Détection de vulnérabilités appliquée à la vérification de code intermédiaire de Java Card

Aymerick Savary

UNIVERSITÉ DE LIMOGES

ECOLE DOCTORALE Science et Ingénierie pour l'Informatique

FACULTÉ DES SCIENCES ET TECHNIQUES

Année : 2016 Thèse N X

Thèse

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE LIMOGES

Discipline : Informatique

présentée et soutenue par

Aymerick Savary

le 30 juin 2016

Détection de vulnérabilités appliquée à la vérification de code intermédiaire de Java Card

Thèse dirigée par Jean-Louis Lanet et Marc Frappier

JURY :

Jean-Louis Lanet	Professeur, Université de Limoges	Directeur
Marc Frappier	Professeur, Université de Sherbrooke	Directeur
Regine Laleau	Professeur, Université Paris-Est Créteil	Rapporteur
Jacques Julliand	Professeur, Université de Besançon	Rapporteur
Christophe Clavier	Professeur, Université de Limoges	Examinateur
Gabriel Girard	Professeur, Université de Sherbrooke	Examinateur

Remerciements

Je tiens à remercier mon directeur canadien, le professeur Marc Frappier de l'Université de Sherbrooke, pour tout le travail accompli durant ces trois années de doctorat ainsi que les deux précédentes. Outre la transmission des connaissances relatives à mes expérimentations, il m'a montré la voie pour devenir un chercheur accompli. Je remercie également mon directeur français, le professeur Jean-Louis Lanet de l'Université de Limoges, qui ma offert l'opportunité de pouvoir travailler avec lui et toute son équipe depuis mes premières années à l'Université. La collaboration avec le professeur Michael Leuschel de l'Université de Düsseldorf, que je remercie, m'a permis d'obtenir des résultats expérimentaux très intéressants. De plus, cette collaboration m'a permis de découvrir une nouvelle méthode de recherche permettant de réunir des chercheurs académiques et industriels. Je remercie Mathieu Lassale pour le travail réalisé, qui m'a permis d'étendre mon cas d'étude. Pour tous ces déplacements, je remercie l'École doctorale de Limoges de m'avoir soutenu financièrement. Finalement, je remercie ma famille de m'avoir soutenu et avoir créé un contexte favorable à la bonne réussite de mon épanouissement depuis mon plus jeune âge.

Table des matières

Table des figures

Chapitre 1 :

Introduction

Contexte

Les outils informatiques sont au cœur des sociétés modernes. Ils nous permettent de travailler plus rapidement et plus efficacement, au point que certaines professions ne peuvent plus s'en passer. Bien que cela nous soit bénéfique, une défaillance informatique, dans une centrale nucléaire par exemple, pourrait entraîner des événements catastrophiques. La sûreté des programmes informatiques ne doit donc pas être négligée. Face à cette problématique, un ensemble de méthodes et d'outils de vérification ont été développés, tels que les tests ou les méthodes formelles. Cependant, certains problèmes ne sont toujours pas résolus de manière satisfaisante.

Les cartes à puce sont actuellement responsables d'une grande partie de la sécurité de certains systèmes vitaux, tels que le système bancaire. Ces systèmes critiques doivent, par conséquent, être protégés contre tous types de défaillance, et en particulier, contre les intrusions perpétrées par des attaquants. Selon la sûreté de fonctionnement, les intrusions sont des fautes dues à l'homme, intentionnelles avec volonté de nuire, opérationnelles, externes et temporaires. De multiples contremesures ont été proposées afin de contrer ces attaques. Cependant, les attaquants disposent de technologies de plus en plus perfectionnées. Il est donc nécessaire de trouver de nouvelles méthodes et de développer de nouveaux outils permettant de prévoir, de détecter et de limiter l'impact de ces intrusions.

Les instituts de certification chargés de vérifier la sécurité d'implémentations n'ont pas toujours accès au code source ou à la version compilée du programme. Dans le cas des cartes à puces, les implémentations sont embarquées. Ainsi, il est seulement possible d'interagir avec ces dernières. Il est donc nécessaire de tester ces programmes en utilisant des stratégies de test en boîte noire. Certains systèmes, de par leur complexité, sont très difficiles à tester et doivent être testés manuellement. Cela est très coûteux et les jeux de tests obtenus ne couvrent qu'une infime partie des défaillances possibles. Le VCI (Vérifieur de Code Intermédiaire, «*bytecode verifier*»), est la première ligne défensive des cartes à puces Java Card. Les méthodes de vérification existantes sont inappropriées pour tester adéquatement des implémentations de VCI embarquées dans des cartes à puce.

