GÉNÉRATION DE TESTS DE VULNÉRABILITÉ POUR VÉRIFIEUR DE BYTE CODE JAVA CARD

par

Aymerick Savary

Mémoire présenté au Départements d'informatique
en vue de l'obtention du grade de maître sciences (M.Sc.)

FACULTÉ DES SCIENCES
UNIVERSITÉ DE LIMOGES
Limoges, France,

FACULTÉ DES SCIENCES
UNIVERSITÉ DE SHERBROOKE
Sherbrooke, Québec, Canada,

24 mai 2013

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-95108-8

Our file Notre référence
ISBN: 978-0-494-95108-8

Le 28 mai 2013

le jury a accepté le mémoire de Monsieur Aymerick Savary dans sa version finale.

Membres du jury

Professeur Marc Frappier
Directeur de recherche
Département d'informatique

Professeur Jean-Louis Lanet
Codirecteur de recherche
Université de Limoges

Monsieur Benoît Fraikin
Évaluateur interne
Département d'informatique

Professeur André Mayers
Président rapporteur
Département d'informatique

Sommaire

Il devient important d'assurer que tout système critique est fiable. Pour cela différentes techniques existent, telles que le test ou l'utilisation de méthodes formelles. S'assurer que le comportement d'un vérifieur de byte code Java Card n'entraînera pas de faille de sécurité est une tâche complexe. L'automatisation totale de cette vérification n'a pour le moment pas été réalisée. Des jeux de tests coûteux ont été produits manuellement, mais ils doivent être refaits à chaque nouvelle spécification.

Les travaux présentés dans ce mémoire proposent une nouvelle méthode pour la génération automatique de tests de vulnérabilité. Ceux-ci reposent sur l'utilisation et la transformation automatique de modèles formels. Pour valider cette méthode, un outil a été développé puis utilisé sur différentes implémentations du vérifieur de byte code Java Card.

Le langage de modélisation que nous avons utilisé est Event-B. Nos modèles représentent le comportement normal du système que l'on souhaite tester. Chaque instruction est modélisée comme un événement. Leur garde représente l'ensemble des conditions que doit satisfaire une instruction pour être acceptable.

À partir de ce modèle initial, une succession de dérivations automatiques génère un ensemble de modèles dérivés. Chacun de ces modèles dérivés représente une faute particulière. On extrait de ces nouveaux modèles les tests de vulnérabilité abstraits. Ceux-ci sont ensuite concrétisés puis envoyés à un système à tester. Ce processus est assuré par notre logiciel qui repose sur les API Rodin, ProB, CapMap et OPAL.

Mots-clés: méthodes formelles ; Event-B ; vérification de modèle ; ProB ; Java Card ; Tests de vulnérabilité ; sécurité et sûteré

SOMMAIRE

Remerciements

Tout d'abord, je tiens à remercier le partenariat entre l'Université de Limoges et l'Université de Sherbrooke. La réunion des compétences des deux laboratoires XLIM et le GRIL, que je remercie, a rendu possible une telle recherche. Je tiens à remercier mes co-directeurs.

Je remercie le professeur Marc Frappier de l'Université de Sherbrooke pour m'avoir accepté dans son laboratoire. Je le remercie pour m'avoir ouvert les portes des méthodes formelles.

Je remercie le professeur Jean-Louis Lanet de l'Université de Limoges pour tout ce qu'il m'a donné durant ces deux années mais aussi au cours des deux précédentes. Je le remercie pour toute l'énergie qu'il a mis dans ce projet.

Je remercie Tiana Razafindralambo ainsi que Josselin Dolhen pour l'aide apportée au développement de notre outil. Je remercie aussi les doctorants de l'équipe Smart Secure Devices (SSD), en particulier Guillaume Bouffard, Amaury Gauthier, David Pequegnot et Nassima Kamel, pour m'avoir aidé dans la compréhension de certaines attaques logiques sur cartes à puce.

Finalement, je remercie toutes les autres personnes qui se sont intéressées à ce projet et ont apporté un regard critique me permettant de progresser rapidement.

Remerciements

Abréviations

APDU Application Protocol Data Unit

CAP Converted APplet

CPA Corelation Power Analysis

CTL Computation Tree Logic

DPA Differential Power Analysis

FIB Focused Ion Beam

VTG Vulnerability Tests Generator

JCRE Java Card Runtime Environement

JVM Java Virtual Machine

LTL Linear Temporal Logic

RID Ressource IDentifier

SEM Scanning Electron Microscop

SPA Simple Power Analysis

TdeV Test de Vulnérabilité

Table des matières

Liste des figures

Liste des tableaux

Introduction

Contexte

Avec l'arrivée des cartes à puce de nouvelle génération, de nouvelles problématiques de sécurité sont apparues. Ces nouvelle cartes, constituant un élément essentiel dans la sécurité de différents systèmes, il est nécessaire de garantir leur fiabilité. Pour cela, un certain nombre de techniques ont été développées. Cependant, face à l'ingéniosité des pirates informatiques, ces techniques ne suffisent pas toujours. Bien que des contre-mesures soient régulièrement intégrées, il est important de s'assurer avant délivrance d'un produit qu'il sera résistant. Pour cela, deux philosophies existent, développer un programme correct par construction ou bien construire le programme puis le tester.

Problématique

La construction de nouvelles techniques ainsi que de nouveaux outils de sécurité est un atout important pour la conception de systèmes sécurisés. L'utilisation de tests de vulnérabilité permet d'augmenter la fiabilité des systèmes face à une tentative d'intrusion. Cependant, la génération de tests de vulnérabilité à l'aide des techniques actuellement existantes est limitée. Leur utilisation dans le domaine du test des cartes à puce est donc difficile.

Méthodologie

Nous avons proposé une méthode de génération de tests de vulnérabilité basée sur l'utilisation de méthodes formelles. Afin de valider cette approche, nous l'avons appliquée au vérifieur de byte code Java Card. Cette solution est toutefois générique et peut s'appliquer à de nombreux problèmes. Cette application a premièrement été réalisée sur le langage de Stephen N. Freund et John C. Mitchell [22]. Ce langage est constitué d'un nombre de mots moins important que le langage Java Card tout en gardant tous les concepts présents dans ce dernier. Dans une deuxième partie, nous l'avons appliquée à un sous-ensemble du langage Java Card.

Les modélisations formelles ont été rédigées dans le langage Event-B [1], grâce à l'outil Rodin [3]. Une phase de manipulation des modèles, permettant de générer les modèles des failles, a été réalisée à l'aide d'un outil que nous avons développé, le VTG (Vulnerability Tests Generator). Pour l'obtention des tests abstraits, nous avons utilisé l'animateur de modèle ProB. Dans le cas du langage Java Card, nous avons aussi pu concrétiser et exécuter ces tests à l'aide d'une autre partie du VTG. Ces tests concrets ont ensuite été utilisés afin de vérifier une partie de la sécurité du vérifieur de byte code off-card de Oracle puis de celui on-card de différentes cartes à puce.

Résultats

Avec un modèle de taille raisonnable, nous avons montré que l'approche ainsi que les outils, utilisés ou développés, permettent de générer des tests de vulnérabilité. L'utilisation du jeu de test sur le vérifieur de byte code off-card n'a révélé aucune faille. Dans le cas des cartes à puce, nous avons constaté que certaines cartes ne contenaient pas toute les fonctions du vérifieur de byte code.

Structure du mémoire

Dans le premier chapitre nous faisons un état de l'art sur les attaques contre les cartes à puces ainsi que sur l'utilisation des méthodes formelles pour la génération

de tests. Dans le second chapitre, nous introduisons quelques notions essentielles à la compréhension de ce mémoire. Dans le troisième chapitre, nous détaillons la partie théorique de notre approche. Dans le quatrième chapitre, nous décrivons comment nous avons mis en œuvre notre approche pour les vérifieurs de byte code Java Card. La première partie du chapitre 5 donne une liste des logiciels que nous avons utilisé. La deuxième partie décrit le logiciel que nous avons mis en place permettant de mettre en application notre approche. Nous donnons dans le chapitre 6 les résultats d'utilisation du logiciel sur trois exemples.

Chapitre 1

État de l'art

1.1 Attaque contre les cartes à puce

Les cartes à puce sont des systèmes renfermant des données très sensibles qui doivent absolument être protégées. De multiples techniques, que nous décrivons dans la section 2.1.2, permettent d'assurer cette sécurité. Malgré toutes ces précautions, tant sur le matériel que sur les logiciels, de nombreuses attaques ont eu lieu. Les premières attaques sont arrivées avec Boneh, DeMillo et Lipton [14]. Malgré toutes les précautions prises afin de rendre les cartes à puce plus résistantes, cela a montré que des failles peuvent toujours persister. Suite à ces attaques, une succession de nouvelles d'attaques et de contremesures ont vu le jour.

1.1.1 Attaques physiques

Ce type d'attaque s'intéresse au comportement du matériel face à des situations non prévues. Cette catégorie peut être re-décomposée avec les attaques invasives et les attaques non invasives.

Les attaques invasives consistent à modifier ou altérer les différentes couches formant une carte à puce afin d'accéder aux composants internes de celle-ci. Ces attaques sont très performantes mais elles détruisent le matériel, le rendant inutilisable par la

suite. La première phase, le *etching*, consiste à utiliser des produits chimiques. Le composant est mis à nu afin d'accéder aux différentes couches du micro-processeur. Par la suite plusieurs techniques existent :

- *SEM* (Scanning Electron Microscop) : l'utilisation d'un microscope électronique permet de savoir quelle partie de la puce est en fonctionnement.
- *probing* : des sondes sont directement placées sur les canaux de données afin de récupérer les informations transmises entre différents composants du micro-processeur
- *FIB* (Focused Ion Beam) : grâce à cet appareil, il est possible de directement supprimer ou créer des liaisons dans le micro-processeur

Les attaques non invasives vont consister en une écoute du matériel lors de son fonctionnement. Contrairement aux attaques invasives, elles ne détruisent pas le produit. Celui-ci reste donc fonctionnel après l'attaque.

Le premier type d'attaque non invasive sont les attaques par canaux cachés. Une simple observation du comportement lors du fonctionnement de l'application permet de retrouver des informations non extractables. La première attaque physique, publiée par Kocher [21], dans laquelle une analyse de temps d'exécution a permis de retrouver une clef cryptographique de l'algorithme RSA. Différents canaux peuvent être exploités tels que le temps d'exécution, la consommation de courant ou encore le champ électromagnétique. Plusieurs techniques permettent ensuite d'analyser ces canaux afin d'en extraire une information. La plus simple des techniques d'analyse est la SPA (Simple Power Analysis). Avec cette technique, les informations secrètes sont directement extraites du canal. La seconde attaque, un peu plus évoluée, est la DPA (Differential Power Analysis) qui à été présenté pour la première fois par Kocher, Jaffe et Jun [25]. Elle consiste à effectuer une hypothèse sur le comportement du canal puis par une analyse différentielle de vérifier si l'hypothèse est valide. La troisième technique, CPA (Correlation Power Analysis) est une amélioration de la DPA qui fut présenté par Brier, Clavier et Olivier [19].

Le deuxième type d'attaque non invasive sont les attaques par faute, aussi appelées PACA dont une première description est donnée dans [4]. Elles vont consister en

une observation de comportement suite à une perturbation volontaire. Là encore, il existe différents canaux utilisables pour l'observation ainsi que pour effectuer la perturbation :

- attaque électrique : la tension d'entrée est brutalement modifiée afin de produire un comportement non attendu,
- attaque électromagnétique : celle-ci consiste en l'application d'un champ électromagnétique sur une zone précise permettant de modifier le contenu des mémoires,
- attaques optiques : l'apport d'énergie dû à l'émission de photons permet de changer le contenu des mémoires,
- attaque par variation de la fréquence d'horloge : modification de la fréquence d'horloge afin de perturber le fonctionnement normal du processeur.

1.1.2 Attaques logiques

Les attaques actuelles contre les cartes à puce sont essentiellement des attaques matérielles. Ces attaques nécessitent généralement des moyens physiques et des outils adaptés (lasers, oscilloscopes, etc.) réduisant par leur sophistication le nombre d'attaquants. De nouvelles attaques, dites logiques, commencent à voir le jour et reposent souvent sur la découverte d'une faille et de son exploitation. Cela consiste à récupérer des données ou fonctionnalités à partir d'injection de données ou de commandes. Contrairement aux attaques physiques, ce type d'attaques est moins onéreux à mettre en place car il ne demande qu'un simple lecteur de cartes et un ordinateur. De plus, les attaques logiques ne détruisent pas le matériel. Cependant elles sont plus complexes et demandent souvent une plus grande connaissance du système que pour les attaques physiques. De plus, elles restent liées à une implémentation particulière.

La première méthode est similaire à celle que nous trouvons sur les PC. Il suffit de trouver des failles dans les logiciels. Pour les cartes, nous avons deux techniques bien utilisées qui sont le *direct protocol attaks* et le *fuzzing*. Comme l'a été proposé par le groupe de recherche Smart Secure Devices de l'Université de Limoges lors de la conférence SAR-SSI 2011 [27] puis lors de la conférence C&ESAR 2011 [11], le *fuzzing* est simple a mettre en place et apporte de bons résultats.

La mise à disposition des cartes dites *ouvertes* offre un nouveau champ d'attaque. Ces cartes autorisant le chargement de nouvelles applications après délivrance. Le but de ces attaques consiste à exploiter ces failles simplement à l'aide d'applications. Ces attaques sont très puissantes si elles fonctionnent, mais restent très difficile à mettre en place. Différentes attaques proposées par Wojciech et Erik [28] utilisent un code mal typé. L'idée est d'exploiter une confusion de type entre les tableaux de types primitifs différents. Il est possible de lire ou d'écrire dans des tableaux de types différents en utilisant une faille dans le mécanisme de transaction de Java Card. De plus, nous avons montré dans le *Journal in Computer Virology* [23] qu'un fichier structurellement mal formé pourrait permettre d'introduire un virus dans une Java Card. Plus récemment, un ensemble d'attaques physiques, [15], [16] et [23], ont été proposées. Elles mettent en avant ce à quoi nous pouvions accéder, données et processus, en utilisant ce type d'attaques. Les recherches proposées dans ce mémoire on été présentées pour la 1ère fois durant la conférence SAR-SSI 2011 [34]. Le but est d'automatiser ce type d'attaques afin de rapidement découvrir de nouvelles failles.

1.2 Test de sécurité à base de modèle

Les méthodes formelles sont généralement utilisées afin de produire des programmes sûrs par construction. La génération des tests, MBT (*Model Base Testing*), peut utiliser ces mêmes méthodes formelles afin de vérifier la conformité d'un système avec une spécification. Le livre *Practical Model-Based Testing : A Tools Approach* [37] propose une vision détaillée des techniques, outils ainsi que des cas d'exemples d'utilisation des MBT.

Afin de produire un modèle permettant le MBT, Utting, Pretschner et Legeard propose dans [38] une classification, reprise dans la figure 1.1. Les travaux de Jeremy et Alain dans [21] puis Bouquet, Grandpierre, Legeard et Peureux dans [17] et enfin Satpathy and Leuschel et Butler dans [33], proposent différentes méthodes permettant de générer des jeux de tests.

Le MBT est largement employé dans le domaine des cartes à puce. Nous pouvons notamment citer le récent projet européen *POSÉ* [26]. Seulement certaines des applications présentes sur carte à puce bénéficient d'un cycle de développement formel.

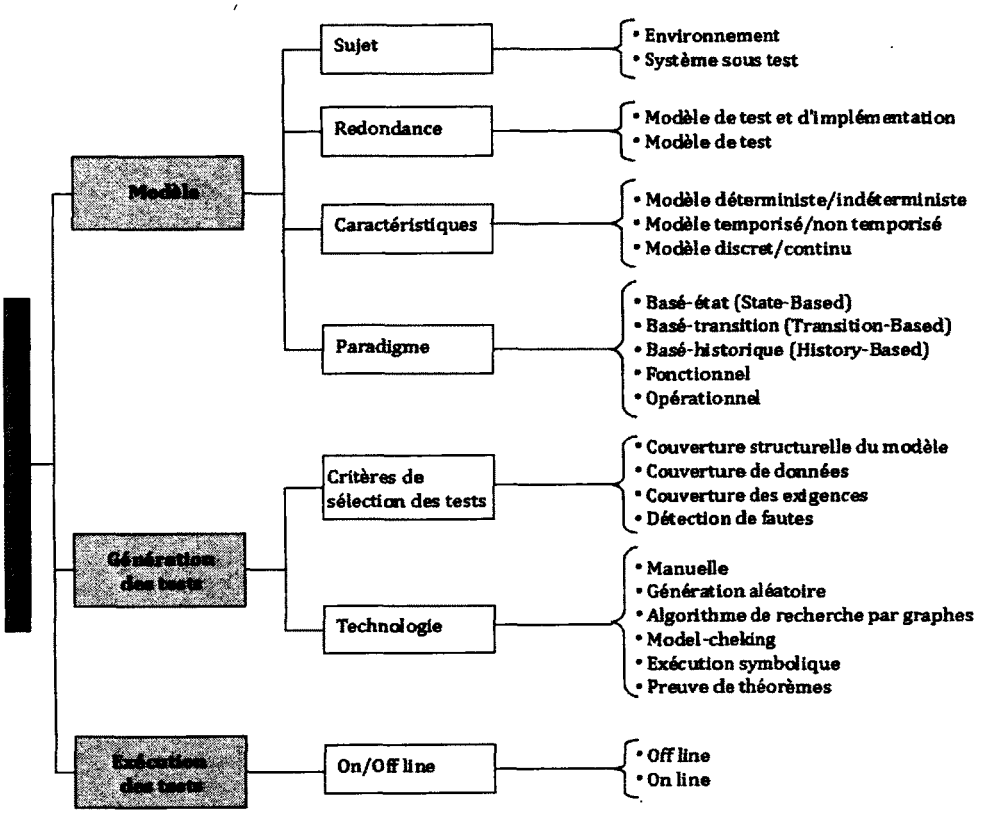

figure 1.1 – Classification MBT

Pour les autres applications, il est nécessaire de les tester pour assurer leur fiabilité. En effet, comme décrit en section 1.1, des failles peuvent toujours persister. De plus, les carte à puces ne peuvent pas être mise à jours si une faille est découverte. L'utilisation de MBT permet de produire des jeux de tests génériques, i.e. indépendants de l'implémentation. Ces jeux de test peuvent ensuite être ré-utilisés indépendamment de l'application afin de garantir son implémentation.

Certaines recherches, telles que celles effectuée par la société smart testing [18], tentent de proposer des approches génériques pour toutes applications. D'autres, comme *Generation of test sequences from formal specifications : GSM 11-11 standard case study* [12] ou encore *Formal specification of the JavaCard API in JML : the APDU class* [31], testent des protocoles de communication standardisés.

Chapitre 2

Fondement

2.1 Carte à puce Java Card

Les cartes à puce sont aujourd'hui monnaie courante : les cartes bancaires, les cartes d'assurance maladie, les passeports, les cartes SIM... Tous ces systèmes suivent une même idée originale, mettre en place un système matériel capable de conserver des informations de façon trés sécurisée tout en proposant la possibilité d'effectuer quelques traitements directement dans la carte.

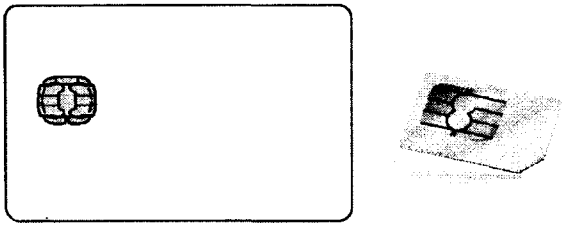

figure 2.1 – Cartes à puce

Une carte à puce est un composant électronique, défini par un ensemble de normes, que l'on peut trouver sous différents formats telles que les cartes bancaires ou encore les cartes SIM comme illustrées en figure 2.1. Bien que ces systèmes soient restreints en capacité processeur ou encore en espace mémoire, il est possible de charger des programmes autonomes de même que dans un ordinateur.

Il existe à l'heure actuelle différentes plateformes logicielles permettant de les utiliser. Java Card est l'une d'entre-elles, offrant le support de Java pour les cartes à puce. Seulement un sous-ensemble des technologies Java est ré-utilisé afin de s'adapter aux contraintes d'espace, de puissance de calcul et de sécurité. Les programmes sont généralement des *applets* bien que depuis la version 3.0 *Connected Edition* il soit possible de charger des *servlets*. Les Java Cards sont dites à *systèmes ouverts* et autorisent donc le chargement de programmes après délivrance. Selon les résultats publiés dans eurosmart 2012 [6], cette plateforme représente à l'heure actuelle 95% des cartes à puce dans le monde.

