculs, il faudrait non seulement conserver

le code intermédiaire, mais aussi les éléments de preuves de programmes, et donc sacrifier de l'espace mémoire.

Deuxième solution Afin d'éviter les inconvénients de la première solution, nous proposons d'établir des sortes de contrats sur la plateforme cible qui empêchent qu'une redéfinition remette en cause les garanties préalablement apportées par le système. On veut donc préserver les garanties apportées par le système sur le temps de réponse. A cause des points d'ouverture on ne pourra pas calculer un WCET définitif. On ne peut seulement donner une expression qui permet de le calculer en fonction des inconnues qui seront les sites d'appels ouverts. Le système doit garder trace des garanties qu'il a pu donner pour l'instant et il décide lorsqu'une nouvelle extension se présente si oui ou non il lui permet de s'installer et de se lier aux applications déjà présentes afin de ne pas remettre en cause l'inéquation.

V.4.2 Génération des contrats

Nous nous appuierons dans ce qui suit sur le patron de conception comportemental *Observateur/Sujet*. Ce mécanisme permet d'informer et de mettre à jour en conséquence plusieurs observateurs enregistrés auprès d'un sujet, lorsque celui-ci change d'état [EGV95]. Le patron de conception peut être vu comme le cas générique du portefeuille électronique que nous avons décrit dans la Section V.2. En effet, la méthode *grantpoints* de *Loyalty* est associée à un évènement particulier dans l'application *Purse*. Notamment, à la fin d'une transaction, les points de fidélité des classes *Loyalty* sont incrémentés en cas de paiement auprès du magasin affilié.

Considérons trois classes A, B et C ainsi qu'une classe abstraite OBSERVER comme le montre la Figure V.8. Les classes B et C, sous-classes de OBSERVER, s'enregistrent auprès de A pour être notifiées à travers la méthode *notifyObservers()* quand un évènement particulier se produit. Les classes B et C spécialisent à leur convenance le code de leur propre méthode de notification *notify()* définissant ainsi les actions effectuées suite à une notification. Comme le montre la Figure V.8, les classes A et Observer sont les premières déployées sur le système à l'instant t. B et C, quant à elles, sont installées après la mise en circulation du TPD respectivement aux instants t' et t".

Le diagramme de séquence dans la Figure V.9 définit les interactions entre les différents objets déployés sur le système.

(i) Tout d'abord, a une instance de A, est installée dans l'environnement d'exécution en appelant sa méthode statique *install()*. Au même moment, la classe abstraite Observer est aussi déployées.

(ii) Ensuite, l'environnement d'exécution appelle la méthode statique *process()* sur l'objet a. Une fois cette méthode appelée, les observateurs enregistrés sont notifiés à travers l'invocation de la méthode *notifyObservers()*.

(iii) Plus tard, b, une instance de B, demande à être installée sur l'environnement d'exécution. Elle redéfinit la méthode *notify()* de la classe abstraite Observer. L'environnement d'exécution vérifie que la méthode *notifyObservers()* exécutée par a serait en mesure de répondre au bout d'un laps de temps préalablement fixé appelé R. Cette vérification consiste à calculer le WCET de la méthode *process()* (noté $wcet(process())$). $wcet(process())$ doit être inférieur à R pour que l'environnement d'exécution installe la nouvelle classe. En effet, une fois installée, b s'enregistre auprès de a. Si a exécute la méthode *notifyObservers()*, il devra appeler *notify()* sur b. Si le sur-coût de l'exécution de la méthode *notify()* sur b

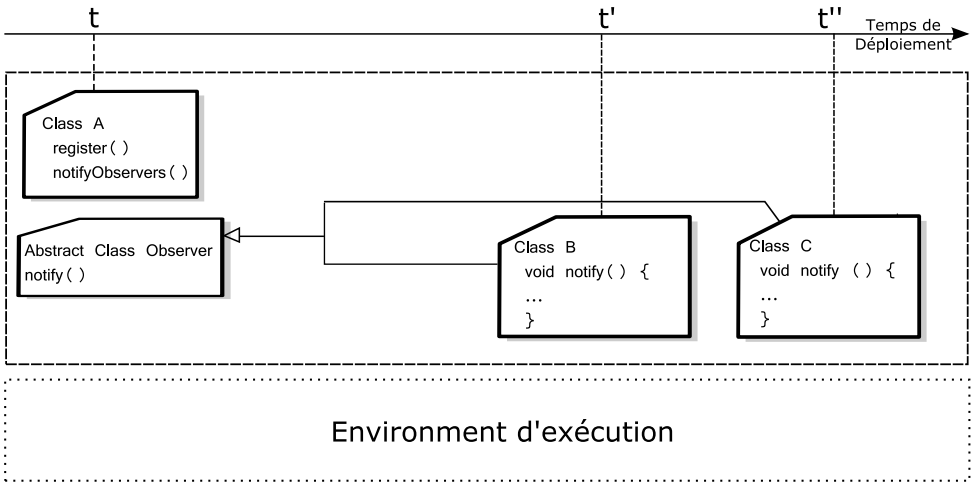


Figure V.8 — *Déploiement d'unité de code dépendantes*

n'implique pas que la méthode *process()* dépasse son budget CPU, b ne sera pas rejetée par l'environnement d'exécution.

Ca qui est important dans ce cas est que l'étape (iii) puisse être répétée sans remettre en cause la garantie ($wcet(a.process()) < R$) assurée par l'environnement d'exécution quand a s'est installé.

