(19) RÉPUBLIQUE FRANÇAISE

INSTITUT NATIONAL
DE LA PROPRIÉTÉ INDUSTRIELLE

PARIS

(11) **N° de publication :** **2 827 974**
(à n'utiliser que pour les commandes de reproduction)

(21) **N° d'enregistrement national :** **01 10210**

(51) Int Cl[7] : **G 06 F 9/30**

(12) DEMANDE DE BREVET D'INVENTION A1

(22) **Date de dépôt :** 26.07.01.

(30) **Priorité :**

(43) **Date de mise à la disposition du public de la demande :** 31.01.03 Bulletin 03/05.

(56) **Liste des documents cités dans le rapport de recherche préliminaire :** *Se reporter à la fin du présent fascicule*

(60) **Références à d'autres documents nationaux apparentés :**

(71) **Demandeur(s) :** *TRUSTED LOGIC Société anonyme* — FR.

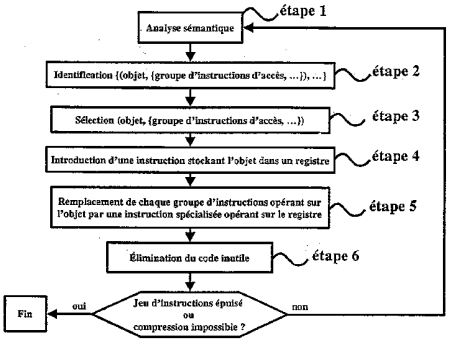
(72) **Inventeur(s) :** LE METAYER DANIEL, MARLET RENAUD, VENET ARNAUD et FREY ALEXANDRE.

(73) **Titulaire(s) :**

(74) **Mandataire(s) :** CABINET MOUTARD.

(54) PROCEDE POUR LA COMPRESSION D'UN CODE INTERPRETE PAR ANALYSE SEMANTIQUE.

(57) Le procédé selon l'invention est utilisable au compactage de programmes en code objet intermédiaire exécutable dans un système embarqué à faibles ressources matérielles. Il comprend l'exploitation d'une information sémantique pour modifier le code objet intermédiaire du programme et l'élimination des instructions du code objet intermédiaire du programme devenues inutiles du fait de cette modification.



FR 2 827 974 - A1

2827974

La présente invention concerne un procédé de compression de code objet intermédiaire par analyse sémantique utilisable, notamment mais non exclusivement, au compactage de programmes en code objet intermédiaire exécutable dans un système embarqué à faibles ressources matérielles tel qu'une carte à puce ou qu'un terminal de paiement comportant un microprocesseur et un environnement matériel et logiciel incluant notamment :

- un compilateur servant à obtenir le code objet intermédiaire à partir d'un programme source, et
- un interpréteur qui effectue une interprétation logicielle des instructions standard du code objet intermédiaire en instructions directement exécutables par le microprocesseur.

D'une manière générale, on sait que, par opposition à l'analyse syntaxique, l'analyse sémantique est un ensemble de techniques permettant d'extraire des propriétés concernant l'exécution d'un programme. Cette technique, qui repose sur des bases théoriques bien connues [CC77] " Patrick Cousot & Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238--252, Los Angeles, California, 1977. ACM Press, New York, NY, USA. ", consiste à construire

un modèle mathématique de l'exécution du programme sous forme d'équations, dites sémantiques, et à appliquer des algorithmes de résolution automatique de ces équations permettant d'extraire l'information désirée. Les techniques d'analyse sémantique permettent par exemple de calculer les intervalles de variation des variables scalaires d'un programme, information utile à des fins de vérification pour prédire les erreurs de dépassement de bornes de tableaux ; elles permettent également de déduire les objets référencés (accédés ou modifiés) à des points de programme donnés.

L'analyse syntaxique est, quant à elle, un ensemble de techniques permettant d'extraire des propriétés concernant la syntaxe d'un programme. Les techniques d'analyse syntaxique permettent par exemple de détecter des répétitions de séquences d'instructions données dans un programme. Contrairement aux analyses sémantiques, les analyses syntaxiques ne reposent pas sur des modèles d'exécution des programmes et n'apportent pas d'informations de nature opérationnelle.

Il s'avère qu'à l'heure actuelle, toutes les techniques existantes en matière de compression de code reposent sur des analyses syntaxiques : Le code objet du programme est traité comme une donnée brute à compresser mais la sémantique du programme (ou son modèle d'exécution) n'intervient jamais dans ces procédés de compression. Certains de ces procédés, tels que par exemple celui qui se trouve décrit dans le brevet FR 2 785 695, reposent sur la recherche de motifs syntaxiques dans le code et sur une factorisation de ces motifs.

Au contraire, l'invention propose d'effectuer une compression en exploitant une information sémantique pour modifier le code objet intermédiaire du programme et éliminer les instructions du code intermédiaire devenues inutiles du fait de cette modification.

Plus particulièrement, le procédé selon l'invention consiste à identifier les accès aux objets manipulés par le programme par une analyse sémantique du code objet du programme et à utiliser cette information pour remplacer dans le code du programme les groupes d'instructions accédant à ces objets par des instructions spécialisées permettant un accès direct aux objets considérés. La réduction de la taille du code provient du remplacement d'un groupe d'instructions par une instruction spécialisée de taille inférieure et/ou par élimination dans le code du programme d'instructions devenues inutiles du fait de l'introduction des instructions spécialisées précédentes.

Les instructions spécialisées employées par ce procédé doivent être intégrées dans l'interpréteur du langage s'il n'en dispose pas.

Conformément au procédé selon l'invention, la compression de programmes code objet exécutable par un interpréteur se décompose en deux parties.

Premièrement, s'il n'en dispose pas déjà, on étend l'interpréteur avec des registres spécifiques et des versions spécialisées de certaines instructions de l'interpréteur qui opèrent implicitement sur ces registres spécifiques plutôt que sur leurs arguments ordinaires (y compris les instructions de simples lecture et écriture d'objets).