Le langage Java étant un langage largement répandu dans l'industrie, son utilisation dans le domaine des cartes à puce lui permet de s'ouvrir à une grande communauté de développeurs. L'utilisation d'un langage plus abstrait que les langages assembleur rend leur utilisation plus aisée. La non dépendance au matériel permet une bonne interopérabilité des applications sur toutes les cartes. De plus, la compatibilité avec les normes, notamment ISO 7816, est simplifiée grâce à l'utilisation des API fournies. Du point de vue de la sécurité, qui est un point très important pour les cartes à puce, le langage Java est sûr. Tous les mécanismes tels que le typage fort ou encore le pare-feux assurent pour les applications une sécurité permanente. Enfin, la prise en charge d'un environnement multi-applicatif permet d'installer plusieurs applications de différents types sur la même carte à puce.

2.1.1 Architecture de la plateforme

Dû au peu de ressources disponibles sur les cartes à puce, la JVM (*Java Virtual Machine*) que l'on peut trouver sur une station de travail a été séparée, comme on peut le voir sur la figure 2.2, en deux :

- une partie à l'extérieur de la carte, aussi appelée *off-card*, généralement sur la station de travail. Cette partie comporte un convertisseur ainsi qu'un vérifieur de byte code.
- une partie dans la carte, aussi appelée *on-card* correspond à l'interpréteur de code qui se charge d'exécuter les applications.

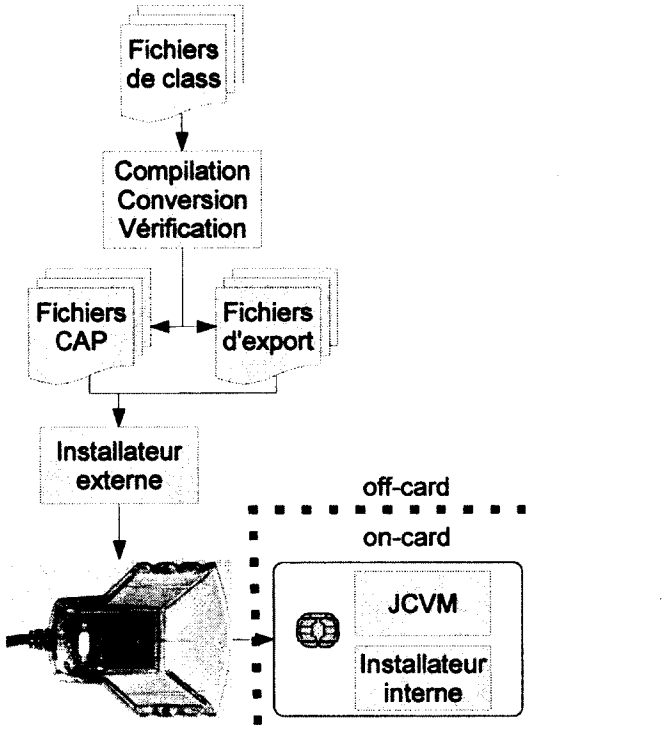

figure 2.2 – Séparation de la machine Java en deux parties

Ces deux parties assurent les mêmes fonctionnalités qu'une JVM classique. Cependant cette séparation est à l'origine d'un grand nombre d'attaques dont certaines ont été citées en section 1.1.

Le cycle de vie d'un programme Java Card est donné en figure 2.3. Dans un premier temps, les codes sources Java sont compilés sur la station de travail pour donner les fichiers de classe (ce code est aussi vérifié afin de s'assurer que le code compilé respecte bien les spécifications.). Une étape de conversion génère ensuite le fichier CAP (Converted APplet), puis ce fichier est stocké ou envoyé sur un réseau. Finalement, il est chargé puis installé dans la carte à puce.

Dans le cas des Java Cards 3.0 *Connected Edition*, l'architecture est un peu différente. En effet, le vérifieur de byte code est obligatoirement embarqué. La carte à puce va donc effectuer une vérification avant installation des nouvelles applications.

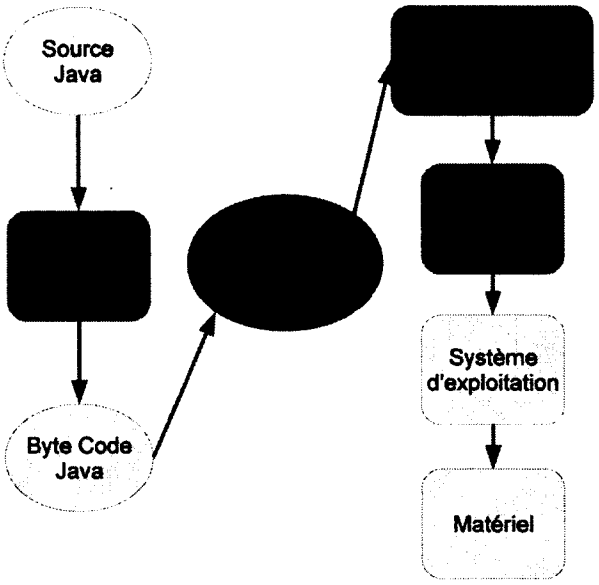

figure 2.3 – Cycle de vie d'un programme Java Card

Applet Java Card

Le cycle de vie d'un *applet* Java Card est composé de six phases représentées sur la figure 2.4 :

- *install* : installe l'applet dans la carte
- *register* : enregistre l'applet auprès du JCRE (Java Card Runtime Environment)
- *select* : active un applet
- *process* : traite des commandes APDU (Application Protocol Data Unit)
- *deselect* : désactive un applet
- *delete* : supprime un applet

Après chargement d'un *applet*, celui-ci est stockée temporairement dans la mémoire temporaire d'entrée. Lors du processus d'installation, *l'applet* est sauvegardé dans une mémoire persistante et l'édition des liens est effectuée. Suite à cela, une instance est créée puis enregistrée auprès du JCRE. À ce stade, *l'applet* est totalement installé et prêt à être utilisé. Comme les Java Cards sont des systèmes multi-applicatif, il faut commencer par sélectionner *l'applet* que l'on souhaite utiliser. Lors de son utilisation la méthode *process* récupère les commandes. À la fin de son utilisation, *l'applet* est

désélectionné. Enfin, l'opération de désinstallation supprime l'accès à *l'applet*.

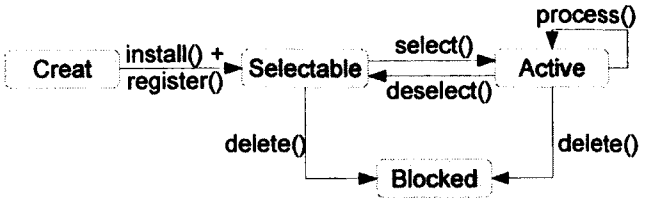

figure 2.4 – Cycle de vie d'un *applet* Java Card

La communication avec un *applet* se fait à l'aide du protocole APDU défini par la norme ISO 7816-4. Les Java Cards fonctionnent comme des serveurs, dans le sens où elles ne peuvent initier des connexions mais simplement répondre à des requêtes. Dans ce protocole, nous avons les commandes APDU qui sont envoyées à la carte puis les réponses APDU qui sont renvoyées par la carte suite à l'envoi d'une commande. Les commandes APDU sont constituées comme suit :

- CLA : classe d'application
- INS : identifiant de l'instruction à exécuter (paramètre utilisable dans *l'applet*)
- P1, P2 : paramètres de l'instruction
- LC : taille du champ de données
- DATA : champ de données transmis à l'*applet*
- LE : nombre d'octets attendus en retour

Les réponses APDU sont constituées comme suit :

- DATA : données en retour à une exécution
- SW1, SW2 : deux octets représentant l'état de la carte après exécution d'une commande

2.1.2 Sécurité de la plateforme Java Card

Java a été conçu pour apporter un niveau de sécurité important. La spécification définit un ensemble de propriétés permettant d'assurer un tel niveau de sécurité. Différents composants sont en charge de vérifier le respect de ces propriétés.

Le vérifieur de byte code est l'un de ces composants. Étant un point très important dans ce mémoire, il sera détaillé dans la section 2.2.

Les Java Cards sont des cartes à puce multi-applicatives. Le pare-feu est en charge de la ségrégation des *applets*, ainsi que de la communication entre ceux-ci si nécessaire. Chaque *package* possède un contexte de sécurité. Touts les *applets* d'un même *package* partagent donc le même contexte de sécurité.

Afin de garantir la mise à jour de certains objets dans la mémoire persistante, des mécanismes de transaction ont été ajoutés. Cela permet notamment de palier à un problème d'arrachage prématuré de la carte du lecteur. Les transactions Java Card vont s'assurer que certains blocs définis dans le programme sont bien atomiques.

2.2 Le vérifieur de byte code

Le vérifieur de byte code est vu comme le composant offensif de la sécurité de la plateforme Java. Cette vérification permet de s'assurer que les règles de sécurité du langage Java sont bien respectées par une analyse statique. Comme nous pouvons le voir dans la figure 2.3, il intervient durant la phase de chargement, avant toute exécution.

Le format de fichier envoyé aux Java Cards est défini dans le chapitre 6 de la spécification de la machine virtuelle Java Card [36] et porte le nom de fichier CAP. Le fichier CAP est la représentation binaire du programme que l'on souhaite envoyer à la carte. Ce fichier est obtenu par conversion des fichiers de classe. Il est l'assemblage de 12 composants. Chacun de ces composants possède un ensemble de contraintes internes et externes représentées sur la figure 2.5.

La vérification de ce fichier s'effectue en deux parties, la vérification de structure que nous détaillerons dans la section 2.2.1 et la vérification de type que nous détaillerons dans la section 2.2.2. Au vu des ressources très limitées des cartes à puce, ce processus se déroule généralement hors de la carte et seule la spécification Java Card 3.0 *Connected Edition* oblige ce processus de vérification à être exécuté directement dans la carte.

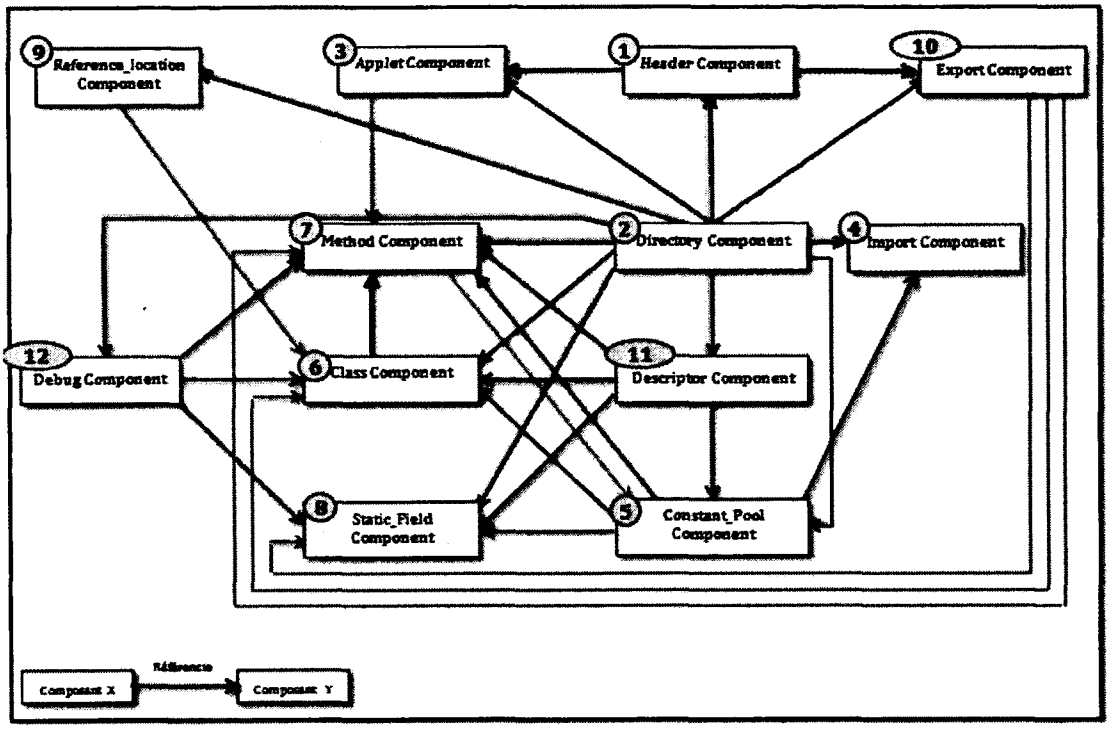

figure 2.5 – Liens d'interdépendances entre les composants du fichier CAP

2.2.1 La vérification de structure

La vérification de structure consiste à vérifier que toutes les dépendances internes et externes de chaque composant sont bien respectées. Si nous prenons par exemple le composant *Applet*, N° 3 sur la figure 2.5 (cette figure est extraite du mémoire de magister de Hamadouche [32]). Il est défini que le champ *size* doit contenir la taille du composant. Cela correspond à une contrainte interne, le composant suffit à lui-même pour valider ou invalider cette propriété. Un autre exemple de contrainte consiste à vérifier que les RID (Ressource IDentifier) contenus dans le tableau AID (Applet IDentifier) de ce composant sont bien identiques à ceux déclarés dans le composant *Header*. À ce moment là, nous avons besoin d'autres composants afin de valider ou invalider la propriété, ce qui correspond à une contrainte externe. Dans ce processus, si une seule propriété est invalide, la vérification de structure s'arrête et le fichier CAP est rejeté.

PC	Instruction	Pile	
		avant	après
1	*sconst_3*	∅	*short*
2	*sconst_1*	*short*	*short*; *short*
3	*sadd*	*short*; *short*	*short*
4	*pop*	*short*	∅
5	*return*	∅	

tableau 2.1 – Byte code valide du premier programme

2.2.2 La vérification de type

La vérification de type va analyser le comportement du programme. Pour cela, elle va récupérer le byte code du programme dans le *Method Component* (N° 7 sur la figure 2.5). Cette vérification va se faire méthode après méthode pour toutes les méthodes contenues dans ce composant.

Pré-conditions

Chaque instruction possède un certain nombre de pré-conditions, que les programmes doivent respecter pour être considérés comme valides. Cependant, le fichier CAP ne renferme pas suffisamment d'informations permettant de le valider *i.e.* de valider toutes les pré-conditions. Il va donc falloir reconstruire ces informations par une exécution symbolique du programme. Durant cette exécution, les pré-conditions ne portant que sur des typages, seules les informations de typage seront importantes et non les valeurs.

Pour illustrer ce processus, prenons deux exemples de programme byte code. Le premier exemple est un programme valide, *i.e.* acceptable par le vérifieur de byte code : *sconst_3*, *sconst_m1*, *sadd*, *pop*, *return*. Le deuxième est un programme non valide, *i.e.* rejetable par le vérifieur de byte code : *aconst_null*, *sconst_m1*, *sadd*, *pop*, *return*. Les tableaux 2.1 et 2.2 représentent l'état de la pile avant exécution et après exécution de chaque instruction.

Les instructions *sconst_1*, *sconst_3*, *aconst_null* possèdent simplement comme

PC	Instruction	Pile	
		avant	après
1	*aconst_null*	$\varnothing$	*ref*
2	*sconst_1*	*ref*	*ref*; *short*
3	*sadd*	*ref*; *short*	??
4	*pop*	??	??
5	*return*	??	??

tableau 2.2 – Byte code invalide du deuxième programme

précondition que la pile ne soit pas pleine. L'instruction *sadd* nécessite deux *shorts* au sommet de la pile. Pour l'instruction *pop*, la pile ne doit pas être vide. Finalement l'instruction *return* ne possède aucune pré-condition particulière. Dans le premier programme, nous pouvons voir que toutes les pré-conditions sont satisfaites. Le programme est donc accepté. Pour le deuxième programme, nous constatons qu'à l'instruction *sadd*, pour $PC = 3$, la pré-condition n'est pas vérifiée. Les types Java sont ordonnés selon un treillis, aussi appelé arbre d'héritage. Le type *Top* représente le sommet de ce treillis et les types *short* et *référence* sont incompatibles. Ce programme sera donc rejeté par le vérifieur de byte code.

Branchements

La vérification est un peu plus complexe lorsque nous avons des branchements dans le programme. Le processus de vérification va s'assurer que dans tous les cas possibles, le programme reste valide. Pour chaque branchement, le vérifieur parcourt les deux branches possibles jusqu'à ce qu'elles se rejoignent. Au point de rencontre, une inférence de type permettra de déterminer si les deux branches sont valides. Si les types au point de jonction ne sont pas compatibles avec la précondition de l'instruction, le programme est rejeté. Si les types sont compatibles, la vérification continue. Si les types ne sont pas identiques, mais compatibles avec la précondition, un algorithme de calcul de point fixe est alors utilisé. Cet algorithme permet de déterminer si, après plusieurs passages par le point de jonction, les types sont toujours compatibles.

Pour illustrer ce processus prenons, deux exemples de programme byte code. Le

PC	Instruction	Pile chemin 1		Pile chemin 2		Inférence
		avant	après	avant	après	
1	*sconst_3*	∅	*short*	∅	*short*	
2	*sconst_0*	*short*	*short*; *short*	*short*	*short*; *short*	
3	*ifeq 3*	*short*; *short*	*short*	*short*; *short*	*short*	
4	*pop*	*short*	∅			
5	*aconst_null*	∅	*reference*			
6	*pop*	*reference*	∅	*short*	∅	*Top*
7	*return*	∅		∅		

tableau 2.3 – Byte code valide du troisième programme

PC	Instruction	Pile chemin 1		Pile chemin 2		Inférence
		avant	après	avant	après	
1	*sconst_3*	∅	*short*	∅	*short*	
2	*sconst_0*	*short*	*short*; *short*	*short*	*short*; *short*	
3	*ifeq 3*	*short*; *short*	*short*	*short*; *short*	*short*	
4	*pop*	*short*	∅			
5	*aconst_null*	∅	*reference*			
6	*s2b*	*reference*	rejet	*short*	rejet	Top
7	*return*	rejet	rejet	rejet	rejet	

tableau 2.4 – Byte code invalide du quatrième programme

premier exemple est un programme valide, *i.e.* acceptable par le vérifieur de byte code : *sconst_3*, *sconst_0*, *ifeq 3*, *pop*, *aconst_null*, *pop*, *return*. Le deuxième est un programme non valide, *i.e.* rejetable par le vérifieur de byte code : *sconst_3*, *sconst_0*, *ifeq 3*, *pop*, *aconst_null*, *s2b*, *return*. Les tableaux 2.3 et 2.4 représente l'état de la pile avant exécution et après exécution de chaque instruction, cela pour les deux branches d'exécution possibles.

L'instruction *ifeq n* permet d'effectuer un branchement conditionnel avec un offset de *n*. Cette instruction nécessite un *short* au sommet de la pile. Dans les deux programmes, la pré-condition de cette instruction est valide. L'erreur est faite au point de branchement. Dans ces deux programmes, nous devons vérifier que les piles présentes après exécution de l'instruction 2 et après l'instruction 4 sont compatibles. Au point de jonction nous avons deux piles différentes. L'inférence de type nous permet de déterminer le type commun qui est *Top*. Ce type est l'élément qui borne le treillis de type Java et il n'est pas utilisable.

Pour le premier programme, la pré-condition de l'instruction *pop* n'imposant rien sur le type de l'élément au sommet de la pile. L'instruction au $PC = 6$ est donc valide et nous pouvons continuer la vérification jusqu'à la fin. Ce programme est donc accepté.

L'instruction *s2b* permet de convertir un *short* en *byte* et nécessite d'avoir un *short* au sommet de la pile. Pour le deuxième programme,LLa pré-condition de l'instruction *s2b* n'est donc pas respectée. En effet, *Top* ne peut pas être considéré comme un *short* selon l'arbre d'héritage de Java (selon le treillis pour être exact). La vérification s'arrête et le programme n'est pas accepté par le vérifieur de byte code. Le terme *rejet* représente le fait que la vérification s'arrête et que les types ne sont plus calculés.

2.2.3 Test de vulnérabilité

Pour le vérifieur de byte code, un test de vulnérabilité consistera à s'assurer que le byte code du programme ne respectant pas la spécification est rejeté. Nous pouvons distinguer les tests pour la partie vérification de structure et la partie vérification de type. Dans ce mémoire nous nous sommes consacrés au deuxième cas. Le programme donné dans les tableaux 2.2 et 2.4 sont donc deux exemples de tests de vulnérabilité

pour la partie vérification de type.

2.3 Test de logiciel

Tout comme il existe différentes façons de développer un logiciel, il en existe de nombreuses permettant de tester un logiciel. Il faut tout d'abord bien cibler quel type de problème on veut tester et ensuite utiliser les méthodes correspondantes.