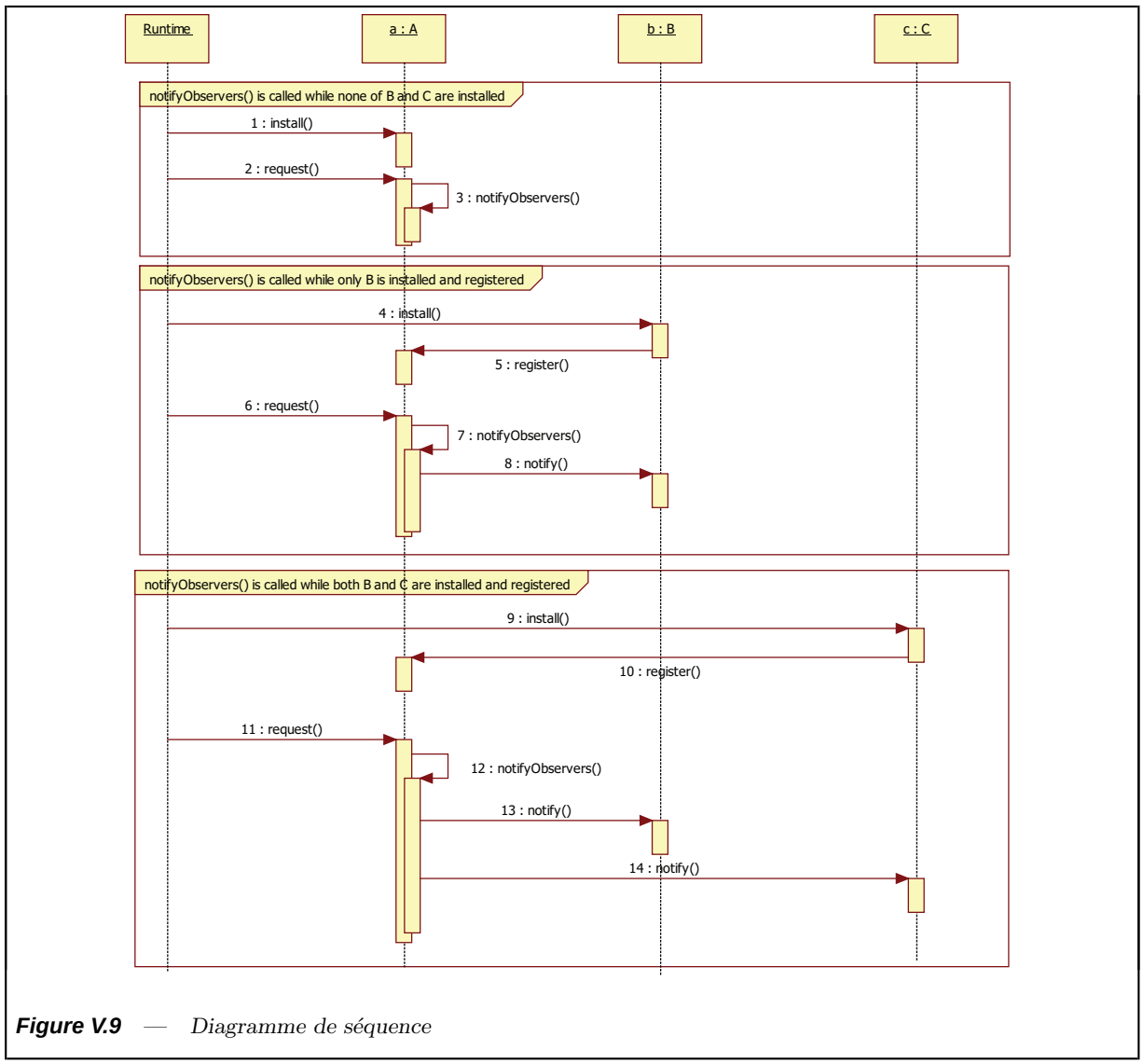


Figure V.9 — *Diagramme de séquence*

VI

Conclusions et perspectives

« *Then world behind and home ahead,*
We'll wander back to home and bed.
Mist and twilight, cloud and shade,
Away shall fade ! Away shall fade !
Fire and lamp, and meat and bread,
And then to bed ! And then to bed ! »

— Tolkien - The Fellowship of the Ring

« *Clore des cycles, fermer des portes, finir des chapitres - peu importe comment nous appelons cela, l'important est de laisser dans le passé les moments de la vie qui sont achevés . . .* »

— Paulo Coelho - *Le Zahir*

A travers cette thèse, nous avons abordé le problème du déploiement sécurisé de logiciels sur des dispositifs ayant des contraintes tant au niveau des ressources matérielles qu'au niveau de la sécurité et la sûreté de fonctionnement. En guise de conclusion à ce mémoire, nous commencerons d'abord, dans la Section VI.1, par résumer et évaluer les différentes contributions apportées par cette thèse afin d'en dresser le bilan. Enfin, nous discuterons dans la Section VI.2 des nouvelles perspectives de recherche sur lesquelles débouchent les contributions que nous avons présenté dans la première section.

VI.1 Bilan

Dans ce mémoire, nous nous sommes préoccupé d'une composante importante et rarement traitée de la sécurité et de la sûreté de fonctionnement qui consiste à maîtriser le temps d'exécution des logiciels destinés à s'exécuter sur un vaste nombre de dispositifs personnels de confiance déjà déployés auprès de leurs utilisateurs. Les dispositifs de confiance auxquels nous faisons référence sont la nouvelle génération de carte à puce, que nous avons désigné par le terme TPD au sein du consortium INSPIRED.

Nous avons commencé par pointer les incompatibilités des solutions de calcul de WCET existantes avec une logique de déploiement a posteriori alors que les TPDs sont en cours d'utilisation par leurs porteurs. De plus, nous avons favorisé dans nos travaux d'apporter les garanties au plus tôt, afin que l'installation échoue si l'environnement d'exécution n'est pas en mesure de répondre aux exigences temporelles exprimées. Ainsi, l'usager est assuré du bon fonctionnement du service auquel il a souscrit en toutes circonstances. En ce sens, nos travaux apportent une réponse aux problème de disponibilité des services, faisant partie des critères de sécurité.

Nous avons mis à profit les avantages de l'analyse statique pour apporter notre première contribution. Celle-ci a permis d'adapter le calcul de WCET en définissant un schéma de calcul distribué entre le producteur de code, d'une part, et le consommateur de code, d'autre part. Le producteur, c'est à dire la station de travail sur laquelle est produit le logiciel, dispose d'importantes ressources matérielles et de tout le temps nécessaire lui permettant d'exécuter des calculs complexes. De son coté, le consommateur (*i.e.* TPD) dispose d'un environnement d'exécution fortement contraint en ressources matérielles et ayant des exigences en terme de sécurité.