Problématique

La vérification de la sécurité des systèmes critiques est primordiale. La sécurité d'un système repose principalement sur le fait qu'il détecte et rejette les tentatives d'intrusion. Cependant, la complexité des implémentations de VCI rend leur vérification difficile. Il est donc nécessaire de déterminer si une méthode de test en boîte noire permettant de vérifier de façon systématique de telles implémentations serait envisageable. Le test de vulnérabilité consiste donc à générer des intrusions afin de détecter la présence de défaillances. De plus, la détection de défaillance doit être précise et permettre d'identifier rapidement son origine, c.-à-d. de trouver facilement la faute.

Méthodologie de recherche

Un modèle est tout d'abord proposé et vérifié à l'aide de preuves et d'exploration de modèle («*model checking*»). Nous utilisons la méthode formelle Event-B pour représenter formellement des parties de la spécification Java Card. Pour simplifier le problème de

départ, un sous-ensemble des mécanismes du VCI est étudié. Une fois les objectifs de test atteints, ce sous-ensemble est étendu. En parallèle de la construction du modèle, la méthode VTG (≪*Vulnerability Test Generation*≫, génération de tests de vulnérabilité) est implémentée. Il est parfois nécessaire d'adapter le modèle ou la méthode. Ces modifications sont réalisées pour prendre en compte de nouvelles caractéristiques du langage Event-B ou du VCI , tout en rendant la méthode toujours plus générique. La méthode proposée ainsi que le cas d'étude sont améliorés empiriquement Les tests obtenus sont utilisés afin de vérifier des implémentations de VCI .

Résultats

Nous proposons une méthode basée sur la mutation de spécification et le MBT (≪*Model-Based Testing*≫, test à base de modèle), appelée VTG. Cette méthode permet de générer tous les tests nécessaires à la vérification d'implémentation du VCI pour les mécanismes étudiés. Une théorie de mutation pour la grammaire Event-B, ainsi qu'une extension des règles de négation que l'on peut trouver dans la littérature, est proposée et partiellement prouvée. Pour la génération des tests, un algorithme de résolution de contraintes, basé sur ProB [1], a été développé en collaboration avec le professeur Michael Leuschel de l'Université de Düsseldorf. ProB est un explorateur de modèle à base de satisfaction de contraintes qui peut être utilisé pour plusieurs langages de spécification, incluant Event-B. De plus, cette méthode est générique et peut être utilisée pour vérifier d'autres types de systèmes.

Le VCI est décomposé en deux parties afin de simplifier son étude. Le premier modèle Event-B représente les contraintes statiques et traite de la vérification de structure. Le second modèle correspond aux contraintes dynamiques de la vérification de type. Ces modèles ont été vérifiés formellement en utilisant des méthodes de preuves et d'exploration de modèle. Ils ont été adaptés afin de fonctionner avec la méthode de génération de tests de vulnérabilité. Différents ensembles de tests ont été produits. Pour la vérification de structure, seuls des tests abstraits ont été produits. Pour la vérification de type, un ensemble de 223 tests concrets a été produit en 45min par le VTG. Ces tests ont permis de caractériser les mécanismes défensifs d'implémentations de VCI embarquées dans des cartes à puce.

Des logiciels ont permis de mettre en œuvre la méthode pour le VCI et de tester des implémentations. Ces outils sont séparés en deux catégories. Les outils pour la méthode sont génériques et peuvent être utilisés pour vérifier différents systèmes. Les outils pour le cas d'étude servent uniquement pour le VCI .

Structure du document

Cette thèse de doctorat est composée de deux grandes parties. La première partie contient : le positionnement par rapport aux travaux existants ainsi que les connaissances essentielles à la compréhension de cette thèse. La seconde partie se divise en trois chapitres. Dans le premier, la méthode proposée est exposée de façon générique. Dans le suivant, la méthode est utilisée afin de résoudre le problème de vérification du VCI . Finalement, le chapitre quatre donne un aperçu des outils développés afin de mettre en application la méthode sur le cas d'étude.