L'obtention d'un jeu de tests comporte trois étapes principales :

- l'élaboration des tests : cette étape consiste à choisir la forme de test (*c.f.* section 2.3.1) ainsi que sa mise en application (*c.f.* section 2.3.2),
- l'exécution des tests : cela consiste à exécuter les tests sur le système que l'on souhaite tester,
- l'interprétation des résultats d'un test : la réponse d'un système à un test peut diverger de la solution attendue. Dans ce cas, il faut pouvoir expliquer la raison de cette différence comportementale. Cette étape comporte aussi une analyse des suites de tests obtenues permettant de vérifier des caractéristiques telles que la couverture ou la qualité du jeux de test.

2.3.1 Forme de test

Nous allons définir trois critères de test qui sont la phase de test, le type de tests et l'ensemble des valeurs à tester.

Phase de test

Le développement d'un logiciel se décompose généralement en différentes phases. Il va donc falloir déterminer la ou les phases sur lesquelles nous souhaitons effectuer des tests.

La première phase que nous pouvons tester se situe au niveau unitaire. Un logiciel est généralement constitué de différentes unités que nous pouvons tester séparément. Les tests unitaires permettent d'assurer le fonctionnement de chaque entité sans prendre en compte les communications. Ce type de tests est très flexible et peut

s'appliquer à une seule méthode ou voire à un d'objet. La personne écrivant ces tests devra définir ce qu'il considère comme étant une unité.

Les tests d'intégration correspondent à tester les comportements de différentes unités communiquant entre elles. Ils sont généralement utilisés après s'être assuré que chaque unité fonctionne correctement. Ils permettent de détecter des problèmes tels que des appels illégaux, le passage de paramètres hors des bornes acceptables.

Les tests systèmes visent à vérifier que le système global fonctionne correctement du point de vue de la cohérence fonctionnelle.

Finalement nous trouvons les tests de recette. Ceux-ci s'occupent de la vérification du cahier des charges. Les fonctionnalités demandées par l'utilisateur sont ainsi testées. Le cahier des charges apporte parfois des scénarios d'utilisation.

Type de tests

Différentes contraintes peuvent être demandées à un logiciel, par exemple une fonctionnalité ou l'absence de faille de sécurité. Les types de tests vont permettre de choisir quelle contrainte nous allons tester.

Le type de test le plus générique est le test de conformité. Ce type de test permet de s'assurer que la spécification a bien été respectée. Pour cela il se base sur des relations de conformité. Ces relations sont très flexibles et doivent être adaptées à chaque logiciel.

Les tests de robustesse vont assurer que le logiciel résistera à des utilisations non conformes qui ne sont généralement pas décrites dans la spécification. Du point de vue de la sureté, ce type de tests est essentiel. Cela permet notamment de s'assurer que le logiciel fonctionnera toujours et ne délivrera pas d'informations importantes, même après une utilisation non prévue.

Les tests en charge permettent de s'assurer que le logiciel assure toujours une cer-

taine qualité de services suite à une quantité de données ou d'opérations demandées. Ce type de test peut être utilisé pour tester les performances d'un logiciel ou bien pour assurer que la montée en charge suite à une utilisation intensive reste convenable.

Finalement les tests de régression assurent le maintien des fonctionnalités après mise à jour. On s'assure ainsi que les modifications ne doivent pas perturber certaines fonctionnalités de base sur lesquelles reposent les logiciels.

Sélection des valeurs

Les espaces d'état à tester sont généralement trop grands pour pouvoir entièrement les parcourir, il faut donc restreindre ces différents espaces. Les erreurs de développement se situent généralement dans une sous-partie correspondant à chaque problème. Cette connaissance nous permettra de mieux définir quel sous-ensemble est à tester.

Tout d'abord nous pouvons simplement prendre toutes les valeurs d'un ensemble. Bien entendu il faut que celui-ci ne soit pas trop grand et qu'il soit vraiment nécessaire de tout tester.

Les tests combinatoires vont générer encore plus de possibilités en effectuant la combinatoire de tous les espaces d'état.

Les tests aléatoires prennent des valeurs au hasard pour constituer des tests. Cela permet de ne pas toujours tester avec les mêmes valeurs.

Les tests aux bornes, ou classe d'équivalence, utilisent des valeurs situées aux bornes des espaces d'état. Si on considère que la majorité des erreurs d'implémentation se situent aux valeurs limites des ensembles, ce type de test est très intéressant et permet de rapidement trouver des erreurs. En plus des bornes, il est généralement intéressant de tester certaines valeurs comme le passage des nombres positifs aux nombres négatifs.

2.3.2 Techniques de mise en pratique

Tout comme il existe différentes formes de tests, il existe différentes façons de les mettre en application pour un logiciel donné. Ces techniques sont cependant similaires

et peuvent être représentées comme sur la figure 2.6.

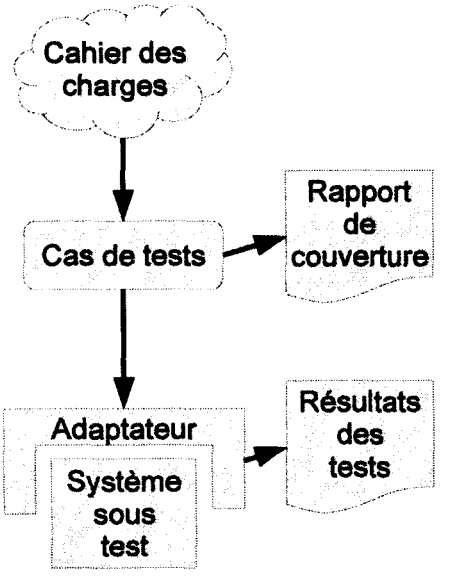

figure 2.6 – Schéma général de test

La première étape va consister, à partir de la spécification, à obtenir un ensemble de cas de tests. Dans cet ensemble nous avons une représentation abstraite des tests. La deuxième étape va donc consister à concrétiser ces tests pour qu'ils puissent être compris par le système sous test. Il est parfois nécessaire de construire un outil permettant d'exécuter ce jeu de test concret.

Test classique

Les tests classiques peuvent être effectués plus ou moins manuellement.

La première possibilité consiste à effectuer toutes les étapes manuellement. Un développeur lit le cahier des charges pour en extraire une suite de tests abstraits (nommé *cas de tests* sur la figure 2.6). De celle-ci, il est rare de pouvoir extraire facilement un rapport de couverture mais au besoin il peut être obtenu manuellement. Ces tests abstraits sont ensuite concrétisés et exécutés. Les réponses du système sont bien entendu automatiques. Finalement les résultats des tests sont décrits manuellement.

Ce type de test a l'avantage d'être très flexible et permet de cibler très précisément certains tests. Cependant, il prend beaucoup de temps car tous ces processus se font manuellement. De plus on ne peut pas réellement garantir la qualité du jeu de test ni des résultats obtenus.

La seconde possibilité consiste à utiliser des scripts afin d'automatiser certaines étapes. La première étape, l'obtention des cas de tests, est un processus qui ne peut s'automatiser. À partir des tests abstraits il est possible de générer automatiquement des tests concrets ainsi que leur exécution sur le système sous test.

Test à base de modèle formel

L'utilisation de modèles formels tente d'automatiser une plus grande partie du processus de tests tout en apportant plus de confiance sur la qualité de celui-ci. La figure 2.7 représente les différentes étapes composant le test à base de modèle formel.

La lecture de la spécification est non automatisable. Cependant, plutôt que de rédiger directement les cas de tests, un modèle formel va être rédigé. En plus de la spécification, un plan de test est nécessaire afin de produire un modèle permettant la génération de tests. L'utilisation d'un modèle formel permet de prouver que le cahier des charges est bien respecté.

Les cas de tests vont être extraits de ce modèle formel. Ce processus est semi-automatique. Un utilisateur va imposer un certain nombre de contraintes. Ces contraintes permettent de choisir quel type de test il souhaite obtenir. Un logiciel va ensuite automatiquement générer les cas de tests voulus ainsi que le rapport de couverture. Comme tous ces processus reposent sur le modèle formel, la qualité des tests obtenue est plus facilement contrôlable.

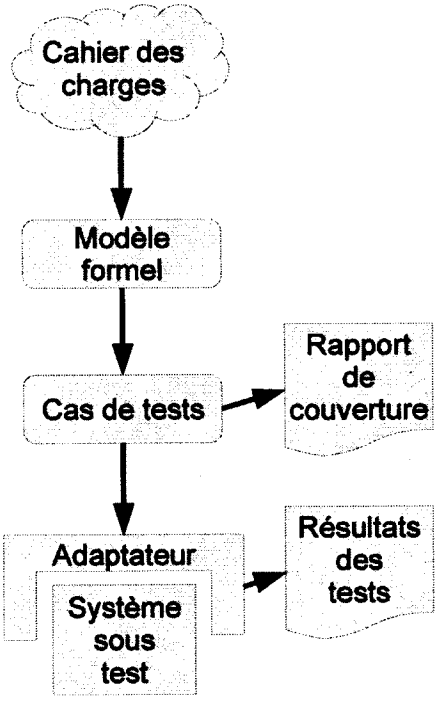

figure 2.7 – Schéma : Test à base de modèle formel

2.4 Méthodes formelles

L'un des principaux objectifs des méthodes formelles est de pouvoir raisonner sur un modèle qui ne comporte pas d'ambiguïté, *i.e.* il ne peut pas y avoir plusieurs interprétations possibles. Cette représentation se base sur des notations mathématiques permettant une manipulation simple des modèles.

Dans cette section, nous décrirons simplement la partie des méthodes formelles que nous avons utilisée dans le cadre de nos recherches.

2.4.1 Contraintes et choix d'outils

Il existe différents types de systèmes et pour chacun différentes façons de les modéliser. Dans le cadre de nos recherches, nous avions différentes contraintes. Tout d'abord, la preuve de modèle était indispensable afin de garantir la validité de nos modèles. Nous devions aussi pouvoir utiliser un vérificateur de modèle (*c.f.* Section 2.4.4) afin d'extraire des traces vérifiant une certaine propriété. De plus, il était intéressant, mais non obligatoire, de pouvoir utiliser les formules LTL (Linear Temporal Logic) 2.4.4, afin de garantir la qualité des traces obtenues. La manipulation des modèles à l'aide de programmes ou d'API, permettant d'automatiser notre approche, était très importante.

Nous avons décidé de prendre comme langage de modélisation Event-B et comme vérificateur de modèle ProB que nous décrirons respectivement dans les sections 2.4.2 et 2.4.4.

2.4.2 Le langage Event-B

Le langage event-B est un langage de modélisation formelle qui est apparu très récemment. Pour la représentation des états, il se base sur la théorie des ensembles. La représentation des différents niveaux d'abstraction du système est contrôlée à l'aide de raffinements prouvés.

Tous les exemples de ce mémoire se réfèreront au modèle décrit dans l'annexe A. Ce modèle représentant toutes les contraintes que nous avons à prendre en compte dans le cadre de ces recherches.

Modèle

Nous ne ferons ici qu'une rapide description du langage Event-B. Pour une meilleur description le lecteur peut se référer au livre *Modeling in Event-B : System and Software Engineering* écrits par Abrial [2], au guide d'utilisation de Rodin [9], aux notations mathématique [29] ou encore à la grammaire du langage [5] . Un modèle Event-B est constitué de deux parties. Tout d'abord, nous avons les contextes dans lesquels sont décrits les constantes et les axiomes. Nous avons ensuite les machines, constituées des variables, des invariants et des événements. Le contexte représente la partie statique du modèle tandis que la machine représente la partie dynamique.

Un modèle peut être constitué de contextes seuls, de machines seules ou bien des deux. Entre les machines et les contextes nous pouvons mettre certaines relations, dont une représentation est donnée en figure 2.8. Tout d'abord nous avons la relation *sees* permettant de lier un ou plusieurs contextes à une machine. Puis nous avons la relation entre deux machines *refines* qui permet de représenter la notion de raffinement. Un modèle ne peut raffiner qu'un seul autre modèle. Finalement nous avons la relation *extends* permettant à un contexte d'étendre un ou plusieurs autres contextes. Ces relations permettent de définir différents niveaux dans nos raffinements et un ensemble d'obligations de preuves est automatiquement généré afin d'assurer la cohérence entre ces différents composants.

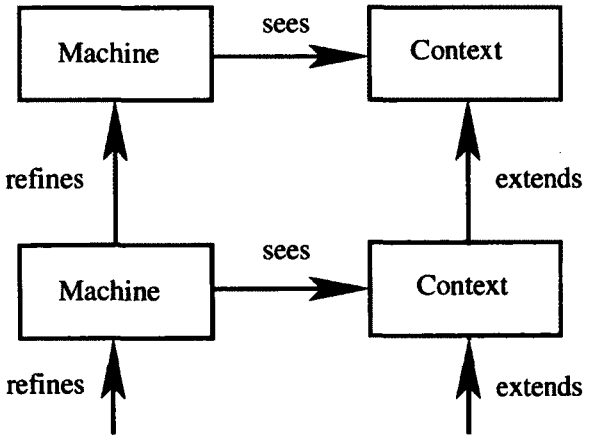

figure 2.8 – Relations entre machines et contextes ([2] p177)

Un exemple de contextes est donné en figure 2.9. Dans cet exemple, nous voyons que nous avons déclaré une constante c et une constante d. Puis nous avons les axiomes $axm1$ et $axm2$ permettant d'imposer une contrainte respectivement sur la constante c et d. Les axiomes sont des prédicats du 1er ordre et ceux-ci doivent être vrais pour pouvoir instancier un contexte. Dans notre exemple, nous voyons qu'il est simple de trouver une valeur pour c tel que l'axiome soit vrai, nous affectons c à la valeur 42. Pour la constante d, plusieurs solutions existent et nous pouvons prendre n'importe laquelle des valeurs entières comprises entre -20 et 50. En Event-B l'ensemble $[-20, 50]$ est noté -20..50

CONTEXT C
CONSTANTS

c

d

AXIOMS

axm1 : $c = 42$

axm2 : $d \in -20 \mathinner{..} 50$

END

figure 2.9 – Exemple de contexte Event-B

Prenons maintenant l'exemple de la machine donnée en figure 2.10 (extrait du modèle de l'annexe A). Dans cette machine, nous avons la variable a. Pour cette variable, nous devons vérifier qu'elle appartient toujours à l'intervalle $[-5, 5]$ ($-5 \mathinner{..} 5$ en notation Event-B). Cela est représenté par l'invariant $inv1$. Tout comme les axiomes, les invariants sont des prédicats du 1er ordre. Une machine est ensuite constituée de différents événements. Le premier événement que l'on trouve obligatoirement correspond à l'initialisation de la machine. Au démarrage de celle-ci, un seul événement peut survenir, cela correspond à l'affectation de valeurs aux différentes variables. Une fois cet événement exécuté il ne pourra plus jamais se ré-exécuter. Les événements Event-B sont représentés par une garde et une action. Une garde est un prédicat qui, s'il est vrai, autorise l'exécution d'un événement et sinon l'interdit. Ce comportement est similaire à une pré-condition, à la différence que dans le cas où une pré-condition est fausse, l'évènement se produit tout de même mais le résultat ne peut pas être

garanti. Les actions vont mettre à jour la valeur de certaines variables.

MACHINE M
VARIABLES
a
INVARIANTS
inv1 : $a \in -5 \mathinner{..} 5$
EVENTS
Initialisation
begin
act1 : $a := 3$
end
Event *add* $\widehat{=}$
when
grd1 : $a < 5$
then
act1 : $a := a + 1$
end
Event *suppr* $\widehat{=}$
when
grd1 : $a - 1 \geq 0$
then
act1 : $a := a - 1$
end
END

figure 2.10 – Exemple de machine Event-B

Plateforme Rodin

La plateforme Rodin, soutenue par le projet européen Deploy [8], met à disposition un ensemble d'outils permettant de manipuler des modèles Event-B. Rodin est basé sur la plateforme Eclipse et permet ainsi de rajouter simplement des nouvelles

fonctionnalités grâce à des extensions. Parmi ces extensions, nous avons notamment des prouveurs qui vont permettre automatiquement ou interactivement de prouver nos modèles. De plus, la mise à disposition d'une API en Java pour la manipulation des modèles nous permet de développer nos propres extensions.

2.4.3 Preuve de modèle

La preuve de modèle permet de garantir que la spécification est bien respectée. Comme nous l'avons dit précédemment, les méthodes formelles reposent sur des notions mathématiques que l'on peut simplement manipuler. Afin de garantir qu'un modèle respecte bien une spécification, un ensemble de propriétés doit être exprimé formellement puis vérifié sur le modèle. Dans le cas du langage Event-B, les contraintes de la spécification sont représentées à l'aide des invariants.

Dans notre exemple, nous avons les trois événements *Initialisation*, *add* et *suppr*. L'événement *add* va incrémenter la valeur de a tant que celle-ci est inférieur à 5, tandis que l'événement *suppr* va la décrémenter tant que a est supérieur à 0. Afin de garantir que le système respecte bien la spécification, un certain nombre de preuves doivent être faites. Les contraintes imposées par la spécification sont représentées grâce aux invariants. Dans notre cas, nous voulons que la variable a appartienne toujours à l'intervalle $-5\,..\,5$. Pour ce faire, nous allons prouver que chaque événement respecte bien cette contrainte. Pour cela, il faut vérifier qu'après chacune de leur exécution l'invariant est toujours vrai. Dans la suite, nous proposons une rapide explication du processus de preuve. Pour plus de précisions le lecteur peut se référer au chapitre 9 de [2].

Pour l'événement *Initialisation*, nous devons vérifier que $a \in [-5, 5]$ pour $a = 3$. Nous devons donc prouver que $3 \in [-5, 5]$. Cette propriété est vraie. L'événement *Initialisation* respecte donc l'invariant $inv1$.

Pour l'événement *add*, nous devons vérifier que si $a < 5$ alors si on effectue $a' = a+1$, nous aurons toujours $a' \in [-5, 5]$. Nous avons ici introduit la nouvelle variable a' représentant a mais après modification. Pour cela, nous allons décomposer la preuve en deux parties, une pour $a' \geq -5$ et une pour $a' \leq 5$. Pour la deuxième partie,

nous utilisons la règle de l'incrément : $n < m \vdash n + 1 \leq m$. Pour la première partie, nous devons considérer que l'événement sera exécuté dans le cas où les invariants sont respectés. Nous avons donc la propriété $a > -5$ et il devient simple de prouver que $a' \geq -5$. Nous procédons de même avec l'événement *suppr* en rajoutant la propriété de décrément.

Nous avons prouvé tous les invariants pour tous nos événements. Nous pouvons maintenant être sûr que les propriétés que nous voulions apporter à notre modèle sont respectées.

2.4.4 Vérificateur de modèle ProB

Certaines propriétés sont à l'heure actuelle relativement difficile à prouver en utilisant la méthode décrite précédemment. Cela est notamment le cas des propriétés sur les traces pour lesquelles nous devons considérer le modèle de façon dynamique. Pour résoudre ces problèmes, la vérification de modèles (*model checking* en anglais) est une technique offrant de nouvelles possibilités.

Dans le cas de Event-B, une trace est une suite d'événements de longueur finie et commençant par l'événement *Initialisation*. Pour les extraire, nous avons utilisé le logiciel ProB [7]. Ce dernier, est capable d'interpréter des modèles formels dans différents langages tels que Z, B ou encore Event-B. De plus, il dispose de différents moyens d'extraire des traces. Parmi ceux-ci nous trouvons l'utilisation de formules LTL qui représentait un point intéressant dans nos recherches.

LTL

La logique temporelle, décrite dans le chapitre 5 de [10], permet de vérifier un certain nombre de propriétés sur un modèle pouvant être dynamique. Pour cela, on ajoute aux connecteurs de la logique classique un ensemble de connecteurs temporel. Dans le cas de la LTL, nous avons les formules suivantes :

- X : demain ou immédiatement après (neXt)
- F : un jour (Finally)
- G : toujours (Globally)
- U : jusqu'à (Until)

- W : à mois que (Unless)
- R : décharge (Release)

Grâce à ces formules, nous pouvons exprimer des contraintes sur le passé, le présent et le futur. Par exemple, une propriété dynamique pourrait être : je vais finir par mourir. Dans ce cas là, on va utiliser la formule F permettant de dire que la propriété, être mort, n'est pas toujours vraie mais qu'elle le deviendra forcément à un moment donné : $F(mort)$.