Le TPD doit être en mesure de garantir l'innocuité des applications chargées *post-issuance*. En sa qualité de dispositif de confiance, il doit offrir en son sein (*i.e.* sans reposer sur la sécurité d'un tiers) l'infrastructure nécessaire pour éviter qu'un nouveau logiciel ne vienne monopoliser le microprocesseur et ainsi diminuer les performances globales de tout le système et de ses applications. Une des contributions de ce travail consiste à embarquer un vérifieur des bornes des boucles qui permet d'assurer que le TPD peut déterminer une borne majorante sur le temps d'exécution du traitement chargé. Ainsi, le TPD est en mesure d'assurer à chacune de ses applications les ressources de calcul nécessaires pour qu'elles s'exécutent dans les délais impartis quelles que soient les conditions d'utilisation. Cette garantie est un critère important de sécurité qui permet de maintenir la disponibilité des services.

Au moment du déploiement d'un nouveau logiciel sur la plateforme cible et en vue de son admission, le TPD doit être en mesure de vérifier sa capacité à fournir les ressources nécessaires à son fonctionnement. Ce besoin en ressources est décrit par le biais d'un contrat. Une approche consiste à surveiller à l'exécution que le logiciel ne va pas utiliser plus de ressources que spécifié dans son contrat. Une autre consiste à formuler les contrats d'une manière statique et figée sous forme de pré et post conditions et de prouver formellement que l'application s'y tient. Nous avons

esquissé une approche originale de la maîtrise du temps d'exécution en monde ouvert. Selon cette approche, il suffit d'extraire des logiciels à installer les contrats à vérifier, de les stocker sur le système sous forme d'équations qui seront vérifiées par le TPD et qui pourront être affinées au fur et à mesure de l'installation de nouvelles extensions.

Mes travaux ont donné lieu à plusieurs publications dans des évènements d'audience internationale avec comité de sélection : Dans [BRDG04] et [BRG04], nous avons présenté notre schéma distribué de calcul de WCET. Puis, dans [BGSR06], nous avons décrit d'un manière plus formelle le calcul de bornes de boucles et sa vérification. Dans [BGB07], nous avons présenté nos résultats expérimentaux dans les conditions expérimentales au plus près des exigences d'un Consortium formé par les principaux acteurs de l'industrie. Dans [BGGSR07], le bénéfice des contrats pour deux problématiques différentes l'une traitant du flot implicite d'information et l'autre traitant de la disponibilité est démontré. Enfin, dans [BGL+08], nous présentons une approche originale de la contractualisation des ressources.

VI.2 Limites des travaux et perspectives

Dans cette section, nous allons présenter quelques aspects de nos contributions qu'une réflexion plus approfondie permettrait d'enrichir :

Portabilité de la chaîne d'outils La solution que nous avons présenté est portable. D'une part, elle ne repose pas sur un langage particulier mais plutôt sur une forme intermédiaire. D'autre part, elle est indifférente au regard des architectures matérielles sous-jacentes. Néanmoins, les outils que nous avons implémenté aussi bien pour déterminer les bornes des boucles que pour le comptage de cycles ne sont pas portables. Un effort de génie logiciel pourrait être fait afin d'accroître la généricité de ces outils notamment grâce à une description formelle du CPU en VHDL.

Code porteur de preuve Nous avons utilisé l'approche du code porteur de preuve pour borner les boucles sur le producteur de code et de permettre au consommateur de vérifier les informations annoncées. Pour cet effet, nous avons joint au code une preuve inférant le type des variables à chaque point de saut. La preuve sur les bornes des boucles n'est pas la première qui existe sur le système. Elle vient, en effet, se rajouter à la preuve de typage correct qui garantit des propriétés d'intégrité et de confidentialité. En d'autres termes, pour chaque propriété que l'on veut garantir, le code se voit adjoindre une nouvelle preuve. On ne peut que s'inquiéter que la taille du code ne devienne négligeable devant l'amoncellement de preuves qui l'accompagnent.

En outre, nous avons pu démontrer que les preuves ne sont pas forcément indépendantes. Dans le Chapitre IV, nous avons démontré que l'utilisation des informations de typage pouvait améliorer les résultats sur le comptage de cycles. Donc, en vue d'assurer un ensemble de propriétés prédéfinies, il est possible d'imaginer qu'une seule preuve où toute redondance est éliminée vienne se rattacher au code.

Détection de bornes des boucles La solution que nous avons présenté dans le Chapitre III, détermine les bornes des boucles quand ils ont une valeur constante dans le code. Ceci n'est pas toujours le cas en pratique, puisqu'il existe des formes plus complexes comme les boucles triangulaires, ou les boucles dépendant d'un paramètre. Dans ces situations, un système de type paramétrique associé d'un calcul

du pire temps d'exécution symbolique (où le WCET d'une méthode s'exprime sous la forme d'une formule paramétrée) permettrait de réduire le pessimisme. En vue des expériences que nous avons mené, les résultats que nous avons obtenus avec la détection de bornes simples sont tout de même satisfaisants. En effet, dans le contexte des TPD, travailler sur les constantes était suffisant, la prise en compte des types paramétrés aurait peut être généré une preuve plus volumineuse, ce qui serait problématique dans ce contexte. Néanmoins, ces informations plus fines seraient certes nécessaires dans le contexte de code mobile.

Prédiction de la consommation énergétique au pire cas Dans le cas des TPD, la maîtrise du temps d'exécution est capitale au vu des ressources de calcul limitées dont il dispose et la gêne éventuellement causé à l'usager. Pour d'autres petits objets comme les capteurs de terrain, c'est plus l'énergie qui est une ressource critique. Des travaux [FFF08] ont démontré que l'énergie est quantifiable par le comptage des instructions. L'algorithme défini dans le Chapitre III pourrait être appliqué à la prédiction des consommations énergétiques au pire cas au même titre que le temps d'exécution tant à des fins d'optimisation qu'à des fins de sûreté de fonctionnement.

Propriétés de temps constant Pour éviter les attaques temporelles sur les TPD, certaines routines système critiques pour la sécurité doivent avoir un temps d'exécution constant afin de ne pas être décelées par un utilisateur malintentionné. A l'aide de l'outil de prédiction que nous avons fourni, il devient possible d'évaluer le temps pris à travers tous les chemins d'exécution possibles afin d'injecter les instructions nécessaires pour atteindre un temps constant.