Deuxièmement, étant donné un interpréteur disposant de registres spécifiques et de versions spécialisées de certaines instructions de l'interpréteur qui opèrent implicitement sur ces registres spécifiques plutôt que sur leurs arguments ordinaires (y compris les instructions de simples lecture et écriture d'objets), le procédé de compression d'un programme code objet exécutable par cet interpréteur est défini par les étapes suivantes :

- une première étape dans laquelle on effectue une analyse sémantique du programme qui met en évidence les objets accédés en chaque point du programme,
- une deuxième étape dans laquelle, à l'aide de l'information calculée par l'analyse sémantique de la première étape, pour tout objet manipulé par le programme, on identifie dans le programme les occurrences de différents groupes d'instructions qui opèrent sur cet objet (c'est-à-dire qui le lisent et éventuellement le transforment) et dont on dispose d'une version spécialisée (c'est-à-dire de sémantique équivalente) en terme d'un registre spécifique dans le jeu d'instruction de l'interpréteur,
- une troisième étape dans laquelle on soumet lesdits objets à un test de comparaison de supériorité d'une fonction d'au moins le nombre d'occurrences des groupes d'instructions correspondants dans ledit programme (comme déterminé à la deuxième étape) à une valeur de référence et, sur réponse positive audit test, pour chaque objet satisfaisant à ladite étape de test,
- une quatrième étape dans laquelle, en un point du programme qui précède, dans toutes les exécutions possibles du programme, les occurrences de groupes d'instructions associées audit objet, on introduit le code correspondant à l'affectation dudit objet dans un des registres spécifiques de l'interpréteur,
- une cinquième étape dans laquelle on remplace dans le programme les groupes d'instruction à chacune desdites occurrences par l'instruction spécialisée par rapport audit registre qui leur correspond,
- une sixième étape dans laquelle on élimine du code du programme les instructions et les constructions qui sont rendues inutiles ou redondantes du fait des transformations du programme indiquées aux quatrième et cinquième étapes.

La taille du programme ainsi transformé est inférieure à celle du programme initial, cette propriété étant garantie par le procédé de choix utilisé dans la troisième étape.

Un mode de mise en œuvre du procédé selon l'invention sur des programmes JavaCard destinés à des cartes à puce sera décrit ci-après, à titre d'exemples non limitatifs, avec référence aux dessins annexés dans lesquels :

La figure 1 est un organigramme du procédé de compression selon l'invention ;

La figure 2 est une représentation schématique représentant l'état de l'interpréteur JCVM juste avant l'exécution de l'appel de *sendBytes* dans la méthode *setBalance*.

Il convient de noter tout d'abord que JavaCard est une architecture logicielle complète (incluant un langage de programmation de haut niveau orienté objet, un code objet interprété, un interpréteur de code objet de type machine à pile, un environnement d'exécution, des bibliothèques standard et un système d'interactions sécurisé entre applets hébergées sur une même carte) pour l'exécution de programmes sur des cartes à puce dans un contexte multi-applications. Ce langage est mis en œuvre à l'aide d'une machine virtuelle qui interprète un programme en code objet (le bytecode JavaCard) dans un environnement sécurisé. Un programme JavaCard est appelé une applet. C'est un programme réactif passif qui interagit avec le terminal auquel la carte est connectée sous la forme d'envoi de messages de type commande-réponse, les commandes étant émises par le terminal uniquement, l'applet renvoyant une réponse pour chaque commande reçue. Une carte à puce peut contenir différentes applets qui peuvent interagir. Les applets peuvent être introduites sur la carte en usine lors de la production en série, ou téléchargées en phase d'utilisation par le possesseur de la carte.

Le langage code objet intermédiaire ici considéré est le bytecode JavaCard. L'interpréteur de bytecode est la machine virtuelle JavaCard (JCVM) qui est embarquée sur les cartes à puce. Les objets manipulés par une applet sont soit les objets JavaCard créés par l'applet elle-même, soit les objets créés par l'environnement d'exécution ou retournés par les méthodes de l'API (Application Programming Interface). L'API définit l'interface d'utilisation d'une bibliothèque donnée (dans ce document c'est l'interface logicielle avec l'environnement d'exécution JavaCard). Elle contient la définition des bibliothèques de types abstraits, leurs fonctionnalités et leur mode d'utilisation. Les analyses sémantiques préalables utilisées pour déterminer les accès aux objets s'inscrivent quant à elles dans le cadre classique des *analyses de pointeurs* ou *analyses d'alias* [De95] "Semantic Models and Abstract Interpretation Techniques for Inductive Data Structures and Pointers, Alain Deutsch. PEPM95 - Proc. ACM SIGPLAN Symposium on Partial evaluation and semantics-based program manipulation June 21 - 23, 1995, La Jolla, CA USA". L'analyse de pointeurs est une analyse sémantique qui décrit l'évolution de la mémoire au cours de l'exécution d'un programme. Ce type d'analyse calcule à chaque instruction du programme l'ensemble des objets qui peuvent être référencés par chaque variable de type pointeur et/ou l'ensemble des variables de type pointeur qui référencent le même objet (on appelle deux pointeurs qui référencent le même objet des *alias*). Dans toute cette section on considèrera une syntaxe simplifiée du bytecode JavaCard afin de faciliter la lecture, sans que cela fausse la description en quoi que ce soit.