Précédemment nous avons parlé des invariants qui peuvent être vu comme des propriétés statiques. Une formule statique p serait équivalente à $G(p)$. En effet, dans le cas des propriétés statiques, la propriété que l'on cherche à satisfaire doit *toujours* (G) être vraie. Cependant si on trouve un contre-exemple, celui-ci sera donné sous forme d'une trace.

2.4.5 Test de vulnérabilité

Dans le cas des méthodes formelles, un test de vulnérabilité correspond à une trace contenant au moins un événement en faute. La trace est obtenue par utilisation d'un vérificateur de modèle. Lors de cette vérification, nous allons imposer d'exécuter au moins une fois un événement en faute. Un événement en faute est une opération dont la garde est fausse mais que l'on exécute tout de même.

Dans notre modèle donné en figure 2.10 (schéma en Annexe A), nous pouvons par exemple extraire comme trace valide : [*INITIALISATION; add; add; suppr*]. Dans le cas d'un test de vulnérabilité, nous pouvons extraire une trace contenant l'événement en faute *add* : [*INITIALISATION; add; add;* ***add***].

Chapitre 3

Approche

L'idée générale consiste à générer des modèles dérivés à partir d'un modèle initial puis à extraire les tests abstraits et finir par les concrétiser. Chacun de ces modèles dérivés représentera une faute particulière. Pour chacun de ces modèles, il va falloir chercher si la faute peut être générée. Pour cela, nous générons des tests de vulnérabilité abstraits que nous concrétisons par la suite.

Nous ne cherchons pas ici à prouver que le modèle est valide ou encore que l'implémentation raffine bien le modèle. Nous voulons générer une suite de tests à l'aide d'un modèle formel nous permettant de tester une implémentation.

Pour ce faire, nous avons développé l'Algorithme 1. Tous les exemples utiliseront le modèle donné en annexe A.

3.1 Critères de tests

Commençons par définir ce qu'est un test de vulnérabilité :

Test de vulnérabilité 1 *Un test de vulnérabilité est un test qui permet de s'assurer qu'un comportement rejeté par le modèle l'est aussi par son implémentation.*

Pour nous, un test sera une trace, *i.e.* une séquence finie d'événements. Dans notre modèle d'exemple, un TdeV (Test de Vulnérabilité) pourrait correspondre à la séquence suivante : [*INITIALISATION; add; add; **add**; suppr; suppr; suppr; halt*].

Input: m : *Event-B* model
Output: M' : set of *Event-B* model
for *each event e of m* **do**
  rewrite the guard of e into two guards :
    grd, the conjunction of all guards of e without suffix "_t";
    grd_t, the conjunction of all guards of e with suffix "_t";
end
for *each event e of m* **do**
  $e.RW := neg(grd\_t)$;
end
for *each event e of m* **do**
  for *each rw in e.RW* **do**
    add a new model m' to M' such that
      $m' := m$;
      $m'.events := m'.events \cup \{e'\}$, where e' is defined as follows :
        $e' := e$;
        replace $e'.grd\_t$ by rw;
        add guard "`eut = FALSE`" to e';
        add action "`eut := TRUE`" to e';
  end
end

Algorithm 1: Dérivation de modèle

Nous considérons que chaque test commencera avec une plateforme toujours dans le même état, *i.e.* il n'y a pas de répercussion des tests précédents sur le test courant. Afin d'identifier le plus précisément possible les erreurs d'implémentation, nous ne ferons qu'une seule faute par test. Dans chaque test, nous autoriserons l'exécution de l'événement non fautif ainsi que l'événement fautif. De plus, nous voulons obtenir des traces comportant la faute à n'importe quelle position, ce qui signifie que nous devons pouvoir trouver une trace conduisant à une faute ainsi qu'une trace continuant après exécution d'une faute.

Dans l'exemple de TdeV précédent, l'événement *add* est utilisé dans une transition valide ainsi que dans une transition invalide et la trace poursuit après exécution de la faute.

3.2 Négation des gardes

Le modèle initial représente l'ensemble des comportements acceptables pour le système sous test. Un comportement acceptable est une suite d'actions pour lesquels nous considérons que le système autorise chacune des ces actions. Dans le cas de Java Card un comportement acceptable est l'acceptation par le JCBCV d'un programme, chaque action étant une instruction. Les gardes des événements sont donc les pré-condition défini dans la spécification [36].

La première étape consiste à générer les modèles dérivés. Chacun de ces modèles nous permettra de tester précisément une seule faille. Pour cela, nous allons effectuer une négation de certaines pré-conditions (gardes dans le cas du langage Event-B) de notre modèle initial.

Un événement peut uniquement survenir si sa garde est satisfaite. Par opposition, la négation de sa garde représente tous les contextes dans lesquels un événement ne peut pas survenir. Nous constatons que si une garde est correctement définie au regard de la spécification, alors il est possible de générer l'ensemble des cas pour lesquels un événement n'est pas autorisé à survenir.

Prenons, par exemple, l'événement *suppr*, il ne peut s'exécuter que si $a - 1 \geq 0$. Il ne devra donc pas pouvoir, dans l'implémentation, s'exécuter lorsque $a - 1 < 0$ ou encore lorsque $\neg(a - 1 \geq 0)$. Si on note *suppr_FAULT_1* l'événement donné en figure 3.1, nous voyons que cet événement pourra s'exécuter lorsque l'événement *suppr* ne le pourra pas.

Event *suppr_FAULT_1* $\widehat{=}$
  when
    grd1_t : $\neg(a - 1 \geq 0)$
    grd2 : $halt = FALSE$
  then
    act1 : $a := a - 1$
  end

figure 3.1 – Événement *suppr_FAULT_1*

Pour un événement donné, nous ne voulons pas toujours tester toutes les parties

de sa garde. Pour cela, nous utilisons des marqueurs permettant de savoir quelles parties de gardes nous devons tester et quelles parties de gardes nous ne devons pas tester. Pour le moment ce marqueur se présente sous la forme d'une convention sur le nommage des parties de grades. Si le nom de la partie de garde fini par $\_t$, cela signifie que nous désirons la tester.

En suivant ce schéma, il est simple de trouver toutes les fautes que l'on peut faire dans le modèle. Cependant, nous constatons que l'on peut très rapidement se trouver face à un problème d'explosion combinatoire. Par exemple, la négation d'une égalité est problématique car elle nous génère un espace de possibilité grand (infini dans le cas des entiers naturels). Il faut donc améliorer cette première idée afin de la rendre applicable.

3.3 Règles de dérivation et règles de ré-écriture

Nous proposons une solution basée sur l'utilisation de règles de réécriture. Considérons l'ensemble de règles suivantes où $p_1...p_n$ sont des prédicats, $b_1...b_n$ sont des variables booléennes, $n_1...n_n$ sont des variables de type entier naturel. La fonction *neg* représente l'application des règles à un prédicat :

- $neg(p_1 \wedge p_2) \rightsquigarrow \{neg(p_1) \wedge p_2,\ p_1 \wedge neg(p_2),\ neg(p_1) \wedge neg(p_2)\}$.
- $neg(p_1 \vee p_2) \rightsquigarrow \{neg(p_1) \wedge neg(p_2)\}$.

- $neg(n_1 = n_2) \rightsquigarrow \{n_1 < n_2,\ n_1 > n_2\}$
- $neg(n_1 > n_2) \rightsquigarrow \{n_1 = n_2,\ n_1 < n_2\}$
- $neg(n_1 < n_2) \rightsquigarrow \{n_1 = n_2,\ n_1 > n_2\}$
- $neg(n_1 \geq n_2) \rightsquigarrow \{n_1 < n_2\}$
- $neg(n_1 \leq n_2) \rightsquigarrow \{n_1 > n_2\}$
- $neg(n_1 \neq n_2) \rightsquigarrow \{n_1 = n_2\}$

- $neg(b_1 = b_2) \rightsquigarrow \{\neg(b_1 = b_2)\}$
- $neg(b_1 \neq b_2) \rightsquigarrow \{b_1 = b_2\}$

Ces règles s'appliquent de façon récursive par un parcours de l'arbre syntaxique du prédicat. Prenons l'exemple de l'événement *supprTwo*. La partie de sa garde qui est à tester est la suivante : $a \neq 1 \wedge a - 2 \geq 0$.

Event *supprTwo* $\widehat{=}$

when

grd1_t : $a \neq 1$

grd2_t : $a - 2 \geq 0$

grd3 : $halt = FALSE$

then

act1 : $a := a - 2$

end

figure 3.2 – Événement *supprTwo*

Nous allons dans ce cas procéder en deux appels récursifs aux règles de réécriture.

$1^{ère}$ itération (règle du $\wedge$ logique) :

$neg(a \neq 1 \wedge a - 2 \geq 0) \rightsquigarrow \{$
$neg(a \neq 1) \wedge \quad (a - 2 \geq 0),$
$(a \neq 1) \wedge neg(a - 2 \geq 0),$
$neg(a \neq 1) \wedge neg(a - 2 \geq 0)$
$\}$

$2^{ème}$ itération (règles du $\neq$ et du $\geq$) :

$neg(a \neq 1) \wedge \quad (a - 2 \geq 0) \rightsquigarrow \{(a = 1) \wedge (a - 2 \geq 0)\}$
$(a \neq 1) \wedge neg(a - 2 \geq 0) \rightsquigarrow \{(a \neq 1) \wedge (a - 2 < 0)\}$
$neg(a \neq 1) \wedge neg(a - 2 \geq 0) \rightsquigarrow \{(a = 1) \wedge (a - 2 < 0)\}$

Au final nous obtenons donc les ré-écritures suivantes :

$neg(a \neq 1 \wedge a - 2 \geq 0) \rightsquigarrow \{$
$(a = 1) \wedge (a - 2 \geq 0),$
$(a \neq 1) \wedge (a - 2 < 0),$
$(a = 1) \wedge (a - 2 < 0)$
$\}$

Grâce à cette règle de ré-écriture, nous allons pouvoir créer trois nouveaux événements

à partir de l'événement *supprTwo*. Chacun de ces modèles correspondra à une faute particulière. Les trois modèles sont donnés dans les figures 3.3, 3.4 et 3.5

Event *supprTwo_FAULT_1* $\widehat{=}$
when
grd_t : $(a = 1) \wedge (a - 2 \geq 0)$
grd : $halt = FALSE$
then
act1 : $a := a - 2$
end

figure 3.3 – Événement *supprTwo_FAULT_1*

Event *supprTwo_FAULT_2* $\widehat{=}$
when
grd_t : $(a \neq 1) \wedge (a - 2 < 0)$
grd : $halt = FALSE$
then
act1 : $a := a - 2$
end

figure 3.4 – Événement *supprTwo_FAULT_2*

Event *supprTwo_FAULT_3* $\widehat{=}$
when
grd_t : $(a = 1) \wedge (a - 2 < 0)$
grd : $halt = FALSE$
then
act1 : $a := a - 2$
end

figure 3.5 – Événement *supprTwo_FAULT_3*

Pour les autres événements, *suppr* et *add*, les ré-écritures sont triviales. Voilà une liste des différents prédicats que l'on peut trouver dans les gardes ainsi que leur négation :

- $neg(a < 5) \rightsquigarrow \{a = 5, a > 5\}$
- $neg(a - 1 \geq 0) \rightsquigarrow \{a - 1 < 0\}$

Si on analyse le résultat de l'application des règles de réécriture, nous pouvons constater trois choses. Tout d'abord, certains prédicats sont trivialement vrais et immédiatement instanciables, exemple : $a = 5$. Ensuite, certains prédicats sont toujours faux, exemple : $(a = 1) \wedge (a - 2 \geq 0)$. Enfin, certains prédicats ne sont pas trivialement instanciables et demanderont une recherche, exemple : $a > 5$. Dans ce dernier cas, comme nous travaillons dans des espaces de taille infinie, nous ne sommes pas sûr de pouvoir trouver une solution.

3.4 Duplication d'événement

La façon de modéliser un événement peut conduire à différents événements dérivés. Ces derniers, représentant dans notre cas des failles de sécurité que on souhaite tester, doivent fidèlement représenter la réalité. Pour illustrer ceci prenons les deux événements *nop* donnés en figures 3.6 et 3.7.

Event *nop1* $\widehat{=}$
when
grd1_t : $a = -4$
then
act1 : $a := a$
end

figure 3.6 – Événement *nop1*

Deux modélisations différentes de l'événement *nop* ont été faites. Les gardes de ces deux événements étant identiques, leur négation sera donc la même. La différence se situe dans les actions. Dans le 1^{er} événement, la valeur de la variable a est affectée

Event *nop2* $\widehat{=}$
when
grd1_t : $a = -4$
then
act1 : $a := -4$
end

figure 3.7 – Événement *nop2*

à sa propre valeur. Lorsque nous aurons nié sa garde, la valeur de a changera donc à chaque exécution de cet événement. Dans le 2^{nd} événement, la variable a est toujours affectée à -4. Lorsque nous aurons nié sa garde, son exécution dans n'importe quel contexte nous obtiendrons $a = -4$. Nous voyons que deux modèles équivalent peuvent entrainer des cas de test très différents. Comme nous ne savons pas comment a été implémenté la spécification, nous devons garder ces deux modélisations pour être certain de couvrir un maximum de cas.

3.5 Extraction des tests abstraits

Maintenant que nous savons comment générer les modèles des fautes, il faut en extraire les tests. Un test classique est composé de quatre parties : le préambule, le corps, la vérification et le postambule. Dans notre cas, nous ne nous intéresserons pas à la partie vérification. En effet, nous ne générons que des tests qui devront être rejetés par le système. La réponse de l'échec d'un test sera donc donnée lors de l'exécution. De plus, le corps se restreint à une exécution simple de la faute dans le modèle dérivé. Bien que nous décrivons dans la suite comment obtenir les postambules, il n'est pas toujours nécessaire de les générer et notre solution sera la plus flexible possible afin de s'adapter à un grand nombre de problèmes.

Afin d'obtenir les préambules et les postambules, nous allons utiliser pour cela un vérificateur de modèle (model checker) qui nous permettra d'obtenir des séquences contenant la faute. Différentes techniques existent et nous allons décrire les deux que nous avons utilisées.

3.5.1 Première approche

Dans une première approche, nous avons tenté d'obtenir des traces les plus précises possibles. Pour cela nous avons utilisé des algorithmes basés sur la vérification de formules LTL. Ces algorithmes sont généralement utilisés afin de trouver, dans un modèle formel, des contre-exemples sur des propriétés temporelles. Dans notre cas, nous les avons utilisés afin d'obtenir les préambules et les postambules. Pour cela, nous commençons par poser comme propriété qu'il est impossible d'exécuter la faute. Si le programme nous trouve un contre-exemple, cela signifie que notre faute peut être atteinte et le programme nous donne explicitement comment attendre la faute. De même pour les postambules en posant qu'il est impossible d'atteindre un état final nous pouvons obtenir une trace.

Dans le cas où nous n'avons pas besoin dans nos tests finaux de postambules, nous avons simplement à extraire les préambules. Dans le cas où nous avons besoin des postambules, nous devons impérativement effectuer une et une seule recherche permettant de trouver le préambule et le postambule en même temps. Si nous ne faisons pas cette recherche en une seule fois, les préambules que nous trouverons risquent de ne pas pouvoir engendrer de solutions pour la recherche de postambule.

Reprenons notre modèle d'exemple. L'événement en faute pour cet exemple sera l'événement *add* dont la dérivation est donnée en figure 3.8.

Event $add\_FAULT\_1 \triangleq$

when

grd_t : $a = 5$

grd : $a \geq 0 \wedge halt = FALSE$

grd_fault : $fault = FALSE$

then

act1 : $a := a + 1$

act_fault : $fault := TRUE$

end

figure 3.8 – Événement *add_FAULT_1*

Pour obtenir le préambule, nous allons utiliser la formule LTL suivante : $G(not(fault =$

$TRUE$)). Voici deux contre-exemples que nous obtenons :

- [*INITIALISATION; add; add;* ***add***]
- [*INITIALISATION; add; opposite; opposite; add;* ***add***]

Ces deux traces correspondent à deux tests abstraits que nous allons pouvoir ensuite concrétiser.

Si nous voulons obtenir un préambule et un postambule, nous devons le rajouter dans notre formule LTL. Dans notre exemple, l'état terminal est représenté par la variable *halt*. Nous allons utiliser la formule LTL suivante : $G(not(faut = TRUE \ \& \ halt = TRUE))$. Voici deux contre-exemples que nous obtenons :

- [*INITIALISATION; add; add;* ***add***; *suppr; supprTwo; halt*]
- [*INITIALISATION; add; add;* ***add***; *suppr; suppr; suppr; halt*]

Grâce à cette méthode, nous voyons que nous allons pouvoir générer des traces de façon très précise. En effet, l'utilisation de formules LTL autorise une très grande flexibilité quant aux propriétés que l'on souhaite obtenir dans nos tests abstraits. Cependant, les outils que nous utilisons permettent d'obtenir qu'un seul contre-exemple donc qu'une seule trace.

3.5.2 Deuxième approche

Dans une deuxième approche, nous avons tenté d'obtenir un plus grand nombre de tests. Pour cela, nous utilisons des algorithmes de génération de tests à base de modèle formel. Ces derniers ne permettent que d'extraire des traces valides, cependant nous avons généré des modèles dérivés contenant les fautes. Ainsi, les traces valides que nous obtenons sont valides au regard du modèle dérivé. Afin d'être sûr de générer uniquement les traces contenant des fautes, nous allons légèrement contraindre le générateur de tests. Dans nos traces, nous imposerons que $fault = TRUE$. Dans le cas où nous voulons aussi obtenir un postambule, nous imposerons aussi la propriété représentant la fin d'une trace. Nous allons aussi contraindre à ce que les traces soient générées dans l'ordre croissant de longueur et en limitant leur longueur maximum.

Reprenons notre modèle d'exemple. L'événement en faute pour cet exemple sera l'événement *nop1* dont une dérivation est donnée en figure 3.9.

Pour obtenir des préambules, voici la liste des possibilités pour une longueur maxi-

Event *nop1_FAULT_2* $\widehat{=}$

when

grd_t : $a > -4$

grd_fault : $fault = FALSE$

then

act1 : $a := a$

act_fault : $fault := TRUE$

end

figure 3.9 – Événement *nop1_FAULT_2*

mum de trois :

- Longueur 0 : []
- Longueur 1 : [] (Ici nous ne pouvons que simplement initialiser la machine)
- Longueur 2 :
  [*INITIALISATION*; ***nop1_FAULT_2***]
- Longueur 3 :
  [*INITIALISATION*; *add*; ***nop1_FAULT_2***] ;
  [*INITIALISATION*; *suppr*; ***nop1_FAULT_2***] ;
  [*INITIALISATION*; *supprTwo*; ***nop1_FAULT_2***]

Dans le cas de traces contenant un préambule et un postambule, nous rajoutons simplement la propriété suivante : $halt = TRUE$. Voici un exemple de traces que nous obtenons :

[*INITIALISATION*; ***nop1_FAULT_2***; *halt*];
[*INITIALISATION*; *add*; ***nop1_FAULT_2***; *suppr*; *halt*] ;
[*INITIALISATION*; *add*; *suppr*; ***nop1_FAULT_2***; *halt*] ;
[*INITIALISATION*; *suppr*; ***nop1_FAULT_2***; *add*; *halt*] ;
[*INITIALISATION*; *suppr*; *add*; ***nop1_FAULT_2***; *halt*] ;
[*INITIALISATION*; *opposite*; *opposite*; ***nop1_FAULT_2***; *halt*]

3.5.3 Comparaison des deux approches

Ces recherches étant une preuve de concepte, nous avons essayé d'utiliser les deux approches afin de déterminer laquelle était la plus efficace. Comme nous l'avons vu, les deux méthodes donnent pratiquement les mêmes résultats. Leurs performances sont de plus relativement équivalentes. La différence que nous pouvons relever se situe au niveau de leur utilisation dans un logiciel. La première est beaucoup plus simple à implanter dans un outil que la seconde. Pour la suite de ces recherches, correspondant au doctorat et donc non décrites dans ce mémoire, nous avons choisi d'utilisé uniquement la première approche.

Chapitre 4

Application au langage de Freund et Mitchell

Afin de valider notre approche de génération de tests de vulnérabilité pour la plateforme Java Card, nous avons commencé par travailler avec le langage de Freund et Mitchell [22]. Afin de bien comprendre ce sous-langage nous en donnerons une première définition informelle puis nous détaillerons comment nous l'avons formalisée. Nous donnerons ensuite un exemple de génération de tests de vulnérabilité. Le modèle complet est donné en annexe B.

4.1 Spécification informelle

Le sous-ensemble que nous avons utilisé s'appelle JVML_i. Il a été défini dans le but d'offrir un langage possédant peu d'instructions mais représentant toutes les caractéristiques du langage Java.

