Ministère de l'Enseignement Supérieur, de la Recherche Scientifique et de l'Innovation
(MESRSI)

- - - - - - - - - - - - - - - - - -

Secrétariat Général

- - - - - - - - - - - - - - - - - -

Université Polytechnique de Bobo -Dioulasso **(U.P.B)**

- - - - - - - - - - - - - - - - - -

Ecole Supérieure d'Informatique **(E.S.I)**

Cycle des Ingénieurs de Conception en Informatique **(C.I.C.I)**

Mémoire de fin de Cycle
présenté par **Adam Ismael Paco SIE**

Thème : « Elaboration d'une Méthode de couverture de code embarqué pour la SmartCard Java »

Soutenu le 07 Avril 2017

Co - directeur de Mémoire
Dr Sadouanouan MALO
(enseignant-chercheur à l'ESI)

Directeur de Mémoire
Pr Jean-Louis LANET
(chercheur à l'INRIA-Rennes)

Année Universitaire : 2016 - 2017

Résumé

La couverture de code est une mesure (fonction charger de quantifier un ensemble) permettant de calculer le pourcentage de code exécuté lors des phases de test. Cette technique consiste à placer des capteurs appelés sondes aux différents points de décision du graphe d'exécution des méthodes en usant de la technique d'instrumentation. Par analogie à l'instrumentation des sciences, l'instrumentation de code consiste à l'ajout d'éléments de mesure (ici les sondes) dans un programme, sans impact sur le code source originel.

Le but de nos travaux consiste à adapter l'outil de couverture JaCoCo (Java Code Coverage) utilisé au niveau de la technologie Java standard pour un usage sur les applications destinées aux cartes à puce java, les applets java card. Notre démarche a consisté à substituer l'outil d'instrumentation utilisé par JaCoCo (ASM), par un outil d'instrumentation expérimental adapté aux applications java card, le module CapMap (Cap Manipulator) développé par la Smart Secure Devices Team (SSD). Le module obtenu CapCov (Converted APplet COVerage) permet d'instrumenter le code binaire d'une applet, de récupérer les données d'exécution et de construire un rendu qui permet d'afficher les différents éléments de mesure telles que la couverture d'instructions, de branches, de lignes et de calcul de complexité cyclomatique.

mots-clés : carte à puce, java card, code binaire, couverture de code, instrumentation.

Abstract

Code coverage is a measure (a function of quantifying a set) used to calculate the percentage of code executed during the test phases. This technique consists in placing sensors called probes at the different decision points of the graph of execution of the methods using the technique of instrumentation. By analogy to the instrumentation of the sciences, code instrumentation consists of the addition of measuring elements (here probes) in a program, without impact on the original source code.

The goal of our work is to adapt the JaCoCo (Java Code Coverage) coverage tool used in standard Java technology for a use on java smart card applications, java card applets. Our approach consisted in replacing the instrumentation tool used by JaCoCo (ASM) with an experimental instrumentation tool adapted for java smartcard applications, the CapMap module (Cap Manipulator) developed by the Smart Secure Devices Team (SSD). The module CapCov (Converted APplet COVerage) allow to instrument the bytecode of applets, to retrieve the execution data and to construct a rendering which allows to display the different elements of measurement such as coverage of instructions, branches, Lines and calculating cyclomatic complexity.

keywords : smartcard, java card, bytecode, code coverage, instrumentation.

Dédicaces

Nous dédions le fruit de notre labeur au/à :
Dieu tout puissant pour toutes ces grâces reçues ;
mes chers parents pour l'éducation reçue, leurs conseils, patience et leurs soutiens indéfectibles ;
mon fils Dassa et à ma fiancée Yolande pour leurs soutiens et affections ;
mon grand frère Aziz, sa femme Saida pour leurs attentions et leurs soutiens ;
ma grande sœur Safi, et son mari ;
mes frères et sœurs Saida, Paton, Rachid, Alioune, Latif, Maimouna ;
ma tante Awa et son mari ;
la famille KOURAOGO ;
mes amis et camarades de classe.

Remerciements

Je remercie l'École Supérieure d'Informatique (E.S.I) de l'Université Polytechnique de Bobo-Bobo-Dioulasso qui, à travers ses enseignants dévoués, m'a offert une formation de qualité. Nous adressons une mention particulière :

- au Pr Jean Louis LANET et au Dr Sadouanouan MALO pour m'avoir désigné pour ces travaux ;
- au Pr Oumarou SIE et M. Didier BASSOLE, pour leurs disponibilités, leurs conseils et pour m'avoir accueilli dans les locaux du LAMI (Laboratoire de Mathématiques et d'Informatique) pour mes travaux ;
- au Dr Mesmin DANDJINOU pour m'avoir permis de percevoir le sens profond du pouvoir de l'éducation (« *the power of education*) ».

Glossaire

AID suite d'octets permettant d'identifier une applet de manière unique. 48

ant moteur de production java open-source impératif. 24

APDU Application Protocol Data Unit. 13, 17, 40, 41, 43–45, 51, 55

API Application Programming Interface. 4

applet Une applet Java Card est une classe Java standard étendant la classe javacard.framework.Applet. 4, 12, 14, 15, 17, 18, 28–30, 34, 35, 40, 41, 46, 50, 52

ASM bibliothèque d'instrumentation de code binaire java. 18, 20, 23, 25, h, j, 50, 51

bundle module d'application. 20

bytecode code machine intermédiaire utilisant un jeu d'instruction indépendant d'une architecture matérielle spécifique. 6, 9–11, 14, 15, 17–19, 21–25, b, 31, h, 35, 39, 41, 42, 48, 49, 51

CapMap bibliothèque de manipulation de bytecode java card. i, 21, 29, 47, 51, 52

CFG Graphe de Contrôle de Flux (Control Flow Graph). 7, 22, 24–27, 50

constant pool pool de constantes. 11, 15

COS Chip Operating System. 3, 4

CPU Central Processing Unit. 2

EEPROM Electrically-Erasable Programmable Read-Only Memory. 3

fichier à extension .cap fichiers contenant du code binaire java card dont l'extension provient de la contraction de Converted et APplet. 10, 14, 21, 29, 41, 44

fichier à extension .class fichiers contenant du code binaire java standard. 10, 11, 14, 48

fichier à extension .exec fichier binaire contenant les résultats d'exécution des tests de JaCoCo. 22, 46, 47

fichier à extension .exp fichier binaire produit lors de la phase de génération du fichier .cap; il contient l'ensemble des informations qui permettront au fichier .cap, d'être à son tour importer dans une autre applet java card. 14, 41

fichier à extension .java fichiers contenant du code source java. 14

GSM Global System for Mobile Communications. 2

JaCoCo outil de couverture de code. i, 20, 22–27, 30, 31, 33, 34, j, 46, 48–51

Java langage de programmation orienté objet fortement typé. 4, 7, 10, 15, 17, 21, 23, 48

Java Card cartes à puces Java. 4, 10, 12, 14–21, 29, 30, 51

JCDK Java Card Development Kit. 51

JCRE Java Card Runtime Environment. 4, 5, 54

JCVM Java Card Virtual Machine. c, 4, 15, 21, a, b, 41

JVM Java virtual Machine (machine virtuelle java). 10

JVMPI Java Virtual Machine Profiler Interface. 9

JVMTI JVM Tool Interface. 9

logging gestion des traces d'exécution. 9

LTE Long Term Evolution. 2

NPU Network Processing Unit. 2

opcode code d'opération machine java. 10, 11, 32, 46, 55

OSGi Open Service Gateway Initiative, ensemble de spécification décrivant un paradigme de programmation orienté composant et une architecture orientée service. 20, 23

package regroupement d'applets. 29, 41, 43

probe appelé sonde. 5

RAM Random Access Memory. 3

ROM Read-Only Memory ou mémoire morte. 3, 4

SIM Subscriber Identity Module. 2, 3

slot espace mémoire atomique de rangement. 10

Smart Card cartes intelligentes. 4, 12, 13, 21, 29, 34, 52

UMTS Universal Mobile Telecommunications System. 2

Sommaire

Introduction générale

Les différentes avancées scientifiques sont à la base de nombreux progrès et innovations technologiques. À l'instar des téléphones intelligents (smartphones), l'apparition des cartes à puces (smartcard) avec des systèmes d'exploitation homogènes et quasi-complets, a permis d'observer un essor considérable de l'ingénierie logicielle dans ce domaine. Ceci se caractérise par l'utilisation de ces cartes par le grand public dans des secteurs variés comme la banque (carte bancaire), les télécommunications (carte SIM), la télévision (carte de décodeur), etc.

Inopportunément, l'essor s'est fait sans tenir compte de certaines procédures standard d'ingénierie logicielle qui n'ont pu être adapter aux particularités des technologies pour smartcard. C'est notamment le cas de la phase des tests ou l'exhaustivité des tests vis à vis des programmes éprouvés ne peut être réaliser par les outils standards existants telles que les outils de couverture de code. De part la nature des applications s'exécutant sur les smartcard (authentification, conservation de clé numérique...), la couverture de code en tant que mesure de la qualité des tests s'impose comme un élément important. Notre contribution vise à pallier dans une certaine mesure à ce manque à travers l'élaboration d'une méthode et d'un outil de couverture de code adapté au smartcard java.

Le présent mémoire qui présente la substance de nos travaux, se subdivise en 3 (trois) chapitres. Le premier chapitre présente la problématique de la couverture de code sous l'environnement Java Card. Le second explore les différentes stratégies de résolutions possibles et détermine la méthode de couverture de code retenue. Le troisième chapitre est consacré à la mise en œuvre et l'évaluation de la méthode choisie. Le mémoire est clôturé par une conclusion.

Chapitre 1

Problématique de la couverture de code en environnement Java Card

1.1 La carte à puce

1.1.1 Présentation

Une carte à puce est, comme son nom l'indique, une carte en plastique dans laquelle est enfouie un circuit intégré doté d'un microcontrôleur (ce qui le différencie des cartes mémoires ordinaires) [9]. L'intérêt particulier porté à ce périphérique vient entre autre de sa capacité à stocker des données et de manière sécurisée, de sa capacité à réaliser des calculs d'algorithme cryptographique le tout dans un micro système embarqué avec des ressources très limitées. Ces caractéristiques lui ont valu d'être largement utilisée dans de nombreux domaines technologiques où la nécessité était d'authentifier des entités distribuées et/ou mobiles de réseaux, où il constituait le maillon important du processus de sécurité.

C'est le cas du domaine des télécommunications où la carte à puce joue le rôle de clé d'authentification d'abonné (Subscriber Identity Module (SIM)), sur les réseaux de type Global System for Mobile Communications (GSM), Universal Mobile Telecommunications System (UMTS) et Long Term Evolution (LTE). Dans le domaine de la télévision payante, la carte joue le rôle de clé de décryptage des signaux (carte de décodeur), dans le domaine bancaire (guichet électronique) elle occupe une place importante dans le processus d'authentification des abonnés (carte bancaire VISA, MASTERCARD...)

Dans ce qui suit nous présentons les aspects matériels et logiciels qui sous-tendent ces cartes.

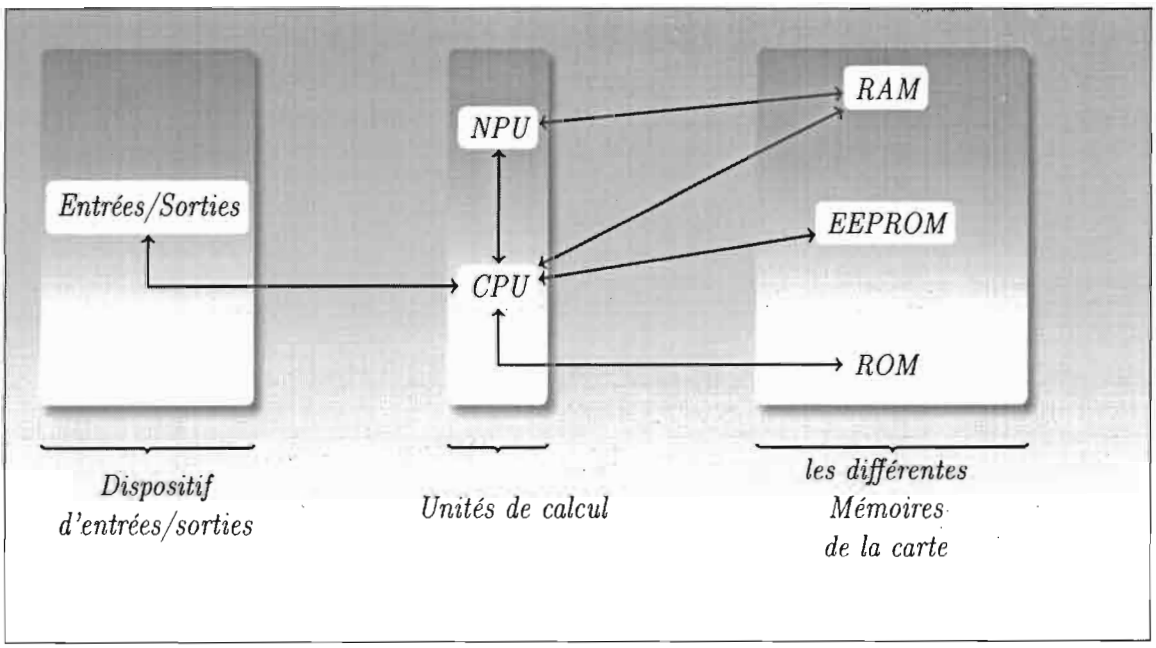
Aspect Matériel

Du concept de la mémoire portative dans les années 1947, elle a progressivement évolué à travers l'intégration de dispositifs de traitement de signaux électriques dans les années 1975, puis le recours à un microprocesseur dans les années 1977, pour de nos jours être à même de prendre en charge la pile de protocole IP et HTTP [12] comme nous l'illustre la Figure 1. Le terme « carte à puce » de nos jours, fait référence à un circuit intégré sur laquelle est bâti un microcontrôleur composé des éléments suivants :

- un microprocesseur (Central Processing Unit (CPU) et Network Processing Unit (NPU) [1]) ;

1. NPU : C'est un processeur conçu pour implémenter directement dans ses circuits certaines fonctions normalement réalisées par du code machine

- une mémoire Random Access Memory (RAM) ;
- une mémoire Read-Only Memory ou mémoire morte (ROM) ;
- une mémoire de stockage Electrically-Erasable Programmable Read-Only Memory (EEPROM) [2] ;
- un dispositif d'entrées sorties.



Figures 1 – Schéma simplifié du microcontrôleur d'une carte à puce

Aspect Logiciel

L'évolution de la composante matériel des cartes à puces permettait l'élaboration de programmes plus complexes. Ceci conduit entre autres, à la mise en place de système permettant une exploitation plus aisée et efficiente des ressources disponibles par les programmes. Ces systèmes d'exploitation pour cartes (Chip Operating System (COS)) aussi appelés masques, ont permis le passage de systèmes monolithiques [3] autour des années 1981, aux systèmes multi-applications prenant en charge la *post-issuance* [4] . A ce jour, il existe une variété de COS que l'on peut répartir en deux grandes catégories :

- les systèmes fermés ;
- les systèmes ouverts.

Les systèmes fermés sont des systèmes très couplés au matériel. Le COS définit un ensemble de commandes devant être utilisées afin d'élaborer les programmes. Ceci a pour avantages de faciliter le développement d'applications, avec le corollaire de limiter le champ des applications possibles (c'est le cas des système GemXplore de Gemalto, CardOS de Siemens). Il faut noter que les cartes implémentant ces systèmes sont généralement mono-applicatif (carte SIM, carte

2. EEPROM : Il s'agit d'une mémoire morte effaçable électriquement et programmable.
3. il s'agit de système ou l'application et le système d'exploitation ne faisait qu'un
4. Possibilité de manipuler (charger, modifier, supprimer) sur les cartes à puces des applications pendant leur utilisation courante.

bancaire).
Les systèmes ouverts quand à eux sont des systèmes qui permettent après la fabrication de la carte, de charger, d'exécuter et de supprimer une multitude d'applications à usages variées écrit dans un langage de programmation évolué spécifique. Les cartes à puces intègrent depuis leurs conception au sein de la ROM, un COS [8], COS qui en fonction des technologies supportées permet de classifier les cartes à puces, on distingue ainsi :

- les cartes à puces Java (Java Card) exécutant des programmes Java (section 1.1.2 page 4) ;
- les Basic Card (Windows Card) qui exécutent des programmes réalisés en visual basic ;
- les MULTOS smart card, qui sont des cartes capables d'exécuter des programmes réalisés en C, Java, visual basic ;
- ...

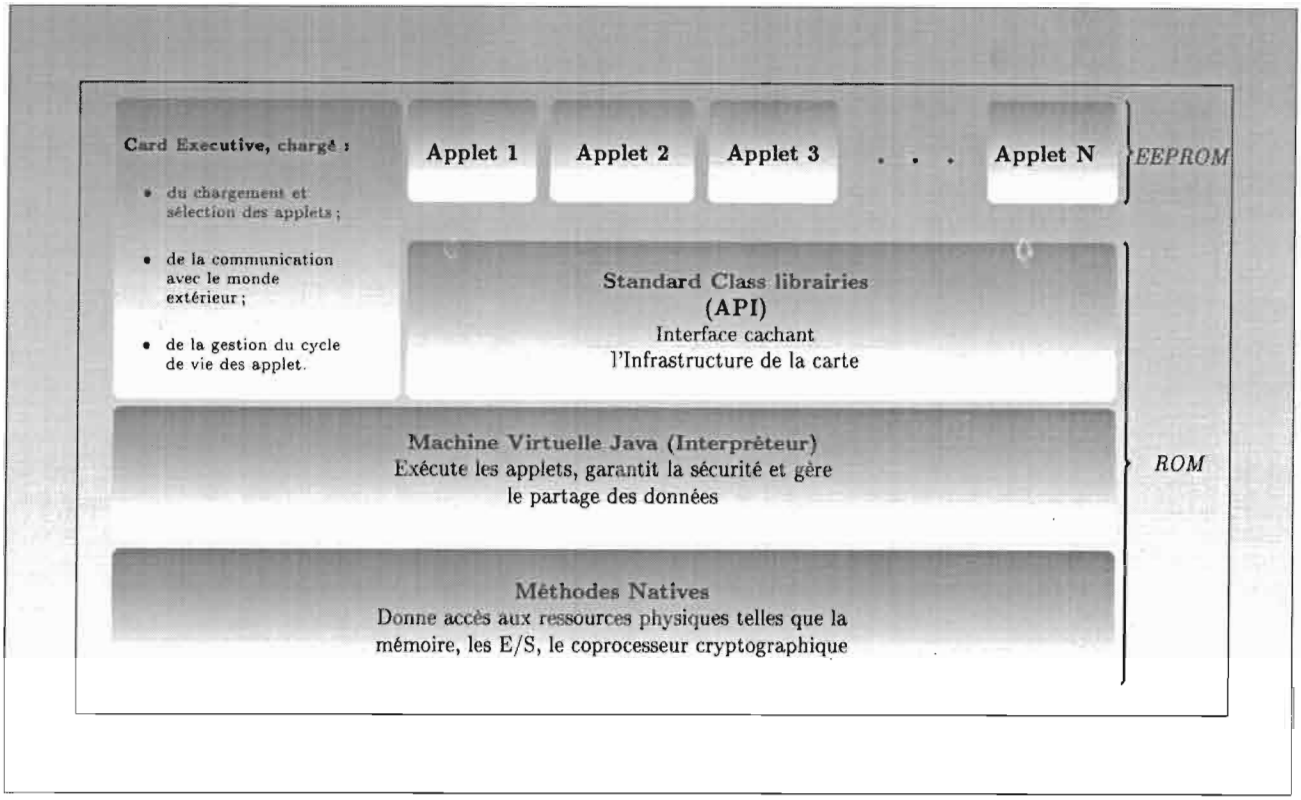
Notre travail sera axé sur le cas spécifique des cartes intelligentes (Smart Card) Java Card. Le choix de cette catégorie de carte car il rentre dans le cadre des recherches de notre structure commanditaire, en plus du fait que la Java Card est le type de carte le plus rependu au monde [4].

1.1.2 Technologie Java Card

La technologie Java Card est une combinaison d'un sous-ensemble du langage de programmation orienté objet fortement typé (Java) avec un environnement d'exécution optimisé pour les Smart Card et les périphériques embarqués disposant de ressources limitées [13]. Une carte à puce est qualifiée de Java Card si elle implémente un environnement d'exécution de programme java en son sein appelé Java Card Runtime Environment (JCRE). Un programme java s'exécutant sur une Java Card est qualifié d'applet. Une applet Java Card est une classe Java standard étendant la classe javacard.framework.Applet (applet) (confère Annexe B pour un exemple d'applet). La JCRE est décrite par un ensemble de spécifications ouvertes et est principalement constituée :

- d'une machine virtuelle ou Java Card Virtual Machine (JCVM) chargée entre autres de l'interprétation, l'exécution des programmes, la sécurité, le partage des données ainsi que l'interfaçage avec les api natives ;
- d'Application Programming Interface (API) standards qui sont des librairies cachant l'infrastructure de la carte ;
- des technologies chargées du chargement des programmes, de leur cycle de vie...

La Figure 2, présente les constituants énumérés.
La technologie Java Card permet au développeur Java standard de pouvoir, à partir de leur connaissance initiale, de développer des applets (en respectant un minimum de contrainte) sans reconversion nécessaire. Cela permit l'ouverture du domaine préalablement réservé aux industriels (en référence au système fermé monolithique), au monde des développeurs tiers.
La nature des applets (programme sensible comme l'authentification...) ainsi que la diversification de leurs sources de productions a conduit à la nécessité d'avoir des outils de vérification de ces applications. De nombreuses techniques ont été mises en œuvre telles que les méthodes formelles [5], la vérification de code binaire d'applet augmenté [11].
Notre objectif à travers nos travaux, est de permettre l'intégration d'une technique d'ingénierie logiciel générique, la couverture de code dans les étapes de création d'applet Java Card. En vue de mieux cerner l'enjeux, nous allons procéder à la présentation de la couverture de code en environnement java.