Contrats et gestion de conflits Nous avons proposé une solution basée sur les contrats afin de contrôler l'admission de nouveaux logiciels sur le système. Dans cette proposition, nous avons admis une hypothèse stipulant que l'ordre de déploiement et prioritaire. C'est à dire qu'une application B pourrait voir son installation échouer si une application A préalablement déployée sur le système a instauré un contrat contraignant. Cet ordre pourrait ne pas correspondre aux voeux de l'usager qui se verrait ainsi priver des services de l'application B alors qu'il n'a peut être même plus besoin de A. Il s'agit dans ce cas d'un conflit dans le sens où deux applications A et B ne peuvent pas cohabiter sur le TPD au même moment. C'est cette gestion de conflits que nous voyons en perspective des investigations menées sur les contrats. Ainsi, pour gérer ces conflits, on pourrait imaginer de remonter des informations au développeur afin qu'il prenne conscience des causes du rejet de son application. De même, on pourrait donner à l'utilisateur un moyen de décider quelle application il voudrait garder sur son système.

Liste de publications

▷▷ Conférences et ateliers internationaux avec comité de lecture et actes

[1] N. Bel Hadj Aissa, C. Rippert, D. Deville and G. Grimaud. A Distributed WCET Computation Scheme for Smart Card Operating Systems. In *4th International Workshop on Worst-Case Execution Time Analysis, in association with the 16th ECRTS conference*, Catania, Italy, 2004.

[2] N. Bel Hadj Aissa, C. Rippert, and G. Grimaud. Distributing the WCET Computation for Embedded Operating Systems. In *25th IEEE International Real-Time Systems Symposium, Work In Progress Session*, Lisbon, Portugal, 2004.

[3] N. Bel Hadj Aissa, G. Grimaud, and D. Simplot-Ryl. A Distributed and Verifiable Loop Bounding Algorithm for WCET Computation on Constrained Embedded Systems. In *14th International Conference on Real-Time and Network System*, Poitiers, France, 2006.

[4] N. Bel Hadj Aissa, G. Grimaud, and V. Bénony. Bringing Worst Case Execution Time Awareness to an Open Smart Card OS. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Daegu, Korea, 2007.

[5] N. Bel Hadj Aissa, D. Ghindici, G. Grimaud, and I. Simplot-Ryl. Contracts as a support to static analysis of open systems. In *1st Workshop on Formal Languages and Analysis of Contract-Oriented Software*, Oslo, Norway, 2007.

▷▷ Manifestations nationales avec comité de lecture et actes

[6] N. Bel Hadj Aissa and G. Grimaud. Calcul de temps d'exécution au pire cas pour code mobile. *Actes des 1ères Rencontres Jeunes Chercheurs en Informatique Temps Réel*, Nancy, France, 2005.

▷▷ En cours de soumission

[7] N. Bel Hadj Aissa, G. Grimaud, A. Linke, L. Manteau, and J.-J. Vandewalle. Secure post-issuance software deployement in Trusted Personal Devices : Ressource availability issues. In *special issue of the IOS PRESS Journal of Computer Security on IST-FP6 research in security, dependability and trust.*

Bibliographie

[ABC+95] J. Arlat, J.-P. Blanquart, A. Costes, Y. Crouzet, Y. Deswarte, J.-C. Fabre, H. Guillermain, M. Kaâniche, K. Kanoun, J.-C. Laprie, J.-C. Mazet, D. Powell, C. Rabéjac, and P. Thévenod. *Guide de la Sûreté de Fonctionnement*. Cépaduès-Editions, 1995.

[ABC+07] A. Armbruster, J. Baker, A. Cunei, C. Flack, D. Holmes, F. Pizlo, E. Pla, M. Prochazka, and J. Vitek. A real-time java virtual machine with applications in avionics. *ACM Transactions on Embedded Computing Systems*, 7(1) :1–49, 2007.

[ACGDZJ04] R.-M. Amadio, S. Coupet-Grimal, S. Dal-Zilio, and L. Jakubiec. A functional scenario for bytecode verification of resource bounds. In *18th International Workshop on Computer Science Logic*, pages 265–279, Karpacz, Poland, 2004.

[ARM04] ARM Ltd. *ARM7TDMI Technical Reference Manual, Revision r4p1 , Document DDI0210C*, 2004.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers : Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[BB06] A. Betts and G. Bernat. Tree-based wcet analysis on instrumentation point graphs. In *9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 558–565, Washington, DC, USA, 2006. IEEE Computer Society.

[BBMP00] I. Bate, G. Bernat, G. Murphy, and P. Puschner. Low-level analysis of a portable java byte code wcet analysis framework. In *7th International Conference on Real-Time Computing Systems and Applications*, pages 39–48, December 2000.

[BBW00] G. Bernat, A. Burns, and A. Wellings. Portable worst-case execution time analysis using java byte code. In *12th EuroMicro Conference on Real-Time Systems*, Stockholm, June 2000.

[BG05] N. BelHadjAissa and G. Grimaud. Calcul de temps d'exécution au pire cas pour code mobile. In *Actes des 1ères Rencontres Jeunes Chercheurs en Informatique Temps Réel*, pages 59–62, Nancy, France, September 2005.

[BGB07] N. BelHadjAissa, G. Grimaud, and V. Benony. Bringing worst case execution time awareness to an open smart card os. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Daegu, Korea, August 2007.

[BGGSR07] N. BelHadjAissa, D. Ghindici, G. Grimaud, and I. Simplot-Ryl. Contracts as a support to static analysis of open systems. In *1st Workshop on Formal Languages and Analysis of Contract-Oriented Software*, Oslo, Norway, 2007.

[BGL+08] N. BelHadjAissa, G. Grimaud, A. Linke, L. Manteau, and J.-J. Vandewalle. Secure post-issuance software deployement in trusted personal devices : Ressource availability issues. *Special Issue of the IOS PRESS Journal of Computer Security on IST-FP6 research in security, dependability and trust*, 2008. Submitted.

[BGSR06] N. BelHadjAissa, G. Grimaud, and D. Simplot-Ryl. A distributed and verifiable loop bounding algorithm for wcet computation on constrained embedded systems.

In *14th International Conference on Real-Time and Network System*, Poitiers, France, March 2006.