Une spécification informelle de chacune de ces instructions est donnée dans les tableaux 4.1 et 4.2. Les pré-conditions doivent notamment permettre de s'assurer que :

- tous les chemins conduisent à une instruction *halt*.
- le programme ne dépasse pas une certaine longueur.

- on ne peut pas effectuer de dépassement de taille de pile : *stack overflow* et *stack underflow.*
- on lit et on écrit les variables locales dans leur domaine.
- le treillis de type est respecté.

4.2 Spécification formelle

Nous avons modélisé le langage $JVML_i$ à l'aide du langage Event-B. La définition des variables est donnée dans le figure 4.1.

Nous nous sommes inspiré des modèles *B* issus des travaux de L. Casset [20]. Cependant, notre modèle n'a pas la même fonction. Dans notre cas, le modèle doit permettre d'extraire des traces et non de proposer une implémentation sûre du vérifieur de byte code.

INVARIANTS

inv1 : $ss \in 1 \mathinner{..} maxpc \pfun (0 \mathinner{..} maxstack - 1 \pfun TYPE)$
inv2 : $vv \in 1 \mathinner{..} maxpc \pfun (1 \mathinner{..} maxlocalvar \pfun TYPE)$
inv3 : $zz \in 1 \mathinner{..} maxpc \pfun 0 \mathinner{..} maxstack$
inv4 : $halt \in BOOL$
inv5 : $dom(ss) = dom(vv) \land dom(ss) = dom(zz) \land dom(vv) = dom(zz)$
inv6 : $pc \in dom(ss)$

figure 4.1 – Invariants du modèle de $JVML_i$

La variable *pc* représente l'index de la prochaine instruction à générer. Cela correspond aussi au nombre d'événements moins un qui ont été exécutés mais également au nombre d'instructions moins une qui composent un test. Cette variable est bornée supérieurement par la constante *maxpc*, afin de garantir que notre recherche ne sera pas infinie. La variable *ss* représente le type des éléments présents dans la pile d'évaluation. Elle se comporte comme un tableau à deux dimensions et va représenter le contenu de la pile pour tout point du programme. Elle est, elle aussi, bornée par *maxpc* en largeur et la taille maximum des pile est bornée grâce à la constante *maxstack*. La constante $TYPE$ est un ensemble contenant les types *Int*, *Obj* et *Uobj* représentant respectivement les entiers naturels, les objets initialisés et les objets non

$push_0$	Charge l'entier 0 en sommet de pile. PRE $size(pile) < pilePleine$ POS $size(pile) + 1 \wedge Int$
inc	Manipule un entier en sommet de pile. PRE Int POS Int
pop	Supprime l'élément se trouvant en sommet de pile. PRE $size(pile) \neq 0$ POS $size(pile) - 1$
if_l	Effectue un saut à l'instruction suivante ou à l'instruction l si l'entier au sommet de pile est 0. PRE Int POS $size(pile) - 1$
$istore_x$	Supprime l'objet du type Int du sommet de la pile et le stocke dans la variable locale x. PRE $Int \wedge x \in dom(VarLoc)$ POS $size(pile) - 1 \wedge VarLoc_x \leftarrow pile(size(pile))$
$iload_x$	Charge l'objet de type Int de la variable locale x au sommet de la pile. PRE $size(pile) < pilePleine \wedge x \in domVarLoc \wedge VarLoc(x) = Int$ POS $size(pile) + 1 \wedge pile(size(pile)) \leftarrow VarLoc(x)$

tableau 4.1 – Instructions du $JVML_i$, première partie

$astore_x$ Supprime l'objet du type Obj du sommet de la pile et le stocke dans la variable locale x.
PRE $Obj \wedge x \in dom(VarLoc)$
POS $size(pile) - 1 \wedge VarLoc_x \leftarrow pile(size(pile))$

$aload_x$ Charge l'objet de type Obj de la variable locale x au sommet de la pile.
PRE $size(pile) < pilePleine \wedge x \in domVarLoc \wedge VarLoc(x) = Obj$
POS $size(pile) + 1 \wedge pile(size(pile)) \leftarrow VarLoc(x)$

halt Termine l'exécution du programme.
PRE $size(pile) = 0$
POS $halt = 1$

new Alloue un nouvel objet non initialisé de type $Uobj$ au sommet de la pile.
PRE $size(pile) < pilePleine$
POS $size(pile) + 1 \wedge Uobj$

init Initialise un objet non initialisé de type $Uobj$ au sommet de la pile.
PRE $Uobj$
POS Obj

use Manipule un objet de type Obj au sommet de la pile, puis le supprime.
PRE Obj
POS $size(pile) - 1$

tableau 4.2 – Instructions du $JVML_i$, deuxième partie

initialisés. La taille de la pile est représentée par la variable *zz* qui gardera en mémoire la taille de la pile à tout point du programme. Nous voyons que la taille de la pile pourrait être obtenue par calcul en utilisant la variable *ss*, *ie.* $card(dom(ran(ss)))$, cependant pour certains types d'attaques, il est nécessaire de posséder une telle structure de données (*ex.* stack underflow). La variable *vv* est similaire à *ss* et stocke le contenu des variables locales. La variable *halt* est une variable booléenne qui représente la fin d'exécution d'un programme. Elle provoquera un blocage (*deadlock*) de la machine permettant de stopper la recherche de solutions. Enfin, l'invariant 7 permet de prouver le bon fonctionnement du modèle.

4.2.1 Initialisation

L'état initial de la machine est donné dans la figure 4.2.

Initialisation

begin

act1 : $pc := 1$
act2 : $ss := \{1 \mapsto \varnothing\}$
act3 : $vv := \{1 \mapsto initlocalvar\}$
act4 : $zz := \{1 \mapsto 0\}$
act6 : $halt := FALSE$

figure 4.2 – Initialisation du modèle de $JVML_i$

La prochaine instruction à être exécutée correspond à $pc := 1$. La pile est initialement vide et sa taille est donc égale à 0. La constante *initlocalvar* est un choix non déterministe sur le contenu des variables locales à l'initialisation de la machine. Pour améliorer les performances, il est possible de fixer la valeur de *initlocalvar* et de ne plus utiliser un choix non déterministe. À l'état initial, la variable *vv* est initialement égale à *initlocalvar*. Finalement, la variable *halt* est initialisée à $FALSE$ pour permettre au programme de commencer.

4.2.2 Prise en compte des branchements

Dans les instructions composant le JVM_l, l'instruction *ifeq* permet d'effectuer des branchements conditionnels. Elle nous oblige à conserver le contenu de la pile ainsi que des variables locales à tout point du programme. Cela explique la présence d'une fonction partielle de fonction partielle pour les variables *ss* et *vv* ainsi qu'une fonction partielle pour la variable *zz*.

Nous allons utiliser la procédure expliquée à la section 2.2.2. Si les piles et variables locales sont différentes, le programme est rejeté et sinon la vérification continue. Nous n'avons pour le moment pas traité le cas des piles compatibles, qui se trouvent donc rejetées. Lorsque nous exécutons l'instruction *ifeq*, nous différentions deux cas.

Dans le premier cas, nous faisons un branchement sur une instruction se trouvant avant. Dans ce cas, nous disposons des informations de typage de la pile et des variables locales dans les variables respectives *ss* et *vv*. Après exécution de cette instruction, l'élément en sommet de pile est retiré et la pile obtenue doit être compatible avec la pile au point de branchement. Les variables locales ne sont pas modifiées mais doivent aussi être compatible. Dans les travaux présentés dans ce mémoire nous avons simplifié la compatibilité par une égalité. Dans les pré-conditions de l'instruction $ifeq$ nous avons donc le prédicat suivant (L est l'indice de branchement) :

$$
\begin{array}{l}
L \in dom(ss) \Rightarrow \\
\quad (\\
\quad\quad \{zz(pc) - 1\} \lhd ss(pc) = ss(L) \wedge \\
\quad\quad zz(pc) - 1 = zz(L) \wedge \\
\quad\quad vv(pc) = vv(L) \\
\quad)
\end{array}
$$

Dans le deuxième cas, nous effectuons un branchement sur une instruction se trouvant après. Dans ce cas, nous ne pouvons pas vérifier si les types sont compatibles car ils ne sont pas encore définis. Il faudra donc effectuer la vérification une fois que l'on sera arrivé au point de branchement. Pour cela, nous plaçons une contrainte à cet endroit du programme grâce à la variable *ss*. Il faut donc que chaque instruction vérifie la présence de cette contrainte. Pour cela nous ajoutons la pré-condition suivante à

tout les événements :

$$
\begin{array}{l}
pc+1 \in dom(ss) \Rightarrow \\
\quad (\\
\quad\quad ss(pc) = ss(pc+1) \wedge \\
\quad\quad zz(pc) = zz(pc+1) \wedge \\
\quad\quad vv(pc) = vv(pc+1)) \\
\quad)
\end{array}
$$

4.2.3 Instruction dupliquées

L'instruction *inc* est représentée en figure 4.3. Cette instruction a été modélisée en deux événements distincts *inc1* et *inc2*. Cela nous permet de décrire les deux comportements les plus probables que l'on peut rencontrer. Ces événements sont équivalents dans un cas normal, cependant comme expliqué dans la section 3.4, lors de leur utilisation pour la génération de tests de vulnérabilité ils n'auront pas le même sens.

Le 1er événement *inc1* va forcer à ce que l'élément en sommet de pile soit un *Int*. L'événement *inc2* ne modifiera pas l'élément en sommet de pile. Ces événements sont des bi-simulations lorsque leurs gardes sont valides. Cependant dans le cas des tests de vulnérabilité, l'un conserve l'élément en sommet de pile tandis que l'autre remet un *Int* quelque soit le type de l'élément en sommet de pile. Travaillant avec des tests en boite noire nous ne pouvons pas savoir comment a été implémenté le JCBCV. Nous devons donc cherchons a couvrir le plus de failles correspondant aux différentes implémentations possibles.

Nous avons appliqué le même processus aux instructions *Istore*, *Iload*, *Astore* et *Aload*.

4.3 Un exemple de génération de tests de vulnérabilité

Nous allons décrire un exemple de test de vulnérabilité pour l'instruction *Istore*1 dont le modèle est donné en figure 4.4.

Event *inc1* $\widehat{=}$

when

grd5 : $halt = FALSE$

grd6 : $pc < maxpc$

grd7_t : $ss(pc)(zz(pc) - 1) = Int$

grd8_t : $pc + 1 \in dom(ss) \Rightarrow$
$(ss(pc) \lhd \{zz(pc) - 1 \mapsto Int\} = ss(pc + 1) \wedge$
$zz(pc) = zz(pc + 1) \wedge$
$vv(pc) = vv(pc + 1))$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \lhd \{pc + 1 \mapsto ss(pc) \lhd \{zz(pc) - 1 \mapsto Int\}\}$

act3 : $zz := zz \lhd \{pc + 1 \mapsto zz(pc)\}$

act4 : $vv := vv \lhd \{pc + 1 \mapsto vv(pc)\}$

end

Event *inc2* $\widehat{=}$

when

grd5 : $halt = FALSE$

grd6 : $pc < maxpc$

grd7_t : $ss(pc)(zz(pc) - 1) = Int$

grd8_t : $pc + 1 \in dom(ss) \Rightarrow$
$(ss(pc) = ss(pc + 1) \wedge$
$zz(pc) = zz(pc + 1) \wedge$
$vv(pc) = vv(pc + 1))$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \lhd \{pc + 1 \mapsto ss(pc)\}$

act3 : $zz := zz \lhd \{pc + 1 \mapsto zz(pc)\}$

act4 : $vv := vv \lhd \{pc + 1 \mapsto vv(pc)\}$

end

figure 4.3 – Instruction *inc*

Event *Istore1* $\widehat{=}$

any

x

where

grd1_t : $x \in 1 \mathinner{..} maxlocalvar$

grd2 : $halt = FALSE$

grd3 : $pc < maxpc$

grd4_t : $ss(pc)(zz(pc) - 1) = Int$

grd5_t : $pc + 1 \in dom(ss) \Rightarrow$
$(\{zz(pc) - 1\} \lhd ss(pc) = ss(pc + 1) \land$
$zz(pc) - 1 = zz(pc + 1) \land$
$vv(pc) \lhd\!\!+ \{x \mapsto Int\} = vv(pc + 1))$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \lhd\!\!+ \{pc + 1 \mapsto \{zz(pc) - 1\} \lhd ss(pc)\}$

act3 : $zz := zz \lhd\!\!+ \{pc + 1 \mapsto zz(pc) - 1\}$

act4 : $vv := vv \lhd\!\!+ \{pc + 1 \mapsto vv(pc) \lhd\!\!+ \{x \mapsto Int\}\}$

end

figure 4.4 – Instruction *Istore1*

La première étape consiste à grouper les parties de garde en séparant les parties à tester et les parties non à tester. Suite à cela, nous obtenons le nouvel événement donné en figure 4.5. Cette étape est appliquée à tous les événements du modèle une seule fois pour ensuite pouvoir les ré-utiliser pour toutes les dérivations.

Il faut maintenant nier la partie de la garde que nous souhaitons tester, *i.e.* celle qui possède le label *grd_t*. Nous obtenons les résultats suivants :

1. $\neg(grd5\_t) \land grd4\_t \land grd1\_t$
2. $grd5\_t \land \neg(grd4\_t) \land grd1\_t$
3. $\neg(grd5\_t) \land \neg(grd4\_t) \land grd1\_t$
4. $grd5\_t \land grd4\_t \land \neg(grd1\_t)$
5. $\neg(grd5\_t) \land grd4\_t \land \neg(grd1\_t)$

Event *Istore1* $\widehat{=}$

any

x

where

grd : $(halt = FALSE) \wedge (pc < maxpc)$

grd_t : $(x \in 1 \mathinner{..} maxlocalvar) \wedge (ss(pc)(zz(pc) - 1) = Int) \wedge (pc + 1 \in dom(ss) \Rightarrow (\{zz(pc) - 1\} \lhd ss(pc) = ss(pc + 1) \wedge zz(pc) - 1 = zz(pc + 1) \wedge vv(pc) \lhd\!\!+ \{x \mapsto Int\} = vv(pc + 1)))$

then

act : ...

end

figure 4.5 – Instruction *Iload1* avec groupage de la garde

6. $grd5\_t \wedge \neg(grd4\_t) \wedge \neg(grd1\_t)$
7. $\neg(grd5\_t) \wedge \neg(grd4\_t) \wedge \neg(grd1\_t)$

Nous obtenons ainsi sept ré-écritures possibles. Nous allons donc construire 7 nouveaux modèles. Chacun de ces modèles possédera le nouvel événement $Iload1\_EUT$ avec l'une des ré-écritures. Prenons la règle 2 pour la suite de notre exemple représenté dans la figure 4.6. Cette ré-écriture est composée de 3 parties. Dans cette règle, seule la deuxième partie est niée (en rouge). Avec cette règle nous voulons tester le cas où il y a un branchement à vérifier, il est valide et l'index dans la table des variables locales est valide, mais le type au sommet de la pile n'est pas un *Int*.

Maintenant que nous avons notre modèle dérivé, il faut extraire une trace, *i.e.* un test abstrait. Pour cet exemple, nous allons chercher une trace contenant un préambule et un postambule. En voici quelques unes :

- [[*INITIALISATION; **Istore1_EUT***; halt]]
- [[*INITIALISATION; New; **Istore1_EUT***; halt]]
- [[*INITIALISATION; New; Inti; **Istore1_EUT***; halt]]

Event *Iload1_EUT* $\widehat{=}$

any

x

where

grd : $(halt = FALSE) \wedge (pc < maxpc)$

grd_t : $(pc + 1 \in dom(ss) \Rightarrow \{zz(pc) - 1\} \lhd ss(pc) \ \wedge \ [...] \ ) \wedge$
$\neg(ss(pc)(zz(pc) - 1) = Int) \wedge$
$x \in 1 \,..\, maxlocalvar)$

then

act : [...]

end

figure 4.6 – Instruction *Iload1_EUT*

Chapitre 4. Application au langage de Freund et Mitchell

Chapitre 5

Application au langage JavaCard

Afin de valider notre approche sur un système réel, nous l'avons appliqué sur le langage Java Card. Cela dans le but de générer des tests de vulnérabilité permettant de tester la sécurité de différents vérifieurs de *byte code*. Pour le moment seules quelques instructions ont été modélisées et un nombre restreint de failles sont couvertes. L'exécution des tests a été effectués sur le vérifieur off-card de Oracle ainsi que sur différentes cartes à puce Java Cards. Le modèle que nous avons utilisé est donné en annexe C.

5.1 Construction du modèle

Ne traitant qu'une sous-partie du langage Java Card, nous ne pouvions tester qu'un ensemble restreint de failles de sécurité. Nous avons donc tenté d'en couvrir un maximum. Cependant toutes celles concernant les branchements ou encore le treillis de type ont été laissées pour les travaux futurs.

Nous avons choisi de tester complètement les instructions *sadd* et *sload*. Un modèle ne contenant que ces instructions ne permet pas de générer de programme. Nous avons donc dû regarder quelle instruction ajouter pour que notre modèle puisse nous générer des programmes.

Tout d'abord nous avons rajouté les instructions *return*, *sconst_m1* et *sconst_0*

... *sconst_5*. Ces instructions permettent de générer des programmes valides. Cependant, lorsque nous voulons générer les tests de vulnérabilité, pour l'instruction *sadd*, nous voyons qu'il nous manque une instruction permettant de placer un élément de type non compatible avec le type *short* en sommet de pile. Nous avons donc rajouté *aconst_null* permettant de placer un pointeur null sur la pile.

Nous voulons toujours finir un test de vulnérabilité avec une pile vide. Lors d'une attaque par dépassement de pile par le haut (stack overflow), l'instruction *pop* permet de revenir dans un état où la pile ne dépasse pas ses capacités.

Le modèle est donc constitué de 12 événements représentant chacun une instruction du langage Java Card :

- *aconst_null*
- *pop*
- *return*
- *sadd*
- *sconst_m1*
- *sconst_0*
- [...]
- *sconst_5*
- *sload*

Nous traitons les failles de sécurité suivantes :

- dépassement de pile par le haut *stack overflow*
- dépassement de pile par le bas *stack underflow*
- faute de typage

5.2 Un exemple de génération de tests de vulnérabilité

Nous allons prendre comme exemple l'instruction *sadd* donnée en figure 5.1. Cette instruction ne peut s'exécuter que lorsque la pile contient au moins deux éléments et que ces éléments sont de type *short*.

Voici un sous-ensemble de règles de ré-écriture que nous obtenons :

Event $sadd \widehat{=}$

any

$forms$

$stackSize$

where

grd1 : $halt = FALSE$

grd2 : $pc < maxpc$

grd3 : $card(dom(ByteArray)) + 1 < maxByteArrayLength$

grd4 : $zz(pc) = stackSize$

grd5 : $forms = 65$

grd1_t : $stackSize \geq 2$

grd2_t : $ss(pc)(zz(pc) - 1) = short$

grd3_t : $ss(pc)(zz(pc) - 2) = short$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \Leftarrow \{pc + 1 \mapsto (\{zz(pc) - 1\} \lhd ss(pc))\}$

act3 : $zz := zz \Leftarrow \{pc + 1 \mapsto zz(pc) - 1\}$

act4 : $vv := vv \Leftarrow \{pc + 1 \mapsto vv(pc)\}$

act5 : $ByteArray := ByteArray \Leftarrow \{card(dom(ByteArray)) \mapsto forms\}$

end

figure 5.1 – Instruction *sadd*

- $stackSize < 2 \wedge ss(pc)(zz(pc) - 1) = short \wedge ss(pc)(zz(pc) - 2) = short$
- $stackSize \geq 2 \wedge ss(pc)(zz(pc) - 1) \neq short \wedge ss(pc)(zz(pc) - 2) = short$
- $stackSize \geq 2 \wedge ss(pc)(zz(pc) - 1) \neq short \wedge ss(pc)(zz(pc) - 2) \neq short$

Pour notre exemple, nous allons prendre la $2^{ème}$ règle. Le préambule doit mettre un *int* en sommet de pile et un *short* juste en dessous. Voici un ensemble de traces que nous pouvons obtenir :

- [[*INITIALISATION; sconst_0; aconst_null;* ***sadd***; pop ; return]]
- [[*INITIALISATION; sconst_3; aconst_null;* ***sadd***; pop ; return]]
- [[*INITIALISATION; sconst_4; pop; sconst_m1; aconst_null;* ***sadd***; pop ; return]]

– [[*INITIALISATION; sconst_1; aconst_null;* ***sadd***; pop; sconst_2; pop; return]]

Nous allons prendre l'exemple de la 1ère trace. Pour concrétiser ce test, nous allons remplir un fichier CAP, que nous avons décrit dans le chapitre 2. Nous avons pour cela fabriqué un fichier CAP prérempli dans lequel nous allons pouvoir insérer notre test. Dans le composant *method* nous utilisons une méthode vierge prévue à recevoir le test. Cette méthode contient initialement seulement l'instruction *return*. Nous allons donc ajouter les instructions $0x03; 0x01; 0x41; 0x3b; 0x7a$ correspondant aux instructions de la trace. Nous obtenons le fichier dont une partie est donnée en figure 5.2.

```
000001F0: 10 80 6B 0A|18 1A 1F 8C|00 06 70 08|11 6E 00 8D
00000200: 00 07 7A 03|32 03 01 41|3B 7A 00 00|00 00 00 00
00000210: 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00
```

figure 5.2 – Partie du fichier CAP contenant le programme

Chapitre 6

Outils utilisés et outils développés

Les programmes qui ont été développés, ainsi que ceux que nous avons utilisés, se décomposent en trois grandes parties. La figure 6.1 représente ces différentes parties. Les programmes développés on été produits par les étudiants Tiana Razafindralambo, Josselin Dolhen ainsi que l'auteur de ce mémoire.