Dans cet exemple, le protocole de communication entre la carte à puce et le terminal qui l'héberge repose sur un format standardisé [ISO7816] "ISO/IEC 7816. First Edition 1995-09-01 *Information Technology – Identification Cards – Integrated Circuit(s) Cards with Contacts*" de paquets de données : les APDU. Les commandes envoyées par le terminal aussi bien que les réponses de l'applet sont codées dans des APDU. Pour gérer cela

l'environnement d'exécution JavaCard (JCRE) met à la disposition des programmeurs un type de données APDU ainsi qu'un unique objet de ce type qui est créé par l'environnement lui-même et qui est utilisé par l'applet aussi bien pour lire les commandes émises par le terminal que pour écrire les données de la réponse. Les données de l'objet APDU sont stockées dans un buffer qui lui est associé. Chaque applet possède nécessairement une méthode *process* qui prend en argument cet objet de type APDU, une méthode étant le nom d'un sous-programme attaché à un type de données dont il définit une fonctionnalité. Au début de l'exécution de la méthode *process*, cet objet contient la commande émise par le terminal, à la fin de l'exécution de la méthode, il contient la réponse renvoyée par l'applet.

Comme illustré par l'organigramme représenté sur la figure 1, la première étape de l'organigramme consiste en une analyse sémantique comprenant une propagation interprocédurale des références à l'objet APDU et à son buffer associé, c'est-à-dire une analyse qui permet d'identifier les références à l'instance de la classe APDU contenue dans l'environnement d'exécution JCRE et son unique buffer associé dans toutes les méthodes de l'applet. La source de l'objet APDU est l'unique argument de la méthode *process* de l'applet. C'est cette valeur qui est propagée dans tout le programme en explorant le graphe d'appel du programme depuis *process*.

Cette première étape comprend également l'identification des références au buffer de l'APDU par les appels de la méthode *getBuffer* de la classe APDU sur une référence à l'objet APDU du JCRE.

L'étape 2 identifie les groupes d'instructions qui lisent ou transforment l'un ou l'autre de ces deux objets (APDU et buffer), le buffer n'étant accessible que par l'intermédiaire de l'APDU.

En fait, l'analyse détermine en chaque point du bytecode JavaCard de l'applet quelles sont les variables locales et les entrées dans la pile qui contiennent une référence à l'APDU ou à son buffer. Différentes optimisations sont alors possibles à partir de cette information (choix de l'objet et du buffer APDU à l'étape 3 de l'organigramme).

En utilisant deux registres de l'interpréteur, on peut accéder directement à l'APDU ou au buffer de l'APDU. Il faut pour cela insérer des opérations d'initialisation de ces registres dans le code de l'applet, par exemple au début de la méthode *process* (étape 4 d'introduction d'une instruction stockant l'objet dans un registre). Un style couramment utilisé dans les applets JavaCard est de passer une référence sur ces deux objets aux méthodes internes de l'applet appelées par *process*. La compression consiste alors (étape 5) à remplacer dans ces méthodes les lectures des variables locales (c'est-à-dire les paramètres de la méthode) contenant ces objets et les opérations qui les suivent par les opérations spécialisées correspondantes (portant sur les deux registres). Il est aussi possible d'éliminer ces deux arguments de la définition de la méthode, puisqu'ils ne sont plus utilisés (étape 6 d'élimination du code inutile). Le gain en espace est ainsi obtenu non seulement par l'emploi d'instructions spécialisées (plus compactes) mais aussi par élimination des instructions d'empilement de ces références lors de chaque appel aux méthodes concernées qui deviennent alors inutiles.

Les étapes du procédé précédemment défini seront illustrées ci-après par les exemples suivants, à savoir :

- un exemple de compression des opérations sur l'objet et le buffer APDU dans une application bancaire de type porte-monnaie électronique (Exemple 1),

- une extension de l'exemple précédent au cas d'applications GSM ("Global System for Mobile Communication" système de radiotéléphonie mobile à transmission numérique) (Exemple 2),
- un exemple de compression des chemins d'accès à des objets créés par l'applet (Exemple 3),
- un exemple de compression des appels de méthodes internes de l'applet (Exemple 4).

Ces exemples font appel aux termes techniques suivants dont les définitions fournies dans le glossaire ci-après tiennent compte de leur contexte :

Bytecode : terme anglo-saxon synonyme de code objet interprété. Ce terme fait référence ici au code objet interprété sous-jacent à la machine virtuelle JavaCard.

Chemin d'accès : séquence d'opérations de sélection de champ dans des données structurées qui permet de référencer un objet du programme qui n'est pas accessible directement depuis une variable.

Classe : dans un langage orienté objet une classe est la définition d'un type de données et des méthodes qui lui sont associées.

EEPROM : mémoire non volatile et réinscriptible embarquée sur une carte à puce. Les objets non volatiles créés par une applet JavaCard sont stockés dans cette mémoire.

JavaCard : langage et environnement de programmation conçus par Sun Microsystems pour l'exécution de programmes sur des cartes à puce.

JCRE : Acronyme de *JavaCard Runtime Environment*. C'est l'environnement d'exécution de l'architecture JavaCard qui est embarqué sur une carte à puce.

Cet environnement contient l'interpréteur de bytecode (la JCVM), les bibliothèques standard, le protocole d'échange de données entre la carte et le terminal ainsi que le système chargé de la sécurité lors des interactions entre applets (pare-feu).

JCVM : Acronyme de *JavaCard Virtual Machine*. Nom de l'interpréteur de code JavaCard (bytecode). Cet interpréteur est un composant du JCRE.

Objet : Zone mémoire munie d'une structure qui est définie par un type de données dans le programme.