[Boo51] A. Booth. A signed binary multiplication technique. *Quarterly Journal of Mechanics and Applied Mathematics*, 4(2) :236–240, 1951.

[Bou07] Bound-t execution time analyzer, 2007. http://www.tidorum.fi/bound-t/.

[BPB02] I. Bate, P. Puschner, and G. Bernat. Java virtual-machine support for portable worst-case execution-time analysis. In *5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 83, 2002.

[BRDG04] N. BelHadjAissa, C. Rippert, D. Deville, and G. Grimaud. A distributed wcet computation scheme for smart card operating systems. In *4th International Workshop on Worst-Case Execution Time Analysis, in association with the 16th ECRTS conference*, Catania, Italy, June 2004.

[BRG04] N. BelHadjAissa, C. Rippert, and G. Grimaud. Distributing the wcet computation for embedded operating systems. In *25th IEEE International Real-Time Systems Symposium, Work In Progress Session*, Lisbon, Portugal, December 2004.

[BS96] D. F. Bacon and P. F. Sweeney. Fast static analysis of c++ virtual function calls. In *11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 324–341, 1996.

[BSF04] M. Q. Beers, C. Stork, and M. Franz. Efficiently verifiable escape analysis. In *9th European Conference on Object-Oriented Programming (ECOOP04)*, pages 75–95, 2004.

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. *ACM SIGOPS Operating Systems Review*, 29(5) :267–283, 1995.

[CBW96] R. Chapman, A. Burns, and A. Wellings. Combining static worst-case timing analysis and program proof. *Real-Time Systems*, 11(2) :145–171, 1996.

[CC77] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CFH+98] A. Carzaniga, A. Fuggetta, R. S. Hall, A. van der Hoek, D. Heimbigner, and A. L. Wolf. A characterization framework for software deployment technologies. Technical Report CU-CS-857-98, Department of Computer Science, University of Colorado, April 1998.

[CG04] M. Corti and T. Gross. Approximation of the worst-case execution time using structural analysis. In *4th ACM international conference on Embedded software*, pages 269–277, 2004.

[CKH99] B. Calder, C. Krintz, and U. Hölzle. Reducing transfer delay using java class file splitting and prefetching. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA99)*, pages 276–291, New York, NY, USA, 1999. ACM Press.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[CLR07] Microsoft corporation. the .NET Common Language Runtime., April 2007. See http://msdn.microsoft.com/net/.

[Col01] A. Colin. *Estimation de temps d'exécution au pire cas par analyse statique et application aux systèmes d'exploitation temps-réel*. PhD thesis, Université de Rennes I, october 2001.

[CP00] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 2 :249–274, 2000.

[CP01a] A. Colin and I. Puaut. A modular and retargetable framework for tree-based wcet analysis. In *13th Euromicro Conference on Real-Time Systems*, pages 37–44, Delft, Netherlands, June 2001.

[CP01b] A. Colin and I. Puaut. Worst case execution time analysis of the rtems real-time operating system. In *13th Euromicro Conference on Real-Time Systems*, pages 191–198, 2001.

[CvE98] G. Czajkowski and T. von Eicken. Jres : a resource accounting interface for java. *SIGPLAN Notices*, 33(10) :21–35, 1998.

[Cza00] G. Czajkowski. Application isolation in the java virtual machine. In *15th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 354–366, New York, NY, USA, 2000. ACM.

[DDHV03] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for java. In *ACM SIGPLAN 2003 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 149–168. ACM Press, 2003.

[DGC95] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *9th European Conference on Object-Oriented Programming (ECOOP95)*, pages 77–101, London, UK, 1995. Springer-Verlag.

[DGGJ03] D. Deville, A. Galland, G. Grimaud, and S. Jean. Smartcard operating systems : Past, present and future. In *5th NORDU/USENIX Conference*, pages 14–28, Västerås, Sweden, 2003.

[DHSR05] D. Deville, Y. Hodique, and I. Simplot-Ryl. Safe collaboration in extensible operating systems : A study on real time extensions. *International Journal of Computers and Applications*, 1 :20–26, january 2005.

[Dij59a] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1 :269–271, 1959.

[Dij59b] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1 :269–271, 1959.

[DLM+00] K. Driesen, P. Lam, J. Miecznikowski, F. Qian, and D. Rayside. On the predictability of invoke targets in java byte code. In *2nd Annual Workshop on Hardware Support for Objects and Microarchitectures for Java*, pages 6–10, Austin, Texas, September 2000.

[EGV95] R. J. E. Gamma, R. Helm and J. Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[EMTP+96] C. Eriksson, J. Mäki-Turja, K. Post, M. Gustafsson, J. Gustafsson, K. Sandström, and E. Brorsson. An overview of realtimetalk, a design framework for real-time systems. *Journal of Parallel and Distributed Computing*, 36(1) :66–80, 1996.

[Eng98] D. R. Engler. *The Exokernel operating system architecture*. PhD thesis, Massachusetts Institute of Technology (MIT), 1998.

[Erm03] A. Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, June 2003.

[FBCG07] D. Frampton, D. Bacon, P. Cheng, and D. Grove. Generational real-time garbage collection. In *21st European Conference on Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 101–125. Springer, 2007.

[FFF08] N. Fournel, A. Fraboulet, and P. Feautrier. Embedded software energy characterization : using non-intrusive measures to annotate application source code. *Journal of Embedded Computing*, in press, 2008.

[Flo67] R. W. Floyd. Assigning meanings to programs. *Mathematical Aspects of Computer Science*, 1967.

[FMC+07] C. Ferdinand, F. Martin, C. Cullmann, M. Schlickling, I. Stein, S. Thesing, and R. Heckmann. New Developments in WCET Analysis. In T. Reps, M. Sagiv, and J. Bauer, editors, *Program Analysis and Compilation. Theory and Practice. Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday*, volume 4444 of *LNCS*, pages 12–52. Springer Verlag, 2007.

[Foo88] B. Foote. Designing to facilitate change with object-oriented frameworks. Master's thesis, University of Illinois at Urbana-Champaign, 1988.

[GESL06] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *27th IEEE Real-Time Systems Symposium*, Rio de Janeiro, Brazil, December 2006.