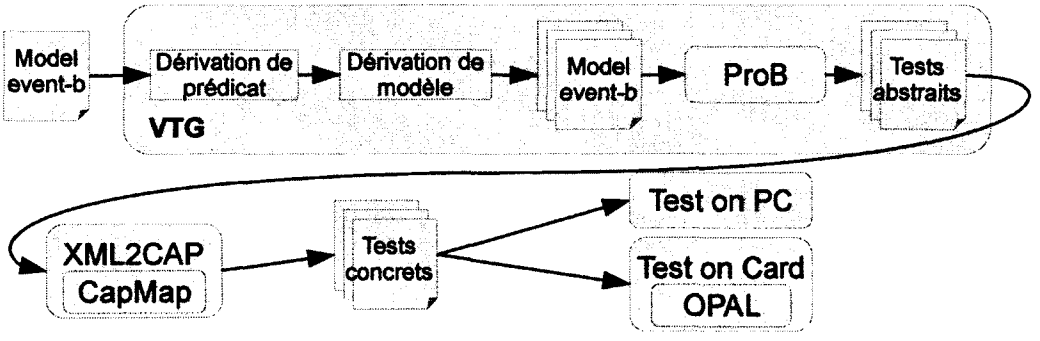

figure 6.1 – Architecture des outils

La première partie, permettant la dérivation de modèles formels à partir d'un modèle initial, se décompose en deux sous-parties : la génération de règles de réécriture à partir de règles de réécriture et la dérivation de modèles avec les nouvelles règles de réécriture. La seconde partie permet d'extraire les tests abstraits à partir de ces modèles en utilisant ProB.

Les résultats obtenus par le VTG sont ensuite utilisés par deux autres programmes. Le premier est le programme XML2CAP qui permet de concrétiser les tests abstraits, soit de générer des fichiers au format CAP. Le deuxième programme correspond à l'exécution des tests sur le vérifieur de byte-code.

6.1 ProB

Une description de ProB a déjà été commencée à la section 2.4.4. L'utilisation de ce logiciel peut se faire à l'aide de différentes interfaces. Dans nos recherches, nous en avons utilisé deux différentes. Tout d'abord, nous avons utilisé l'interface graphique classique nous permettant de simplement vérifier des propriétés sur notre modèle ou encore extraire quelques traces semi-automatiquement. Puis nous avons utilisé l'interface en ligne de commande permettant d'automatiser certaines tâches.

6.2 OPAL

OPAL, présenté a la conférence *e-Smart 2011* [13], est une bibliothèque Java open-source permettant de communiquer avec des cartes à puce. Elle a été développée par l'équipe SSD de l'Université de Limoges dans le cadre des recherches concernant les attaques de cartes à puce. L'un des principaux avantages de OPAL est d'offrir une API simple à utiliser dans n'importe quel logiciel et qui reconnaît toute les cartes à puces. De plus, elle propose une implémentation pour toutes les spécifications de GlobalPlatform (incluant notamment les dernières spécifications de communication en HTTP) et elle est multi-plateforme.

6.3 CapMap

Le CapMap, aussi appelé *CAP file manipulator*, est une bibliothèque Java permettant de manipuler les *fichiers CAP*. Il as été présenté lors de la conférence MajecSTIC [30]. Lui aussi développé par l'équipe SSD de l'Université de Limoges, il est notamment utilisé pour effectuer des attaques logiques sur les cartes à puce. Comme nous l'avons vu dans la section 2.2.1, les *fichiers CAP* ont un formalisme très contraint. Le

CapMap permet de manipuler ces fichiers tout en conservant ou non les dépendances entre les différents composants. Si on conserve ces dépendances, cela nous permet de modifier à la main le contenu des *fichiers CAP* tout en gardant un *fichier CAP* valide (du point de vue du vérifieur de structure). Dans le cas où on ne conserve pas les dépendances, il est possible d'introduire des fautes dans le *fichier CAP*. Ce dernier cas est notamment très intéressant pour effectuer des attaques logiques sur la structure du *fichier CAP*.

6.4 VTG

La mise en oeuvre de la génération de tests de vulnérabilité ne pouvait se faire sans un outil permettant de valider la solution. Nous en avons donc proposé une implémentation appelée VTG. Ce logiciel a été présenté pour la première fois lors du workshop Deploy [35].

Les étudiants Tiana Razafindralambo et Josselin Dolhen on largement contribué à cette implémentation durant leur formation ainsi que lors de différents stages qu'il ont réalisé.

Le VTG peut fonctionner semi automatiquement ou entièrement automatique. L'automatisation totale permet de rapidement générer une suite de tests tandis qu'une génération semi-automatique permet de mieux paramétrer cette génération.

L'interaction avec le VTG se fait à l'aide d'une simple application web comportant quatre pages. La première page permet de sélectionner le modèle avec lequel nous allons effectuer la génération de tests de vulnérabilité. La deuxième page, donnée en figure 6.2, correspond aux règles de réécritures. Elle sont automatiquement générée mais l'utilisateur peut en rajouter, en modifier ou en supprimer pour s'adapter au mieux aux besoins.

6.4.1 Limitations

Ce logiciel nous a permis d'effectuer nos premières expérimentations mais il n'est pour le moment pas fini. Parmis les fonctionnalités manquantes, il serait intéressant de pouvoir générer des TdeV pour des contextes ou encore pour des modèles qui sont

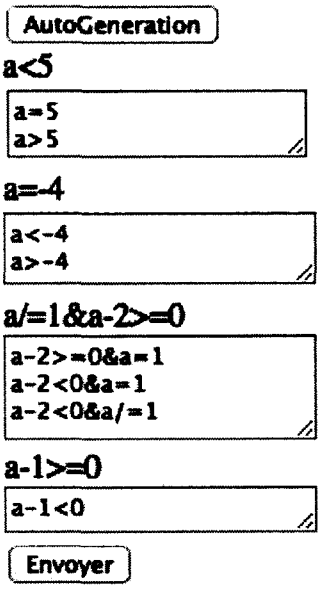

figure 6.2 – Deuxième page du VTG

obtenus par raffinement. L'algorithme de négation de règles ne comporte actuellement pas toutes les règles dont nous aurions besoin.

6.5 Divers petits programmes

6.5.1 XmlToCap

Après la génération des tests abstraits, le programme *XmlToCap* nous permet de concrétiser ces tests pour Java Card. Bien entendu, il fonctionne uniquement avec un modèle prévu pour ce type de plateforme. Il prend en entrée un fichier XML ou un répertoire contenant des fichiers XML. Pour tous les fichiers correspondant à des XML qu'il sait concrétiser il va générer les tests concrets sous format de fichier CAP.

6.5.2 TestOnPC et TestOnCard

Nous avons deux programmes permettant d'exécuter les tests de vulnérabilité. Le premier, *TestOnPC*, les exécute sur le vérifieur de byte code de Oracle. Le deuxième,

TestOnCard, les exécute sur une ou plusieurs cartes à puce.

Nous fournissons à ces programmes un fichier CAP ou un répertoire contenant un ensemble de fichiers CAP. Ces fichiers sont envoyés au vérifieur et les résultats sont sauvegardés dans différents fichiers. Le premier fichier contient l'ensemble des fichiers acceptés. Le second contient l'ensemble des fichiers rejetés. Enfin le dernier fichier contient l'explication du rejet des fichiers CAP. Si le vérifieur fonctionne correctement tou les fichiers doivent être rejetés.

Ces deux implémentations ont été réalisées par Josselin Dolhen.

Chapitre 6. Outils utilisés et outils développés

Chapitre 7

Évaluation de l'approche

Dans ce chapitre nous présentons les résultats expérimentaux. Toutes les mesures ont été réalisées sur un MacBook Pro possédant un processeur 2,3 GHz Core i5 dual-core, 4Go de RAM et un disque dur 5400 $t.m^{-1}$. Toutes les mesures ont été réalisées trois fois et nous donnons la moyenne de ces résultats. Les exécutions sur carte à puce ont été réalisées sur des JC 2.1 implémentant GP 2.0.1.

Notre but étant de tester chaque instruction du langage Java, nous avons réalisé la génération telle que pour chaque faute possible nous ayons au moins un test. Le critère de couverture représentera donc le pourcentage de défaillances que nous sommes capable de tester.

7.1 Le modèle de référence

7.1.1 Génération naïve

Dans un premier temps nous avons utilisé le logiciel sans chercher à optimiser les résultats.

Les résultats de l'algorithme de négation de prédicat ont été utilisés sans modification. Cela a conduit à lancer la recherche de traces sur des modèles dérivés pour lesquels il n'existe pas de solution. Par exemple la première négation de $a \neq 1 \wedge a - 2 \geq 0$ est $a - 2 \geq 0 \wedge a = 1$. Nous avons tout de même laissé toutes ces ré-écritures afin de

Profondeur de recherche	Temps de génération (s)	Nb tests	Vitesse (nb/s)	Vitesse (nb/min)
3	10,8	5	0,463	28
5	19,4	68	3,503	210
7	125,7	772	6,142	369
9	1789,9	8045	4,495	270
11	9680,0	25021	2,585	155

tableau 7.1 – Temps de génération des TdeV pour le modèle ExpVTG, génération naïve

Profondeur de recherche	Modèles Couverts
3	20%
5	40%
7	80%
9	80%
11	80%

tableau 7.2 – Couverture pour le modèle ExpVTG, génération naïve

caractériser notre logiciel lors d'une utilisation naïve.

Pour la recherche des traces, nous n'avons pas non plus cherché à optimiser les paramètres de cette recherche. Nous avons simplement utilisé le paramètre global.

Le tableau 7.1 représente les temps d'exécution et le nombre de tests obtenus pour différentes profondeurs. Le tableau 7.2 représente la couverture des tests pour différentes profondeurs.

Jusqu'à une profondeur de 7 nous augmentons le nombre de tests que nous générons par minute ainsi que le nombre de modèles couverts. Passé cette profondeur nous n'augmentons plus cette couverture et la vitesse diminue. Les 20 % de couverture restant sont dû à des modèles pour lesquels il n'existe aucune solution. Le premier nécessite de satisfaire le prédicat $a - 2 \geq 0 \wedge a = 1$ qui sera toujours faux. Le deuxième nécessite de satisfaire le prédicat $a > 5$. Comme nous pouvons le voir sur le diagramme d'état transition donné à la fin de l'annexe A, il est impossible de trouver un préambule conduisant dans un état ou ce prédicat est vrai.

Pour les profondeurs supérieurs à 7 nous avons donc atteint notre maximum de modèles couverts. Cependant, comme nous pouvons le voir sur le tableau 7.3, le nombre de tests augmente toujours. Même si la couverture n'augmente plus, il existe toujours d'autres tests plus longs et plus complexes par rapport à ceux déjà trouvés.

Profondeur de recherche	Tests par modèle
3	0,5
5	6,8
7	77,2
9	804,5
11	2502,1

tableau 7.3 – Nombre moyen de tests par modèle pour ExpVTG, génération naïve

7.1.2 Génération plus précise

Une étude plus précise du modèle et des modèles dérivés nous permet de mieux paramétrer le VTG. Tout d'abord nous savons qu'il n'est pas utile de rechercher des traces de profondeur supérieure à 7. Ensuite nous pouvons supprimer des modèles pour lesquels nous ne trouverons jamais de solution. Finalement, pour chaque modèle dérivé, nous souhaitons obtenir au moins un test. Le tableau 7.4 donne la liste des modèles et de leur profondeur respective à partir de laquelle nous obtenons une solution.

La génération de ces tests a produit 11 tests en $33,2$ s. Nous avons donc une vitesse de $0,34$ $t.s^{-1}$ ou encore 20 $t.m^{-1}$.

7.2 Le modèle d'une partie du langage Java Card

Nous présentons ici les résultats obtenus à partir du modèle donné dans le chapitre 5. Ces expérimentations se décomposent en trois parties. Dans un premier temps nous avons généré un jeu de TdeV abstraits à l'aide du VTG. Dans un second temps,

Modèles	Profondeur
M_add_1_EUT	6
M_suppr_3_EUT	5
M_supprTwo_5_EUT	6
M_supprTwo_6_EUT	5
M_nop1_7_EUT	7
M_nop1_8_EUT	2
M_nop2_9_EUT	7
M_nop2_10_EUT	2

tableau 7.4 – Profondeur optimale pour le modèle ExpVTG, génération optimale

Profondeur de recherche	Temps de génération (s)	Nb tests	Vitesse (nb/s)	Vitesse (nb/min)
2	53,6	2	0,037	2
3	148,7	30	0,202	12,1
4	1380,2	432	0,313	17,7
5	11286,7	10133	0,898	53,9

tableau 7.5 – Temps de génération des TdeV pour le modèle Langage_JC, génération naïve

nous avons dû concrétiser ces tests. Finalement nous avons exécuté les TdeV sur des vérifieurs de byte code Java Card.

7.2.1 Obtention des tests abstraits

Les tableaux 7.5 et 7.6 présentent les résultats pour des exécutions du VTG avec des paramètres de recherche choisis naïvement. Nous voyons que l'utilisation naïve du VTG n'est pas toujours possible.

Nous avons donc utilisé des paramètres plus précis afin d'obtenir des TdeV de façon plus efficace. Pour chaque modèle dérivé, nous avons tenté d'obtenir au moins un TdeV. Nous avons pour cela regardé la garde de chaque événement sous test. S'il n'était pas possible de les satisfaire nous n'avons pas cherché de TdeV pour ces modèles. Sinon nous avons cherché la profondeur minimum à laquelle nous devions trouver un premier TdeV. Le tableau 7.7 donne les valeurs que nous avons utilisées.

Profondeur de recherche	Modèles Couverts
2	8%
3	8%
4	22%
5	65%

tableau 7.6 – Couverture pour le modèle Langage_JC, génération naïve

Cette génération a duré 5283s et nous a permis d'économiser 1/3 du temps de calcul tout en couvrant autant de failles de sécurité.

7.2.2 Concrétisation des tests

Pour simplifier l'étape de concrétisation, nous sommes partis d'un fichier CAP pré-rempli. Nous ne faisons qu'ajouter les instructions composant le programme. Le tableau 7.8 donne les temps de génération de ces tests pour les différentes exécutions du VTG.

7.2.3 Execution des tests sur vérifieurs de byte code

Les fichier CAP obtenus ont finalement été exécutés sur différents vérifieurs de byte code. Nous les avons tout d'abord exécutés sur le vérifieur off-card de Oracle. Tous nos TdeV ont été rejetés. Nous n'avons donc pas trouvé de failles dans ce dernier. Le tableau 7.9 donne les tests d'exécution des différents jeux de tests. Ce vérifieur nous a permis d'exécuter en moyenne 180 $t.min^{-1}$.

Dans le cas des cartes à puce les résultats étaient très différents. Beaucoup de TdeV ont été acceptés par les cartes et ont donc pu être exécutés. Dû à la sensibilité des résultats, nous ne donnerons ici ni la liste des failles trouvées, ni la liste des cartes sur lesquelles nous avons travaillé.

Modèles	Profondeur
M_aconst_null_1	5
M_pop_2	2
M_sadd_3	>5
M_sadd_4	4
M_sadd_5	4
M_sadd_6	>5
M_sadd_7	>5
M_sadd_8	>5
M_sadd_9	4
M_sconst_n1_10	5
M_sconst_0_11	5
M_sconst_1_12	5
M_sconst_2_13	5
M_sconst_3_14	5
M_sconst_4_15	5
M_sconst_5_16	5
M_sload_17	>5
M_sload_18	2
M_sload_19	>5
M_sload_20	>5
M_sload_21	>5
M_sload_22	5
M_sload_23	5

tableau 7.7 – Profondeur optimale pour le modèle Langage_JC, génération optimale

Profondeur de recherche	Temps (s)
2	0,8
3	1,7
4	7,2
5	113,6
*	86,9

tableau 7.8 – Temps de génération des fichiers CAP, * :profondeur optimale

Profondeur de recherche	Temps (s)
2	0,8
3	9,9
4	143,2
5	3682,0
*	2415,3

tableau 7.9 – Temps d'exécution des TdeV sur le vérifieur off-card de Oracle, * :profondeur optimale

Conclusion

Ces travaux ont montré la faisabilité de l'approche. Ils ne se veulent pas complets et proposent simplement une première étude appliquée à une sous-partie du vérifieur de bytecode Java Card.

Il serait tout d'abord intéressant de voir comment se comporterait notre méthode sur l'ensemble du langage Java Card. L'animation d'un modèle à 12 instructions ne pose pas de problème mais notre méthode risque de ne pas supporter le passage à l'échelle. De plus, toutes les fonctionnalités du vérifieur de byte code Java Card ne sont pas modélisées. Parmi celles-ci, nous avons le treillis de type. Sa modélisation permettrait d'ouvrir la porte à de nouvelles attaques. L'une des autres fonctionnalités serait les branchements avec le calcul de point fixe. Notre proposition n'est pour le moment appliquée qu'à une seule utilisation, le test du vérifieur de bytecode. Il serait intéressant de voir son impact sur d'autres problèmes. Pour rester dans le domaine de la carte, il pourrait être intéressant de tester des *Applets*, constituant un problème de plus haut niveau pour lequel les réponses du système ne sont pas simplement binaires.

Afin de garantir une grande flexibilité de l'approche, il serait intéressant de proposer des règles de réécriture pour toute la grammaire des gardes Even-B. De plus les règles de réécriture que nous avons proposées n'ont pas été prouvées. Il serait intéressant par une méthode systématique d'assurer qu'une règle de réécriture ne rentre pas en conflit avec les autres ou encore qu'une règle couvre bien tous les cas possibles.

Nos résultat ont montré que cette méthode permettait de produire des tests de vulnérabilité. Ces tests nous ont notamment permis de trouver des défaillances dans différentes implémentations du vérifieur de bytecode Java Card. L'implémentation d'un outil a permis de passer de la théorie à la pratique pour une partie du vérifieur de byte code Java Card. L'application à différentes implémentations du vérifieur a permis de trouver plusieurs failles.