Figures 2 – Structure d'une JCRE

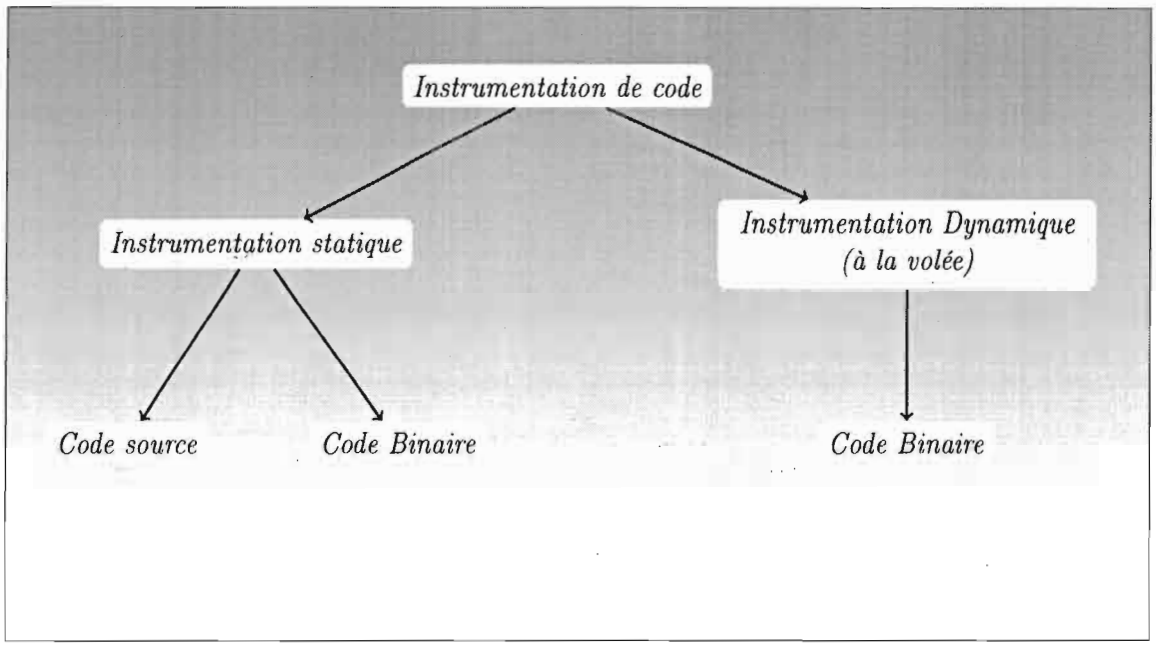
1.2 La Couverture de Code en Java

La couverture de code est une mesure qui permet d'identifier le pourcentage du code testé [7]. Elle permet de ce fait de déterminer la qualité des tests que l'on fait subir aux programmes. Il existe plusieurs méthodes de couverture de code, dont certaines sont la résultante de la combinaison d'autres méthodes de couverture basiques. En termes de méthodes basiques nous distinguons :

- la couverture des instructions[5] (Statement Coverage) : elle permet de déterminer le taux de couverture au niveau instructions du programme ;
- la couverture des fonctions (Method/function Coverage) : c'est une métrique qui permet de déterminer le taux de couverture des fonctions du programme ;
- la couverture des structures de contrôles (Condition/branch Coverage) : elle permet de déterminer le taux de couverture des structures de choix conditionnel tels que *if*, *switch*... ;
- la couverture des chemins d'exécution (Path Coverage) : elle permet de déterminer le taux de couverture de l'ensemble des chemins d'exécution possibles ;

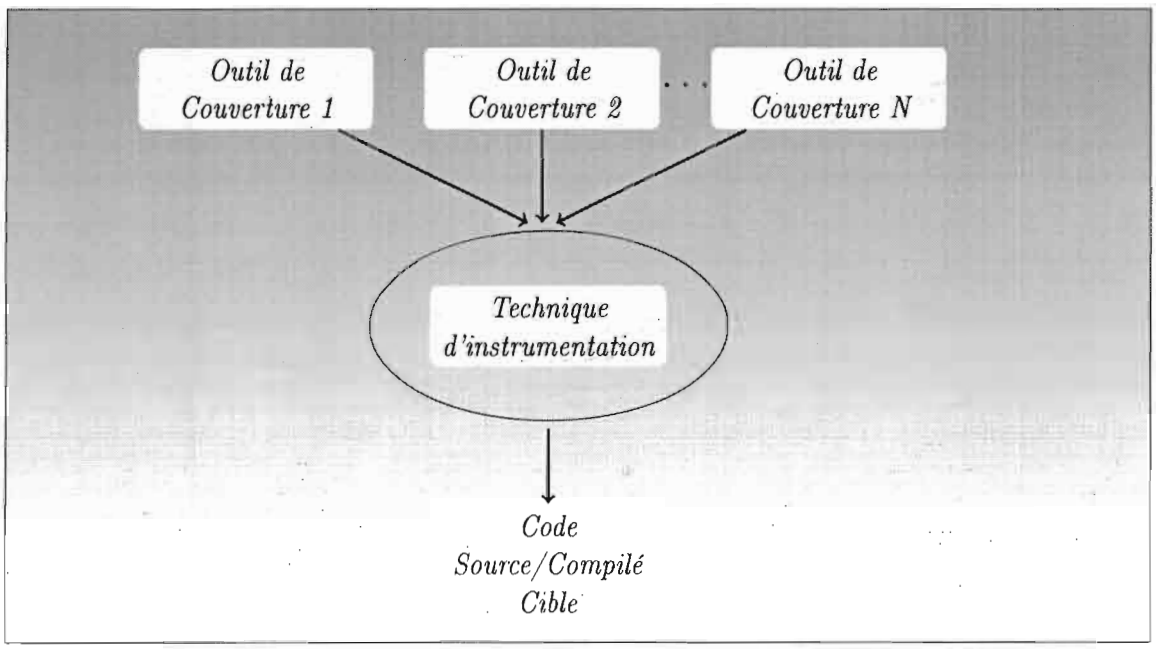
La mise en œuvre technique des méthodes de couverture de code s'appuie sur la technique d'instrumentation. Par analogie avec l'instrumentation des sciences, l'instrumentation de code consiste à l'ajout d'élément de mesure appelé sonde (probe) dans un programme, sans impact sur le code source originel. L'instrumentation peut être réalisée avant ou au cours de l'exécution du programme ; on parle respectivement d'instrumentation statique et/ou dynamique comme l'illustre la Figure 3.

5. une instruction est une commande basique native ou non (instruction intermédiaire), pouvant être directement exécutée par un processeur.



Figures 3 – Techniques d'instrumentation

Les techniques d'instrumentation employées permettent de caractériser les outils de couverture de code au-delà de leurs aspects propres (rendu...) comme illustré au niveau de la Figure 4. De par ce constat nous avons opté de présenter les différentes techniques d'instrumentation.



Figures 4 – Positionnement des techniques d'instrumentation au niveau de la couverture de code.

1.2.1 Instrumentation statique

L'instrumentation statique peut être réalisée en se servant soit du code source ou du programme déjà compilé (code machine intermédiaire utilisant un jeu d'instruction indépendant d'une architecture matérielle spécifique (bytecode)).

1.2.1.1 Instrumentation du code source

Ce type d'instrumentation débute par une analyse du code source du programme en se servant d'analyseurs lexical et syntaxique adaptés au langage utilisé. Cette analyse permet d'identifier le Graphe de Contrôle de Flux (Control Flow Graph) (CFG) (ensemble des chemins possibles lors de l'exécution d'un programme) et de placer ainsi des sondes suivant les nécessités de la méthode de couverture de code implémentée.

L'avantage de cette technique est de produire un code qui reste exploitable par la grande majorité des développeurs d'applications.

L'inconvénient de cette technique est sa limite lors de l'usage par exemple de bibliothèques propriétaires ou importées où l'accès au code source est restreint. A cela s'ajoute la complexité d'analyse lors de sa mise en œuvre. L'un des outils implémentant cette technique est l'outil de couverture Clover de Atlassian [1].

1.2.1.2 Instrumentation du code compilé

Cette technique est basée sur le parcours et l'analyse des instructions machine résultantes de la phase de compilation en vue de placer des sondes conformément à la méthode de couverture de code devant être implémentée.

L'avantage de ce type d'instrumentation est qu'il est assez répandu dans l'univers Java. Cette forte prévalence se justifie par le concept de machine virtuelle mise en œuvre au niveau de la technologie Java, garantissant ainsi la portabilité des programmes instrumentés sur diverses plateformes.

L'inconvénient de cette méthode réside dans le fait que les résultats ne restent compréhensifs uniquement qu'au programmeur du domaine et le taux d'erreurs est également assez élevé.

1.2.1.3 Approche orientée aspect

La programmation orientée aspect, est un paradigme de programmation dans lequel les préoccupations transversales (appelé aspect, etc) d'un programme (telles que par exemple la sécurité, la gestion des logs...) sont traitées séparément. Le principe est de permettre l'expression simple des préoccupations métier tout en étant déchargé des implications techniques induites. Le tisseur d'aspects est un programme qui a la charge d'intégrer les différents codes des aspects appelés greffons, à des points précis du programme, déterminés ou choisis (les points de jonction) à l'aide de la technique d'instrumentation.

Dans notre contexte d'étude qui est la couverture de code, la préoccupation transversale est la collecte de données lors de l'exécution du programme.

L'avantage est que le greffon remplace la sonde en nous donnant les informations requises, réduisant considérablement la tâche du développeur.

L'inconvénient majeur est la condition d'exécution du greffon. En effet, celui-ci est fonction d'élément déclencheur comme l'appel d'une fonction pouvant occasionner une grande perte d'informations par rapport aux méthodes d'instrumentation statique classique.

La technique d'instrumentation basée sur l'approche orientée aspect est très utilisée par la plupart des outils de couverture de code car indépendante dans une certaine mesure de la plateforme (portable). D'autres techniques d'instrumentation offrent une plus grande portabilité, car mise en œuvre lors de la phase d'exécution, telle que l'instrumentation dynamique.

1.2.2 Instrumentation Dynamique

L'instrumentation dynamique consiste à l'ajout de sondes soit durant la phase de chargement du programme, ou par interception des différents appels au cours de la phase d'exécution.

1.2.2.1 Instrumentation lors de la phase de chargement

La machine virtuelle Java utilise pour charger ses classes, la classe *ClassLoader*. L'idée est de surcharger [6] cette classe afin de modifier les octets des fichiers compilés (fichier .class) ciblés lors du chargement, comme illustré dans Code Source 1.

```
public static class MemoryClassLoader extends ClassLoader {

        private final Map<String, byte[]> definitions;
        definitions = new HashMap<String, byte[]>();

        /**
        * Add a in-memory representation of a class.
        *
        * @param name
        *            name of the class
        * @param bytes
        * class definition
        */
        public void addDefinition(final String name, final byte[] bytes) {
                definitions.put(name, bytes);
        }

        @Override
        protected Class<?> loadClass(final String name, final boolean resolve)
                        throws ClassNotFoundException {

                final byte[] bytes = definitions.get(name);

                if (bytes != null) {
                        return defineClass(name, bytes, 0, bytes.length);
                }
                        return super.loadClass(name, resolve);
                }

        }
```

Code Source 1 – Chargeur d'un ClassLaoder modifié du plugin d'instrumentation Java ASM

Depuis la version 5 de Java, la machine virtuelle offre de façon native, la possibilité en plus de la surcharge du ClassLoader, de créer des programmes appelés agents Java à même d'instrumenter le code lors de l'exécution. Un agent Java est un programme sous la forme d'un fichier jar (Java archive), dont l'une des classes implémente les méthodes *premain(...)*, point

6. Concept orienté objet qui consiste à définir une classe qui redéfinit les propriétés d'une autre classe en vue de les étendre.

d'entrée de l'agent. Code Source 2 nous donne un exemple d'agent Java chargé de réaliser la gestion des traces d'exécution (logging).

```
/* la methode premain */
public class MyAgent {
    public static void premain(String agentArgs, Instrumentation inst);
        inst.addTransformer(new PrintingTransformer());
    }
}

/* un exemple de tranformer */
public class PrintingTransformer implements ClassFileTransformer {

    public byte[] transform(ClassLoader loader, String fullyQualifiedClassName,
                            Class classBeingRedefined, ProtectionDomain
                               protectionDomain, byte[] classofileBuffer)
                               throws IllegalClassFormatException {

        String className = fullyQualifiedClassName.replaceAll(".*/", "");
        String package=fullyQualifiedClassName.replaceAll("/[a-zA-Z$0-9_]*$","");
        System.out.printf("Class: %s in: %sn", className, package);
        return null;

    }
}
```

Code Source 2 – Chargeur d'un ClassLaoder modifié du plugin d'instrumentation Java ASM

1.2.2.2 Instrumentation dynamique à l'exécution

Le principe est d'utiliser les interfaces d'écoute de la machine virtuelle telles que Java Virtual Machine Profiler Interface (JVMPI) ou JVM Tool Interface (JVMTI) (disponible à partir de la version 5 de Java) afin d'intercepter un nombre défini d'événements en plaçant des crochets (hooks) pour l'écoute des évènements suivants :

- le chargement de classes ;
- le démarrage et fermeture de *thread* ;
- l'entrée et la sortie de méthodes ;
- le chargement et le déchargement des méthodes compilées.

Le talon d'Achille de cette technique réside fondamentalement au niveau du nombre restreint de modèles d'écoute contrairement à l'instrumentation classique statique.

1.2.2.3 Instrumentation dynamique hybride

Le principe de cette solution est l'usage des interfaces d'écoute mise à disposition par la machine virtuelle pour instrumenter le bytecode en se servant d'un environnement natif en C/C++. Cette technique tend à créer des machines virtuelles personnalisées causant ainsi la perte de portabilité, qui est l'un des piliers fondamentaux de la technologie Java «*write once read anywhere* ».

1.2.2.4 Tissage d'aspect dynamique

Le tissage d'aspect dynamique fonctionne de manière similaire à sa version statique à la seule différence que le tissage a lieu lors de l'exécution du programme. Cette méthode est très utilisée au niveau des serveurs d'applications pour les besoins de l'exécution des programmes. Le tissage d'aspect dynamique peut être associé à d'autres technologies telles que hotswap (pour exploiter les possibilités offertes par la machine virtuelle). Parmi les solutions implémentant ce paradigme d'instrumentation nous pouvons citer JBoss AOP ,AspectWerkz.

L'ensemble de ces techniques d'instrumentation sont dans le principe applicable à la plupart des langages de développement dont Java. Néanmoins, leur mise en œuvre induit des changements dont la nature peut varier d'une technologie Java spécifique (Java standard) à une autre (Java Card). Dans la partie suivante, nous allons présenter les structures internes de la technologie Java standard et Java Card, afin de percevoir les changements que pourrait causer une instrumentation de code.

1.3 Étude des structures internes des programmes Java et Java Card

Afin de mieux appréhender l'essence de nos travaux, nous allons procéder à une présentation des différentes structures de la technologie Java, notamment sa machine virtuelle, ses variantes Java standard et Java Card.

1.3.1 Présentation de la machine virtuelle Java [3]

Le principe en Java est que chaque programme est exécuté dans une tâche (*thread*). Chaque tâche dispose pour son exécution d'une pile (*stack*) ; cette pile est décomposée en cadre (*frame*). Chaque frame est un espace alloué lors de l'invocation d'une fonction au cours de l'exécution et est subdivisé en deux parties :

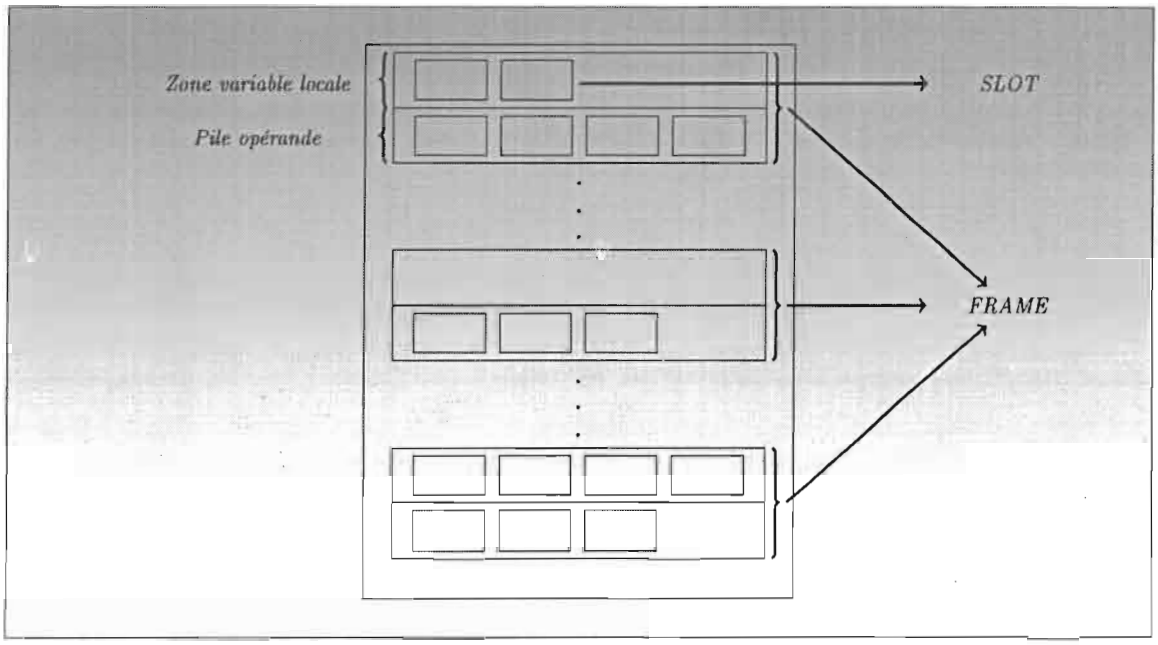
- une partie pour les variables locales ;
- une autre qui est en fait une petite pile destinée aux opérandes [7].

Chaque partie est composée d'espace mémoire atomique de rangement (slot) permettant de stocker les données. Pour ce qui est des variables locales, le nombre de *slot* nécessaires est fonction de la donnée à stocker (booléen codé sur 1 *slot* et réel sur 2 *slot* dans le cadre du langage Java standard). La Figure 5 illustre la description de la pile.

Lors de la phase de compilation, le code Java est converti en bytecode, la description de certaines instructions machine est donnée par l'Annexe C.

Le bytecode Java, est un code intermédiaire généré par un compilateur Java avec un jeu d'instructions propre à la Java virtual Machine (machine virtuelle java) (JVM) indépendant de toutes plateformes (linux, Windows, Mac...). Le bytecode est stocké dans des fichiers contenant du code binaire java standard (fichier à extension .class) et des fichiers contenant du code binaire java card dont l'extension provient de la contraction de Converted et APplet (fichier à extension .cap). Une instruction bytecode (illustré par le pseudo Code Source 3) est composée d'une ou de deux partie(s), le code d'opération machine java (opcode), ainsi que d'arguments optionnels selon la nature de l'opcode.

7. un opérande est une valeur sur laquelle une instruction, nommée par sa mnémonique, opère. Cet opérande peut être un registre, une adresse mémoire, une valeur littérale ou une étiquette



Figures 5 – Pile d'exécution de thread

```
<Opcode> [argument 1] [argument 2] [argument 3] . . . [argument x]
```

Code Source 3 – Pseudo Code d'une Instruction Bytecode

Les codes d'opération sont codés sur un octet non signé (0-255) ou chacune des valeurs représente une instruction et est désignée par une mnémonique. Les arguments sont quant à eux les paramètres de l'opcode qui sont connus et stockés de façon statique dès la phase de compilation contrairement aux variables locales qui ne sont elles connues que durant la phase d'exécution. Une description plus exhaustive du fonctionnement de la machine virtuelle est faite dans l'Annexe A.

1.3.2 Technologie Java Standard

Dans la technologie Java standard une classe compilée (fichier à extension .class) ne contient que le bytecode d'une seule classe. Si le fichier de source d'origine de la classe contient une classe interne, un autre fichier sera créé pour la description de cette classe, et une référence sera simplement faite au niveau du fichier à extension .class. Un fichier à extension .class est obtenu après le processus décrit au niveau de la Figure 6.

Lorsqu'une classe est compilée, il est réservé une section pour décrire (sous forme de méta-données) la classe, ses champs ainsi que ses méthodes. La description du contenu d'un fichier .class est faite par le Tableau 1[2].

Le pool de constantes (constant pool) est un tableau dont les valeurs peuvent être soit un élément, soit un pointeur vers d'autres éléments. Elle contient les informations de description de la classe (version, nom de la classe, nom, type des attributs, accesseurs...). Une description plus complète de la classe peut être obtenue en utilisant la commande Code Source 4.

```
javap -v -p -s -sysinfo -constants nom_classe.class
```

Code Source 4 – Commande descriptive du contenant d'un fichier ".class"

Modificateurs, nom, super classe, interfaces	
Zone des constantes (constant pool)	
Nom du fichier source (optionnel)	
Référence de la classe englobant	
Annotation*	
Attribut*	
Classe interne*	Nom
Champ*	Modificateurs, nom, type Annotation* Attribut*
Méthode*	Modificateurs, nom, type de retour Annotation* Attribut* Code compilé

Tableau 1 – Contenu des différents champs d'un fichier « .class »

Les échanges entre la carte et les terminaux externes se font sous forme d'Application Protocol Data Unit (APDU) ou unité de données de niveau application [10]. Une APDU est une séquence d'octets échangées entre une Smart Card et une application cliente. On distingue deux types d'APDU :

- les APDU de commande (Tableau 2) ;
- les APDU de réponse (Tableau 3).

Champs	**Taille (en octet)**	**description**
CLA	1	Classe d'instruction : indique le type de la commande, Inter industrie (réservée) ou propriétaire
INS	1	Code d'instruction : indique le code de commande, "write data" par exemple
P1-P2	2	Paramètres d'instructions pour la commande, par exemple la position du curseur (offset) du fichier où écrire des données
Lc	0, 1 ou 3	Définit le nombre (Nc) d'octets envoyés par la commande
Données envoyés	Nc	Nc octets
Le	0, 1, 2 ou 3	Définit le nombre (Ne) maximum d'octets attendus dans la réponse

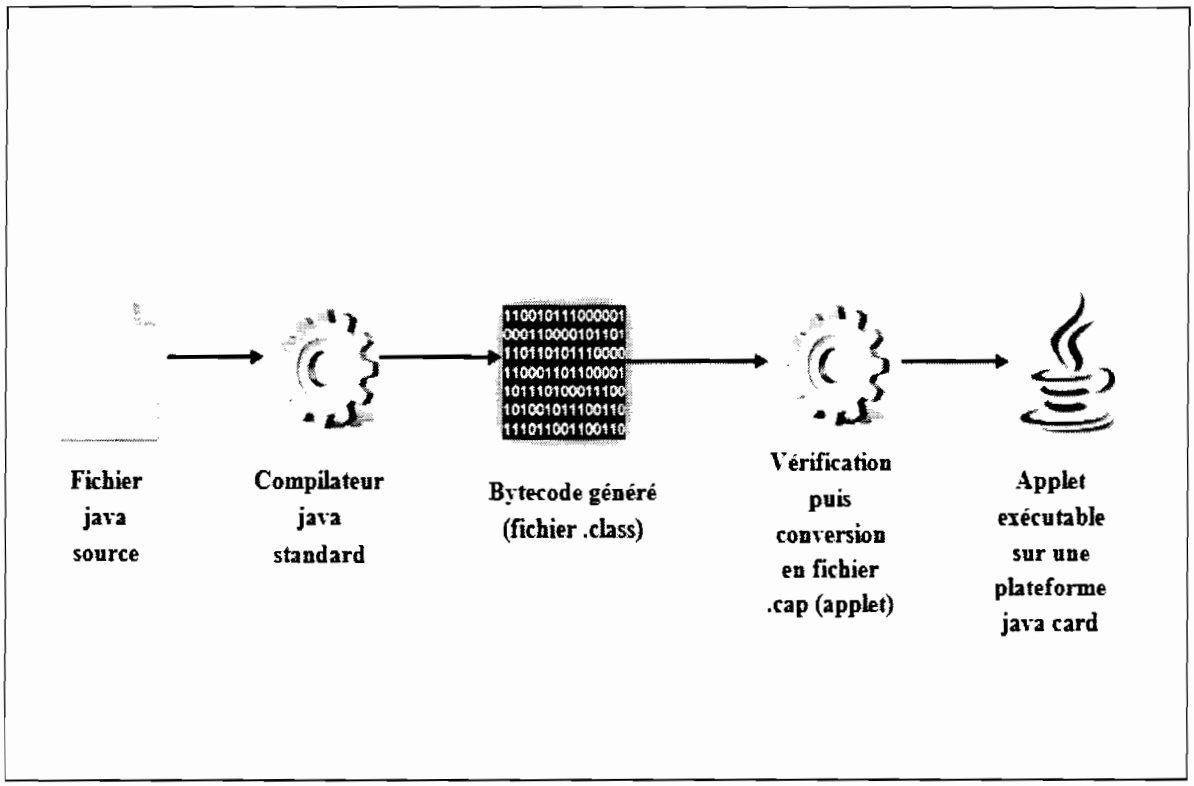
Tableau 2 – APDU de commande[10]

Champs	Taille	description
Réponse	Nr (au maximum Ne)	Donnée de réponse
SW1-SW2(Statut de la réponse)	2	Statut résultant de la commande, par exemple 90 00 (hexadécimal) indique que l'opération a été effectuée avec succès.