[GHSR07] G. Grimaud, Y. Hodique, and I. Simplot-Ryl. On the use of metatypes for safe embedded operating system extension. *International Journal of Parallel, Emergent and Distributed Systems*, 22 :1–13, 2007.

[GLV99] G. Grimaud, J.-L. Lanet, and J.-J. Vandewalle. Facade : A typed intermediate language dedicated to smart cards. In *7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 476–493, 1999.

[GMTB96] J. Gustafsson, J. Mäki-Turja, and E. Brorson. Benefits of type inference for an object-oriented real-time language. *OOPS Messenger*, 7(1), January 1996.

[Gri00] G. Grimaud. *CAMILLE : un Système d'Exploitation Ouvert pour Carte à Microprocesseur*. PhD thesis, Université des Sciences et Technologies de Lille, 2000.

[Gro06a] J. . E. Group. Jsr 202 : Java class file specification update. http://jcp.org/en/jsr/detail?id=202, 2006.

[Gro06b] T. C. Group. Tcg tpm specification version 1.2, revisions 62-94 (design principles, structures of the tpm, and commands), 2003-2006. https://www.trustedcomputinggroup.org/specs/TPM/.

[GS05] P. Giambiagi and G. Schneider. Memory consumption analysis of java smart cards. In *Proceedings of XXXI Latin American Informatics Conference*, page 12, Cali, Colombia, 2005.

[Gus94a] J. Gustafsson. *Calculation of Execution Times in Object-Oriented Real-Time Software - A study focused on RealTimeTalk*. PhD thesis, Department of Machine Elements, The Royal Institute of Technology, Sweden, 1994.

[Gus94b] J. Gustafsson. Calculation of execution times in realtimetalk-an object-oriented language for real-time. *words*, 00 :153, 1994.

[Gus02] J. Gustafsson. Worst case execution time analysis of object-oriented programs. In *7th IEEE International Workshop on Object-oriented Real-time Dependable Systems*, pages 71–76, San Diego,CA, January 2002.

[HALGP05] M. V. Hermenegildo, E. Albert, P. López-García, and G. Puebla. Abstraction carrying code and resource-awareness. In *7th ACM SIGPLAN international conference on Principles and Practice of Declarative Programming (PPDP05)*, pages 1–11, New York, NY, USA, 2005. ACM Press.

[Hav97] P. Havlak. Nesting of reducible and irreducible loops. *ACM Transaction on Programming Language and Systems*, 19(4) :557–567, 1997.

[HBW02a] E. Y.-S. Hu, G. Bernat, and A. Wellings. Addressing dynamic dispatching issues in wcet analysis for object-oriented hard real-time systems. In *5th IEEE International Symposium on Object-oriented Real-time distributed Computing*, pages 109–116, Crystal City, VA, USA, 2002.

[HBW02b] E. Y.-S. Hu, G. Bernat, and A. Wellings. A static timing analysis environment using java architecture for safety critical real-time systems. In *7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, page 77, 2002.

[HGSC04] M. Huisman, D. Gurov, C. Sprenger, and G. Chugunov. Checking absence of illicit applet interactions : A case study. In *7th International Conference in Fundamental Approaches to Software Engineering*, volume 2984 of *Lecture Notes in Computer Science*, pages 84–98, Barcelona, Spain, 2004. Springer.

[HKM+99] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. Plan : a packet language for active networks. *ACM SIGPLAN Notices*, 34(1) :86–93, 1999.

[HLS00] N. Holsti, T. Långbacka, and S. Saarine. Worst-case execution time analysis for digital signal processors. In *10th European Signal Processing Conference*, Tampere, Finland, September 2000.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580, 1969.

[HS02] N. Holsti and S. Saarine. Status of the bound-t tool. In *2nd International Workshop on Worst-Case Execution Time Analysis*, Technical University of Vienna, Austria, June 2002.

[HSR+00] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2/3) :129–156, May 2000.

[HSRW98] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *4th Real-Time Technology and Applications Symposium*, pages 12–21, Denver, CO, 1998.

[HWB03] E. Y.-S. Hu, A. Wellings, and G. Bernat. Gain time reclaiming in high performance real-time java systems. In *6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 249, 2003.

[Ins99] E. T. S. Institute. Digital cellular telecommunications system (phase 2+) ; specification of the subscriber identity module mobile equipment (sim-me) interface (gsm 11.11 version 6.2.0 release 1997), 1999.

[Ins04] Integrated secure platform for interactive trusted personal devices, 2004. http://www.inspiredproject.com.

[Jav07] Java card 2.2.2 specification, 2007. http://java.sun.com/products/javacard/.

[Jav08] Java card specification 3.0 final release for documentation, 2008. http://java.sun.com/javacard/3.0/.

[Kil73] G. A. Kildall. A unified approach to global program optimization. In *1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206, New York, NY, USA, 1973. ACM.

[KM01] K. Krishna and M. Montgomery. A simple(r) interface distribution mechanism for java card. In *Revised Papers from the First International Workshop on Java on Smart Cards : Programming and Security*, pages 114–120, London, UK, 2001. Springer-Verlag.

[LB98] S. Liang and G. Bracha. Dynamic class loading in the java virtual machine. In *13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA98)*, pages 36–44, New York, NY, USA, 1998. ACM Press.

[LBJ+95] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon, and C. S. Kim. An accurate worst case timing analysis for risc processors. *IEEE Transactions on Software Engineering*, 21(7) :593–604, 1995.

[Ler03] X. Leroy. Java bytecode verification : Algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4) :235–269, 2003.

[Lim06] M. Limited. Multos developer's guide mao-doc-tec-005 v1.35. http://www.multos.com/downloads/technical/mdg.pdf, 2006.

[LM95] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Workshop on Languages, Compilers & Tools for Real-Time Systems*, 1995.

[LPC+08] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, and D. M. Zimanderman. *JML Reference Manual (DRAFT)*, 2008. http://www.jmlspecs.org/OldReleases/jmlrefman.pdf.

[LR91] W. Landi and B. G. Ryder. Pointer-induced aliasing : A problem classification. In *18th Annual ACM Symposium on Principles of Programming Languages (POPL1991)*, pages 93–103, January 1991.

[LS99] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2-3) :183–207, November 1999.