Exemple 1

Dans cet exemple, on suppose que :

a) la méthode *process* de l'applet a la forme classique suivante en JavaCard :

```
public void process(APDU apdu) {
  byte[] buffer = apdu.getBuffer();
  ...
  switch(buffer[ISO7816.OFFSET_INS]) {
    case SET_BALANCE: {
      setBalance(apdu, buffer);
      return;
    }
    case ...
  }
  ...
}
```

et que,

b) la méthode *setBalance* qui modifie le solde du porte-monnaie a la forme suivante :

```
private void setBalance(APDU apdu, byte[] buffer) {
  short currency = buffer[OFFSET_CURRENCY];
  short amount = Util.getShort(buffer,OFFSET_AMOUNT);
  ...
  apdu.sendBytes((short)0,(short)2); // new balance
}
```

La méthode *process* décode la commande stockée dans le buffer de l'objet APDU et, en fonction du code de la commande (donné notamment par l'octet situé à l'index *ISO7816.OFFSET_INS* du buffer de l'APDU), appelle différentes méthodes internes qui sont chargées de traiter chaque commande. Ces méthodes internes reçoivent en argument à la fois l'objet APDU et son buffer afin de pouvoir lire les paramètres de la commande et écrire la réponse. Cette méthode *setBalance* n'a pas été détaillée dans son intégralité, seules ont été reprises quelques lignes qui lisent dans le buffer le type de monnaie (à l'index *OFFSET_CURRENCY*) et le montant à créditer au porte-monnaie électronique (à l'index *OFFSET_AMOUNT*).

Après compilation de l'applet en bytecode la méthode *process* a typiquement la forme suivante (les commentaires sont précédés du symbole « // ») :

```
aload 1
invokevirtual "getBuffer()"
astore 2
...
aload 2
sconst 1    // ISO7816.OFFSET_INS
baload
```

```
switch ...
...
aload 0
aload 1
aload 2
invokespecial "setBalance(APDU, byte[])"
return
...
```

Après compilation en bytecode, la méthode *setBalance* a la forme suivante :

```
aload 2
sconst OFFSET_CURRENCY
baload
sstore 3
aload 2
sconst OFFSET_AMOUNT
invokestatic "Util.getShort(byte[],short)"
sstore 4
...
aload 1
sconst 2
sconst 0
invokevirtual "sendBytes(short,short)"
return
```

La machine virtuelle JavaCard utilise une pile pour stocker les arguments et les résultats de calculs ainsi que des variables JCVM numérotées permettant à la fois de stocker les arguments des méthodes et de stocker des résultats dont la durée de vie est supérieure à celle de la pile. Chaque variable dans le

programme JavaCard (variable locale ou argument) est traduite comme une variable JCVM dans le bytecode.

Dans la méthode *process,* le paramètre *apdu* est stocké dans la variable JCVM 1, la variable JavaCard *buffer* dans la variable JCVM 2 et la référence à l'objet applet dans la variable JCVM 0. Dans la méthode *setBalance* le paramètre *apdu* est stocké dans la variable JCVM 1, le paramètre *buffer* dans la variable JCVM 2, la variable JavaCard *currency* dans la variable JCVM 3, la variable JavaCard *amount* dans la variable JCVM 4, et la référence à l'objet applet dans la variable JCVM 0.

L'instruction *aload n* empile la valeur de la variable JCVM *n*, l'instruction *astore n* écrit dans la variable JCVM *n* la valeur de type objet se trouvant au sommet de la pile d'opérandes, et l'instruction *sstore n* écrit dans la variable JCVM *n* la valeur de type entier se trouvant au sommet de la pile, l'instruction *sconst c* empile la constante *c*, les instructions *invokevirtual* et *invokespecial* représentent des appels de méthode selon divers modes non détaillés ici, l'instruction *baload* est l'instruction de lecture d'un élément de tableau, les instructions *switch* et *return* représentent respectivement le choix multiple et le retour de méthode. Toutes les instructions du bytecode JavaCard prennent leurs arguments sur la pile d'opérandes.

La figure 2 est une représentation schématique montrant l'état de l'interpréteur JCVM juste avant l'exécution de l'appel de *sendBytes* dans la méthode *setBalance*.

Selon cette représentation :

- le bloc 1 représente la mémoire allouée à l'interpréteur qui comporte ici l'objet applet qui correspond au programme en cours d'exécution, l'objet APDU et le buffer associé à l'APDU,

- le bloc 2 est la pile d'opérandes,
- le bloc 3 représente les variables JCVM 0 à JCVM 4 utilisées par la méthode *setBalance*, et
- le bloc 4 montre les variables JCVM 0 et JCVM 1 utilisées par la méthode *process*.

La pile d'opérandes (bloc 2) contient trois valeurs en partant du bas de la pile : une référence sur l'objet APDU, l'entier 2 et l'entier 0.

Chaque appel de méthode engendre un environnement d'exécution qui définit des variables JCVM utilisées par la méthode.

Que ce soit pour la pile d'opérandes ou pour les environnements d'exécution des méthodes, on a représenté par des flèches les références aux objets se trouvant en mémoire.

Dans le cas où l'analyse sémantique préalable (étape 1 du procédé) a déterminé qu'en tout point de la méthode *setBalance* la variable JCVM 1 contenait une référence à l'APDU et la variable JCVM 2 une référence au buffer de l'APDU, l'étape 2 détermine alors notamment, parmi les instructions mentionnées ci-dessus, les occurrences de groupes d'instructions qui opèrent sur l'objet et le buffer APDU. Dans le cas où ces deux objets sont choisis à l'étape 3, alors, selon la suite du procédé, ces objets vont être stockés dans deux registres de l'interpréteur, notés *A* et *B*. Les instructions qui effectuent la lecture des registres correspondants sont notées *aload_A* et *aload_B*. De même, sont notées *astore_A*, *astore_B* les instructions qui affectent à ces mêmes registres une référence stockée sur la pile d'opérandes.