Tableau 3 – APDU réponse[10]

La compilation d'une applet suit les étapes suivantes (expliciter par la Figure 8) :

1. Écriture de l'applet avec un environnement de développement Java standard ;
2. Compilation des fichiers contenant du code source java (fichier à extension .java) en bytecode Java (fichier à extension .class) ;
3. Vérification du bytecode du fichier à extension .class et conversion du bytecode vers un fichier à extension .cap.



Figures 8 – Étape de compilation d'une applet

Le fichier à extension .cap est accompagné d'un autre fichier. Il s'agit d'un fichier binaire produit lors de la phase de génération du fichier .cap; il contient l'ensemble des informations qui permettront au fichier .cap, d'être à son tour importer dans une autre applet java card (fichier à extension .exp).

Le bytecode d'un fichier à extension .cap [13] est structuré en composants. Un composant est lui même un fichier à extension .cap chargé du stockage (sous forme binaire) des constituants (classe, méthode...) de l'applet. Le Tableau 4, décrit les différents composants standard d'une applet Java Card.

Fichier	Composant	Description
Header.cap	Composant Header	Il contient les informations générales sur le fichier « .cap » ainsi que les packages qui y sont définis.
Directory.cap	Composant Directory	"Liste la taille de chaque composant du fichier « .cap » (composant header directory...)."
Applet.cap	Composant Applet	Contient une entrée pour chaque applet défini dans ce package.
Import.cap	Composant Import	Contient la liste de l'ensemble des packages importés par les classes du fichier « .cap » courant.
ConstantPool.cap	Composant Constant Pool	"Il contient une entrée pour chaque classe méthode champ référencé par une méthode."
Class.cap	Composant Class	Décrit chaque classe et interface de ce fichier « .cap »
Method.cap	Composant Method	Décrit chaque méthode présent dans le fichier « .cap » sauf les déclarations d'interface et block d'initiation statique
StaticField.cap	Composant Static field	Contient toutes les informations requises pour créer et initialiser tous les éléments statiques du fichier « .cap ».
RefLocation.cap	Composant Reference Location	Contient les informations permettant d'identifier dans le composant Method les opérandes des opcodes representant des index d'éléments du composant Constant Pool exemple d'instruction (getstatic_a, getfield_t)
Export.cap	Composant Export	"Liste tous les éléments statiques de la classe pouvant être exportés dans d'autres packages. Il s'agit des (éléments avec la visibilité publique et statique par exemple)."
Descriptor.cap	Composant Descriptor	Fournit les informations nécessaires pour l'analyse et la vérification de tous les éléments d'un fichier « .cap »
Debug.cap	Composant Debug	Contient toutes les métadonnées nécessaires pour le débogage.

Tableau 4 – Structure interne d'un fichier « .cap »

1.3.4 Problématique

L'étude structurale des deux technologies Java et Java Card nous a permis de déceler de grandes divergences notamment :

1. la présence d'un constant pool par classe au niveau du Java standard et d'un espace unique pour toutes les classes au niveau de Java Card ;
2. le bytecode des méthodes mutualisées au sein d'un même fichier sous Java Card, tandis que sous Java standard chaque fichier de classe contient le bytecode de ses méthodes ;
3. l'information de débogage mutualisée au sein d'un même fichier sous Java Card, tandis qu'il accompagne chaque bytecode de méthodes sous Java standard ;
4. l'introduction du concept de composant au niveau des applets ;
5. le jeu d'instruction réduit au niveau du Java Card par rapport à Java standard ;
6. la non prise en charge de l'instrumentation dynamique par la JCVM.

La divergence des structures internes des programmes Java (Java standard versus Java Card) empêche l'usage des outils d'instrumentation Java standard au niveau de Java Card.

Ceci a pour corollaire d'invalider de prime abord l'usage des outils de couverture standard au niveau de la technologie Java Card.

Afin de solutionner ce problème, nous allons proposer une stratégie issue de l'analyse du fonctionnement des deux technologies (Java standard et Java Card). L'idée maîtresse ici est d'adapter un outil de couverture de code standard à un usage sous Java Card en résolvant les disparités là où elles seront constatées.

Chapitre 2

Détermination et description d'une solution de résolution du problème

2.1 Analyse et choix d'une Solution

Nous allons débuter par la présentation de solutions potentielles décrittent en stratégies ; puis procéder au choix de l'une d'entre elle sur la base de critères qui seront détaillés.

2.1.1 Analyse de stratégies possibles

L'analyse des schémas de compilation des technologies Java standard et Java Card nous ont permis d'élaborer deux stratégies possibles.

2.1.1.1 Stratégie 1 : instrumentation pré-génération d'applet

Présentation

Cette stratégie tire sa substance de l'analyse comparée du schéma de compilation d'une applet (Figure 8) et du schéma de compilation d'un programme Java standard (Figure 6). Le but ici est d'intercepter le bytecode généré lors de la phase de compilation standard, afin d'y instrumenter les sondes conformément aux spécifications d'un outil de couverture de code que nous aurons préalablement choisi et étudié. Ce bytecode instrumenté sera ensuite converti en applet comme illustré au niveau de la Figure 9.

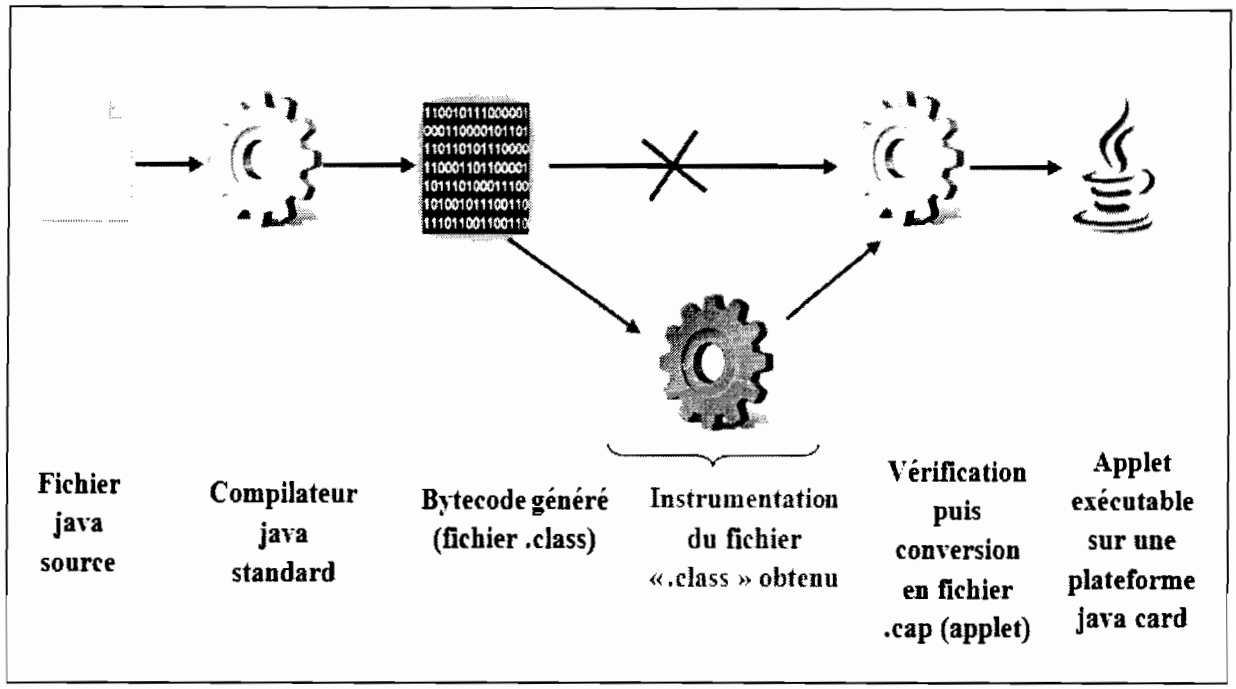
Avantage

L'avantage de cette stratégie réside dans le fait que les outils nécessaires pour réaliser l'instrumentation peuvent être des outils standard de couverture, donc déjà existants.

Inconvénients

Les inconvénients sont quant à eux multiples. Bien que les outils standards puissent être employés, la grande difficulté induite serait la récupération de données collectées lors des phases de test. Toute communication externe étant réalisée à partir d'APDU. Cela nécessiterait donc :

- préalablement l'étude de l'outil de couverture choisie en vue d'identifier la structure des sondes ;
- la mise en place d'un algorithme en vue de détecter les sondes placées ;
- la mise en place d'une stratégie de récupération des données après les tests effectués.



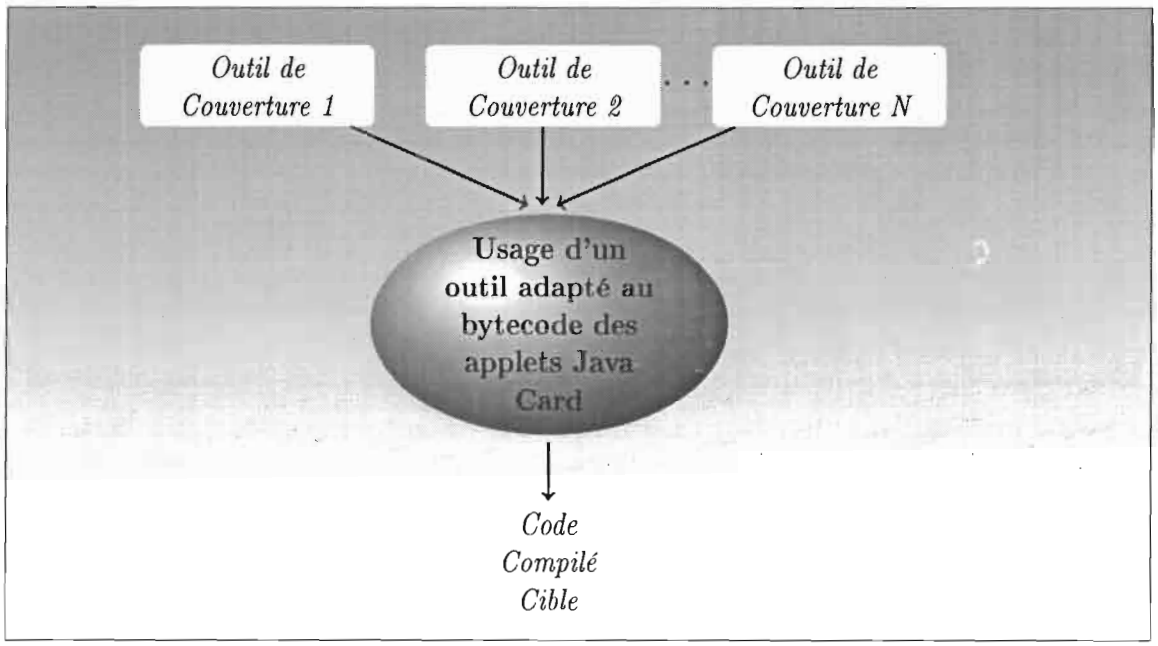
Figures 9 – Instrumentation du bytecode java (stratégie 1)

La mise en œuvre de l'ensemble des points cités ci-dessus nécessiterait donc l'usage d'un deuxième outil d'instrumentation propre à la structure d'une applet Java Card. Un autre point faible de cette démarche, réside dans le fait quelle pose la contrainte primordiale de disposer du code source des applets.

2.1.1.2 Stratégie 2 : instrumentation post-génération d'applet

Présentation

Pour cette seconde stratégie, l'idée consiste à exploiter la modularité existante au sein de certains outil de couverture de code telle JaCoCo ou encore Cobertura, comme indiqué au niveau de la Figure 10. Ces outils délèguent l'étape d'instrumentation à des librairies spécialisées telle que la bibliothèque d'instrumentation de code binaire java (ASM) ou Apache BCEL. La stratégie serait donc de remplacer l'outil d'instrumentation d'une de ces bibliothèques par un outil adapté à la manipulation du bytecode des applets comme l'illustre la Figure 10. Les résultats obtenus après la phase des tests pourront ensuite être mis en forme et réacheminés vers l'outil de couverture choisi en vue du rendu.



Figures 10 – Instrumentation du bytecode java card

Avantages

Tout comme la première stratégie, cette méthode permet l'utilisation des outils de couverture déjà existants. La modification s'opérant principalement sur le module d'instrumentation, le caractère portable du module de couverture demeure (vis à vis des différentes plateformes ou il sera exécuté). Cette stratégie permet également de gérer de façon intégrale la mise en œuvre de l'instrumentation des sondes permettant ainsi une bonne prise en compte des spécificités de la plateforme.

Inconvénients

Cette stratégie induit les contraintes suivantes :

- nécessité de définir un algorithme chargé de repérer les points d'insertion des sondes adaptées au bytecode Java Card ;
- nécessité de définir une structure de sonde adaptée aux spécifications de l'outil de couverture employé ;
- nécessité également de définir une stratégie adaptée de récupération des données après tests.

2.1.2 Stratégie retenue

L'analyse des avantages et inconvénients des deux stratégies précédemment présentées nous a permis de faire le choix de la stratégie 2. Le choix s'est fait en raison des critères assez importants et intéressants qu'elle revêt à savoir :

- son caractère portatif de la solution ;
- son caractère modulaire (adaptable a plusieurs outils de couverture) ;
- son caractère autonome (module faiblement couplé avec un module tiers).

la section suivante sera consacrée à la description des outils nécessaires à la mise en œuvre de la stratégie retenue.

2.2 Description des outils nécessaires à la mise en œuvre de la stratégie

Nous allons dans cette section préalablement débuter par la détermination des outils nécessaires, puis à une analyse du fonctionnement de l'outil de couverture retenu (pierre angulaire de cette stratégie).

2.2.1 Choix des Outils

Les outils nécessaires à la mise en œuvre de la stratégie retenue seront principalement :

- un outil de couverture de code Java ;
- un outil d'instrumentation adapté aux applets Java Card ;
- les outils divers nécessaires à l'utilisation des outils précédant (outils d'accompagnement).

2.2.1.1 Outil de couverture

Notre choix pour cette étape s'est porté sur l'outil de couverture de code (JaCoCo) (Java Code Coverage) [6]. Ce Choix d'abord proposé par notre directeur de mémoire, se justifie par l'adéquation quasi complète de l'outil avec de la problématique. En effet, cet outil est caractérisé par les éléments suivants, classés par ordre de pertinence :

- la prise en compte de l'instrumentation statique (mais aussi dynamique) ;
- l'outil de couverture de code et d'instrumentation sous licence open-source ;
- l'usage d'un outil d'instrumentation autonome ASM [2] ;
- l'outil composé de module d'application (bundle) faiblement couplé, norme Open Service Gateway Initiative, ensemble de spécification décrivant un paradigme de programmation orienté composant et une architecture orientée service (OSGi) ;
- le projet purement Java ne dépendant que très peu de librairie externe (ASM) ;
- la documentation très fournie et accessible ;
- la prise en compte des méthodologies de couverture par classe, fonction, ligne, bloc basic ;
- la possibilité de fusionner des résultats de test distinct ;
- une bibliothèque disponible sous forme de plugin pour certains IDE ;
- l'usage possible pour les projets à grand échelle et aussi pour ceux à échelle réduite ;
- la possibilité de couvrir les tests de classe ou projet entier ;
- le format varié de rendu des résultats après test (texte, xml, html).

Le couplage faible vis-à-vis de la librairie ASM, permet ainsi de le remplacer avec une librairie adaptée au contexte Java Card.

2.2.1.2 Outil d'instrumentation

Le choix de l'outil d'instrumentation quant à lui ne souffre pas de grandes discussions. Cela s'explique par le fait que les outils existants sont mis en œuvre dans le cadre de projet d'expérimentation. Il a été mis à notre disposition une bibliothèque de manipulation de bytecode java card (CapMap) [15]. Cette librairie développée par l'équipe SSD (Smart Secure Devices Team) du laboratoire XLIM Labs de l'Université de Limoges France, permet de lire puis d'instrumenter le bytecode Java Card.

2.2.1.3 Outils complémentaires

Les outils complémentaires sont quant à eux constitués de l'ensemble des outils qui devraient accompagner la mise en œuvre de la stratégie choisie. Il s'agit :

- d'un lecteur de carte à puce, permettant d'assurer la communication physique entre la Smart Card et l'application cliente ;
- d'une Smart Card Java Card ;
- d'une bibliothèque pour les échanges logiques avec la Smart Card ;
- d'un environnement de développements ;
- d'une bibliothèque de développements d'applets ;
- d'un environnement Java Card.

Les choix opérés sont consignés au niveau du Tableau 5.

Besoins	Choix
Lecteur de carte à puce	Gemalto PC Twin Reader
Carte à puce Java Card	Gemalto TOP IM GX4
Bibliothèque pour la communication avec la Java smartcard	OPAL 4.0 [16] & PC/SC Lite 1.8.14
Environnement de développement	Eclipse mars, Eclipse Neon
Plugin de développements d'applet sous eclipse	Eclipse-JCDE version 0.3
Outils d'exploration de classe pour éclipse	ObjectAid version 1.1.13
Environnement Java Card	Java Card 2.2.1
Décompilateur Java	JD-GUI 1.4.0, JD-Core 0.7.1
Editeur de fichier binaire	GHex 3.18.0
Système d'exploitation Utilisé	Ubuntu 14.04 & 16.04

Tableau 5 – Choix des outils d'accompagnement

2.2.2 Le module CapMap

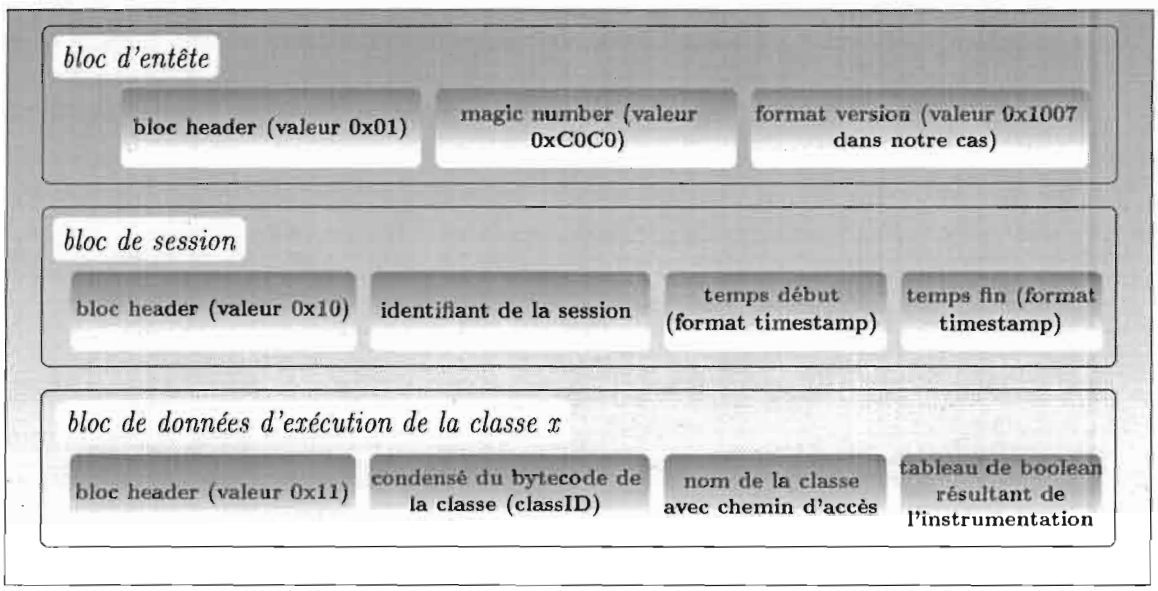
le module CapMap est un programme écrit en Java, qui implémente sous forme de classe les éléments de la structure d'un fichier à extension .cap défini au sein de la spécification JCVM[13]. Ceci lui permet de lire un fichier à extension .cap passé en argument, de l'interpréter afin de retrouver les différents composants et leurs constituants internes afin de le rendre manipulable. Il permet d'instrumenter de façon statique (ajouter, remplacer, supprimer) des instructions dans le bytecode des méthodes, ou de modifier explicitement d'autres composants comme le composant Reference Location. Son but initial est de permettre de réaliser des attaques sur des programmes Java Card afin d'éprouver la sûreté des Smart Card.

2.2.3 Fonctionnement logique de JaCoCo

« Une sonde est une séquence d'instructions de code intermédiaire qui peuvent être insérée dans une méthode Java. Lorsque la sonde est exécutée, ce fait est enregistré et peut être signalé dans l'exécution de la couverture. La sonde ne doit pas modifier le comportement du code original. » [1]. Une sonde sera assimilée à un drapeau (*flag*), qui ne se déclenche que si elle est exécutée **au moins une fois** lors des tests. Les sondes devront pouvoir :

- enregistrer les exécutions ;
- être identifiées ;
- pouvoir être exécutées dans un environnement *multi thread* (*thread-safe*) ;
- ne pas avoir d'impact sur le fonctionnement du programme d'origine ;
- avoir un temps d'exécution négligeable.