Conclusion

Annexe A

Modèle d'exemple

VARIABLES

a

halt

INVARIANTS

inv1 : $a \in -5 \mathinner{..} 5$

inv2 : $halt \in BOOL$

EVENTS

Initialisation

begin

act1 : $a := 3$

act2 : $halt := FALSE$

end

Event $add \mathrel{\widehat{=}}$

when

grd1 : $a \geq 0$

grd2_t : $a < 5$

grd3 : $halt = FALSE$

then

act1 : $a := a + 1$

end

Event *suppr* $\widehat{=}$

when

grd1_t : $a - 1 \geq 0$

grd2 : $halt = FALSE$

then

act1 : $a := a - 1$

end

Event *supprTwo* $\widehat{=}$

when

grd1_t : $a \neq 1$

grd2_t : $a - 2 \geq 0$

grd3 : $halt = FALSE$

then

act1 : $a := a - 2$

end

Event *opposite* $\widehat{=}$

when

grd1 : $a \neq 2$

grd2 : $a \neq -2$

grd3 : $halt = FALSE$

then

act1 : $a := a * -1$

end

Event *nop1* $\widehat{=}$

when

grd1_t : $a = -4$

then

act1 : $a := a$

end

Event *nop2* $\triangleq$

when

grd1_t : $a = -4$

then

act1 : $a := -4$

end

Event *Halt* $\triangleq$

when

grd1 : $a = 3$

grd2 : $halt = FALSE$

then

act1 : $halt := TRUE$

end

END

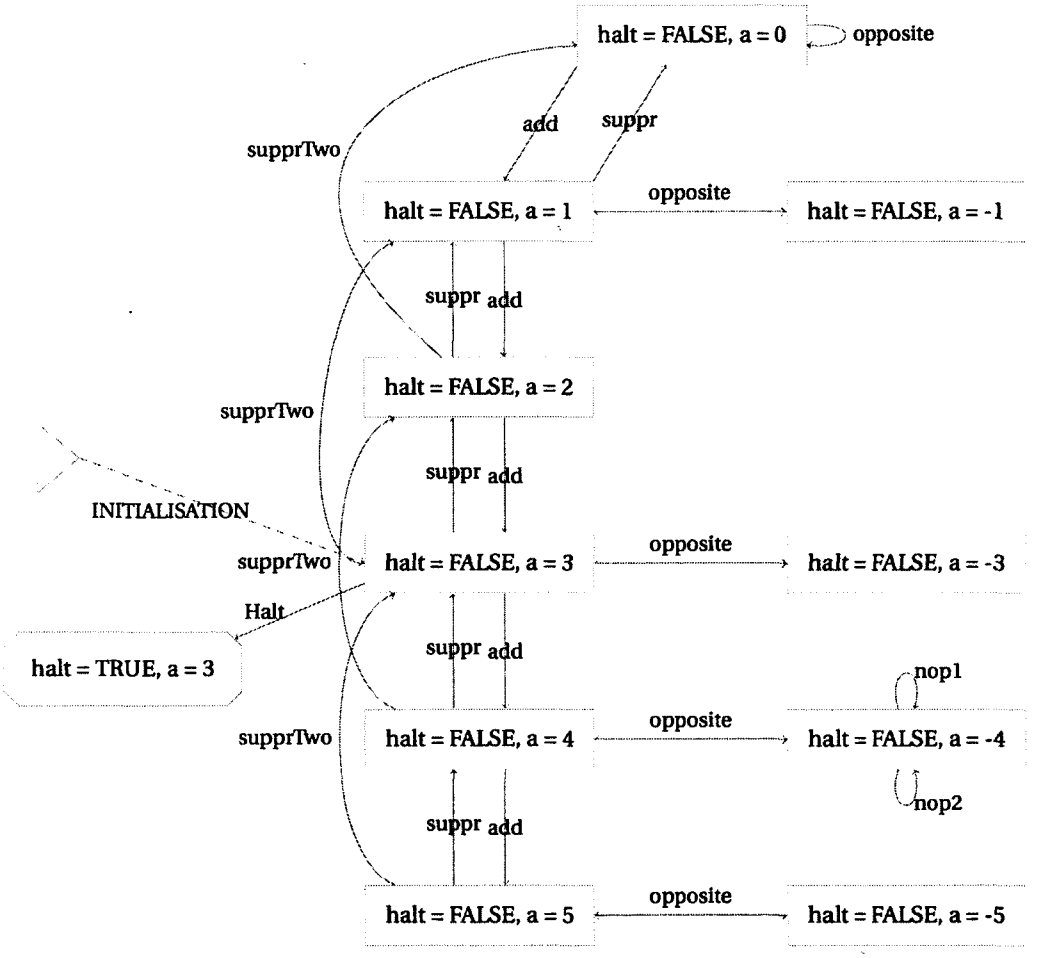

figure A.1 – États et transitions valides

Annexe B

Modèle du langage de Freund et Mitchell

CONTEXT C

SETS

ELEMENT

CONSTANTS

TYPE

Int

Obj

Uobj

maxpc

maxstack

maxlocalvar

initlocalvar

AXIOMS

axm1 : $TYPE \subseteq ELEMENT$

axm1_1 : $TYPE = \{Int, Obj, Uobj\}$

axm1_2 : $Int \neq Obj \wedge Int \neq Uobj$

axm1_3 : $Obj \neq Uobj$

axm2 : $maxpc = 20$

axm3 : $maxstack = 5$

axm4 : $maxlocalvar = 3$

axm5 : $initlocalvar \in 1 \mathinner{..} maxlocalvar \pfun TYPE$

END

MACHINE M
SEES C
VARIABLES

pc

ss

vv

zz

halt

INVARIANTS

inv1 : $pc \in 1 \mathinner{..} maxpc$

inv2 : $ss \in 1 \mathinner{..} maxpc \pfun (0 \mathinner{..} maxstack - 1 \pfun TYPE)$

inv3 : $vv \in 1 \mathinner{..} maxpc \pfun (1 \mathinner{..} maxlocalvar \pfun TYPE)$

inv4 : $zz \in 1 \mathinner{..} maxpc \pfun 0 \mathinner{..} maxstack$

inv5 : $halt \in BOOL$

inv6 : $dom(ss) = dom(vv) \land dom(ss) = dom(zz) \land dom(vv) = dom(zz)$

inv7 : $pc \in dom(ss)$

EVENTS
Initialisation

begin

act1 : $pc := 1$

act2 : $ss := \{1 \mapsto \varnothing\}$

act3 : $vv := \{1 \mapsto initlocalvar\}$

act4 : $zz := \{1 \mapsto 0\}$

act6 : $halt := FALSE$

end

Event *inc1* $\widehat{=}$

when

grd5 : $halt = FALSE$

grd6 : $pc < maxpc$

grd7_t : $ss(pc)(zz(pc) - 1) = Int$

The element on tos is an Int

grd8_t : $pc + 1 \in dom(ss) \Rightarrow (ss(pc) \lhd\!\!\!- \{zz(pc) - 1 \mapsto Int\} = ss(pc + 1) \wedge zz(pc) = zz(pc + 1) \wedge vv(pc) = vv(pc + 1))$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \lhd\!\!\!- \{pc + 1 \mapsto ss(pc) \lhd\!\!\!- \{zz(pc) - 1 \mapsto Int\}\}$

act3 : $zz := zz \lhd\!\!\!- \{pc + 1 \mapsto zz(pc)\}$

act4 : $vv := vv \lhd\!\!\!- \{pc + 1 \mapsto vv(pc)\}$

end

Event *inc2* $\widehat{=}$

when

grd5 : $halt = FALSE$

grd6 : $pc < maxpc$

grd7_t : $ss(pc)(zz(pc) - 1) = Int$

The element on tos is an Int

grd8_t : $pc+1 \in dom(ss) \Rightarrow (ss(pc) = ss(pc+1) \wedge zz(pc) = zz(pc+1) \wedge vv(pc) = vv(pc + 1))$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \lhd\!\!\!- \{pc + 1 \mapsto ss(pc)\}$

act3 : $zz := zz \Leftarrow \{pc + 1 \mapsto zz(pc)\}$

act4 : $vv := vv \Leftarrow \{pc + 1 \mapsto vv(pc)\}$

end

Event *push* $\widehat{=}$

when

grd6_t : $zz(pc) < maxstack$

grd7 : $halt = FALSE$

grd8 : $pc < maxpc$

grd9_t : $pc + 1 \in dom(ss) \Rightarrow (ss(pc) \Leftarrow \{zz(pc) \mapsto Int\} = ss(pc + 1) \wedge zz(pc) + 1 = zz(pc + 1) \wedge vv(pc) = vv(pc + 1))$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \Leftarrow \{pc + 1 \mapsto ss(pc) \Leftarrow \{zz(pc) \mapsto Int\}\}$

act3 : $zz := zz \Leftarrow \{pc + 1 \mapsto zz(pc) + 1\}$

act4 : $vv := vv \Leftarrow \{pc + 1 \mapsto vv(pc)\}$

end

Event *pop* $\widehat{=}$

when

grd6_t : $zz(pc) > 0$

grd7 : $halt = FALSE$

grd8 : $pc < maxpc$

grd9_t : $pc+1 \in dom(ss) \Rightarrow (\{zz(pc)-1\} \lhd ss(pc) = ss(pc+1) \wedge zz(pc)-1 = zz(pc+1) \wedge vv(pc) = vv(pc+1))$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \Leftarrow \{pc + 1 \mapsto (\{zz(pc) - 1\} \lhd ss(pc))\}$

act3 : $zz := zz \Leftarrow \{pc + 1 \mapsto zz(pc) - 1\}$

act4 : $vv := vv \Leftarrow \{pc + 1 \mapsto vv(pc)\}$

end

Event *If* $\widehat{=}$

FINI AVEC Marc ICI 03-02-2012

any

L

where

grd5_t : $L \in 1 \mathinner{..} maxpc$

grd11 : $halt = FALSE$

grd12 : $pc < maxpc$

grd14_t : $ss(pc)(zz(pc) - 1) = Int$

The element on top of stack is an Int

grd15_t : $pc + 1 \in dom(ss) \Rightarrow (\{zz(pc) - 1\} \lhd ss(pc) = ss(pc + 1) \wedge zz(pc) - 1 = zz(pc + 1) \wedge vv(pc) = vv(pc + 1))$

grd16_t : $L \in dom(ss) \Rightarrow (\{zz(pc) - 1\} \lhd ss(pc) = ss(L) \wedge zz(pc) - 1 = zz(L) \wedge vv(pc) = vv(L))$

Vérification du point de branchement

then

act1 : $pc := pc + 1$

act2 : $ss := ss \lhd\!\!+ \{pc + 1 \mapsto (\{zz(pc) - 1\} \lhd ss(pc)), L \mapsto (\{zz(pc) - 1\} \lhd ss(pc))\}$

act3 : $zz := zz \lhd\!\!+ \{pc + 1 \mapsto zz(pc) - 1, L \mapsto zz(pc) - 1\}$

act4 : $vv := vv \lhd\!\!+ \{pc + 1 \mapsto vv(pc), L \mapsto vv(pc)\}$

end

Event *Istore1* $\widehat{=}$

Version where the rewrite with an Int

any

x

where

grd6_t : $x \in 1 \mathinner{..} maxlocalvar$

grd11 : $halt = FALSE$

grd12 : $pc < maxpc$

grd14_t : $ss(pc)(zz(pc) - 1) = Int$

The element on top of stack is an Int

grd15_t : $pc + 1 \in dom(ss) \Rightarrow (\{zz(pc) - 1\} \lhd ss(pc) = ss(pc + 1) \land zz(pc) - 1 = zz(pc + 1) \land vv(pc) \lhd\!\!+ \{x \mapsto Int\} = vv(pc + 1))$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \lhd\!\!+ \{pc + 1 \mapsto \{zz(pc) - 1\} \lhd ss(pc)\}$

act3 : $zz := zz \lhd\!\!+ \{pc + 1 \mapsto zz(pc) - 1\}$

act4 : $vv := vv \lhd\!\!+ \{pc + 1 \mapsto vv(pc) \lhd\!\!+ \{x \mapsto Int\}\}$

end

Event *Istore2* $\widehat{=}$

Version where the rewrite with le element on the tos

any

x

where

grd6_t : $x \in 1 \mathinner{..} maxlocalvar$

grd11 : $halt = FALSE$

grd12 : $pc < maxpc$

grd14_t : $ss(pc)(zz(pc) - 1) = Int$

The element on top of stack is an Int

grd15_t : $pc+1 \in dom(ss) \Rightarrow (\{zz(pc)-1\} \lhd ss(pc) = ss(pc+1) \land zz(pc)-1 = zz(pc + 1) \land vv(pc) \lhd\!\!+ \{x \mapsto ss(pc)(zz(pc) - 1)\} = vv(pc + 1))$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \lhd\!\!+ \{pc + 1 \mapsto \{zz(pc) - 1\} \lhd ss(pc)\}$

act3 : $zz := zz \lhd\!\!+ \{pc + 1 \mapsto zz(pc) - 1\}$

act4 : $vv := vv \lhd\!\!+ \{pc + 1 \mapsto vv(pc) \lhd\!\!+ \{x \mapsto ss(pc)(zz(pc) - 1)\}\}$

end

Event *Iload1* $\widehat{=}$

any

x

where

grd5__t : $x \in 1 \mathinner{..} maxlocalvar$

grd8__t : $zz(pc) < maxstack$

grd9 : $halt = FALSE$

grd10 : $pc < maxpc$

grd12__t : $vv(pc)(x) = Int$

The element on the vaiable x is an Int

grd13__t : $pc + 1 \in dom(ss) \Rightarrow (ss(pc) \lhd\!\!\!- \{zz(pc) \mapsto Int\} = ss(pc + 1) \wedge$
$zz(pc) + 1 = zz(pc + 1) \wedge vv(pc) = vv(pc + 1))$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \lhd\!\!\!- \{pc + 1 \mapsto ss(pc) \lhd\!\!\!- \{zz(pc) \mapsto Int\}\}$

act3 : $zz := zz \lhd\!\!\!- \{pc + 1 \mapsto zz(pc) + 1\}$

act4 : $vv := vv \lhd\!\!\!- \{pc + 1 \mapsto vv(pc)\}$

end

Event *Iload2* $\widehat{=}$

any

x

where

grd5__t : $x \in 1 \mathinner{..} maxlocalvar$

grd8__t : $zz(pc) < maxstack$

grd9 : $halt = FALSE$

grd10 : $pc < maxpc$

grd12__t : $vv(pc)(x) = Int$

The element on the vaiable x is an Int

grd13_t : $pc + 1 \in dom(ss) \Rightarrow (ss(pc) \lhd \{zz(pc) \mapsto vv(pc)(x)\} = ss(pc + 1) \wedge zz(pc) + 1 = zz(pc + 1) \wedge vv(pc) = vv(pc + 1))$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \lhd \{pc + 1 \mapsto ss(pc) \lhd \{zz(pc) \mapsto vv(pc)(x)\}\}$

act3 : $zz := zz \lhd \{pc + 1 \mapsto zz(pc) + 1\}$

act4 : $vv := vv \lhd \{pc + 1 \mapsto vv(pc)\}$

end

Event *halt* $\widehat{=}$

when

grd1 : $halt = FALSE$

grd10 : $pc < maxpc$

grd2 : $zz(pc) = 0$

then

act1 : $halt := TRUE$

end

Event *new* $\widehat{=}$

when

grd6_t : $zz(pc) < maxstack$

grd7 : $halt = FALSE$

grd8 : $pc < maxpc$

grd9_t : $pc + 1 \in dom(ss) \Rightarrow (ss(pc) \lhd \{zz(pc) \mapsto Uobj\} = ss(pc + 1) \wedge zz(pc) + 1 = zz(pc + 1) \wedge vv(pc) = vv(pc + 1))$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \lhd \{pc + 1 \mapsto ss(pc) \lhd \{zz(pc) \mapsto Uobj\}\}$

act3 : $zz := zz \lhd \{pc + 1 \mapsto zz(pc) + 1\}$

act4 : $vv := vv \lhd \{pc + 1 \mapsto vv(pc)\}$

end

Event *init1* $\widehat{=}$

when

grd8 : $halt = FALSE$

grd9 : $pc < maxpc$

grd11 : $ss(pc)(zz(pc) - 1) = Uobj$

The element on top of stack is an Uobj

grd12_t : $pc + 1 \in dom(ss) \Rightarrow (ss(pc) \lhd\!\!\!- \{zz(pc) - 1 \mapsto Obj\} = ss(pc + 1) \wedge zz(pc) = zz(pc + 1) \wedge vv(pc) = vv(pc + 1))$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \lhd\!\!\!- \{pc + 1 \mapsto ss(pc) \lhd\!\!\!- \{zz(pc) - 1 \mapsto Obj\}\}$

act3 : $zz := zz \lhd\!\!\!- \{pc + 1 \mapsto zz(pc)\}$

act4 : $vv := vv \lhd\!\!\!- \{pc + 1 \mapsto vv(pc)\}$

end

Event *Use* $\widehat{=}$

when

grd7 : $halt = FALSE$

grd8 : $pc < maxpc$

grd11_t : $ss(pc)(zz(pc) - 1) = Obj$

The element on top of stack is an Obj

grd12_t : $pc + 1 \in dom(ss) \Rightarrow (\{zz(pc) - 1\} \lhd ss(pc) = ss(pc + 1) \wedge zz(pc) - 1 = zz(pc + 1) \wedge vv(pc) = vv(pc + 1))$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \lhd\!\!\!- \{pc + 1 \mapsto \{zz(pc) - 1\} \lhd ss(pc)\}$

act3 : $zz := zz \lhd\!\!\!- \{pc + 1 \mapsto zz(pc) - 1\}$

act4 : $vv := vv \lhd\!\!\!- \{pc + 1 \mapsto vv(pc)\}$

end

Event *Astore1* $\widehat{=}$

Version where the rewrite with an Int

any

x

where

grd6__t : $x \in 1 \mathinner{..} maxlocalvar$

grd11 : $halt = FALSE$

grd12 : $pc < maxpc$

grd14__t : $ss(pc)(zz(pc) - 1) = Obj$

The element on top of stack is an Int

grd15__t : $pc + 1 \in dom(ss) \Rightarrow (\{zz(pc) - 1\} \lhd ss(pc) = ss(pc + 1) \land zz(pc) - 1 = zz(pc + 1) \land vv(pc) \lhd\!\!+ \{x \mapsto Obj\} = vv(pc + 1))$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \lhd\!\!+ \{pc + 1 \mapsto \{zz(pc) - 1\} \lhd ss(pc)\}$

act3 : $zz := zz \lhd\!\!+ \{pc + 1 \mapsto zz(pc) - 1\}$

act4 : $vv := vv \lhd\!\!+ \{pc + 1 \mapsto vv(pc) \lhd\!\!+ \{x \mapsto Obj\}\}$

end

Event *Astore2* $\widehat{=}$

Version where the rewrite with le element on the tos

any

x

where

grd6__t : $x \in 1 \mathinner{..} maxlocalvar$

grd11 : $halt = FALSE$

grd12 : $pc < maxpc$

grd14__t : $ss(pc)(zz(pc) - 1) = Obj$

The element on top of stack is an Int

grd15_t : $pc+1 \in dom(ss) \Rightarrow (\{zz(pc)-1\} \lhd ss(pc) = ss(pc+1) \land zz(pc)-1 = zz(pc+1) \land vv(pc) \lhd\!\!+ \{x \mapsto ss(pc)(zz(pc)-1)\} = vv(pc+1))$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \lhd\!\!+ \{pc + 1 \mapsto \{zz(pc) - 1\} \lhd ss(pc)\}$

act3 : $zz := zz \lhd\!\!+ \{pc + 1 \mapsto zz(pc) - 1\}$

act4 : $vv := vv \lhd\!\!+ \{pc + 1 \mapsto vv(pc) \lhd\!\!+ \{x \mapsto ss(pc)(zz(pc) - 1)\}\}$

end

Event *Aload1* $\widehat{=}$

any

x

where

grd5_t : $x \in 1 \mathinner{..} maxlocalvar$

grd8_t : $zz(pc) < maxstack$

grd9 : $halt = FALSE$

grd10 : $pc < maxpc$

grd12_t : $vv(pc)(x) = Obj$

The element on the vaiable x is an Int

grd13_t : $pc + 1 \in dom(ss) \Rightarrow (ss(pc) \lhd\!\!+ \{zz(pc) \mapsto Obj\} = ss(pc + 1) \land zz(pc) + 1 = zz(pc + 1) \land vv(pc) = vv(pc + 1))$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \lhd\!\!+ \{pc + 1 \mapsto ss(pc) \lhd\!\!+ \{zz(pc) \mapsto Obj\}\}$

act3 : $zz := zz \lhd\!\!+ \{pc + 1 \mapsto zz(pc) + 1\}$

act4 : $vv := vv \lhd\!\!+ \{pc + 1 \mapsto vv(pc)\}$

end

Event *Aload2* $\widehat{=}$

any

x

where

grd5_t : $x \in 1 \,..\, maxlocalvar$

grd8_t : $zz(pc) < maxstack$

grd9 : $halt = FALSE$

grd10 : $pc < maxpc$

grd12_t : $vv(pc)(x) = Obj$

The element on the vaiable x is an Int

grd13_t : $pc + 1 \in dom(ss) \Rightarrow (ss(pc) \lhd\!\!\!\!- \{zz(pc) \mapsto vv(pc)(x)\} = ss(pc + 1) \wedge zz(pc) + 1 = zz(pc + 1) \wedge vv(pc) = vv(pc + 1))$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \lhd\!\!\!\!- \{pc + 1 \mapsto ss(pc) \lhd\!\!\!\!- \{zz(pc) \mapsto vv(pc)(x)\}\}$