[LY99] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[MAL04] M. Montgomery, A. Ali, and K. Lu. Secure network card. implementation of a standard network stack in a smart card. In *6th International Conference on Smart Card Research and Advanced Applications*, pages 193–208, 2004.

[Mey92] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10) :40–51, 1992.

[Mic00] S. Microsystems. JAVA *2 Platform Micro Edition Building Blocks for Mobile Devices - White Paper on* KVM *and the Connected, Limited Device Configuration (*CLDC*)*, 2000. http://java.sun.com/products/cldc/wp/KVMwp.pdf.

[MM03] K. Markantonakis and K. Mayes. An overview of the globalplatform smart card specification. *Information Security Technical Report*, 8 :17–29, 2003.

[MM08] K. Mayes and K. Markantonakis. *Smart Cards, Tokens, Security and Applications*. Springer Verlag, 2008.

[Mue00] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems - Special issue on worst-case execution-time analysis*, 18(2-3) :217–247, 2000.

[MWCG99] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system f to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3) :527–568, 1999.

[MY02a] A. K. Mok and W. Yu. Enforcing resource bound safety for mobile snmp agents. In *18th Annual Computer Security Applications Conference*, pages 69–77, Las Vegas, Nevada, USA, December 2002.

[MY02b] A. K. Mok and W. Yu. Tinman : A resource bound security checking system for mobile code. In *7th European Symposium on Research in Computer Security*, pages 178–193, Zurich, Switzerland, October 2002.

[MY03] A. K. Mok and W. Yu. Formal specification and verification of resource bound security using pvs. In *International Symposium on Software Security*, volume 3233 of *LNCS*, pages 113–133, Tokyo, Japan, November 2003. Springer.

[Nec97] G. C. Necula. Proof-carrying code. In *24th ACM Symposium on Principles of Programming Languages*, January 1997.

[NL96] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *2nd USENIX symposium on Operating systems design and implementation (OSDI96)*, pages 229–243, New York, NY, USA, 1996. ACM Press.

[NL98] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *1998 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 333–344. ACM, 1998.

[Org99] I. S. Organisation. Integrated circuit(s) cards with contacts, parts 1 to 9, 1987-1999.

[Org01] I. S. Organisation. Iso/iec 14443 : Identification cards - contactless integrated circuit cards - proximity cards, 2001.

[Par93] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1) :31–62, 1993.

[PB01] P. Puschner and G. Bernat. Wcet analysis of reusable portable code. In *13th Euromicro Conference on Real-Time Systems*, page 45, 2001.

[Pic05] D. Pichardie. *Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certiés*. PhD thesis, Université Rennes 1, Rennes, France, december 2005.

[PK89a] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Journal of Real-Time Systems*, 1(2) :159–176, 1989.

[PK89b] P. Puschner and C. Koza. Calculating the maximum, execution time of real-time programs. *Real-Time Systems*, 1(2) :159–176, 1989.

[PP07] I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard real-time systems : a quantitative comparison. In *Conference on Design, Automation and Test in Europe*, pages 1484–1489, San Jose, CA, USA, 2007. EDA Consortium.

[Pre29] M. Presburger. Uber die volstandigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation. In *C. R. 1er congres des Mathématiciens des pays slaves*, pages 92–101, Varsovie, 1929.

[PS91] C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing schema. *Computer*, 24(5) :48–57, 1991.

[Pug91] W. Pugh. The omega test : a fast and practical integer programming algorithm for dependence analysis. In *4th ACM/IEEE conference on Supercomputing*, pages 4–13, 1991.

[Ram99] G. Ramalingam. Identifying loops in almost linear time. *ACM Transaction on Programming Language and Systems*, 21(2) :175–188, 1999.

[Ram02] G. Ramalingam. On loops, dominators, and dominance frontiers. *ACM Transaction on Programming Language and Systems*, 24(5) :455–490, September 2002.

[Ric53] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2) :358–366, 1953.

[Ros03] E. Rose. Lightweight bytecode verification. *Journal of Automated Reasoning (JAR)*, 31(3-4) :303–334, 2003.

[Ryd03] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *12th International Conference on Compiler Construction (CC2003)*, volume 2622 of *Lecture Notes in Computer Science*, pages 126–137. Springer, April 2003.

[SA00a] F. Stajano and R. Anderson. The grenade timer : Fortifying the watchdog timer against malicious mobile code. In *7th International Workshop on Mobile Multimedia Communications*, 2000.

[SA00b] F. Stappert and P. Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Euromicro Journal of Systems Architecture*, 46(4) :339–355, 2000.

[SBC00] V. C. Sreedhar, M. Burke, and J.-D. Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *ACM SIGPLAN 2000 conference on Programming language design and implementation (PLDI00)*, pages 196–207, New York, NY, USA, 2000. ACM.

[Sch00] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1) :30–50, 2000.

[Sch04] G. Schneider. A constraint-based algorithm for analysing memory usage on java cards. Technical Report RR-5440, INRIA, December 2004.

[SEE01] F. Stappert, A. Ermedahl, and J. Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *4th international conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 132–140, 2001.

[SGL96] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. Identifying loops using dj graphs. *ACM Transaction on Programming Language and Systems*, 18(6) :649–658, 1996.

[Sha80] M. Sharir. Structural analysis : A new approch to flow analysis in optimizing compilers. *Computer Languages*, 5(3) :141–153, 1980.

[SHR+00] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Valle-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for java. In *15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 264–280, 2000.

[Sie07] F. Siebert. Realtime garbage collection in the JamaicaVM 3.0. In *5th international workshop on Java Technologies for Real-time and Embedded Systems*, pages 94–103, New York, NY, USA, 2007. ACM.

[Sma02] Y. Smaragdakis. Layered development with (unix) dynamic libraries. In *7th International Conference on Software Reuse*, pages 33–45, London, UK, 2002. Springer-Verlag.

[Sta88] J. A. Stankovic. Misconceptions about real-time computing : a serious problem for next-generation systems. *IEEE Computer*, 21(10) :10–19, October 1988.

[SWE07] Worst case execution times (wcet) project homepage, 2007. www.mrtc.mdh.se/projects/wcet.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2) :285–309, 1955.