Selon l'étape 4 du procédé, on insère au début de la méthode *process* la séquence de bytecode qui initialise les registres globaux *A* et *B*. Le début de la méthode *process* devient :

```
aload 1
astore_A
aload 1
invokevirtual "getBuffer()"
astore 2
aload 2
astore_B
```

Selon l'étape 5 du procédé, on remplace les occurrences des groupes d'instruction accédant à l'objet et au buffer APDU par les instructions de lecture spécialisées correspondantes, aussi bien dans *process* :

```
aload_B
sconst 1    // ISO7816.OFFSET_INS
baload
switch ...
...
aload 0
aload_A
aload_B
invokespecial "setBalance(APDU, byte[])"
return
...
```

que dans *setBalance* :

```
aload_B
sconst OFFSET_CURRENCY
baload
sstore 3
```

```
aload_B
sconst OFFSET_AMOUNT
invokestatic "Util.getShort(byte[],short)"
sstore 4
...
aload_A
sconst 2
sconst 0
invokevirtual "sendBytes(short,short)"
return
```

Enfin, selon l'étape 6 du procédé, le programme est simplifié par élimination des arguments désormais inutiles de la méthode *setBalance*. L'appel de cette méthode dans *process* devient :

```
aload 0
invokespecial "setBalance()"
return
...
```

et la signature de la méthode *setBalance* est modifiée par suppression de ses arguments.

On obtient un gain en taille du fait de l'élimination des instructions d'empilement des arguments de la méthode *setBalance* au moment de son appel.

Spécialisation d'opérations sur les tableaux

Les instructions de manipulation de tableaux *bastore* (écriture d'un élément) et *baload* (lecture d'un élément) sont fréquemment utilisées pour accéder au

buffer de l'APDU. Créer des instructions spécialisées qui manipulent directement le buffer stocké dans un registre permet d'éliminer l'opération d'empilement de la référence à ce buffer avant chaque *bastore/baload.*

Ainsi, si dans l'étape 3 du procédé on prend en compte également la lecture du tableau dans la méthode *setBalance* parmi les groupes d'instructions opérant sur le buffer de l'APDU, l'étape 5 récrit *setBalance* de la manière suivante :

```
sconst OFFSET_CURRENCY
baload_B
sstore 3
```

L'instruction de lecture d'un élément du tableau a été spécialisée en l'instruction `baload_B` qui lit son argument dans le registre B, l'empilement du buffer APDU a été économisé.

Spécialisation sur l'objet d'appels virtuels à l'API

Un appel virtuel consiste à appeler une méthode sur un objet dont elle définit une fonctionnalité.

Certaines méthodes de l'API comme *javacard.framework.APDU.sendBytes* (envoi de données au terminal auquel est reliée la carte) sont toujours appelées sur l'objet APDU. On peut employer une instruction spécialisée pour faire de tels appels et ainsi éviter d'empiler la référence à l'APDU avant l'appel de la méthode virtuelle.

Ainsi, si dans l'étape 3 du procédé on prend en compte également l'appel virtuel à *sendBytes* dans la méthode *setBalance* parmi les groupes d'instructions opérant sur l'objet APDU, l'étape 5 récrit *setBalance* de la manière suivante :

```
sconst 2
sconst 0
invokevirtual_A "sendBytes(short,short)"
return
```

L'instruction d'appel de méthode virtuelle a été spécialisée en l'instruction `invokevirtual_A` qui va lire l'objet sur lequel la méthode est appelée dans le registre A, on économise ainsi l'empilement de l'objet APDU.

Spécialisation sur les arguments d'appels à l'API

Certaines méthodes de l'API comme *javacard.framework.Util.getShort* (lecture d'un entier 16 bits dans un tableau d'octets) *ou* *javacard.framework.Util.arrayCopy* (copie d'un tableau d'octets) sont souvent utilisées avec le buffer de l'APDU en argument. On peut créer des instructions spécialisées qui évitent d'empiler la référence au buffer pour l'argument correspondant.

Ainsi, si dans l'étape 3 du procédé on prend en compte également l'appel à *getShort* dans la méthode *setBalance* parmi les groupes d'instructions opérant sur le buffer APDU, l'étape 5 récrit *setBalance* de la manière suivante :

```
sconst OFFSET_AMOUNT
invokestatic_B "Util.getShort(byte[],short)"
sstore 4
```

L'instruction d'appel de méthode statique a été spécialisée en instruction `invokestatic_B` qui lit son premier argument dans le registre B, on économise ainsi l'empilement du buffer APDU.

Exemple 2

Les applications GSM n'utilisent pas le protocole classique d'échanges d'APDU mais une API de plus haut niveau qui prend en charge les différents échanges d'APDU. Cette API repose sur quatre classes qui représentent les différents types de messages que la carte peut échanger avec le terminal (ici un téléphone mobile) :

- *EnvelopeHandler*
- *EnvelopeResponseHandler*
- *ProactiveHandler*
- *ProactiveResponseHandler*

Ces types de données sont utilisés de la même façon que le type APDU pour échanger des données entre le terminal et la carte. Il n'existe qu'une seule instance de chacune de ces classes qui est construite et gérée par l'API. Une référence à ces différentes instances peut être obtenue à l'aide de la méthode *getTheHandler* qui est implantée par chacune de ces classes. Toutes les techniques de compression décrites dans la section précédente pour l'APDU et son buffer peuvent donc se transposer directement au cadre des applications GSM. L'analyse sémantique est de la même façon une propagation de références interprocédurale qui se charge d'identifier les références à ces différents objets dans tout le code de l'applet.

Exemple 3

Compression des chemins d'accès à des objets créés par l'applet

Typiquement une applet peut créer des objets en EEPROM au moment de son installation sur la carte et les stocker dans des variables de l'applet. L'analyse sémantique permet d'associer à chaque groupe de bytecodes qui accèdent à un

objet quel est l'objet accédé chaque fois qu'elle le peut (cela dépend de la puissance et de la précision de l'analyse sémantique utilisée). Lorsque l'analyse permet de déterminer quel est l'objet accédé, on peut remplacer le groupe d'opérations utilisées pour y accéder par une seule opération. Pour ce faire, on peut utiliser un registre dans lequel on stocke une référence à l'objet et remplacer le groupe de bytecodes par une opération de lecture de ce registre.