Pour satisfaire ces propriétés, les sondes sous JaCoCo sont représentées par un tableau de boolean [2], où chaque entrée du tableau correspond à une sonde instrumentée dans le CFG. JaCoCo a pour stratégie de créer un tableau de sondes par classe. Pour ce faire, pour chaque classe il dénombrera le nombre de sondes à insérer, initialisera le tableau de boolean avec la taille adéquate, puis procédera à l'insertion des sondes. Au début d'exécution des tests (appelé session), JaCoCo procédera préalablement à la création d'une structure qui sera chargée d'identifier la session de test (bloc de session). Puis pour chaque classe, il constituera une structure (bloc de données) qui sera chargée de conserver les résultats obtenus (état du tableau de sonde) à la fin de la session de test. Tous ces éléments seront persistés dans un fichier binaire contenant les résultats d'exécution des tests de JaCoCo (fichier à extension .exec) dont la description est faite par la Figure 11 ; ce fichier servira ensuite en association avec le bytecode et éventuellement le code source des programmes à générer des rapports aux formats telles que csv ou html...



Figures 11 – Structure d'un fichier d'exécution (.exec)

1. Traduction de la definition " A probe is a sequence of bytecode instructions that can be inserted into a Java method. When the probe is executed, this fact is recorded and can be reported by the coverage runtime. The probe must not change the behavior of the original code."[6]

2. type de données ne pouvant prendre que deux valeurs possibles, vrai ou faux

Le module JaCoCo est composé d'un ensemble de sous modules plus ou moins indépendants (norme OSGi), jouant un rôle défini dans les séquences d'étapes nécessaires à la couverture de code d'un programme. Nous distinguons :

- le module *org.jacoco.core* ;
- le module *org.jacoco.jacocoagent* ;
- le module *org.jacoco.ant* ;
- le module *org.jacoco.report.*

2.2.3.1 La librairie ASM

ASM est un framework léger de manipulation de bytecode (provenant du code compilé de classe Java standard) développé par l'Institut National de Recherche en Informatique et en Automatique (INRIA) et France Telecom sous licence libre. Il offre la possibilité :

- de lire et interpréter le bytecode de classe Java ;
- d'instrumenter aussi bien de manière dynamique que statique le bytecode d'une classe ;
- de créer entièrement une classe à partir d'instructions bytecode ;
- ...

ASM est conçu sur le modèle de conception *Visitor* (confère Annexe D), ce qui facilite l'ajout, la réutilisation de fonctionnalité du module sans nécessité sa modification, en ajoutant simplement de nouvelle classe appelé adaptateur. JaCoCo définit des adaptateurs afin d'interagir avec ASM lorsqu'il est nécessaire d'instrumenter ou de lire des informations du bytecode Java. Son poids (assez léger), ses performances ainsi que son modèle de conception lui ont valu d'être utilisé dans de nombreux autres projets Java, tel que le populaire *framework Spring.* Dans la section suivante nous présenterons les principaux modules de JaCoCo.

2.2.3.2 Modules principaux

JaCoCo comme présenté au niveau de la section 2.2.1.1 page 20 est organisé sous forme de module (*bundle*). Nous allons dans cette section présenter ces modules de bases telle que jacoco core, report, agent, ant.

org.jacoco.core

Il s'agit du module cœur de JaCoCo, il contient de nombreux sous modules (package) structuré par fonction. Il s'agit entre autre du module *org.jacoco.core.internal.instr* ; il permet de rechercher et placer respectivement les points d'insertion et d'insérer les sondes requises dans le bytecode par le biais d'ASM. Il est chargé de réaliser les instrumentations aussi bien statique que dynamique par l'usage de classes telles que :

- *org.jacoco.core.internal.instr.ProbeArrayStrategyFactory* pour la définition de la stratégie d'insertion du tableau de sonde JaCoCo ;
- *org.jacoco.core.internal.instr.ProbeInserter* pour l'insertion des sondes ;
- *org.jacoco.core.internal.instr.MethodInstrumenter* pour l'instrumentation dans les méthodes ;
- ...

Le package *org.jacoco.core.internal.flow* contient les classes permettant d'analyser le CFG du bytecode des méthodes fournies à travers des classes comme :

- *org.jacoco.core.internal.flow.Instruction* pour modéliser les instructions en mémoire ;
- *org.jacoco.core.internal.flow.ClassProbesAdapter* qui permet de calculer le nombre de sondes pour chaque type de méthode ;
- ...

Les packages *org.jacoco.core.data* et *org.jacoco.core.internal.data* qui sont chargés de la gestion du tableau de sondes après la phase d'instrumentation, à travers des classes comme :

- *org.jacoco.core.internal.CRC64* implémente l'algorithme de hachage dont le résultat tient sur 64 bits ;
- *org.jacoco.core.ExecutionDataStore* charger du chargement des fichier d'exécution après la phase de test.

Les packages *org.jacoco.core.analysis* et *org.jacoco.core.internal.analysis* qui sont chargés de l'analyse des différentes classes après la phase de tests dans l'optique de produire des structures nécessaires à la production d'état statistique.

org.jacoco.agent
Ce module est utilisé lors de la phase d'instrumentation dynamique ; il ne rentre pas dans le cadre de nos travaux.

org.jacoco.report
Ce module quand à lui à pour fonction principale d'analyser le fichier .exec produit afin de le rendre sous un format plus intuitif (format csv, html...) .

org.jacoco.ant
Ce module tient lieu de chef d'orchestre car définit l'ordre d'enchaînement des différentes étapes, par le biais du moteur de production java open-source impératif (ant). Ces taches sont les suivantes (pour ce qui est de l'instrumentation statique) :

1. import des librairies nécessaires dont (*org.jacoco.ant*) ;
2. compilation du programme cible avec les options de débogage ;
3. récupération des fichiers .exec et appel du module *org.jacoco.ant* pour les rendus.

2.2.4 Analyse du fonctionnement interne de JaCoCo

Dans l'optique de réaliser des statiques de couverture, JaCoCo prend en compte les niveaux d'instrumentation suivants :

- couverture niveau instruction (*C0 Coverage*) ;
- couverture niveau branche (*C1 Coverage*) ;
- *complexité cyclomatique*[3] ;

3. la complexité cyclomatique est un outil de métrologie logicielle développé par Thomas McCabe en 1976 pour mesurer la complexité d'un programme informatique.

- couverture niveau ligne ;
- couverture niveau méthode ;
- couverture niveau classe.

Afin de mieux percevoir les techniques mises en œuvre par JaCoCo lors de la phase d'instrumentation, nous allons préalablement présenter le CFG, pierre angulaire de la stratégie d'instrumentation JaCoCo. Nous utiliserons pour besoin d'illustration le petit programme Java (Code Source 5) nommé programme x1.

```
public static void example() {
    a();
    if (cond()) {
        b();
    } else {
        c();
    }
    d();
}
```

Code Source 5 – Programme d'illustration x1[6]

2.2.4.1 Graphe de Contrôle de Flux

Le bytecode cible d'une compilation peut être représenté sous forme d'un graphe ; le graphe de contrôle de flux. Ce graphe permet de mettre en évidence les différents schémas d'exécution possible d'un programme. Le Code Source 6 nous présente le bytecode du programme x1 (Code Source 5) obtenu par l'usage ASM.

```
public static void example() {
    a();
    if (cond()) {
        b();
    } else {
        c();
    }
    d();
}
```

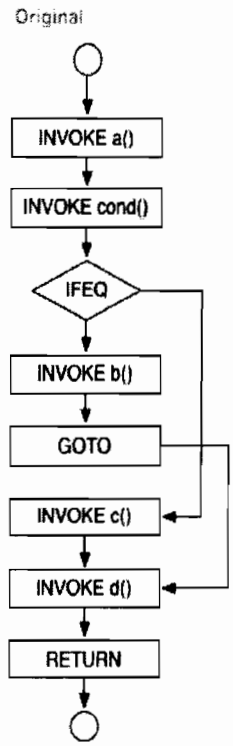
```
public static example()V
        INVOKESTATIC a()V
        INVOKESTATIC cond()Z
        IFEQ L1
        INVOKESTATIC b()V
        GOTO L2
   L1:  INVOKESTATIC c()V
   L2:  INVOKESTATIC d()V
        RETURN
```

Code Source 6 – Bytecode du programme x1[6]

La Figure 12 présente le CFG induit par le programme x1.

Selon la nature du nœud source et/ou de destination, les chemins « atomiques » peuvent être classés en 6 types, décrits au niveau du Tableau 6

La version actuelle de JaCoCo (version 0.7.6.201505271001 du 27 mai 2015) ne considère



Figures 12 – CGF du programme x1[6]

Type	Source	Cible	Observation
ENTRY	Aucune instruction	Première instruction de la méthode	
SEQUENCE	INSTRUCTION, except GOTO, xRETURN, THROW, TABLESWITCH et LOOKUPSWITCH	Instruction ultérieur	
JUMP	instruction GOTO, IFx, TABLESWITCH ou LOOKUPSWITCH	Instruction cible	TABLESWITCH et LOOKUPSWITCH définirons des arêtes multiples
EXHANDLER	toutes instructions dans la portée du manageur d'exception	Instruction Cible	
EXIT	xRETURN or THROW instruction	Aucune instruction	
EXEXIT	toutes instructions	Aucune instruction	Exception non encadré

Tableau 6 – Type des chemins du CFG

pour la couverture de code que les types suivants :

- SEQUENCE ;
- JUMP conditionnel (if else...) et inconditionnel (goto...) ;
- EXIT.

La gestion des exceptions (encadré ou non) des nœuds virtuels d'entrée ne sont pas couverts présentement par JaCoCo.

2.2.4.2 Politique d'instrumentation des sondes

Le principe utilisé par JaCoCo consiste à ajouter des sondes (probe) au niveau des arcs du CFG. Les observations suivantes ont été réalisées sur le CFG :

- si un arc est visité alors le nœud source de cet arc a été aussi visité ;
- si un nœud est exécuté et est cible d'un seul arc alors cet arc a été visité.

Ces observations ont permis d'aboutir à la stratégie suivante pour l'insertion des sondes dans le CFG :

- une sonde est placée sur toute sortie possible de méthodes (fin de méthode, instruction return) ;
- une sonde est placée sur chaque arc cible multiple d'un même nœud source (instruction swith case...) ;
- lorsque les informations de débogage sont disponibles (numéro de ligne), JaCoCo place une sonde au niveau de chaque ligne (du code source), afin d'avoir entre autre une précision maximum même quand une exception non encadré est levée.

L'ensemble de ces descriptions sont présentées et complétées au niveau du Tableau 7.

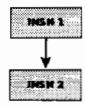
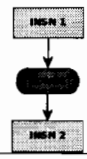
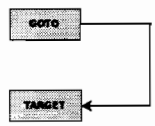
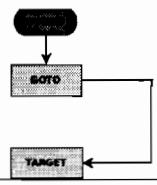
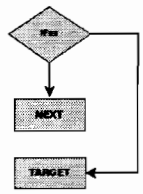
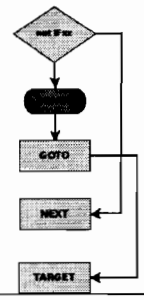
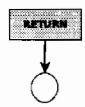
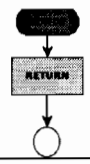
Type	CFG avant	CFG après	Observations
SEQUENCE			
JUMP INCONDITIONNEL			
JUMP CONDITIONNEL			Dans le cas d'un saut conditionnel, nous inversons la sémantique de l'opcode et ajoutons la sonde juste après le saut conditionnel. Avec une instruction GOTO subséquente nous allons à l'objectif initial (target).
EXIT			

Tableau 7 – Disposition des sondes dans le CFG

Dans le chapitre suivant nous aborderons la mise en œuvre concrète de la stratégie retenue.

Chapitre 3

Mise en œuvre de la solution : phase pré-chargement

La mise en œuvre de la stratégie retenue requiert la vérification d'un ensemble d'assertions énumérées comme suit :

- une applet est couverte que si au moins une classe étend la classe *javacard.framework.Applet* ;
- le nombre de sondes ne peut excéder 2^{15}-1 (32767) entrées ([13] paragraphe 2.2.4.3) par classe ;
- seules les classes non abstraites sont candidates à la couverture.

Le module chargé de la mise en œuvre de la stratégie dont la structure logique est présentée par la Figure 13, réalise de manière séquentielle les tâches suivantes :

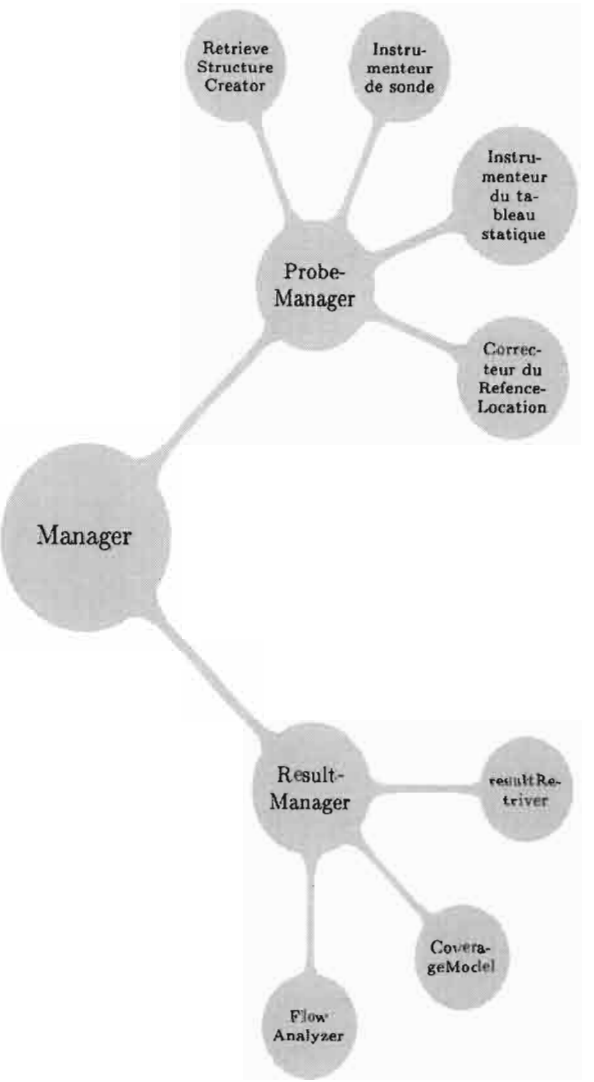
1. chargement du programme cible et dénombrement du nombre de sondes requises :
2. instrumentation de(s) structure(s) chargée(s) de la sauvegarde de l'état des sondes ;
3. instrumentation des sondes et code de récupération des résultats ;
4. construction des rendues.

Ce module porte le nom capCov pour converted applet Coverage. Au cours des sections suivantes nous utiliserons quelques formalismes du langage C [1] afin de représenter les structures binaires des applets, Il s'agit de :

1. la structure qui est un regroupement de variable de type homogène ou distinct identifié par un même nom de variable ;
2. l'union, bien qu'ayant les mêmes caractéristiques que la structure, diffère de celle-ci par le fait que sa taille en mémoire est celle de la taille de sa plus grande variable (en terme de type).

Nous aurons également recourt à une notation un peu particulière de la forme ux, ou x est un entier naturel permettant de désigner une suite consécutive d'octets non signés. Exemple *u2* pour 2 (deux) octets consécutifs.

1. le langage C est un langage de programmation impératif et généraliste. Inventé au début des années 1970 pour réécrire UNIX , le langage C est devenu un des langages les plus utilisés. De nombreux langages plus modernes comme C++ , Java et PHP reprennent des aspects de C (source wikipedia)



Figures 13 – Structuration logique du module de couverture (capCov)

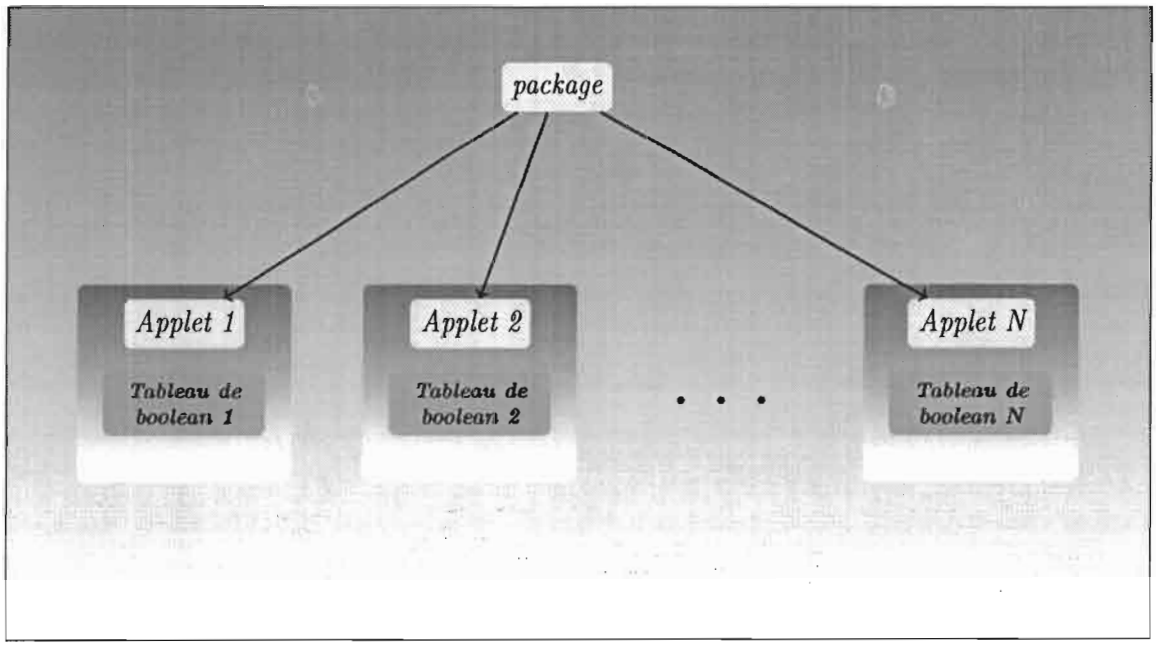
Le *probeManager* a la charge de la gestion de tout ce qui est tache instrumentation sur la Smart Card ; le *ResultManager* gère la partie récupération des résultats et réalisation des rendues.

3.1 Dénombrement du nombre de sondes requis

La mise en œuvre de cette étape nécessite le chargement en vue de la manipulation du programme compilé (fichier à extension .cap). Cette tache sera réalisée par le module CapMap mis à notre disposition.

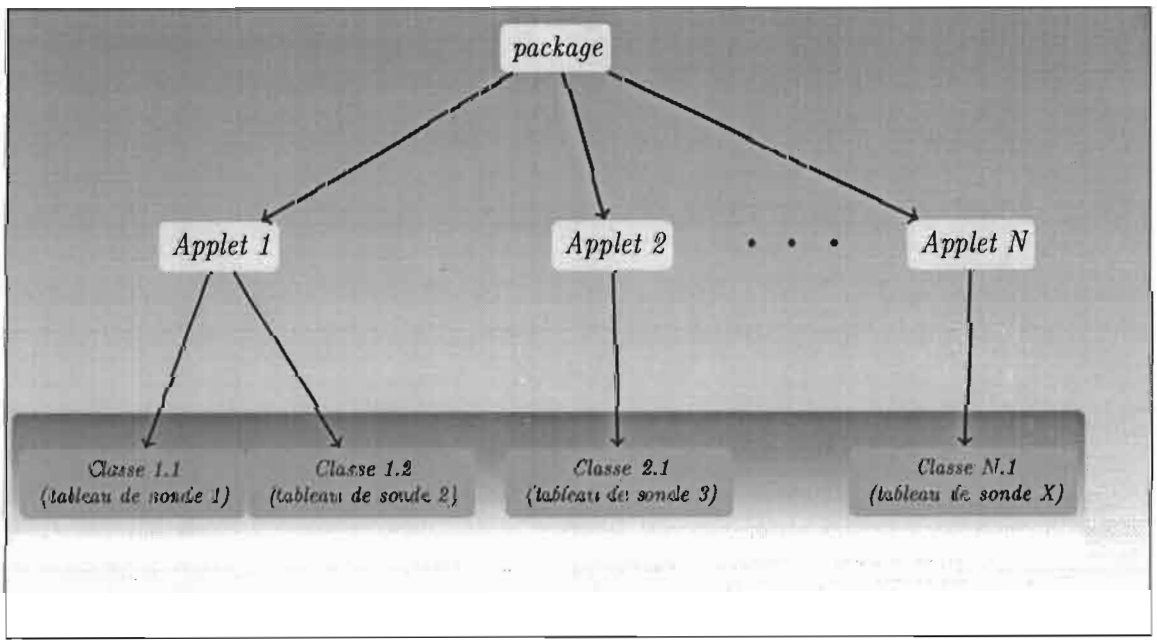
Un programme Java Card dans l'absolu, est un regroupement d'applets (package) c'est à dire pouvant être constitué d'un ensemble de classe Java standard étendant la classe *javacard.framework.Applet*. Au vue des nombreuses contraintes (taille de programme très réduit, Smart Card à fonctionnement réactif normé, isolation des applets,...) auxquelles nous sommes soumis, une approche optimale aurait constitué à utiliser une structure d'enregistrement d'exécution par applet du package comme présenté par la Figure 14. Cette approche aurait comme avantage de réduire le nombre de tableau instrumenté (avec un maximum de 256 tableaux, correspondant au nombre maximum d'applets possible dans un programme Java Card), réduisant du même coût, les instrumentations nécessaires en vue de la récupération des états des tableaux. La mise en œuvre de cette approche nécessiterait l'identification préalable des classes candidates à la couverture par applet, puis pour chaque méthode des classes identifiées, de

dénombrer le nombre de sondes requises. Lors de l'étude de la spécification Java Card[13] nous n'avons pas identifié de structure ou mécanisme permettant d'identifier de manière exhaustive et sûre toutes les classes par applet donnée, d'où l'abandon de cette approche.