[The00] H. Theiling. Extracting safe and precise control flow from binaries. In *7th International Conference on Real-Time Systems and Applications*, pages 23–30, Washington, DC, USA, 2000. IEEE Computer Society.

[The02] H. Theiling. Ilp-based interprocedural path analysis. In *2nd International Conference on Embedded Software (EMSOFT02)*, pages 349–363, 2002.

[Tur37] A. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proc. Lecture Notes in Mathematics, Series 2*, 42 :230–265, 1936-1937.

[VHMW01] E. Vivancos, C. Healy, F. Mueller, and D. Whalley. Parametric timing analysis. In *ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, pages 88–93, 2001.

[WEE+08a] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools. 7(3), 2008. ACM Transactions on Embedded Computing Systems (TECS).

[WEE+08b] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem — overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3) :1–53, 2008.

[WLAG93] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *14^{th} ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, North Carolina, United States, 1993.

[XN05] J. Xue and P. H. Nguyen. Completeness analysis for incomplete object-oriented programs. In *14^{th} International Conference on Compiler Construction (CC2005)*, volume 3443 of *Lecture Notes in Computer Science*, pages 271–286. Springer, April 2005.

Table des figures

Liste des tableaux

Liste des abréviations

ACC	*Abstraction Carrying Code*
APDU	*Application Protocol Data Unit*
API	*Application Programming Interface*
ASIC	*Application Specific Integrated Circuit*
CFG	*Control Flow Graph*
CLDC	*Connected Limited Device Configuration*
CLR	*Common Language Runtime*
DES	*Data Encryption Standard*
DMA	*Direct Access Memory*
EEPROM	*Electrically-Erasable Programmable Read-Only Memory*
FPGA	*Field Programmable Gate Array*
GPRS	*General Packet Radio Service*
GSM	*Global System for Mobile communications*
HTTP	*Hyper Text Transfer Protocol*
HTTPS	*Secure Hyper Text Transfer Protocol*
INSPIRED	*INtegrated Secure Platform for Interactive tRusted pErsonel Device*
IPET	*Implicit Path Enumeration Technique*
ISO	*International Organization for Standardization*
JDK	*Java Development Kit*

KVM *Kilo Virtual Machine*

MMC *Multi Media Card*

MMU *Memory Management Unit*

MULTOS *Multi Application Operating System*

NFC *Near Field Communication*

PCC *Proof Carrying Code*

PDA *Personal Digital Assistant*

PIN *Personal Identification Number*

PIV *Personal Identification Verification*

POPS Petits Objets Portables et Sécurisés

RAM *Random Access Memory*

RFID *Radio Frequency IDentification*

RISC *Reduced Instruction Set Computer*

RNG *Random Number Generator*

ROM *Read Only Memory*

SIM *Subscriber Identity Module*

TAL *Typed Assembly Language*

TCB *Trusted Computing Base*

TCP/IP *Transport Control Protocol / Internet Protocol*

TLB *Translation Lookaside Buffer*

TPD *Trusted Personal Device*

UAL Unité Arithmétique et Logique

USB *Universal Serial Bus*

WCET *Worst Case Execution Time*

TITRE: *Maîtrise du temps d'exécution de logiciels déployés dans des dispositifs personnels de confiance*
RÉSUMÉ: La prolifération de petits équipements ouverts tels que les dispositifs personnels de confiance a favorisé l'essor des environnements d'exécution dynamiquement adaptables. Ainsi, de nouveaux logiciels peuvent être déployés à la volée après que les équipements ne soient délivrés à leurs porteurs. Par nos travaux, nous aspirons à garantir que chaque nouveau logiciel, dont le déploiement aboutit, est en mesure de délivrer les réponses aux requêtes qui lui sont adressées dans un délai maximal préalablement établi. Les garanties apportées sont cruciales tant en terme de sûreté de fonctionnement que de sécurité. À cet effet, nous avons proposé de distribuer le calcul du temps d'exécution au pire cas à la manière des approches de codes porteurs de preuve. Le fournisseur de code se voit attribuer les calculs gourmands en ressources matérielles ne supposant aucune connaissance préalable de l'environnement d'exécution sur lequel s'exécutera son logiciel, en l'occurrence la détermination des bornes des boucles. Quant au consommateur, il vérifie grâce à la preuve les bornes inférées par le fournisseur et calcule le temps d'exécution au pire cas. Nous avons évalué expérimentalement le bien-fondé de notre démarche sur une architecture matérielle et logicielle répondant aux exigences des dispositifs personnels de confiance. Enfin, nous nous sommes préoccupés du cas où plusieurs logiciels, émanant d'entités différentes, coexistent. Nous avons mis l'accent sur l'impact de l'interaction entre ces logiciels sur les garanties préalablement apportées par le système sur le temps d'exécution au pire cas et nous avons ébauché une solution basée sur les contrats pour maintenir ces garanties.

MOTS CLÉS: Systèmes embarqués, Sécurité des logiciels, Déploiement, Calcul du temps d'exécution au pire cas

---

TITLE: *Secure software deployment on Trusted Personal Devices :*
ABSTRACT: The proliferation of small and open objects such as personal trusted devices has encouraged the spread of dynamically adaptable runtime environments. Thus, new software can be deployed on the fly after the devices are delivered to their holders. Through our work, we aim to ensure that each new software, whose deployment is successful, will be able to deliver responses within a maximum delay previously established. These guarantees are crucial in terms of safety and security. To this end, we propose to distribute the computation of worst case execution time. Our solution embraces a proof carrying code approach making distinction between a powerful but untrusted computer used to produce the code, and a safe but resource constrained code consumer. The producer does not assume any prior knowledge of the runtime environment on which its software will be executed. The code is statically analyzed to extract loop bounds and a proof containing this information is joint to the software. By a straightforward inspection of the code, the consumer can verify the validity of the proof and compute the global worst case execution time. We experimentally validate our approach on a hardware and software architecture which meets the requirements of trusted personal devices. Finally, we address the challenges raised when software from different service providers potentially untrusted can coexist and interact in a single device. We focus on the impact of the interaction between different software units on the guarantees previously given by the system on the worst case execution time and we outline a solution based on contracts to maintain these guarantees.

KEYWORDS: Embedded systems, Security, Deployment, Worst case execution time