Cette méthode nécessite de déterminer une position dans le corps de l'applet où insérer l'opération d'initialisation du registre à l'objet en question dans ses différents contextes d'utilisation. Cette position doit satisfaire deux contraintes : elle doit précéder tout accès à la valeur de l'objet concerné et elle ne doit précéder elle-même aucune modification de la valeur de l'objet. Il s'agit donc d'une position dans le programme où l'objet a acquis sa valeur définitive (une valeur qui restera constante jusqu'à la fin du programme) et où il n'a pas encore été accédé. Le calcul de telles positions dans le programme relève des techniques standard d'analyse statique.

A titre d'exemple d'application, on considère le cas de la méthode *install* qui est appelée une seule fois dans la durée de vie de chaque applet au moment de sa création sur la carte. Cette méthode se charge de mettre en place toutes les données qui vont être utilisées par l'applet. En général tous les objets manipulés par l'applet sont créés dans cette méthode. On se place dans ce cas de figure et on suppose que l'analyse sémantique permet de déterminer les chemins d'accès aux objets issus des variables de l'applet qui ne sont plus modifiables après exécution de la méthode *install*. Dans ce cas, il est licite de stocker les valeurs de ces chemins dans des registres en insérant les opérations appropriées au début de la méthode *process*.

Soit le fragment de code JavaCard suivant :

```
this.f.array[0] = 1;
this.f.array[1] = 2;
```

Le bytecode correspondant après compilation a la forme suivante :

```
getfield_this "f"
getfield "array"
sconst 0
sconst 1
sastore
getfield_this "f"
getfield "array"
sconst 1
sconst 2
sastore
```

Les instructions *getfield* et *getfield_this* empilent la valeur de champs d'objets (de champs de l'objet applet courant pour *getfield_this*) et l'instruction *sastore* écrit un entier dans un tableau.

Si l'analyse sémantique a permis de déterminer que l'objet référencé par *this.f.array* est uniquement déterminé après installation de l'applet et s'il est stocké dans le registre *A*, alors en reprenant les notations de l'exemple 1, le code obtenu après compaction a la forme suivante (en reprenant les notations de l'Exemple 1) :

```
aload_A
sconst 0
sconst 1
sastore
aload_A
```

```
sconst 1
sconst 2
sastore
```

Il faut également insérer à l'endroit approprié (au début de la méthode *process* par exemple) la séquence d'initialisation suivante :

```
getfield_this "f"
getfield "array"
astore_A
```

Il est à noter que cette optimisation en espace est aussi une optimisation en temps de calcul, car il y a moins d'accès à la mémoire dans la forme compactée du bytecode.

Exemple 4

On trouve généralement dans le corps de l'applet de nombreux appels aux méthodes internes de l'applet. Ces appels de méthodes (par l'opération *invokespecial*) nécessitent l'empilement de la référence à l'objet applet par l'intermédiaire de l'instruction *aload 0* (en JavaCard la référence à l'applet est par convention toujours stockée dans la variable 0 de chaque méthode de l'applet). La méthode de compaction proposée concerne les méthodes internes de l'applet qui ne sont pas accessibles directement ou indirectement depuis une autre applet. En effet dans un contexte multi-application une méthode qui est utilisable depuis un autre contexte (ici une autre applet) ne peut pas subir de modifications qui intègrent des spécificités du contexte de l'applet qui la définit.

Le système de compaction opère alors comme suit :

- On utilise un registre *A* pour enregistrer la référence à l'objet applet.
- On remplace dans chacune des méthodes candidates chaque opération *aload 0* qui fait référence à l'applet par l'opération de lecture du registre *A*.
- Dans toutes les autres méthodes de l'applet, on identifie les groupes de bytecodes qui correspondent à un appel à une des méthodes ainsi transformées. Dans chacun de ces groupes :

    a. On élimine l'instruction *aload 0* qui correspond à l'empilement de la référence à l'objet applet courant ;
    b. On remplace l'instruction *invokespecial* par une nouvelle instruction *invokespecial_applet* qui possède la même sémantique, à la seule différence que la référence à l'objet applet est lue non pas dans la pile mais dans le registre *A*.

- On insère une opération d'initialisation du registre *A* dans le code de l'applet, par exemple au début de la méthode *process*.

Soit le groupe de bytecodes suivant correspondant à un appel de méthode interne :

```
aload 0
aload 2
sconst 23
invokespecial "privateMethod(byte[],short)"
```

Si la méthode *privateMethod* satisfait aux conditions requises pour la compression, ce groupe de bytecodes devient après application de la compression :

```
aload 2
sconst 23
invokespecial_applet "privateMethod(byte[],short)"
```

Sachant que, dans la méthode *privateMethod*, toutes les instructions *aload 0* ont été remplacées par des instructions de lecture de registre *aload_A* (en reprenant les notations de l'exemple 1), et qu'on a inséré au début de la méthode *process* la séquence de bytecode suivante :

```
aload 0
astore_A
```

Revendications

1. Procédé pour la compression du code d'un programme en code objet intermédiaire, ce procédé étant utilisable au compactage de programmes en code objet intermédiaire exécutable dans un système embarqué à faibles ressources matérielles,
caractérisé en ce qu'il comprend l'exploitation d'une information sémantique pour modifier le code objet intermédiaire du programme et l'élimination des instructions du code objet intermédiaire du programme devenues inutiles du fait de cette modification.

2. Procédé selon la revendication 1,
caractérisé en ce qu'il comprend l'identification des accès aux objets manipulés par le programme par une analyse sémantique du code objet du programme et l'utilisation de cette information pour remplacer dans le code du programme les groupes d'instructions accédant à ces objets par des instructions spécialisées permettant un accès direct aux objets considérés.