Figures 14 – Approche de couverture par applet

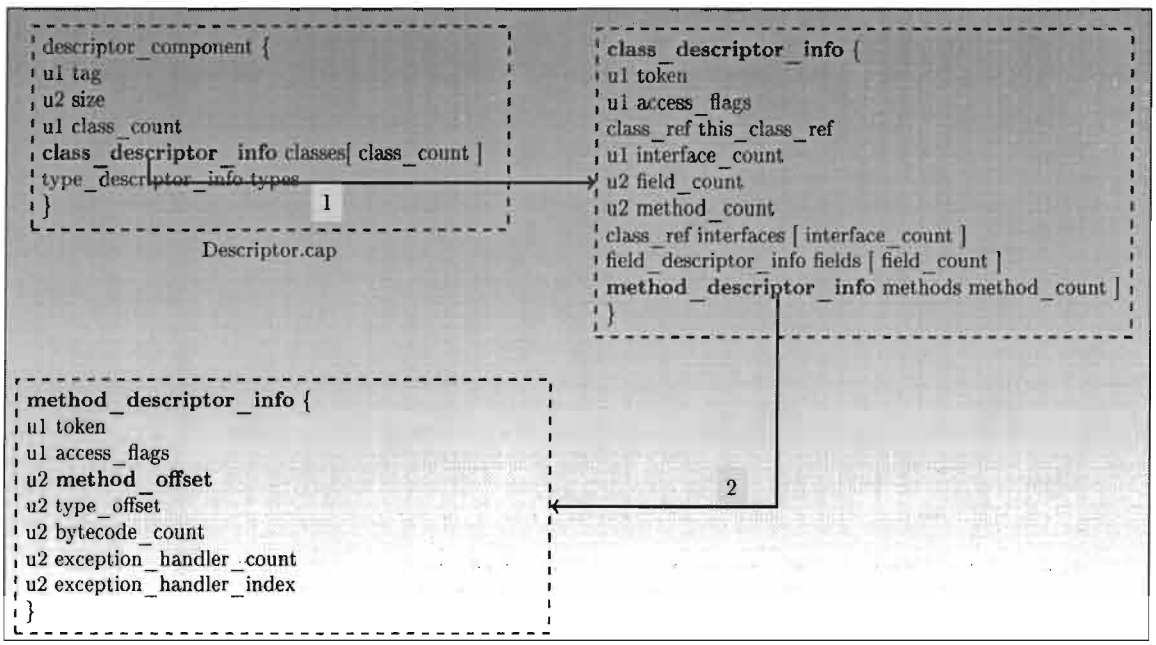
L'approche adoptée fut celle de JaCoCo dont le principe est d'instrumenter une structure d'enregistrement par classe (Figure 15). Cette approche à pour avantage de repousser les limites du nombre de sondes instrumentables de 2^{15}-1 à une limite fonction du nombre classe candidate (classe candidate * 2^{15}-1) pour toute l'applet. L'inconvénient de cette méthode provient du fait qu'il peut nécessiter plusieurs tableaux de sondes pour une seule application, induisant d'instrumenter plus de codes (pour chaque tableau un mécanisme de récupération est nécessaire), dans un environnement à ressources restreintes.



Figures 15 – Approche de couverture retenue

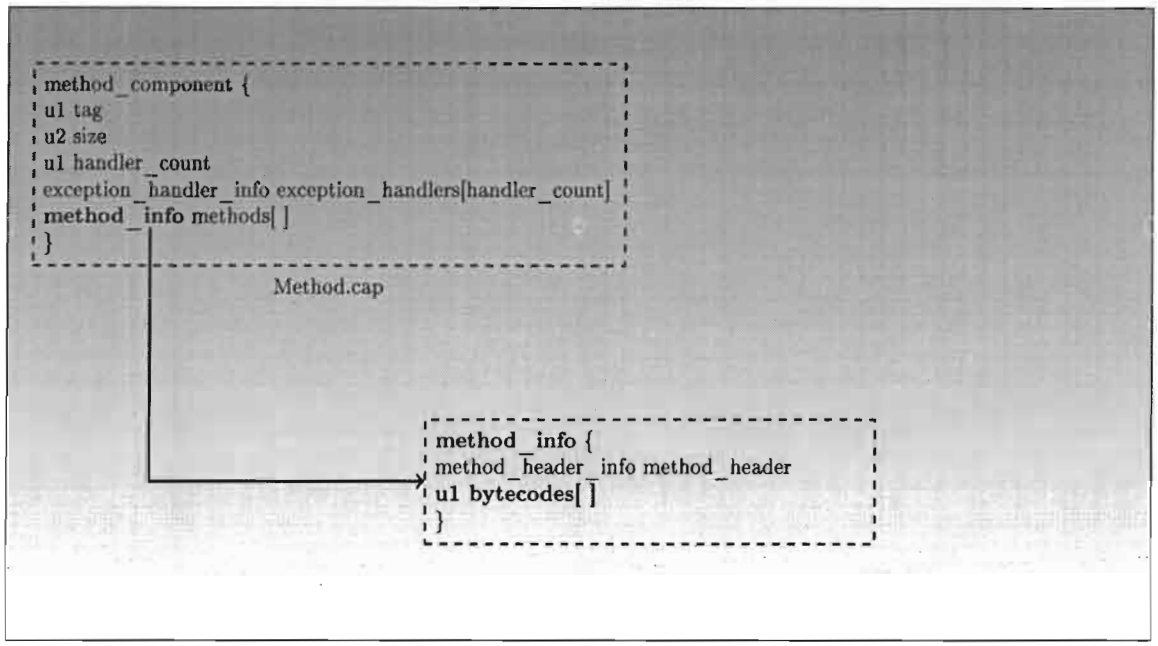
L'approche étant déterminée, nous allons pour chaque classe concrète (n'étant pas une interface ou une classe abstraite) identifier la liste de ces méthodes non abstraites en utilisant la

structure class_descriptor_info (information de description de classe) du composant descriptor dont la structure interne est présentée par la Figure 16. La structure class_descriptor_info contient la structure method_descriptor_info qui permet de décrire chaque méthode de la classe, en donnant des informations sur sa nature méthode abstraite ou non (champ flag), l'adresse dans le composant Method permettant de retrouver le bytecode de la méthode (champ method_offset),...



Figures 16 – Recherche des méthodes de classe

Nous allons ensuite, en s'aidant du champ method_offset retrouver le bytecode de chaque méthode stockée au sein du composant Method (champ bytecode[]), dont la structure est présentée par la figure Figure 17. Le bytecode sera parcouru et l'analysé en vue de dénombrer le nombre de sondes requis pour la méthode, puis pour l'ensemble de la classe. L'analyse a pour objet de repérer les instructions cibles conformément à la politique définie par JaCoCo (section 2.2.4.2 page 26) ; il s'agit des instructions des structures conditionnelles *if_xxx*, *goto_xxx*, *switch case* et de fin de méthode *xreturn* présentés par le Tableau 8.



Figures 17 – Structure du composant Method

Mnémonique de l'instruction	Valeur hexadécimal	Description
IFEQ	0x60	Ensemble des instructions machine utilisé pour la représentation du si (if).
IFNE	0x61	
IFLT	0x62	
IFGE	0x63	
IFGT	0x64	
IFLE	0x65	
IFNULL	0x66	
IFNONNULL	0x67	
IF_ACMPEQ	0x68	
IF_ACMPNE	0x69	
IF_SCMPEQ	0x6A	
IF_SCMPNE	0x6B	
IF_SCMPLT	0x6C	
IF_SCMPGE	0x6D	
IF_SCMPGT	0x6E	
IF_SCMPLE	0x6F	
IFEQ_W	0x98	
IFNE_W	0x99	
IFLT_W	0x9A	
IFGE_W	0x9B	
IFGT_W	0x9C	
IFLE_W	0x9D	
IFNULL_W	0x9E	
IFNONNULL_W	0x9F	
IF_ACMPEQ_W	0xA0	
IF_ACMPNE_W	0xA1	
IF_SCMPEQ_W	0xA2	
IF_SCMPNE_W	0xA3	
IF_SCMPLT_W	0xA4	
IF_SCMPGE_W	0xA5	
IF_SCMPGT_W	0xA6	
IF_SCMPLE_W	0xA7	
GOTO	0xA8	Ensemble des instructions machine utilisé pour la représentation du décalage (goto).
GOTO_W	0x70	
ARETURN	0x77	Ensemble des instructions machine utilisé pour la représentation du return.
SRETURN	0x78	
IRETURN	0x79	
RETURN	0x7A	
ILOOKUPSWITCH	0x76	Ensemble des instructions machine utilisées pour la représentation du switch case.
SLOOKUPSWITCH	0x75	
ITABLESWITCH	0x74	
STABLESWITCH	0x73	
ATHROW	0x93	Cas des exceptions

Tableau 8 – opcode cible pour l'insertion de sonde

La méthode Java chargée de réaliser le dénombrement des sondes par méthodes présentées au niveau du Code Source 7.

```
//Fonction charger de la recherche des emplacements
public short method_probe_count(){

        /*
         * On utilisera la liste obtenu par la methode getOpcodeMap(), car
         * la cle du map correspond au pc reel du code de la methode
         *
         */
        Map<Short, OpCode> liste_opcode_method = this.methodInfo.getOpcodeMap();

        // l'entree de methode etant instrumenter
        //this.methodProbeCount++;

        Iterator it = liste_opcode_method.entrySet().iterator();
        while (it.hasNext()) {

           Map.Entry pair = (Map.Entry)it.next();
           short pc=(short)pair.getKey();
           OpCode opCode=(OpCode)pair.getValue();

           //si il s'agit d'un IF, RETURN ou GOTO, SWITCH on incremente le compteur de sonde
           if((listeIfArgument.contains(opCode.getValue())) || (opCode.getValue() == IRETURN
             || opCode.getValue() == SRETURN || opCode.getValue() == RETURN
               || opCode.getValue() == ARETURN) || (GOTO == opCode.getValue()
                 || GOTO_W == opCode.getValue() || opCode.getValue() == SLOOKUPSWITCH
                   || opCode.getValue() == ILOOKUPSWITCH || opCode.getValue() == ITABLESWITCH
                     || opCode.getValue() == STABLESWITCH) || opCode.getValue() == ATHROW){

                        this.methodProbeCount=(short)(this.methodProbeCount+1);

            }

         }

         return this.methodProbeCount;

       }
```

Code Source 7 – Fonction chargée du dénombrement du nombre de sondes dans une methode

3.2 Instrumentation de la structure d'enregistrement

La structure de données utilisées par le module JaCoCo afin d'enregistrer les informations d'exécutions est un tableau de boolean à accessibilité privée [2],statique [3] ,synthétique [4] et transient [5] dont le code est présenté par le Code Source 8 (code obtenu en partie en désassemblant un programme instrumenté par JaCoCo à l'aide du module JD-GUI section 2.2.1.3 page 21 et par l'étude de la classe *FieldProbeArrayStrategy* du package *org.jacoco.core.internal.instr*).

2. propriété d'accessibilité empêchant une propriété ou méthode Java d'être accessible directement en dehors de la classe.

3. propriété d'accessibilité permettant à une propriété ou méthode Java d'être utilisée sans nécessité de créer un objet explicite de la classe qui la contient.

4. L'accesseur synthetic permet d'indiquer à la machine virtuelle qu'il s'agit d'une propriété/méthode générée lors de la phase de compilation.

5. L'accesseur transient indique que la propriété ne peut être persistée par un gestionnaire de persistance.

```
/*Exemple de declaration d'un tableau de sonde pour une classe*/
synthetic private transient final static boolean[] $jacocoData;

/*Exemple de declaration d'un tableau de sonde pour une interface*/
synthetic publique final static boolean[] $jacocoData_;


/* Fonction chargee de l'initialisation
   du tableau de sonde sous \gls{jacoco} */



synthetic private static boolean[] $jacocoInit(){

    if($jacocoData == null){

        static boolean[] $jacocoData = new boolean[taille];

    }

   return $jacocoData;
}
```

Code Source 8 – Fonction d'initialisation du tableau de sonde jaCoCo (généré par *org.jacoco.core*)

3.2.1 Stratégie

3.2.1.1 Stratégie tableau d'instance

Cette stratégie a pour objet d'instrumenter un tableau de boolean à accessibilité privée[6] dans la classe déclarant l'applet (classe étendant la classe *javacard.framework.Applet*). L'avantage de cette stratégie résidait dans le faite que le tableau n'était initialisé que si l'applet était utilisée, optimisant l'utilisation des ressources. Mais cette stratégie atteignait ses limites lorsque le programme à « couvrir » comporte :

- une méthode statique, qui par principe est indépendante de toutes entités non statiques, donc ne pouvant comporté à son sein des références (sonde) vers notre tableau d'instance de la classe ;
- des classes qui ne sont pas « quotedes points d'entrées » d'applet (classe étendant la classe *javacard.framework.Applet*), posant le problème de la récupération de l'état du tableau en dehors de la Smart Card, pour analyse.

3.2.1.2 Stratégie retenue, le tableau statique

Cette stratégie consiste à rester dans la logique du module JaCoCo, donc à instrumenter un tableau de boolean statique mais à accès publique n'ayant pas les propriétés synthétique ni transient. (L'accesseur *synthetic* bien qu'étant supporter lors de la phase de compilation (Figure 8) n'est pas implicitement représenté dans le programme obtenu). Nous allons dans la section suivante décrire concrètement la démarche d'instrumentation.

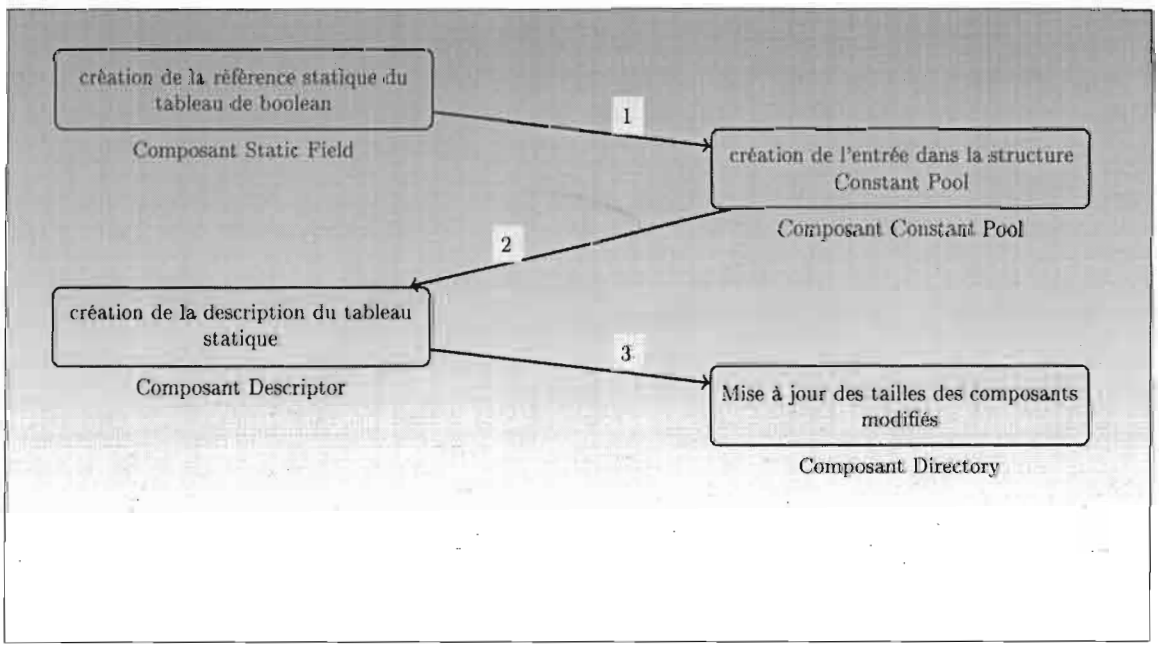
6. propriété d'accessibilité permettant à une propriété ou méthode Java de n'être pas accessible directement en dehors de la classe.

3.2.2 Processus d'instrumentation d'un tableau de sondes

L'instrumentation d'attribut nécessite la modification d'un certain nombre de composant de l'applet. La mise en œuvre de cette étape impliquera la mise à jour et/ou modification d'un certains nombre de composants dont le *Static Field*, *Constant Pool*, *Descriptor* et *Directory* (voir le Tableau 4 pour une présentation des différents composants). La démarche consistera aux étapes séquentielles suivantes :

1. création de la référence statique du tableau de boolean dans le composant Static Field ;
2. création d'une entrée dans la structure Constant Pool, afin de permettre au tableau crée dans le composant statique, d'être référencé dans le bytecode des méthodes sous forme d'index ;
3. ajout de la description du tableau statique (classe d'incrustation,...) au composant *Descriptor* ;
4. notification au composant *Directory* de la nouvelle taille des composants mis à jour.

L'ensemble de ces étapes est présenté par la Figure 18. Au cours des sous-sections suivantes, nous présenterons pour chaque étape la procédure employée.



Figures 18 – Étape de création d'une structure d'enregistrement

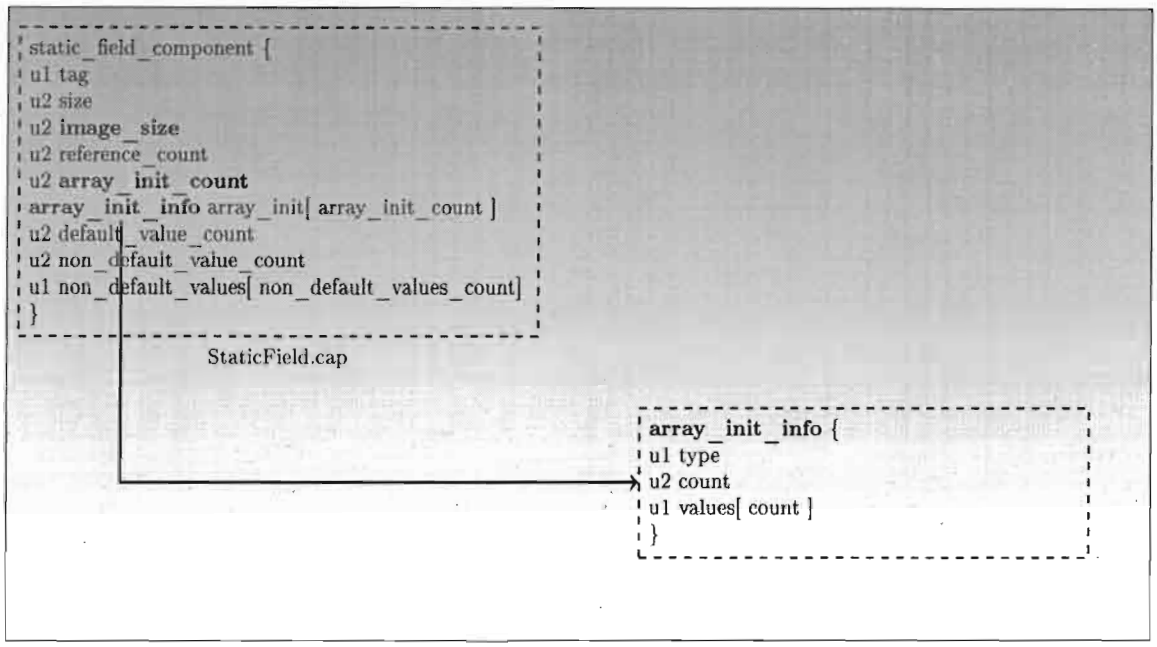
3.2.3 Ajout d'une référence au composant Static Field

le composant static Field en rappel, contient l'ensemble des informations permettant d'initialiser les champs statiques du programme. Les éléments de la structure de ce composant concerné par notre procédure seront :

- le champ *array_init_count* qui dénombre le nombre de tableaux statiques du package, que nous incrémenterons ;
- la structure *array_init_info*, qui permet de décrire chaque tableau statique du package (type et nombre d'éléments contenus), ou l'on ajoutera la description de notre tableau ;

- le champ *image_size* qui permet d'indiquer le nombre d'octets requis pour initialiser l'ensemble des champs statiques du package, et également de référencer un champ statique depuis autre composant. Nous le mettrons à jour pour la prise en compte de notre tableau crée (image_size = (reference_count * 2) + default_value_count + non_default_value_count).

la Figure 19 tout en présentant la structure du composant Static Field, permet de présenter la structure les champs énumérés ci-dessus.



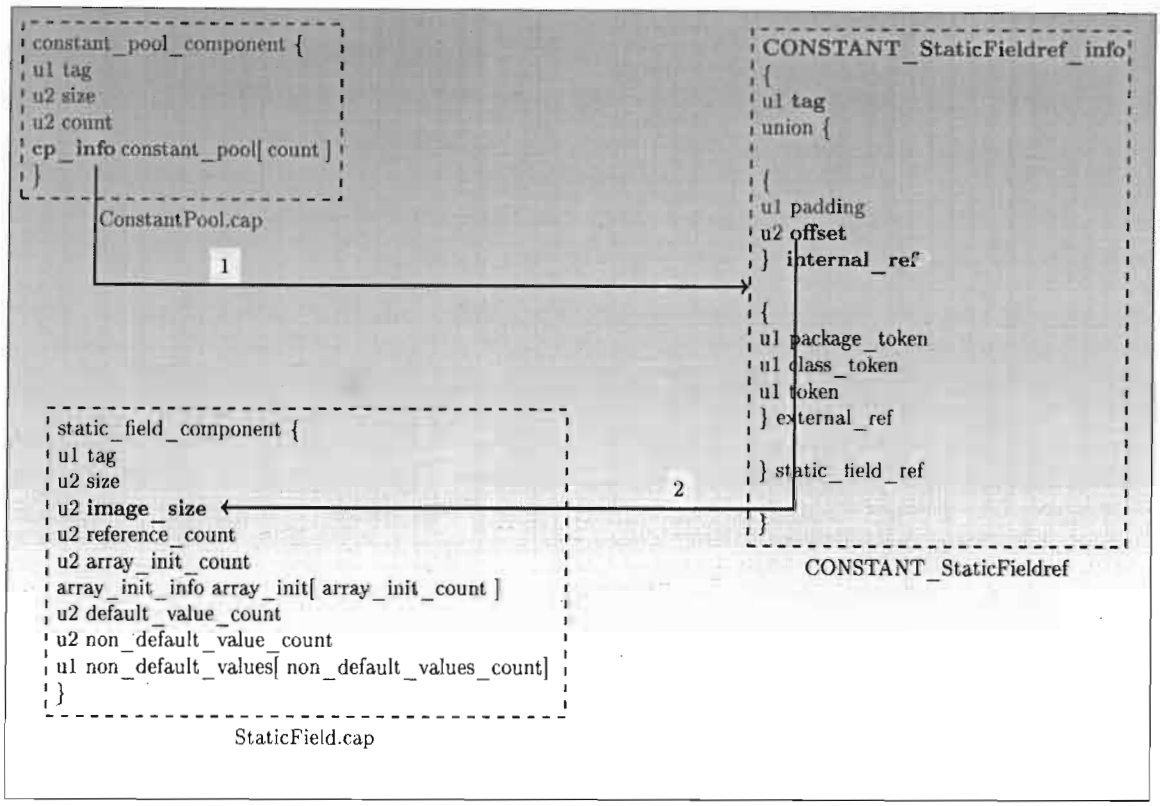
Figures 19 – Le composant Static Field

Notre tableau statique étant crée, nous allons procéder à la création de son entrée de référencement dans le pool de constante.

3.2.4 Création d'une entrée dans le composant constant pool

Le composant constant pool présenté par la Figure 20, peut être perçu comme principalement constituer d'un tableau de structure (champ *cp_info*) ou chaque structure a une taille de 3 (trois) octets. Ces structures permettent de décrire la nature des objets quelle représente (champ, méthode et classe statique ou d'instance) à travers la valeur de leurs champs *tag*, puis d'identifier l'élément représenté dans son composant de description par les 2 (deux) octets restants.

Au cour de cette étape nous allons créer puis ajouter une référence de notre tableau statique dans le pool de constante. Cette référence de type statique, sera représenté par une structure de type *CONSTANT_StaticFieldref_info*. Cette structure présentée dans la Figure 20, aura la valeur de son champ *tag* égal à 5 (cinq) (confère [13] TABLE 23 CAP file constant pool tags). Puis nous indiquerons l'adresse de notre tableau dans le composant Static Field (plus précisément son champ *image_size*) dans le champ *offset*, de l'union *internal_ref* .