act3 : $zz := zz \lhd\!\!\!\!- \{pc + 1 \mapsto zz(pc) + 1\}$

act4 : $vv := vv \lhd\!\!\!\!- \{pc + 1 \mapsto vv(pc)\}$

end

END

Annexe C

Modèle du langage Java Card

CONTEXT C

SETS

ELEMENT

CONSTANTS

TYPE

shart

ref

maxByteArrayLength

maxpc

maxstack

maxlocalvar

initlocalvar

AXIOMS

axm1 : $TYPE \subseteq ELEMENT$

axm1_1 : $TYPE = \{shart, ref\}$

axm1_2 : $shart \neq ref$

axm2 : $maxByteArrayLength = 40$

axm3 : $maxpc = 20$

axm4 : $maxstack = 3$

axm5 : $maxlocalvar = 2$

axm6 : $initlocalvar \in 1 \mathinner{..} maxlocalvar \rightarrow TYPE$

END

MACHINE M

SEES C

VARIABLES

pc

ss

vv

zz

halt

ByteArray

INVARIANTS

inv1 : $pc \in 1 \mathinner{..} maxpc$

inv2 : $ss \in 1 \mathinner{..} maxpc \nrightarrow (0 \mathinner{..} maxstack - 1 \nrightarrow TYPE)$

inv3 : $vv \in 1 \mathinner{..} maxpc \nrightarrow (1 \mathinner{..} maxlocalvar \nrightarrow TYPE)$

inv4 : $zz \in 1 \mathinner{..} maxpc \nrightarrow 0 \mathinner{..} maxstack$

inv5 : $halt \in BOOL$

inv6 : $ByteArray \in 0 \mathinner{..} maxByteArrayLength \nrightarrow 0 \mathinner{..} 255$

inv7 : $dom(ss) = dom(vv) \wedge dom(ss) = dom(zz) \wedge dom(vv) = dom(zz)$

inv8 : $pc \in dom(ss)$

EVENTS

Initialisation

begin

act1 : $pc := 1$

act2 : $ss := \{1 \mapsto \varnothing\}$

act3 : $vv := \{1 \mapsto initlocalvar\}$

act4 : $zz := \{1 \mapsto 0\}$

act6 : $halt := FALSE$

act7 : $ByteArray := \varnothing$

end

Event *aconst_null* $\widehat{=}$

any

forms

stackSize

where

grd1 : $halt = FALSE$

grd2 : $pc < maxpc$

grd3 : $card(dom(ByteArray)) + 1 < maxByteArrayLength$

grd4 : $zz(pc) = stackSize$

grd5 : $forms = 1$

grd6_t : $stackSize \leq maxstack - 1$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \lhd\!\!+ \{pc + 1 \mapsto ss(pc) \lhd\!\!+ \{zz(pc) \mapsto ref\}\}$

act3 : $zz := zz \lhd\!\!+ \{pc + 1 \mapsto zz(pc) + 1\}$

act4 : $vv := vv \lhd\!\!+ \{pc + 1 \mapsto vv(pc)\}$

act5 : $ByteArray := ByteArray \lhd\!\!+ \{card(dom(ByteArray)) \mapsto forms\}$

end

Event *pop* $\widehat{=}$

any

forms

stackSize

where

grd1 : $halt = FALSE$

grd2 : $pc < maxpc$

grd3 : $card(dom(ByteArray)) + 1 < maxByteArrayLength$

grd4 : $zz(pc) = stackSize$

grd5 : $forms = 59$

grd_t : $stackSize \geq 1$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \Leftarrow \{pc + 1 \mapsto (\{zz(pc) - 1\} \lhd ss(pc))\}$

act3 : $zz := zz \Leftarrow \{pc + 1 \mapsto zz(pc) - 1\}$

act4 : $vv := vv \Leftarrow \{pc + 1 \mapsto vv(pc)\}$

act5 : $ByteArray := ByteArray \Leftarrow \{card(dom(ByteArray)) \mapsto forms\}$

end

Event *return* $\widehat{=}$

any

forms

ByteArrayLength

where

grd1 : $halt = FALSE$

grd2 : $pc < maxpc$

grd3 : $card(dom(ByteArray)) + 1 < maxByteArrayLength$

grd4 : $forms = 122$

grd5 : $ByteArrayLength = card(dom(ByteArray)) + 1$

then

act1 : $halt := TRUE$

act2 : $ByteArray := ByteArray \Leftarrow \{card(dom(ByteArray)) \mapsto forms\}$

end

Event *sadd* $\widehat{=}$

any

forms

stackSize

where

grd1 : $halt = FALSE$

grd2 : $pc < maxpc$

grd3 : $card(dom(ByteArray)) + 1 < maxByteArrayLength$

grd4 : $zz(pc) = stackSize$

grd5 : $forms = 65$

grd__t : $stackSize \geq 2$

grd2__t : $ss(pc)(zz(pc) - 1) = shart$

grd3__t : $ss(pc)(zz(pc) - 2) = shart$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \Leftarrow \{pc + 1 \mapsto (\{zz(pc) - 1\} \lhd ss(pc))\}$

act3 : $zz := zz \Leftarrow \{pc + 1 \mapsto zz(pc) - 1\}$

act4 : $vv := vv \Leftarrow \{pc + 1 \mapsto vv(pc)\}$

act5 : $ByteArray := ByteArray \Leftarrow \{card(dom(ByteArray)) \mapsto forms\}$

end

Event *sconst_n1* $\widehat{=}$

any

forms

stackSize

where

grd1 : $halt = FALSE$

grd2 : $pc < maxpc$

grd3 : $card(dom(ByteArray)) + 1 < maxByteArrayLength$

grd4 : $zz(pc) = stackSize$

grd5 : $forms = 2$

grd__t : $stackSize \leq maxstack - 1$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \lhd\!\!+ \{pc + 1 \mapsto ss(pc) \lhd\!\!+ \{zz(pc) \mapsto shart\}\}$

act3 : $zz := zz \lhd\!\!+ \{pc + 1 \mapsto zz(pc) + 1\}$

act4 : $vv := vv \lhd\!\!+ \{pc + 1 \mapsto vv(pc)\}$

act5 : $ByteArray := ByteArray \lhd\!\!+ \{card(dom(ByteArray)) \mapsto forms\}$

end

Event *sconst_0* $\widehat{=}$

any

forms

stackSize

where

grd1 : $halt = FALSE$

grd2 : $pc < maxpc$

grd3 : $card(dom(ByteArray)) + 1 < maxByteArrayLength$

grd4 : $zz(pc) = stackSize$

grd5 : $forms = 3$

grd_t : $stackSize \leq maxstack - 1$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \lhd\!\!+ \{pc + 1 \mapsto ss(pc) \lhd\!\!+ \{zz(pc) \mapsto shart\}\}$

act3 : $zz := zz \lhd\!\!+ \{pc + 1 \mapsto zz(pc) + 1\}$

act4 : $vv := vv \lhd\!\!+ \{pc + 1 \mapsto vv(pc)\}$

act5 : $ByteArray := ByteArray \lhd\!\!+ \{card(dom(ByteArray)) \mapsto forms\}$

end

Event *sconst_1* $\widehat{=}$

any

forms

stackSize

where

grd1 : $halt = FALSE$

grd2 : $pc < maxpc$

grd3 : $card(dom(ByteArray)) + 1 < maxByteArrayLength$

grd4 : $zz(pc) = stackSize$

grd5 : $forms = 4$

grd__t : $(stackSize \leq maxstack - 1)$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \Leftarrow \{pc + 1 \mapsto ss(pc) \Leftarrow \{zz(pc) \mapsto shart\}\}$

act3 : $zz := zz \Leftarrow \{pc + 1 \mapsto zz(pc) + 1\}$

act4 : $vv := vv \Leftarrow \{pc + 1 \mapsto vv(pc)\}$

act5 : $ByteArray := ByteArray \Leftarrow \{card(dom(ByteArray)) \mapsto forms\}$

end

Event $sconst\_2 \triangleq$

any

$forms$

$stackSize$

where

grd1 : $halt = FALSE$

grd2 : $pc < maxpc$

grd3 : $card(dom(ByteArray)) + 1 < maxByteArrayLength$

grd4 : $zz(pc) = stackSize$

grd5 : $forms = 5$

grd__t : $(stackSize \leq maxstack - 1)$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \Leftarrow \{pc + 1 \mapsto ss(pc) \Leftarrow \{zz(pc) \mapsto shart\}\}$

act3 : $zz := zz \lhd\!\!+ \{pc + 1 \mapsto zz(pc) + 1\}$

act4 : $vv := vv \lhd\!\!+ \{pc + 1 \mapsto vv(pc)\}$

act5 : $ByteArray := ByteArray \lhd\!\!+ \{card(dom(ByteArray)) \mapsto forms\}$

end

Event *sconst_3* $\widehat{=}$

any

forms

stackSize

where

grd1 : $halt = FALSE$

grd2 : $pc < maxpc$

grd3 : $card(dom(ByteArray)) + 1 < maxByteArrayLength$

grd4 : $zz(pc) = stackSize$

grd5 : $forms = 6$

grd_t : $(stackSize \leq maxstack - 1)$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \lhd\!\!+ \{pc + 1 \mapsto ss(pc) \lhd\!\!+ \{zz(pc) \mapsto shart\}\}$

act3 : $zz := zz \lhd\!\!+ \{pc + 1 \mapsto zz(pc) + 1\}$

act4 : $vv := vv \lhd\!\!+ \{pc + 1 \mapsto vv(pc)\}$

act5 : $ByteArray := ByteArray \lhd\!\!+ \{card(dom(ByteArray)) \mapsto forms\}$

end

Event *sconst_4* $\widehat{=}$

any

forms

stackSize

where

grd1 : $halt = FALSE$

grd2 : $pc < maxpc$

grd3 : $card(dom(ByteArray)) + 1 < maxByteArrayLength$

grd4 : $zz(pc) = stackSize$

grd5 : $forms = 7$

grd_t : $(stackSize \leq maxstack - 1)$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \Leftarrow \{pc + 1 \mapsto ss(pc) \Leftarrow \{zz(pc) \mapsto shart\}\}$

act3 : $zz := zz \Leftarrow \{pc + 1 \mapsto zz(pc) + 1\}$

act4 : $vv := vv \Leftarrow \{pc + 1 \mapsto vv(pc)\}$

act5 : $ByteArray := ByteArray \Leftarrow \{card(dom(ByteArray)) \mapsto forms\}$

end

Event $sconst\_5 \widehat{=}$

any

$forms$

$stackSize$

where

grd1 : $halt = FALSE$

grd2 : $pc < maxpc$

grd3 : $card(dom(ByteArray)) + 1 < maxByteArrayLength$

grd4 : $zz(pc) = stackSize$

grd5 : $forms = 8$

grd_t : $stackSize \leq maxstack - 1$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \Leftarrow \{pc + 1 \mapsto ss(pc) \Leftarrow \{zz(pc) \mapsto shart\}\}$

act3 : $zz := zz \Leftarrow \{pc + 1 \mapsto zz(pc) + 1\}$

act4 : $vv := vv \Leftarrow \{pc + 1 \mapsto vv(pc)\}$

act5 : $ByteArray := ByteArray \lhd\!\!+ \{card(dom(ByteArray)) \mapsto forms\}$

end

Event *sload* $\widehat{=}$

any

forms

index

stackSize

where

grd1 : $halt = FALSE$

grd2 : $pc < maxpc$

grd3 : $card(dom(ByteArray)) + 1 < maxByteArrayLength$

grd4 : $zz(pc) = stackSize$

grd5 : $forms = 22$

grd__t : $stackSize \leq maxstack - 1$

grd1__t : $index \in 1 \mathinner{..} maxlocalvar$

grd2__t : $vv(pc)(index) = shart$

then

act1 : $pc := pc + 1$

act2 : $ss := ss \lhd\!\!+ \{pc + 1 \mapsto ss(pc) \lhd\!\!+ \{zz(pc) \mapsto shart\}\}$

act3 : $zz := zz \lhd\!\!+ \{pc + 1 \mapsto zz(pc) + 1\}$

act4 : $vv := vv \lhd\!\!+ \{pc + 1 \mapsto vv(pc)\}$

act5 : $ByteArray := ByteArray \lhd\!\!+ \{card(dom(ByteArray)) \mapsto forms, card(dom(ByteArray)) + 1 \mapsto index\}$

end

END

Bibliographie

[1] Jean-Raymond ABRIAL.
Modeling in Event-B - System and Software Engineering.
Cambridge University Press, 2010.

[2] Jean-Raymond ABRIAL.
Modeling in Event-B : System and Software Engineering.
Cambridge University Press, 1st édition, 2010.

[3] Jean-Raymond ABRIAL, Michael BUTLER, Stefan HALLERSTEDE, Thai Son HOANG, Farhad MEHTA et Laurent VOISIN.
« Rodin : An Open Toolset for Modelling and Reasoning in Event-B ».
International Journal on Software Tools for Technology Transfer (STTT), 12(6):447–466, 2010.

[4] Frederic AMIEL, Karine VILLEGAS, Benoit FEIX et Louis MARCEL.
« Passive and Active Combined Attacks : Combining Fault Attacks and Side Channel Analysis ».
Dans *Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 92–102, 2007.

[5] « A Concise Summary of the Event-B mathematical toolkit : `http://lfm.iti.uni-karlsruhe.de/download/EventB_Summary.pdf`, avril 2013 ».

[6] « EuroSmart 2012 : `http://www.eurosmart.com/`, avril 2013 ».

[7] « ProB : `http://www.stups.uni-duesseldorf.de/ProB`, avril 2013 ».

[8] « Projet Deploy : `http://www.deploy-project.eu`, avril 2013 ».

[9] « Rodin Users Handbook : http://handbook.event-b.org/current/pdf/rodin-doc.pdf, avril 2013 ».

[10] Christel BAIER et Joost-Pieter KATOEN.
Principles of model checking.
MIT Press, 2007.

[11] Mathieu BARREAUD, Guillaume BOUFFARD, Nassima KAMEL et Jean-Louis LANET.
« Fuzzing on the HTTP protocol implementation in mobile embedded web server ».
C&ESAR 2011, 2011.

[12] Eddy BERNARD, Bruno LEGEARD, Xavier LUCK et Fabien PEUREUX.
« Generation of test sequences from formal specifications : GSM 11-11 standard case study ».
Software : Practice and Experience, 34(10):915–948, 2004.

[13] Anis BKAKRIA, Guillaume BOUFFARD, Julien IGUCHI-CARTIGNY et Jean-Louis LANET.
« OPAL : an open-source global platform Java Library which includes the remote application management over HTTP, ».
Dans *e-Smart 2011*, septembre 2011.

[14] Dan BONEH, Richard A. DEMILLO et Richard J. LIPTON.
« On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract) ».
Dans *EUROCRYPT*, pages 37–51, 1997.

[15] Guillaume BOUFFARD et Jean-Louis LANET.
« The Next Smart Card Nightmare Logical Attacks, Combined Attacks, Mutant Applications and other Funny Things ».
Dans David NACCACHE, éditeur, *Cryptography and Security : From Theory to Applications*, volume 6805 de *Lecture Notes in Computer Science*, chapitre The Next Smart Card Nightmare Logical Attacks, Combined Attacks, Mutant Applications and other Funny Things, pages 405–424. Springer, 2012.

[16] Guillaume BOUFFARD, Jean-Louis LANET et Julien IGUCHY-CARTIGNY.
« Combined Software and Hardware Attacks on the Java Card Control Flow ».
Dans Emmanuel PROUFF, éditeur, *Smart Card Research and Advanced Applications*, volume 7079 de *Lecture Notes in Computer Science*, pages 283–296. Springer, septembre 14th to 16th 2011.

[17] Fabrice BOUQUET, Christophe GRANDPIERRE, Bruno LEGEARD et Fabien PEUREUX.
« A test generation solution to automate software testing ».
Dans *Proceedings of the 3rd international workshop on Automation of software test*, AST '08, pages 45–48. ACM, 2008.

[18] Fabrice BOUQUET, Bruno LEGEARD, Fabien PEUREUX et Eric TORREBORRE.
« Mastering Test Generation from Smart Card Software Formal Models ».
Dans Gilles BARTHE, Lilian BURDY, Marieke HUISMAN, Jean-Louis LANET et Traian MUNTEAN, éditeurs, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 de *Lecture Notes in Computer Science*, pages 70–85. Springer Berlin / Heidelberg, 2005.

[19] Eric BRIER, Christophe CLAVIER et Francis OLIVIER.
« Correlation Power Analysis with a Leakage Model ».
Dans *Cryptographic Hardware and Embedded Systems*, pages 16–29, 2004.

[20] Ludovic CASSET et Jean Louis LANET.
« How to Formally Specify the Java Bytecode Semantics Using the B Method ».
Dans *Proceedings of the Workshop on Object-Oriented Technology*, pages 104–105. Springer-Verlag, 1999.

[21] Jeremy DICK et Alain FAIVRE.
« Automating the Generation and Sequencing of Test Cases from Model-Based Specifications ».
Dans *Proceedings of the First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, FME '93, pages 268–284. Springer-Verlag, 1993.

[22] Stephen N. FREUND et John C. MITCHELL.
« A type system for object initialization in the Java bytecode language ».

SIGPLAN Notices, 33(10):310–327, octobre 1998.

[23] Julien IGUCHI-CARTIGNY et Jean-Louis LANET.
« Developing a Trojan applets in a smart card ».
Journal in Computer Virology, 6:343–351, 2010.

[24] Paul Carl KOCHER.
« Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems ».
Dans *CRYPTO*, pages 104–113, 1996.

[25] Paul Carl KOCHER, Joshua JAFFE et Benjamin JUN.
« Differential Power Analysis ».
Dans *CRYPTO*, pages 388–397, 1999.

[26] Pierre-Alain MASSON, Marie-Laure POTET, Jacques JULLIAND, Régis TISSOT, Georges DEBOIS, Bruno LEGEARD, Boutheina CHETALI, Fabrice BOUQUET, Eddie JAFFUEL, Lionel VAN AERTRICK, June ANDRONICK et Amal HADDAD.
« An Access Control Model Based Testing Approach for Smart Card Applications : Results of the POSÉ Project ».
JIAS, Journal of Information Assurance and Security, 5(1):335–351, 2010.

[27] Barreaud MATHIEU, Julien IGUCHI-CARTIGNY et Jean-Louis LANET.
« Analysis Vulnerabilities in Smart Card Web Server ».
SAR-SSI 2011 Network and Information Systems Security, 2011.

[28] Wojciech MOSTOWSKI et Erik POLL.
« Malicious Code on Java Card Smartcards : Attacks and Countermeasures ».
Dans *Proceedings of the 8th IFIP WG 8.8/11.2 international conference on Smart Card Research and Advanced Applications*, CARDIS '08, pages 1–16. Springer-Verlag, 2008.

[29] Christophe MÉTAYER et Laurent VOISIN.
« *The Event-B Mathematical Language : http://deploy-eprints.ecs.soton.ac.uk/11/4/kernel_lang.pdf, avril 2013* ».

[30] Agnes NOUBISSI, Julien IGUCHI-CARTIGNY, Jean-Louis LANET, Guillaume BOUFFARD et Julien BOUTET.
« Carte à puce : attaques et contremesures ».

Manifestation des Jeunes Chercheurs en Sciences et Technologies de l'Information et de la Communication (MajecSTIC), 2009.

[31] Erik POLL, Joachim van den BERG et Bart JACOBS.
« Formal specification of the JavaCard API in JML : the APDU class ».
Computer Networks, 36(4):407 – 421, 2001.

[32] Hamadouche SAMIYA.
« Étude de la sécurité d'un vérifieur de byte code et génération de tests de vulnérabilité », 2012.
Université M'Hamed Bougara de Boumerdes, Algerie.

[33] Manoranjan SATPATHY, Michael LEUSCHEL et Michael BUTLER.
« ProTest : An Automatic Test Environment for B Specifications ».
Electronic Notes in Theoretical Computer Science, 111(0):113 – 136, 2005.

[34] Aymerick SAVARY, Marc FRAPPIER et Jean-Louis LANET.
« Automatic Generation of Vulnerability Tests for the Java Card Byte Code Verifier ».
Dans *Network and Information Systems Security (SAR-SSI), 2011 Conference on*, pages 1 –7, may 2011.

[35] Aymerick SAVARY, Jean-Louis LANET, Marc FRAPPIER, Tiana RAZAFINDRALAMBO et Josselin DOLHEN.
« VTG - Vulnerability Test Generator, a Plug-in for Rodin ».
Workshop Deploy, février 2012.

[36] Sun MICROSYSTEMS.
« Virtual Machine Specification Java Card Platform, May 2009, `http://www.oracle.com` ».

[37] Mark UTTING et Bruno LEGEARD.
Practical Model-Based Testing : A Tools Approach.
Morgan Kaufmann Publishers Inc., 2007.

[38] Mark UTTING, Alexander PRETSCHNER et Bruno LEGEARD.
« A taxonomy of model-based testing approaches ».
Software Testing, Verification and Reliability, 22(5):297–312, 2012.