3. Procédé selon la revendication 2,
caractérisé en ce que les susdites instructions spécialisées sont intégrées dans l'interpréteur servant à effectuer une interprétation logicielle des instructions standard du code intermédiaire en instructions directement exécutables par un microprocesseur, dans le cas où cet interpréteur n'en dispose pas.

4. Procédé selon la revendication 3,
caractérisé en ce que l'interpréteur est étendu avec des registres spécifiques et des versions spécialisées de certaines instructions de l'interpréteur qui opèrent implicitement sur ces registres spécifiques plutôt que sur leurs arguments ordinaires.

5. Procédé selon la revendication 4,

caractérisé en ce qu'il comprend les étapes suivantes :

- une première étape dans laquelle on effectue une analyse sémantique du programme qui met en évidence les objets accédés en chaque point du programme,
- une deuxième étape dans laquelle, à l'aide de l'information calculée par l'analyse sémantique de la première étape, pour tout objet manipulé par le programme, on identifie dans le programme les occurrences de différents groupes d'instructions qui opèrent sur cet objet et dont on dispose une version spécialisée en terme d'un registre spécifique dans le jeu d'instruction de l'interpréteur,
- une troisième étape dans laquelle on soumet lesdits objets à un test de comparaison de supériorité d'une fonction d'au moins le nombre d'occurrences des groupes d'instructions correspondants dans ledit programme, comme déterminé à la deuxième étape, à une valeur de référence et, sur réponse positive audit test, pour chaque objet satisfaisant à ladite étape de test,
- une quatrième étape dans laquelle, en un point du programme qui précède, dans toutes les exécutions possibles du programme, les occurrences de groupes d'instructions associées audit objet, on introduit le code correspondant à l'affectation dudit objet dans un des registres spécifiques de l'interpréteur,
- une cinquième étape dans laquelle on remplace dans le programme les groupes d'instruction à chacune desdites occurrences par l'instruction spécialisée par rapport audit registre qui leur correspond,
- une sixième étape dans laquelle on élimine du code du programme les instructions et les constructions qui sont rendues inutiles ou redondantes du fait des transformations du programme indiquées aux quatrième et cinquième étapes.

6. Procédé selon l'une des revendications précédentes,
caractérisé en ce que dans le cas d'un compactage d'un programme en code objet intermédiaire JavaCard exécutable dans le cadre des échanges entre une carte à puce et un terminal de lecture, l'analyse sémantique consiste en une propagation interprocédurale des références à l'objet APDU, analyse qui permet d'identifier les références de l'unique instance de la classe APDU contenue dans le JCRE et son unique buffer associé dans toutes les méthodes de l'applet, cette analyse déterminant en chaque point du bytecode JavaCard de l'applet quelles sont les variables locales et les entrées dans la pile qui contiennent une référence à l'APDU ou à son buffer.

7. Procédé selon la revendication 6,
caractérisé en ce que l'interpréteur comprend deux registres permettant d'accéder directement à l'APDU ou au buffer de l'APDU en insérant des opérations d'initialisation de ces registres dans le code de l'applet.

8. Procédé selon l'une des revendications précédentes,
caractérisé en ce que la compression consiste à remplacer dans les méthodes internes de l'applet, les lectures des variables locales contenant ces objets et les opérations qui les suivent par les opérations spécialisées correspondantes.

Fig. 1

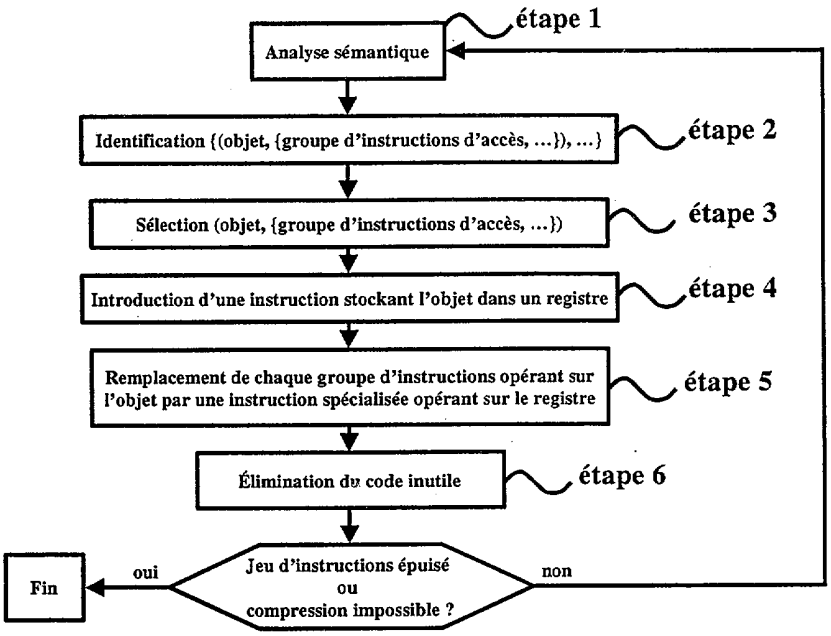
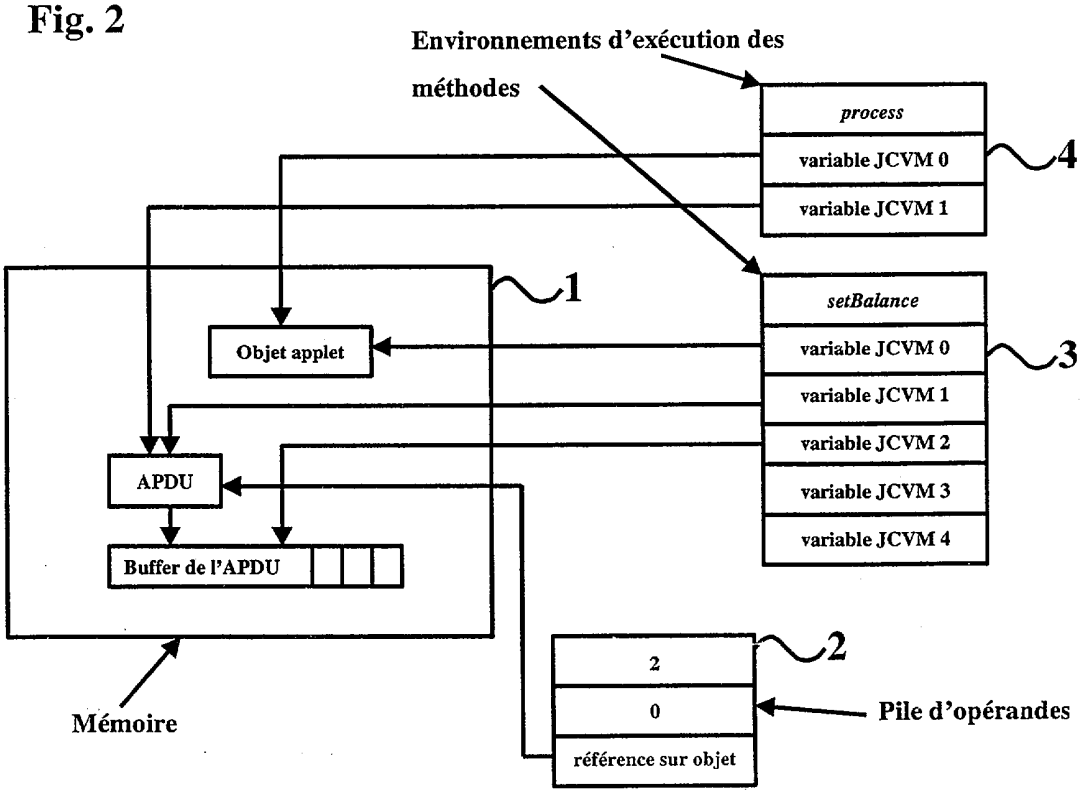