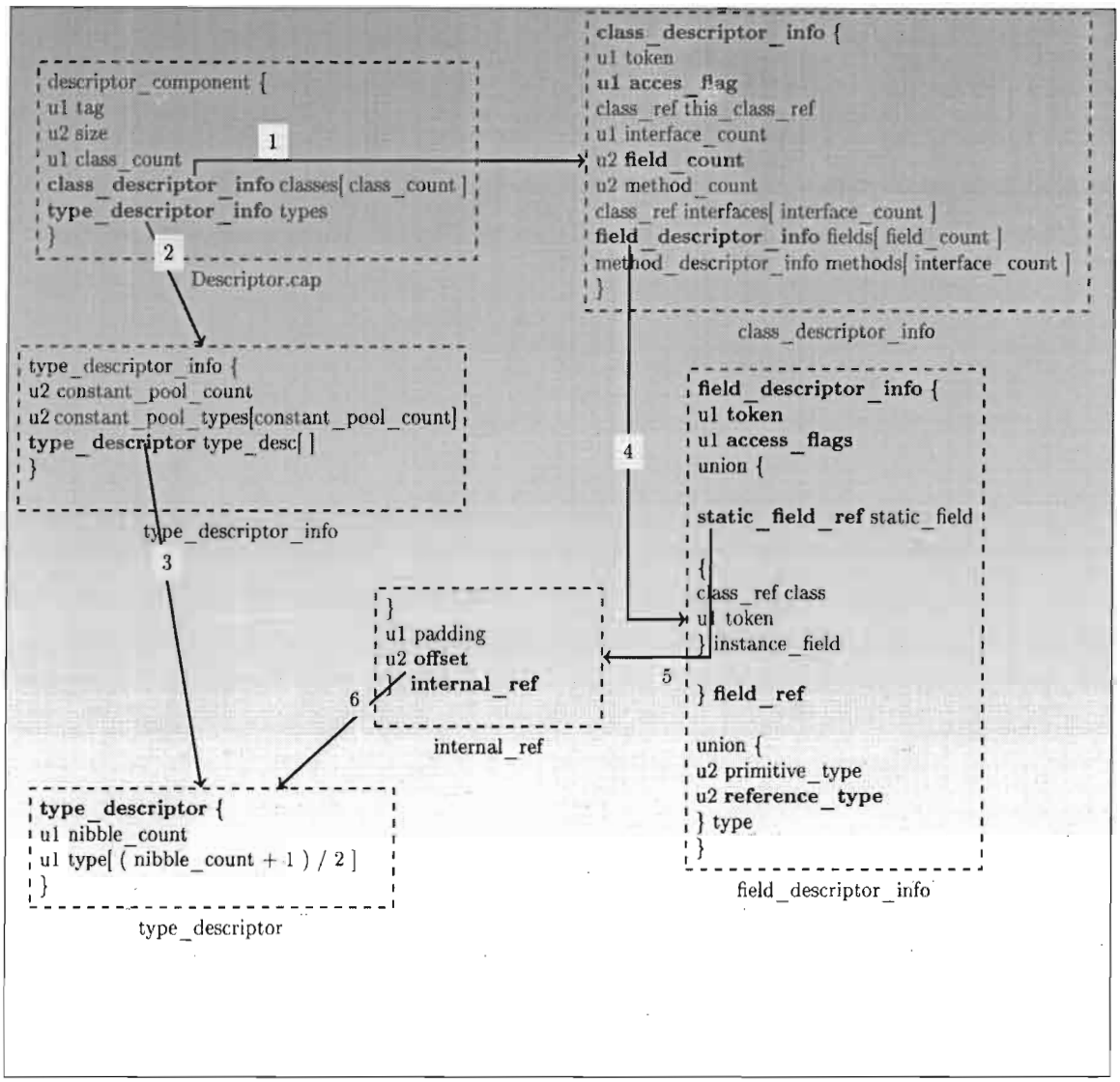
Figures 20 – Le composant Constant Pool (structure CONSTANT_StaticFieldref)

3.2.5 Description du tableau de sonde dans le composant descriptor

Le composant descriptor présenté au niveau de la Figure 21, fournit des méta-données sur d'autre composant permettant leurs descriptions pour certains et leurs complétions pour d'autres. Le but de notre processus à ce niveau, sera de rechercher et choisir la classe qui abritera notre tableau, ainsi que réaliser la mise à jour des autres composants impactés par cette instrumentation. Pour ce faire, nous allons procéder au parcourt de chacune des classes explicitement définies dans le package, à travers le parcourt des structures *class_descriptor_info*, qui permettent de décrire le contenu de chaque classe, à travers entre autres ces champs *field_descriptor_info* pour la description des variables de la classe, *method_descriptor_info* pour ce qui concerne les méthodes. Dans la structure *class_descriptor_info* de chaque classe non abstraite (que l'on vérifie à travers l'analyse du champ *flag* qui permet de déterminer si la classe est une interface, abstraite...) on incrémentera la valeur du champ *field_count*, qui nous renseigne sur le nombre de champs contenus dans la classe, puis nous procéderons à la création et l'ajout de la structure *field_descriptor_info* devant décrire notre tableau statique. La structure créée aura les valeurs caractéristiques suivantes (voir « [13] 6.13.2 field_descriptor_info » pour une description plus détaillée) :

- elle sera identifiable à travers la valeur du champ *token*, généré en conformité avec la description faite par la spécification.
- au niveau du champ *acces_flag*, comportera les propriétés public et static (valeur) ;
- au niveau de l'union *field_ref*, le choix sera porté sur la structure *static_field_ref* (référence du champ statique comme précédemment défini dans le constant pool) ;
- nous procéderons à la description du type de notre tableau statique que l'on ajoutera à la fin de la structure *type_descriptor_info*, ainsi que sa mise à jour.

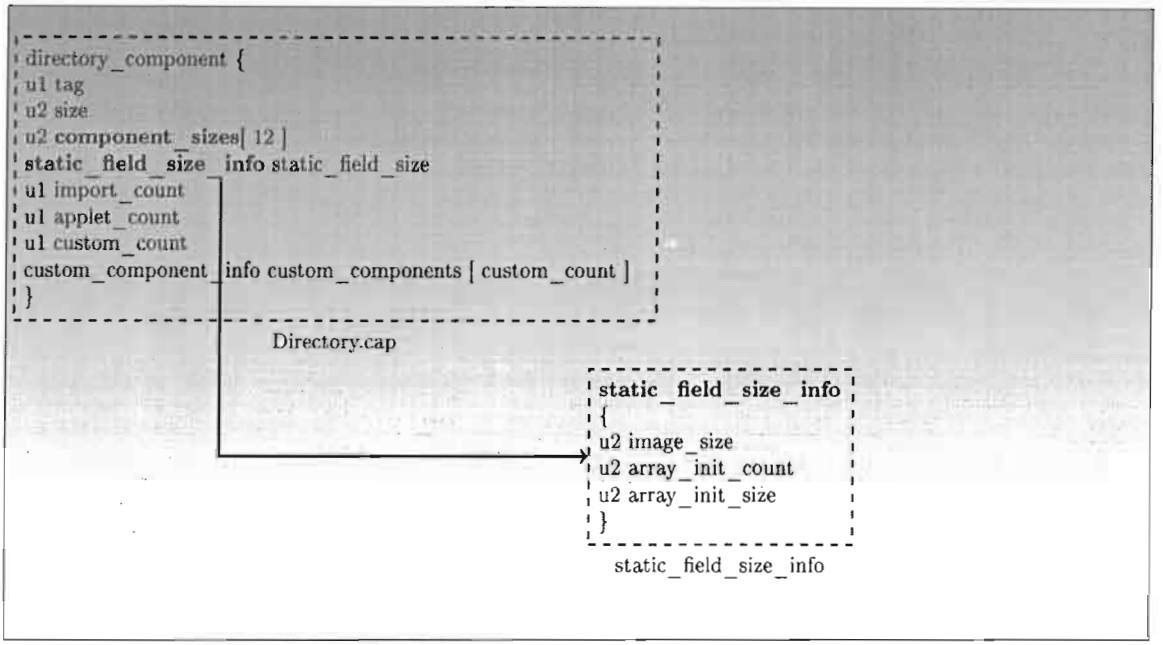
- au niveau de l'union *type*, on donnera l'adresse dans la structure *type_descriptor_info*, permettant d'identifier les éléments de description du tableau.



Figures 21 – le composant descriptor

3.2.6 Mise à jour du composant directory

Le composant directory permet de tenir un état des tailles de tous les composants présents dans le package. Cet état est tenu à l'aide du tableau *component_sizes*, contenant 12 (douze) entrées, ou chaque entrée correspond à un composant du package dont l'index est défini par la spécification ([13] TABLE 19 CAP file component tags). Nous mettrons également à jour la structure *static_field_size_info* permettant de tenir un état sur certains champs du composant static Field, dont la taille des variables statiques, (*image_size*) ainsi que le nombre et la taille des tableaux statiques du package. La Figure 22 présente cette structure.



Figures 22 – Le composant Directory

L'instrumentation du tableau de sondes ayant été réalisé, nous allons au cour de la section suivante passer à la présentation de la partie du processus concernant l'instrumentation des sondes dans le bytecode des méthodes.

3.3 Instrumentation des sondes et récupération des résultats

3.3.1 Instrumentation des sondes

La mise en place d'une sonde consiste préalablement à récupérer et à mettre sur la pile d'opérande (présentée au niveau de la section 1.3.1 page 10), la référence vers le tableau statique de boolean couvrant la classe, en utilisant l'instruction *GETSTATIC_A* suivi de l'index du tableau dans le pool de constante préalablement crée. Puis on charge sur la pile d'opérande, l'index du tableau statique correspondant au numéro de la sonde que l'on souhaite instrumenter, en se servant des instructions *SCONST_X, BSPUSH* ou *SSPUSH* selon la taille de l'index. On met ensuite à vrai (*true*) la valeur de l'entrée du tableau en chargeant le nombre 1 (un) sur la pile d'opérande grâce à l'instruction *SCONST_1*. Puis nous réalisons l'enregistrement de ces informations dans le tableau statique à l'aide de l'instruction *BASTORE*. La taille d'une sonde peut varier de 5 à 7 octets, pour une description du fonctionnement des instructions (*GETSTATIC_A, SCONST_X, BSPUSH, SSPUSH, SCONST_1, BASTORE*) confère l'Annexe B.

le Code Source 9 permet de présenter la méthode de capCov réalisant l'instrumentation de sonde.

```
private  int instrument_probe(int pc){

  MethodInfoEditable methodInfoEditable = new MethodInfoEditable(this.capFile);

  ReferenceLocationComponent referenceLocationComponent =
                      capFile.getReferenceLocationComponent();

  ArrayList<Integer> bytes =
                       referenceLocationComponent.getOffsetsToByteIndicesGlobal();


  //on recupere la reference dans le constantPool vers le tableau de sonde
  pc = methodInfoEditable.insertInstruction(pc, Instruction.GETSTATIC_A,
          new byte[]{(byte)(this.probeArrayIndexConstantPool  >> 8 & 0xFF),
               (byte)(this.probeArrayIndexConstantPool & 0xFF)});

  //on indique l'indice du tableau concerner par cette sonde
  //on verifi si l'indice dans le tableau ne depasse pas 5
  if(this.comptageNombreSondesMethodes <= 5){

        pc = methodInfoEditable.insertInstruction(pc,
                          this.lInstructions[this.comptageNombreSondesMethodes],
                              new byte[]{});

  //on verifi si l'indice dans le tableau depasse 5 et pas 255
  }else if(this.comptageNombreSondesMethodes > 5
           && this.comptageNombreSondesMethodes <= 255){

        pc = methodInfoEditable.insertInstruction(pc, Instruction.BSPUSH,
                         new byte[]{(byte)(this.comptageNombreSondesMethodes & 0xFF)});

  //si taille superieur a 255
  }else{

        pc = methodInfoEditable.insertInstruction(pc, Instruction.SSPUSH,
                    new byte[]{(byte)(this.comptageNombreSondesMethodes  >> 8 & 0xFF),
                          (byte)(this.comptageNombreSondesMethodes & 0xFF)});
  }


  // on met la valeur 0x01 (true) au niveau du tableau
  pc = methodInfoEditable.insertInstruction(pc, Instruction.SCONST_1, new byte[]{});

  //on procede a l'enregistrement dans le tableau avec l'instruction bastore
  //stack before ...,arrayref, index, value
  //Stack after ...,
  pc = methodInfoEditable.insertInstruction(pc, Instruction.BASTORE, new byte[]{});

  //methodInfo=capFile.getMethodComponent().getMethods().get(numeroMethod);
  this.comptageNombreSondesMethodes++;

  return pc;

}
```

Code Source 9 – Code responsable de l'instrumentation de sonde

La partie suivante présentera la stratégie de récupération des états des sondes après tests.

3.3.2 Récupération des états des tableaux statiques

Après la réalisation des tests, l'une des étapes importantes consiste à récupérer les valeurs des tableaux statiques en vue de réaliser un rendu convivial et rapidement exploitable (statistique de couverture). A cette étape plusieurs pistes furent explorées dont l'instrumentation d'un nouvel applet dont la méthode de traitement des APDU, la méthode *process* (confère Annexe B) serait intégralement consacrée au APDU de récupération des sondes. L'avantage de cette stratégie réside dans la conservation quasi assurée de la logique originale du programme initial. L'inconvénient de cette stratégie bien qu'idéal, provient de sa complexité de mise en œuvre

trop élevée ; en effet sa mise en œuvre nécessiterait la modification de la quasi totalité des composants avec un impact plus important sur les composants Method et class. Cette stratégie nécessitera donc d'instrumenter une quantité de codes assez considérable dans un système ou les ressources sont faibles.

La seconde stratégie a consisté à l'exploration de la possibilité qu'offre JCVM de définir de nouveaux composants personnalisés (dans le fichier à extension .cap) dont la définition est faite au niveau du composant directory par le champ *custom_component_info* (Figure 22). A cause de la faible documentation disponible (la spécification ne couvrant pas ce volet), cette exploration fût abandonnée.

La stratégie retenue consiste à instrumenter à la fin de la méthode *process* du premier applet du package étudié, un code qui permettrait de traiter une APDU particulière dont les informations permettent d'identifier le tableau statique cible et d'en retourner l'état.

La mise en œuvre de la stratégie suit les étapes suivantes :

- identification de l'adresse bytecode de la méthode *process* dans le composant Method ;
- identification des références dans le pool de constante des références aux méthodes de traitement des APDU ;
- instrumentation des structures de récupération de résultats.

3.3.2.1 Identification du bytecode de la méthode process

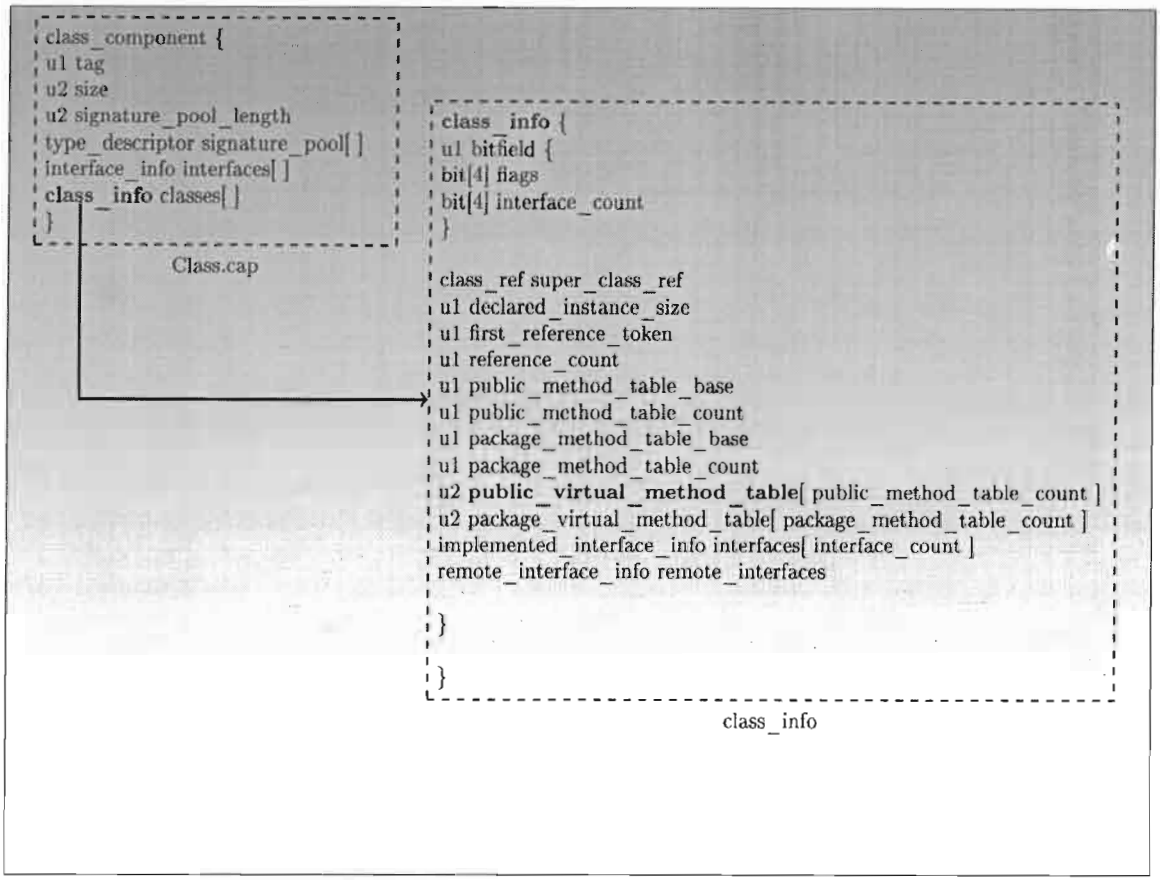
Une technique simple pour identifier l'adresse du bytecode de la méthode *process* est d'utiliser les données de débogage localisé dans le composant debug. L'inconvénient de cette méthode est de limiter le fonctionnement de notre module aux applets disposant des informations de débogage. Nous avons plutôt opté pour une méthode utilisant plutôt les métadonnées associées à chaque méthode, plus distinctement le champ *token* qui jout un rôle d'identifiant relatif à la catégorie de la méthode ; méthode statique, d'instance... Cela nécessite l'analyse de la méthode *process* en vue d'identifier son *token* de façon unique et sûre. La méthode *process* est en fait une méthode de la classe *javacard.framework.Applet*, que notre applet doit redéfinir lorsqu'elle implémente la classe *javacard.framework.Applet*. Selon le "[13] 4.3.7.6 Virtual Methods" qui stipule que si une méthode redéfinie une méthode de sa super classe, il est assigné le même numéro de *token* que la méthode de la super classe. L'analyse du fichier à extension .exp du package *javacard.framework* contenant la classe *Applet* par l'usage de la commande exp2text (confère "[14] CHAPTER 6 Viewing an Export File"), nous a permis d'identifier le *token* de la méthode comme présenté par le Code Source 10. Ce *token* à une valeur égale à 7 (sept).

```
class_info {            // javacard/framework/Applet
        token   3
        access_flags    public abstract
        name_index      191             // javacard/framework/Applet
        export_supers_count     1
        supers {
                constant_pool_index     5       // java/lang/Object
        }
        export_interfaces_count 0
        interfaces {
        }
        export_fields_count     0
        fields {
        }
        export_methods_count    10
        methods {
                method_info {
                        token   0
                        access_flags    protected
                        name_index      97              // <init>
```

```
                Descriptor_Index        1       // ()V
        }
        method_info {
                token   1
                access_flags    public static
                name_index      184             // install
                Descriptor_Index        115     // ([BSB)V
        }
        method_info {
                token   0
                access_flags    public
                name_index      100             // equals
                Descriptor_Index        101     // (Ljava/lang/Object;)Z
        }
        method_info {
                token   1
                access_flags    protected final
                name_index      185             // register
                Descriptor_Index        1       // ()V
        }
        method_info {
                token   2
                access_flags    protected final
                name_index      185             // register
                Descriptor_Index        115     // ([BSB)V
        }
        method_info {
                token   3
                access_flags    protected final
                name_index      186             // selectingApplet
                Descriptor_Index        91      // ()Z
        }
        method_info {
                token   4
                access_flags    public
                name_index      82              // deselect
                Descriptor_Index        1       // ()V
        }
        method_info {
                token   5
                access_flags    public
                name_index      187             // getShareableInterfaceObject
                Descriptor_Index  158 // (Ljavacard/framework/AID;B)
                                        Ljavacard/framework/Shareable;
        }
        method_info {
                token   6
                access_flags    public
                name_index      80              // select
                Descriptor_Index        91      // ()Z
        }
        method_info {
                token   7
                access_flags    public abstract
                name_index      188             // process
                Descriptor_Index        189     // (Ljavacard/framework/APDU;)V
        }
    }
}
```

Code Source 10 – description de la classe javacard.framework.Applet dans son fichier d'export (.exp)

Le *token* ayant été identifié, nous avons procédé à la localisation de l'adresse du bytecode de la méthode *process* dans le composant method. Pour accomplir cette tâche, nous avons d'abord procédé à l'identification des données de description dans le composant class, de la classe étendant la méthode *Applet*. Ceci est réalisé en s'aidant de l'adresse de la méthode d'installation de l'applet (méthode *install*, confère Annexe B) définie au niveau du composant Applet, associés aux métadonnées décrites dans le composant descriptor (confère Figure 21).



Figures 23 – Le composant Class

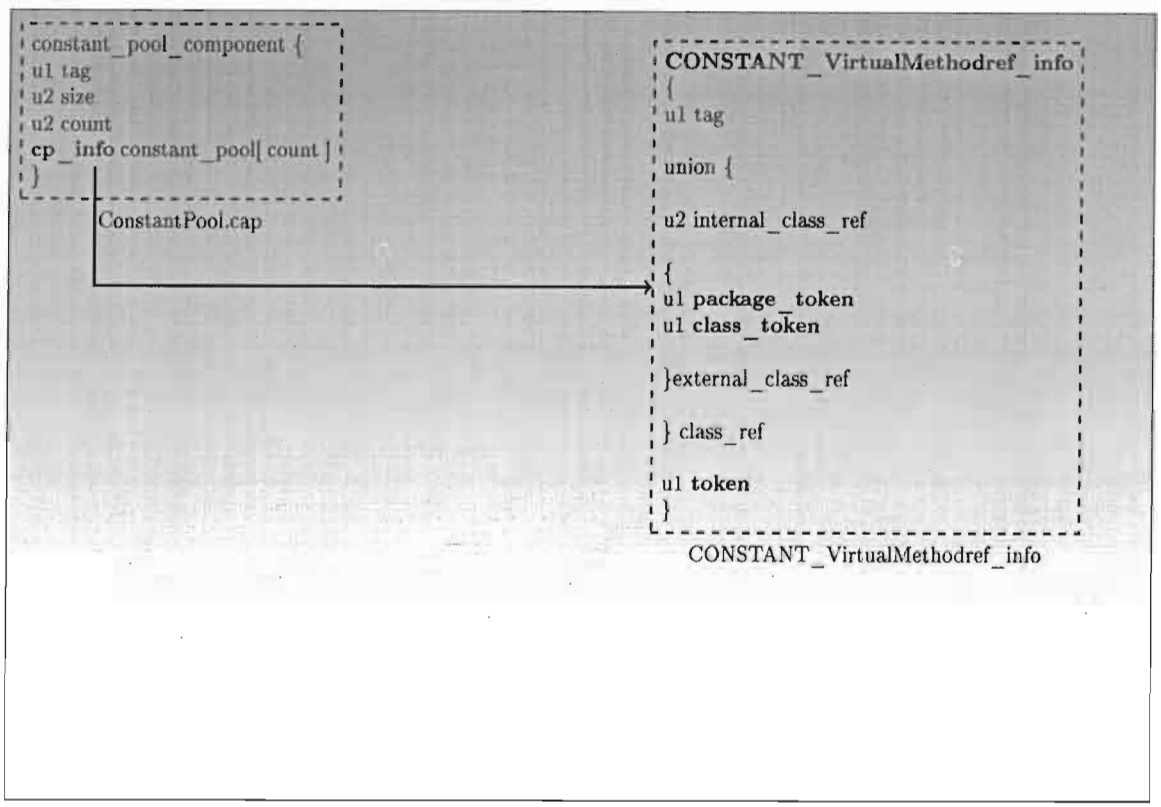
La prochaine partie sera consacrée à l'identification des références des méthodes dans le constant pool, permettant de traiter les APDU reçues.

3.3.2.2 Identification des méthodes dédiées au traitement d'APDU dans le pool de constante

Les méthodes concernées par cette recherche sont les méthodes de la classe *APDU* du package *javacard.framework* permettant :

- de récupérer le tableau d'octets jouant le rôle de vecteurs de données entre la carte et l'environnement externe, la méthode *getBuffer* ;
- de définir ou créer la fonction faisant référence au nombre d'octets devant être retournés, la méthode *setOutgoingAndSend.*

Pour identifier ces méthodes nous allons préalablement rechercher leurs *token*, le *token* de classe et de package importé (structure *external_class_ref*, contenant les champs *class_token* et *package_token*) en utilisant la même démarche que pour la méthode *process*. Puis à partir des informations trouvées, on identifie les références des méthodes dans le pool de constante, où l'on en crée si inexistante. La Figure 24 présente la structure du composant constant pool analysée, la structure *CONSTANT_VirtualMethodref_info*.



Figures 24 – Le composant Constant Pool (structure CONSTANT_StaticFieldref)

L'ensemble des rudiments étant réunis, la prochaine étape va présenter le principe de fonctionnement de la structure instrumentée dans la méthode *process* en vue de récupérer les résultats d'instrumentation.

3.3.2.3 Instrumentation de la structure de récupération de résultat

La structure de récupération va être constituée des éléments suivants :

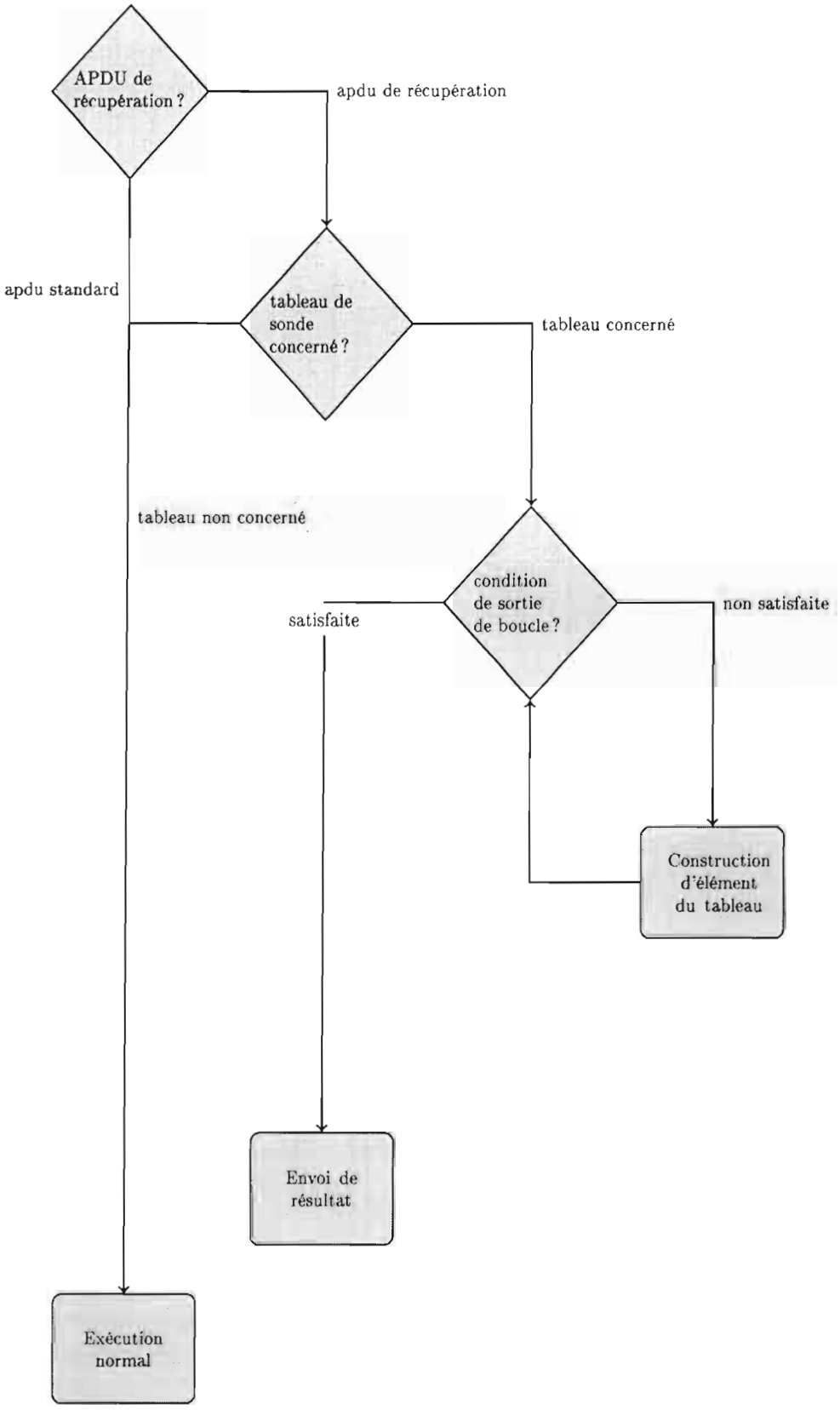
- une structure d'APDU de commande particulière en charge de la récupération des données ;
- une structure conditionnelle qui permettra de vérifier si l'APDU saisie est une APDU de récupération de résultat et identifier le tableau de sonde à retourner ;
- d'un ensemble d'instructions permettant de construire l'APDU de réponse contenant le tableau.

L'un des aspects délicats et très important est la mise au point de l'APDU de commande pour la récupération des données. L'APDU tout en étant valide doit être particulière, en ce sens qu'elle ne doit pas faire partie de l'ensemble des APDU potentiellement utilisable (classe (*CLASS*) propriétaire) ou utilisée (classe (*CLASS*) inter industrielle) confère Tableau 2. Nous avons opté pour une APDU dont les champs sont décrits par le Tableau 9.

Champ	Valeur	Description
CLASS	0x20	Il s'agit d'identifiant de classe inter industrielle réservé pour des usages futures (confère [10] 5.1.1 Class byte)
INS	0xFD	Il s'agit d'une instruction d'un APDU de classe inter industrielle non utilisé (confère [10] 5.1.2 Instruction byte)
P1	*token*	Il s'agit du jeton d'identification dans le fichier à extension .cap du tableau de sonde
P2 et *Lc*	0x00	Les autres champ *P2* et *Lc* sont mis à zero (0x00)

Tableau 9 – Description de l'APDU de récupération des données de tableau de sondes

L'APDU de récupération étant défini, la tâche suivante consistera à définir la structure de récupération dont l'algorigramme est présenté par la Figure 25.



Figures 25 – algorithme de récupération d'un tableau de sondes

Chapitre 4

Mise en œuvre de la solution : phase post-exécution des tests

L'objectif de ce chapitre est de présenté les étapes suivies pour construire des structures de données utilisables comme informations d'entrées pour le module JaCoCo.

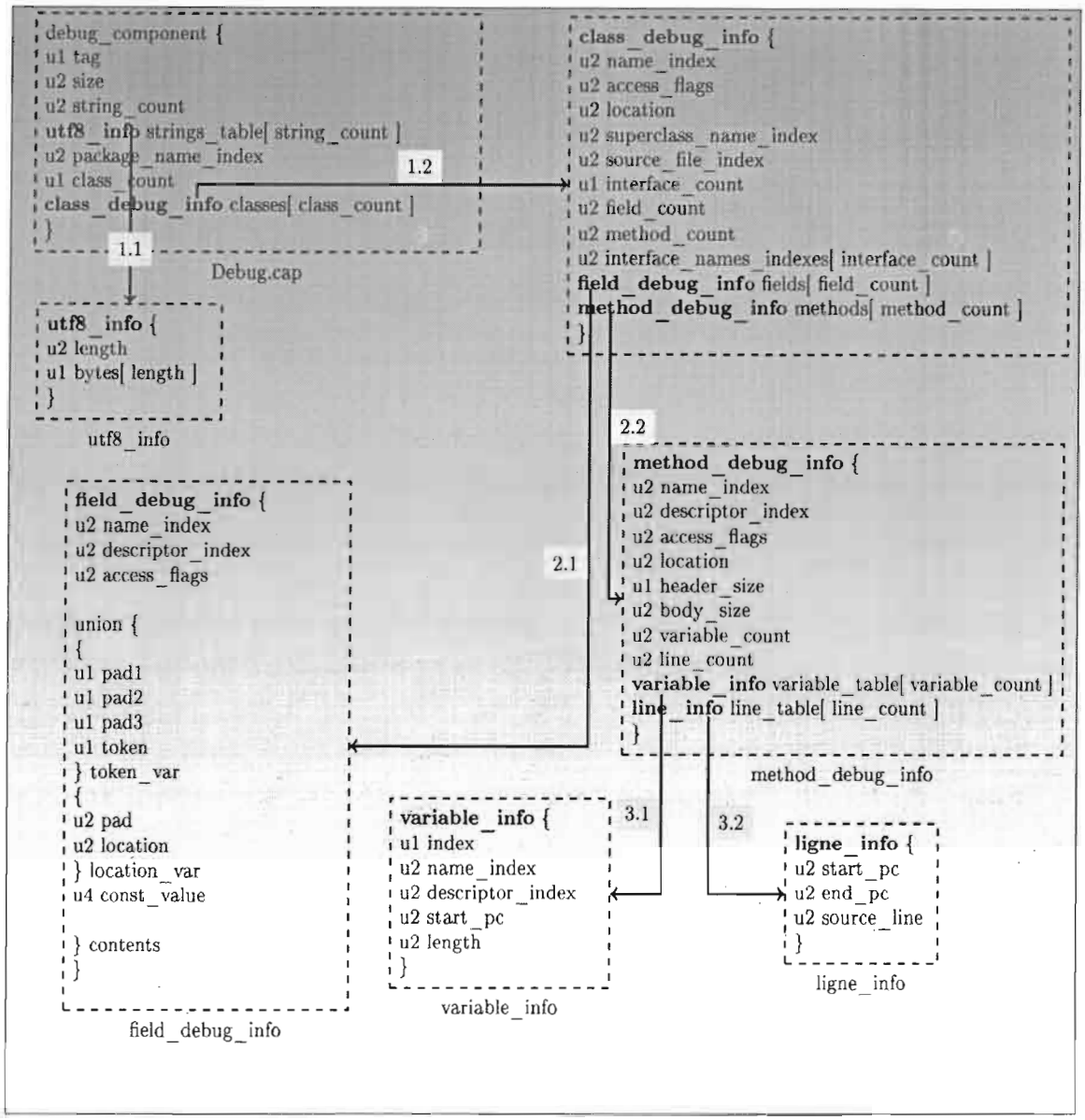
4.1 Construction des rendues

Dans cette étape nous allons à partir des états des tableaux de sondes récupérées, construire la structure fichier à extension .exec (présenté au niveau de la Figure 11), qui est un fichier d'entrée utilisé par le module *org.jacoco.report* pour la génération des rendues. Ce fichier dans la norme se construit sur la base de données de débogages.

4.1.1 Données de débogage

Le débogage consiste à l'ajout d'informations supplémentaires dans un programme lors de la phase de compilation. Ces informations sont des métadonnées qui permettent de faire le lien entre les objets (variables, méthodes, classes...) du fichier source et leurs équivalents générés après la phase de compilation. Au niveau des applets, c'est le composant Debug (Figure 26) qui contient les informations de débogage lorsque l'applet les contient. Ce mécanisme s'avère utile car il permet :

- d'identifier le nom (codé en utf8) des différents composants du fichier source (nom de classe, de méthode , de champ et de variable) contenu dans la structure *utf8_info* ;
- d'identifier les opcodes correspondants à une ligne, ou une variable du code source obtenue après la phase de compilation respectivement les structures *ligne_info* et *variable_info* ;
- d'apporter d'autres informations de descriptions complémentaires.



Figures 26 – Structure du composant Debug

4.1.2 Fabrication des fichiers de rendu

Le module CapMap ne chargeant pas ces données (car optionnel et n'étant pas charger sur la carte à puce), nous allons procéder à la construction d'abord du fichier à extension .exec, puis à la construction des classes chargées de retrouver les points d'insertion des sondes et de savoir en fonction du tableau de sonde, quelles sont les sondes qui ont été activées.

4.1.2.1 Construction du fichier .exec

Le fichier à extension .exec comme présenté au niveau de la section 2.2.3 page 22 permet de stocker l'état des tableaux de sondes après chaque session de test. Ces fichiers sont utiles dans ce sens qu'ils permettent de fusionner le résultats de plusieurs séries de tests (fusion des tableaux de sondes) pour des codes compilés identiques. Comme présenté au niveau de la Figure 11, ce fichier est composé d'un bloc d'entête, de sessions et plusieurs éventuels blocs de données. Certains de ces blocs subiront une modification dont le but sera l'ajout ou la modification d'informations. La Figure 27 présente la nouveau format de fichier d'exécution, dont les modifications et informations additionnelles sont marquées en fond rouge.

Ajout d'une nouvelle en tête (bloc d'entête capcov)

Ce nouveau bloc de données permet d'ajouter au bloc de données standard, la suite d'octets permettant d'identifier une applet de manière unique (AID) de la première applet (contenant la structure de récupération du tableau de sonde).

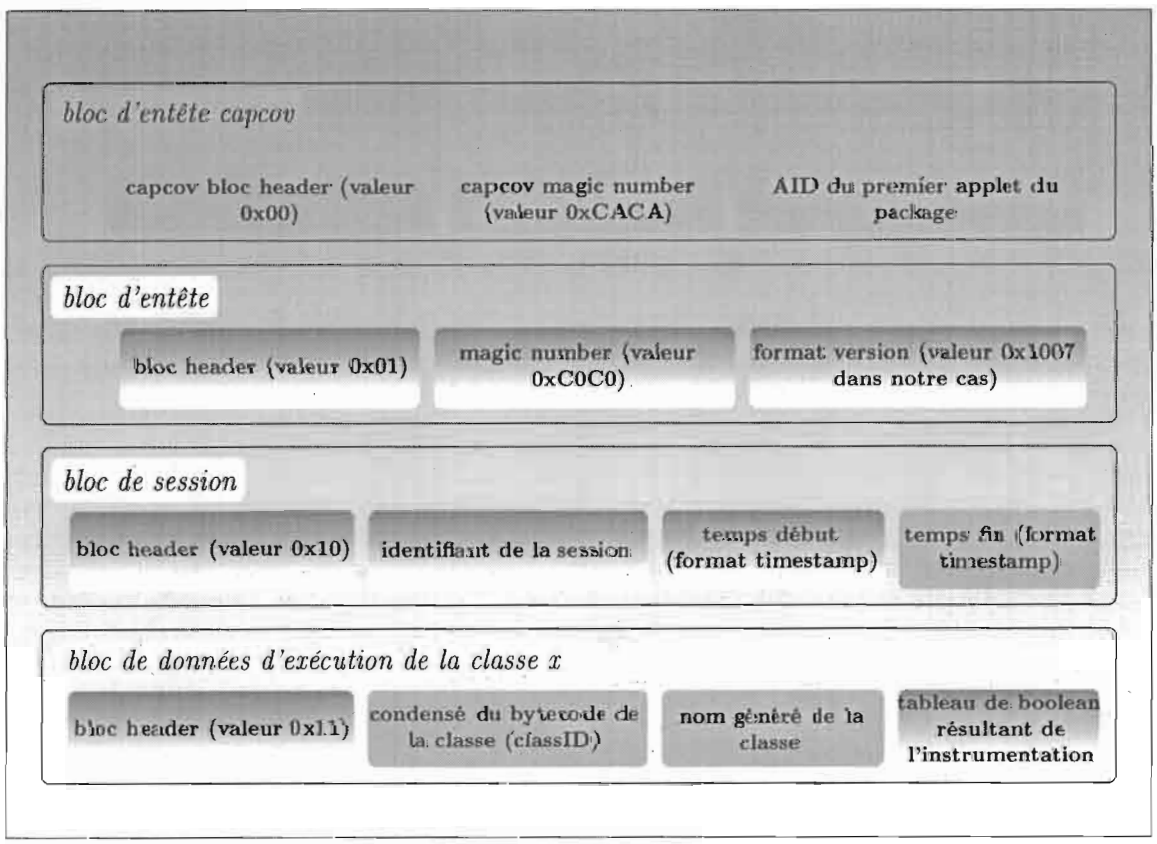
Modification du bloc de session

A ce niveau la difficulté résidait dans le fait que la date de fin de tests, n'est connu que lorsque l'on décide de récupérer les données d'exécution depuis la carte ; nous avons donc opté de lui donner la même valeur que la date de début de test plus une marge de temps fixe.

Modification du bloc de données d'exécution de la classe

Dans le fonctionnement standard de JaCoCo l'identifiant de la classe (classID) est obtenu en calculant le condensé du bytecode de la classe grâce à la fonction de hachage CRC64. Le but étant de s'assurer de l'identité de la classe (classe initiale instrumenté).

En Java standard un fichier à extension .class étant autonome (contenant le bytecode de toutes ces méthodes...), l'identifiant de la classe « reflète son contenu ». Dans le cadre de la création de nos fichiers .exec, nous avons opté pour le calcul de l'identifiant de la classe à partir du flux binaire séquentiel de la structure *class_descriptor_info* (confère Figure 21) du composant descriptor. Ce choix est motivé par le caractère représentatif de cette structure vis à vis des composantes de la classe. Le nom de la classe étant initialement composé du nom de la classe précédée du chemin absolu (au sens de la machine virtuelle) vers son package, nous avons dû procéder à sa modification en composant le nom de la classe avec son jeton associé dans le composant descriptor comme suit « Class_jeton ».



Figures 27 – Structure d'un fichier d'exécution (.exec) capcov

4.1.2.2 Construction du modèle de couverture *coverage model*

Dans le but de mieux percevoir notre démarche, nous allons débuter par une présentation de la stratégie mise en œuvre par JaCoCo, pour la génération des résultats.

Le but est de retrouver pour chaque entrée du tableau de sondes d'une classe donnée, les instructions dont elle gère la couverture, en vue de générer les états. Pour ce faire, JaCoCo va procéder après la phase d'instrumentation et de test, au chargement des classes initiales (non instrumentées) afin de les analyser pour retrouver les lieux d'insertion des sondes. JaCoCo va ensuite procéder à un appel récursif de visiteur afin de retrouver les points d'insertions de sondes, puis pour chaque point trouvé, de vérifier son état de couverture (la valeur de son entrée dans le tableau de boolean). Concrètement le module va procéder à la lecture puis à l'interprétation du bytecode des fichiers de classe à partir de la classe *ClassReader* Annexe D ; puis pour chaque section identifiée (label, instruction conditionnelle)..., il va procéder à l'appel des méthodes équivalentes du visiteur *ClassProbeAdaptor* dont le rôle est d'identifier les lieux d'insertion de sondes. Le visiteur *ClassProbeAdaptor* prend également en argument un autre visiteur de type *ClassProbeVisitor*, dont le rôle est de réaliser les traitements nécessaires lors de l'identification de sondes (ces traitements peuvent être l'instrumentation de sonde *ClassInstrumenter*...). En ce qui nous concerne, le visiteur sera *ClassAnalyzer* qui se chargera pour chaque sonde identifiée dans une méthode, de vérifier son état de couverture (à partir de son tableau de sondes provenant des tests qu'il prend en paramètre) par le biais de sa classe d'analyse de méthode *MethodAnalyzer*. C'est la classe *MethodAnalyzer* qui pour chaque instruction, détermine le numéro de ligne associé (obtenu à partir des informations de débogage) dans le code source, les instructions qui la précèdent (prédécesseurs) , les différents chemins (*branch*) possibles (informations qui seront stockées dans la classe *org.jacoco.core.internal.flow.Instruction*). Le Code Source 11 présente la fonction invoquée à la fin de la visite de chaque méthode par *MethodAnalyzer*.

```
/* Fonction qui permettra de construire mon noeud methode */
@Override
public void visitEnd() {
        // Wire jumps:
        for (final Jump j : jumps) {
                LabelInfo.getInstruction(j.target).setPredecessor(j.source);
        }

        // on met toutes les instructions anterieurs au instructions
        // coveredProbes a couverte
        // Propagate probe values:
        for (final Instruction p : coveredProbes) {
                p.setCovered();
        }
        // Report result:
        coverage.ensureCapacity(firstLine, lastLine);
        for (final Instruction i : instructions) {
                final int total = i.getBranches();
                final int covered = i.getCoveredBranches();
                final ICounter instrCounter = covered == 0 ? CounterImpl.COUNTER_1_0
                                : CounterImpl.COUNTER_0_1;
                final ICounter branchCounter = total > 1
                                ? CounterImpl.getInstance(total - covered, covered)
                                : CounterImpl.COUNTER_0_0;
                coverage.increment(instrCounter, branchCounter, i.getLine());
        }
        coverage.incrementMethodCounter();
}
```

Code Source 11 – Couverture des instruction antérieurs (MethodAnanlyzer)

Notre travail à consister à linéariser les appels « semi-récursifs » des adaptateurs (dont cer-

tains ont été listés ci-dessus) imposés par le modèle Visitor ; afin de remplacer l'analyseur d'états des sondes (*MethodAnalyzer*) très couplé à ASM pour construire la structure hiérarchique de données qui fera l'association entre tableau de sonde et lieu d'insertion dans le CFG.