étape 1
Analyse sémantique
Identification {(objet, {groupe d'instructions d'accès, ...}), ...}
étape 2
Sélection (objet, {groupe d'instructions d'accès, ...})
étape 3
Introduction d'une instruction stockant l'objet dans un registre
étape 4
Remplacement de chaque groupe d'instructions opérant sur l'objet par une instruction spécialisée opérant sur le registre
étape 5
Élimination du code inutile
étape 6
Jeu d'instructions épuisé ou compression impossible ?
oui
non
Fin

Fig. 2


Environnements d'exécution des méthodes
process
variable JCVM 0
variable JCVM 1
4
setBalance
variable JCVM 0
variable JCVM 1
variable JCVM 2
variable JCVM 3
variable JCVM 4
3
1
Objet applet
APDU
Buffer de l'APDU
Mémoire
2
0
référence sur objet
Pile d'opérandes

RÉPUBLIQUE FRANÇAISE

INPI
INSTITUT NATIONAL DE LA PROPRIETE INDUSTRIELLE

2827974

N° d'enregistrement national

FA 609241
FR 0110210

RAPPORT DE RECHERCHE PRÉLIMINAIRE

établi sur la base des dernières revendications déposées avant le commencement de la recherche

DOCUMENTS CONSIDÉRÉS COMME PERTINENTS		Revendication(s) concernée(s)	Classement attribué à l'invention par l'INPI
Catégorie	Citation du document avec indication, en cas de besoin, des parties pertinentes		
X	EP 0 778 521 A (SUN MICROSYSTEMS INC) 11 juin 1997 (1997-06-11) * page 4, ligne 37 - page 5, ligne 11 *	1,2	G06F9/30
A	US 5 613 121 A (BLAINEY ROBERT J) 18 mars 1997 (1997-03-18) * colonne 2, ligne 17 - dernière ligne *	1-8	
			DOMAINES TECHNIQUES RECHERCHÉS (Int.CL.7)
			G06F

Date d'achèvement de la recherche	Examinateur
30 mai 2002	Bijn, K

CATÉGORIE DES DOCUMENTS CITÉS

X : particulièrement pertinent à lui seul
Y : particulièrement pertinent en combinaison avec un autre document de la même catégorie
A : arrière-plan technologique
O : divulgation non-écrite
P : document intercalaire

T : théorie ou principe à la base de l'invention
E : document de brevet bénéficiant d'une date antérieure à la date de dépôt et qui n'a été publié qu'à cette date de dépôt ou qu'à une date postérieure.
D : cité dans la demande
L : cité pour d'autres raisons

.....

& : membre de la même famille, document correspondant

EPO FORM 1503 12.99 (P04C14)

ANNEXE AU RAPPORT DE RECHERCHE PRÉLIMINAIRE RELATIF A LA DEMANDE DE BREVET FRANÇAIS NO. FR 0110210 FA 609241

La présente annexe indique les membres de la famille de brevets relatifs aux documents brevets cités dans le rapport de recherche préliminaire visé ci-dessus.
Les dits membres sont contenus au fichier informatique de l'Office européen des brevets à la date du 30-05-2002
Les renseignements fournis sont donnés à titre indicatif et n'engagent pas la responsabilité de l'Office européen des brevets, ni de l'Administration française

Document brevet cité au rapport de recherche	Date de publication	Membre(s) de la famille de brevet(s)	Date de publication
EP 0778521 A	11-06-1997	US 5794044 A	11-08-1998
		AU 712005 B2	28-10-1999
		AU 7188996 A	12-06-1997
		CA 2191411 A1	09-06-1997
		CN 1158459 A	03-09-1997
		EP 0778521 A2	11-06-1997
		JP 10040107 A	13-02-1998
		SG 75109 A1	19-09-2000
US 5613121 A	18-03-1997	AUCUN	

EPO FORM P0465