4.2 Quelques Résultats

L'ensemble des travaux que nous avons réalisé, nous a permis d'obtenir un module fonctionnant en ligne de commande. Le module permet à travers ses options :

- d'instrumenter une applet qui lui est fourni en arguments ;
- de récupérer les résultats de tests après la phase d'instrumentation pour stockage dans une structure JaCoCo encapsulé (confère Figure 27) ;
- de produire à partir du fichier d'exécution et l'applet d'origine le résultat des tests en mode ligne de commande.

Après la phase d'instrumentation, les résultats de la phase de tests permettent d'obtenir des informations sur les mesures suivantes :

- couverture niveau instruction (*C0 Coverage*) ;
- couverture niveau branche (*C1 Coverage*) ;
- calcul de la complexité cyclique *cyclomatic Complexity* ;
- couverture niveau méthode ;
- couverture niveau classe.

La couverture niveau ligne n'est pas prise en compte, du fait de l'absence des informations de débogage. Le Code Source 12 nous donne un aperçu des données produites après l'aide d'une commande d'analyse.

```
// execution de la commande permettant la production de resultats
./capcov.sh -p projet.cap jacoco_capcov.exec

****************** Result of the CapCov coverage task ******************

Coverage of class Class_0
16 of 84 instructions missed
17 of 30 branches missed
0 of 0 lines missed
0 of 5 methods missed
16 of 25 complexity missed

****************** End of Result of the operation ******************
```

Code Source 12 – production de résultats à l'aide de capcov

4.3 Bilan et perspectives

4.3.1 Bilan

L'importance de la couverture de code aussi bien dans l'étude (nous avons eu recours à la couverture pour faciliter l'étude du code source de JaCoCo) que l'élaboration de logiciel n'est plus à démontrer.

Au cours de nos travaux nous avons rencontré certaines difficultés telles que la panne de notre carte à puce. Pour réaliser nos tests nous avons eu recours au vérificateur de bytecode Java Card en mode *off-card* (hors de la carte), l'outil verifycap présenté au niveau de [14] chapitre 7 vérification des fichiers CAP et Export. Ce module de la Java Card Development Kit (JCDK) nous a été très utile, car il permet de vérifier la consistance et la cohérence aussi bien sur le plan structurel, syntaxique que sémantique du bytecode qui lui est soumis. Au niveau du composant Method, verifycap permet de simuler l'exécution des différentes méthodes en utilisant une analyse de flux de données ou *Data flow analyses* (comme présenté au niveau de la sous-section 8.1.1 du [2] de titre *Data flow analyses*). Le principe est de calculer l'état de la frame (pile d'opérande et du tableau de variable local) lors de l'exécution de chaque instruction afin d'analyser les états obtenus, pour en vérifier la correction. Nous avons également exploré la piste de la simulation afin de simuler une carte physique (avec connexion, envoie d'APDU), ce qui nous à conduit à étudier des simulateurs tels que *JCardSim* (Java Card Runtime Environment Simulator), *vJCRE* (virtual JCRE) et *vsmartcard* (virtual Smart Card), que nous avons essayé mais qui ne répondaient de façon exhaustive a nos besoins.

Nous avons également dû mettre en œuvre certaines procédures correctives (en vue de compléter le module CapMap), afin de corriger les décalages que causait l'instrumentation continue des sondes au niveau des composants method et reference Locator. Cette tache nous à amené à étudier les sources du vérificateur *verifycap* en plus du module CapMap, à fin d'élaborer les fonctions correctives. Au niveau du composant method nous avons procédé après chaque phase d'instrumentation à la correction de la représentation mémoire des *offset* des instructions. Le cas où on instrumentait un code dans une section concernée par un saut (réaliser par un opcode goto, if, switch) en est un exemple.

```
ifeq // rel:+3

return

aload_2
//
```

```
ifeq // rel:+9
getfield_a_this //reference
sconst_0
sconst_1
bastore
return
aload_2
```

Code Source 13 – Anomalie d'instrumentation

Au niveau du composant referenceLocator nous procédons à la fin de chaque phase d'instrumentation à la reconstruction du composant (contenant les *offset* d'instruction faisant référence à un élément du pool de constante). Pour ce faire nous avons étudié et réutilisé le mécanisme de construction du referenceLocator réalisé par le vérificateur *verifycap*.

Ces travaux nous ont également permis à travers l'étude des spécifications, d'en mesurer l'importance et les limites dans un domaine où l'on veut l'interopérabilité des technologies.

L'étude assez laborieuse des codes sources des librairies intégrées (JaCoCo, ASM, CapMap,..), nous a permis à travers les styles de programmation nouveaux et système d'optimisa-

tion des programmes, d'étendre nos champs de connaissances et de nous porté vers de nouvelles perspectives.

4.3.2 Perspectives

Ce projet ouvre la voie à de nombreuses perspectives en ce qui concerne les cartes à puces de notre point de vue. Il s'agit entre autres :

- la complétion du module CapMap afin de permettre d'interpréter les données de débogage, ce qui permettra d'améliorer les rendues (graphiques) ;
- l'optimisation des méthodes de calcul des nouvelles tailles de piles d'opérande et tableaux de variable, lors de l'ajout d'instructions (en lieu et place d'une augmentation systématique) ;
- l'extension du module capCov en vue de faire du profilage [1] en plus de la couverture de code, en vue de permettre au développeur de mieux optimiser leurs programmes ;
- le développement d'une interface graphique en vue de faciliter l'utilisation du module, ou même le convertir en plugin pour un usage direct au sein d'environnement de développement ;
- le couplage du module à un simulateur autonome, afin de le rendre autonome, et d'éviter ainsi la nécessité de posséder une carte à puce physique.

L'ensemble de ces extensions serait la bienvenue au vue de la vulgarisation de l'usage des Smart Card de plus en plus performante, permettant de ce fait de réaliser des applets de plus en plus complexes.

1. Le profilage fonctionne sur le même principe que l'instrumentation, à la différence que la structure d'enregistrement, enregistre le nombre de fois ou chaque sonde est exécutée

Conclusion

La mise en œuvre de tests (unitaire, d'intégration, etc) lors du cycle de développement d'une application est un gage d'une certaine qualité d'application. S'assurer de l'exhaustivité de ces derniers pour certains types de système tels que les systèmes bancaires, temps réel revêt d'un caractère primordial. Il est donc nécessaire de combiner des outils de couverture de code aux outils de tests, afin de mesurer la couverture des fonctionnalités des applications lors des tests.

L'absence d'outils de couverture de code grand publique adapté au développement d'applet java card a motivé notre étude. Nos travaux ont donc consisté à adapter un outil de couverture de code existant, afin de permettre son utilisation lors des tests d'applet java card.

Dans un premier temps, nous avons procédé à une étude des techniques de couverture de code, ce qui nous à permis d'élaboration un scénario dans lequel nous adaptons un outils de couverture de code standard à un usage sur les cartes à puces. Nous avons ensuite, à partir des outils mis à notre disposition (bibliothèques, lecteur de carte à puce, carte à puce) procédé à une implémentation du scénario retenue.

Pour vérifier la correction de nos algorithmes a défaut d'avoir une carte à puce fonctionnelle, nous avons procédé a la vérification de nos applications instrumentées à l'aide de la librairie capverifier (offerte par SUN), qui nous a permis de déboguer nos fichiers .cap instrumentés enfin d'en vérifier la consistance et la cohérence.

Les travaux réalisés nous ont permis d'étudier de nouveaux modèle de conception telle que le modèle de conception visitor utilisé par JaCoCo et ASM ; d'étudier le fonctionnement des émulateurs d'exécutions (que nous avons dû réaliser pour identifier les sources d'erreur). La réalisation de ce projet nous à également emmené à réaliser une immersion dans l'univers de la recherche, que nous avons côtoyé par la lecture de nombreux articles, thèses (doctorat), et posts qui nous ont permis d'éclaircir notre point de vue sur le sujet.

Les cartes à puces de par leurs utilisations présentes et futures que laissent présager les spécifications (telle que *7816-7 : Interindustry commands for Structured Card Query Language (SCQL)*), la conception de machine incorporant des ports pour carte à puces témoignent de la vulgarisation et de la complexité possible des applications. La couverture de code comme techniques du génie logiciel apporté dans ce secteur, permettra d'améliorer la qualité des applications produites dans un monde de plus en plus porté vers les objets connectés.

Table des figures

Liste des tableaux

Liste des Codes Sources

Bibliographie

[1] Atlassian, Couverture du code java et groovy.

[2] Consortium, O. (2011) *ASM 4.0 A Java bytecode engineering library.*

[3] Corporation, O. (2013) *The Java Virtual Machine Specification Java SE 7 Edition.*

[4] de Jong, E., Hartel, P. P., Peyret, P., and Cattaneo, P., Java card : an analysis of the most successful smart card operating system to date.

[5] Dufay, G. (2003) *Vérification formelle de la plate-forme Java Card.* Ph.D. thesis, UNIVERSITÉ DE NICE.

[6] EclEmma, Jacoco java code coverage library.

[7] Gantaume, B. (2016), Définition de la couverture de code illustrée avec junit.

[8] Guo, H., Smart cards and their operating systems.

[9] Iguchi-Cartigny, J. (2014) *Contributions à la sécurité des Java Card.* Ph.D. thesis, Université de Limoges.

[10] ISO/IEC *7816-4 (Part 4 : Organization, security and commands for interchange).*

[11] Parzian, A. (2015) *Java Card Bytecode Verification Designing a novel verification system.* Ph.D. thesis, University of Twente.

[12] Pascal Urien, A. T., Hayder Saleh (2016), Cartes à puce internet, état de l'art et perpectives.

[13] Sun Microsystems, I. (2003) *Java Card Platform Specification 2.2.2.*

[14] Sun Microsystems, I. (2016) *Development Kit User's Guide For the Binary Release with Cryptography Extensions Java CardTM Platform, Version 2.2.2.*

[15] Team, S. S. D., capmap library.

[16] Team, S. S. D., Opal library.

Annexe A : Fonctionnement de la JVM standard

Cet annexe présente de façon abstraite, le fonctionnement d'élément de la machine virtuelle java utilisée aussi bien au niveau du java standard que du java card.

les deux principaux types de java sont :

- Les types primitifs (byte, boolean , short, char, int) ;
- Les types de références (classe, tableaux...).

La taille occupée par les éléments de type primitif est présentée par le Tableau 10.

Type	Taille en octet
byte	1
boolean	1
short	2
char	2
int	4

Tableau 10 – Taille des types primitifs

Pour ce qui est des types de référence ils peuvent être de type classe, tableau (unidimensionnelle dans notre contexte), une interface,...
Afin de pouvoir exécuter les programmes qui lui sont soumis, la jvm gère un ensemble de structures en mémoire vive connu sous le nom de *Run-Time Data Areas*. Parmi ces zones nous distinguons :

- JVM stack ;
- JVM Heap ;
- JVM Non Heap ;
- Method Area ;
- Frame ;
- Dynamic Linking ;
- ...

Un thread est une zone mémoire permettant l'exécution d'un programme java. sous la JCVM.

JVM STACK

Chaque thread possède une pile (stack) chargée de la gestion des frames , créée en même temps que le thread.

JVM Heap

Il s'agit d'un tas[2] partagé entre tous les threads de la JVM. Le tas est la zone mémoire ou sont créés et stockés toutes les instances de classes et tableaux (trop volumineux pour

2. Un tas est une structure de données organisées sous forme d'arbre (binaire parfait, complet par exemple) et peut même être représentable dans un tableau

être stocker au même endroit que les opérandes des opcodes en mémoire centrale). Le tas est principalement divisé en 3 zones :

- **Eden Space :** Il s'agit de la zone mémoire du tas ou la plupart des objets sont initialement aloués ;
- **Survivor Space :** Il s'agit de la zone mémoire contenant les objets ayant survécu au passage du garbage collector (ramasse miette) ;
- **Tenured Space :** Il s'agit de la zone mémoire contenant les objets ayant survécu pendant un certains temps dans l'espace de survit.

JVM Non Heap

La *JVM Non Heap* contient une zone mémoire lui permettant de stocker le bytecode des méthodes, la *Method Area*. Elle est également considérée comme la zone mémoire dans laquelle les données propres au fonctionnement de la JVM sont stockées.

Method Area

Il s'agit d'une zone mémoire partagée entre tous les threads de la JVM. C'est la zone où est située le code compilé des méthodes . Il stock par classe des structures telles que :

- **Run-Time Constant Pool :** sous la JCVM, chaque méthode contient une référence vers le pool de constante du package ;
- **Données de champs et méthodes ;**
- **Code de méthodes et constructeurs ;**
- **Les méthodes spéciales.**

Frame

C'est la zone mémoire située dans la pile du thread affectée de façon bijective à une méthode unique. La *Frame* est composé :

- d'une pile permettant de stocker les opérandes des opcodes ;
- d'une zone pour les variables locales (C'est un tableau dont l'accès se fait par indexation) ;
- d'une référence vers le run-time constant pool ;
- d'un champ pour la valeur de retour de la fonction.

Lors de la phase de compilation la taille maximal nécessaire pour l'exécution de la fonction (donc de la frame), est calculée et ajoutée a son bytecode. La taille d'une frame peut être étendue par des informations additionnelles, telles que des informations de débogage.
La frame joue le rôle de cadre de travail lors de l'exécution de la méthode. En effet, la zone des variables locales stock initialement les valeurs des paramètres d'entrées données lors de l'appel de la fonction (la première *slot* du tableau est réservée à la référence vers l'objet encapsulant la fonction dans le cas d'une fonction non static).
La Pile d'opérande comme son nom l'indique, contient l'opérande nécessaire pour l'opcode du PC Register du thread, en cours d'exécution.

Chaque *thread* possède son propre PC (*Program Counter*) registre. Chaque *thread* n'exécute qu'une seule fonction à la fois, qui est nommée « *current method* » ou méthode courante ; en rappel, sous la JCVM la machine virtuelle n'exécute qu'un seul *thread* à un moment donné. Le PC contient au cours de l'exécution du thread, l'adresse d'une instruction (opcode) de la méthode courante.

Dynamic Linking

Chaque frame contient une référence vers le run-time constant pool. Lors de la phase de compilation, toutes les références vers des objets ou des méthodes sont convertie en lien symbolique et sont stockées au niveau du constant pool. La résolution des liens symboliques vers des zones mémoires concrètes peut être réalisée lors de la phase qui suit le chargement de la classe en MC (lors de la phase de vérification...).

Annexe B : Exemple du code d'une applet

```
import javacard.framework.*;

public class HelloWorld extends Applet {

    // Initialisation de la variable
    // avec 'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd'.
    private final static byte[] message
          = { 0x48, 0x65, 0x6c, 0x6c, 0x6f,
                0x20, 0x77, 0x6f, 0x72, 0x6c, 0x64 };

    public static void install(byte[] bArray,
            short bOffset, byte bLength) { new HelloWorld(); }

    protected HelloWorld() { register(); }

    public void process(APDU apdu) {
        if (selectingApplet()) {
            // Retourne le status a OK
            return;
        }
        byte[] buffer = apdu.getBuffer() ;
        if ( buffer[ISO7816.OFFSET_CLA] != (byte)(0x80) )

            // Retourne le status word CLA NOT SUPPORTED
            ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
            switch(buffer[ISO7816.OFFSET_INS]) {
            case 0x10 :
                // Copie du contenu du message dans le buffer de la reponse
                Util.arrayCopy(message, (byte)0, buffer,
                      ISO7816.OFFSET_CDATA, (byte)message.length);

                // Envoi de la reponse
                apdu.setOutgoingAndSend(ISO7816.OFFSET_CDATA,
                        (byte)message.length);
                break;
            default:
                // Retourne le status a la valeur INS NOT SUPPORTED
                ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
        }
    }
}


}
```

Code Source 14 – code source d'une applet java card

Annexe C : Quelques Instructions Java Card

Mnemonic (opcode)	valeur en hexadécimale	valeur en décimal	argument(s)	pile avant->après	Description
aaload	24	36		..., arrayref, index ->..., value	Load reference from array
aastore	37	55		..., arrayref, index, value ->...	Store into reference array
aload	15	21	index	...->..., objectref	Load reference from local variable
aload_n	aload_0 = 18 aload_1 = 19 aload_2 = 1a aload_3 = 1b	aload_0 = 24 aload_1 = 25 aload_2 = 26 aload_3 = 27		...->..., objectref	Load reference from local variable where index is between [|0,3|]
areturn	77	119		..., objectref ->[empty]	Return reference from method
arraylength	92	146		..., arrayref ->..., length	Get length of array
astore	28	40	index		Store reference into local variable
astore_n	astore_0 = 2b astore_1 = 2c astore_2 = 45 astore_3 = 2e	astore_0 = 43 astore_1 = 44 astore_2 = 45 astore_3 = 46		..., objectref ->...	Store reference into local variable where index is between [|0,3|]
athrow	93	147		..., objectref ->objectref	Throw exception or error
baload	25	37		..., arrayref, index ->..., value	Load byte or boolean from array
bastore	38	56		..., arrayref, index, value ->...	Store into byte or boolean array
bspush	10	16		...->..., value	Push short on the operand stack
getstatic_a	7b	123	indexbyte1, indexbyte2	...->..., value	Get static field from class
goto	70	112	branch	No change	Branch always
goto_w	a8	168	branchbyte1, branchbyte2	No change	Branch always (wide index)
if_acmp <n >	if_acmpeq = 68 if_acmpne = 69	if_acmpeq = 104 if_acmpne = 105	branch	..., value1, value2 ->...	Branch if reference comparison succeeds.
if_acmp_w <n >	if_acmpeq_w=a0 if_acmpne_w=a1	if_acmpeq_w=160 if_acmpne_w=161	branchbyte1, branchbyte2	..., value1, value2 ->...	Branch if reference comparison succeeds (wide index)
if_scmp <n >	if_scmpeq = 0x6a if_scmpne = 6b if_scmplt = 0x6c if_scmpge = 0x6d if_scmpgt = 0x6e if_scmple = 0x6f	if_scmpeq = 106 if_scmpne = 107 if_scmplt = 108 if_scmpge = 109 if_scmpgt = 110 if_scmple = 111	branch	..., value1, value2 ->...	Branch if short comparison succeeds

if_scmp_w <n >	if_scmpeq_w=a2 if_scmpne_w=a3 if_scmplt_w=a4 if_scmpge_w=a5 if_scmpgt_w=a6 if_scmple_w=a7	if_scmpeq_w=162 if_scmpne_w=163 if_scmplt_w=164 if_scmpge_w=165 if_scmpgt_w=166 if_scmple_w=167	branchbyte1, branchbyte2	..., value1, value2 ->...	Branch if short comparison succeeds (wide index)
if <n >	ifeq = 60 ifne = 61 iflt = 62 ifge = 63 ifgt = 64 ifle = 65	ifeq = 96 ifne = 97 iflt = 98 ifge = 99 ifgt = 100 ifle = 101	branch	..., value ->...	Branch if short comparison with zero succeeds
if_w <n >	ifeq_w = 98 ifne_w = 99 iflt_w = 9a ifge_w = 9b ifgt_w = 9c ifle_w = 9d	ifeq_w = 152 ifne_w = 153 iflt_w = 154 ifge_w = 155 ifgt_w = 156 ifle_w = 157	branchbyte1, branchbyte2	..., value ->...	Branch if short comparison with zero succeeds (wide index)
ifnonnull	67	103	branch	..., value ->...	Branch if reference not null
ifnonnull_w	9f	159	branchbyte1, branchbyte2	..., value ->...	Branch if reference not null (wide index)
ifnull	66	102	branch	..., value ->...	Branch if reference is null
ifnull_w	9e	158		..., value ->...	Branch if reference is null (wide index)
invokespecial	8c	140	indexbyte1, indexbyte2	..., objectref, [arg1, [arg2 ...]] ->...	Invoke instance method; special handling for superclass, private, and instance initialization method invocations
invokevirtual	8b	139	indexbyte1, indexbyte2	..., objectref, [arg1, [arg2 ...]] ->...	Invoke instance method; dispatch based on class
ireturn	79	121		..., value.word1, value.word2 ->[empty]	Return int from method
itableswitch	74	116	itableswitch, defaultbyte1, defaultbyte2, lowbyte1, lowbyte2, lowbyte3, lowbyte4, highbyte1, highbyte2, highbyte3, highbyte4, jump offsets...	..., index ->...	Access jump table by int index and jump
return	7a	122		...->[empty]	Return void from method

sconst_n	sconst_m1 = 2 sconst_0 = 3 sconst_1 = 4 sconst_2 = 5 sconst_3 = 6 sconst_4= 7 sconst_5 = 8	sconst_m1 = 2 sconst_0 = 3 sconst_1 = 4 sconst_2 = 5 sconst_3 = 6 sconst_4= 7 sconst_5 = 8		...->...,	Push short constant
sinc	59	89	index,const	No change	Increment local short variable by constant
sipush	13	19	byte1,byte2	...->..., value1.word1, value1.word2	Push short
slookupswitch	75	117	slookupswitch, defaultbyte1, defaultbyte2, npairs1, npairs2, match-offset pairs...	..., key ->...	Access jump table by key match and jump
sreturn	78	120		..., value ->[empty]	Return short from method
sspush	11	17	byte1,byte2	...->..., value	Push short
stableswitch	73	115	stableswitch, defaultbyte1, defaultbyte2, lowbyte1, lowbyte2, highbyte1, highbyte2, jump offsets...	..., index ->	Access jump table by short index and jump

Tableau 11: Description des instructions utilisées dans le document

Annexe D : Le patron de conception Visitor

Le patron de conception *Visitor* est un modèle de conception dont l'objectif est de permettre l'extension ou la réutilisation (bien au delà du simple héritage) d'un programme sans nécessiter sa modification. Supposons deux classes *ClasseA* et *CLasseB*, ou *CLasseA* produit un ensemble d'informations devant être analysées/traitées, et *CLasseB* réalise des traitements. Afin de réaliser les traitements, la classe *CLasseB* va implémenter une interface Java définie par *CLasseA* (que l'on nommera *ClasseAvisitor*) sur la base de la structure du modèle de données qu'il produit. la classe *CLasseA* défini ensuite une méthode *accept(CLasseAvisitor classeAvisitor[,...])* qui sera chargée d'appeler les méthodes de l'interface *ClasseAvisitor* afin de réaliser les traitements. Définir de nouveaux traitements pour la *ClasseA* consiste simplement à définir une nouvelle classe qui implémente les méthodes de l'interface *ClasseAvisitor* et à la passer en paramètre à la *ClasseA*.

Une illustration de ce modèle est faite par la classe *ClassReader*. *ClassReader* est une classe d'ASM qui prend en entrée un flux de données binaire représentant le bytecode d'une classe, l'interprète en vue de retrouver les informations stockées (en accord avec la spécification). Pour chaque élément de la classe interprétée (annotation, information de débogage, instruction conditionnelle,...) il fera appel au méthode d'une classe (le visiteur) qui sera chargé de « consommer » les informations produites en vue de réaliser des traitements déterminés. Une classe implémentant le pattern *Visitor* doit être défini

Table des matières