Chapitre 2 :

Revue de littérature

Ce chapitre permet de situer notre problématique et nos contributions par rapport à l'existant. Les cartes à puces sont des systèmes dont la sécurité a été très étudiée. La première section donne un aperçu des différents travaux existants. Dans la section suivante, nous étudierons plus en détail les travaux relatifs à notre cas d'étude, le VCI . La dernière section décrit notre positionnement général.

2.1 Sécurité des cartes à puce Java

Les cartes à puce sont des systèmes embarqués disposant de très peu de capacité de calcul et de stockage. Malgré cela, elles constituent un élément essentiel de la sécurité de systèmes plus grands, tels que les banques ou la téléphonie. Leur sécurité ne doit donc pas être compromise. Dans nos travaux, nous nous intéressons aux cartes à puces Java Card. Cette technologie sera décrite dans la section 3.1.

La sécurité informatique est une perpétuelle compétition entre les attaquants et les défenseurs. Afin d'améliorer la résistance des systèmes face aux attaques, il existe deux catégories de sécurité. La sécurité défensive consiste à proposer des systèmes sécurisés ou à combler les failles de sécurité après leurs publications. La sécurité offensive consiste à découvrir ces failles dans les systèmes sécurisés avant qu'elles ne soient exploitées. Cette dernière méthode permet notamment aux instituts de certification de vérifier la résistance d'un produit de sécurité. Dans nos recherches, nous nous intéresserons à la sécurité offensive appliquée aux cartes à puce Java Card en nous inspirant des attaques existantes.

2.1.1 Les trois classes d'attaques pour Java Card

Dans le contexte des cartes à puce, il existe trois types d'attaques : les attaques physiques, les attaques logiques et les attaques combinées. Les attaques actuelles contre les cartes à puce sont essentiellement des attaques *physiques*, voir [2, 3]. 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 dans un logiciel et de son exploitation. Cela consiste à récupérer des données ou fonctionnalités à partir d'injection de commandes ou de données. 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. Cependant, ces attaques sont plus complexes, demandent souvent une plus grande connaissance du système que pour les attaques physiques et restent généralement liées à une implémentation particulière. Dans le cadre de cette thèse, nous nous intéressons uniquement aux attaques logiques. Celles-ci seront décrites plus précisément dans la section 2.1.2. Plus récemment, une nouvelle classe d'attaques, les attaques *combinées* [4], exploite des attaques physiques et des attaques logiques dans la même attaque.

2.1.2 Les attaques logiques

Les attaques logiques exploitent des défaillances au niveau des logiciels. Ces défaillances peuvent provenir d'une spécification ou d'une implémentation non sûre. Nous

parlerons respectivement *d'attaques de spécification* et *d'attaques d'implémentation*. Les attaques de spécification sont les plus dangereuses, car elles rendent généralement toutes les implémentations non sûres.

Le premier champ d'attaque pour les attaques logiques est similaire à celui que nous trouvons sur les ordinateurs. Pour cela, il suffit de trouver des failles dans l'application afin de récupérer ses données ou ses droits d'exécution. Le but est d'interagir avec l'application en lui envoyant des commandes malveillantes. De nombreux travaux traitent de ce problème. Parmi ceux-ci, nous pouvons citer les attaques d'implémentation de [5, 6], qui ont testé la résistance de serveurs web embarqués dans les cartes à puce *Java Card 3.0 Connected Edition*.

2.1.2.1 Attaque de cartes ouvertes

La mise à disposition des cartes dites *ouvertes* offre un nouveau champ d'attaque. Ces cartes autorisent le chargement de nouvelles applications après délivrance. Le but de ces attaques [7] consiste à installer un programme malveillant dans la carte, qui récupère des données ou des fonctionnalités.

L'attaque sur le mécanisme de transaction [8] tire profit d'une faille dans l'environnement d'exécution Java Card (≪*Java Card Runtime Environment*≫). Si une transaction est avortée, toutes les références vers les objets doivent être nullifiées et le système est restauré. Cependant, le mécanisme de suppression n'est pas clairement spécifié et peut conduire à des implémentations non sûres. Une autre attaque [9] exploite une faille sur les interfaces de partage. Dans la spécification, la vérification de type des arguments des méthodes de partage n'est pas imposée. Ces deux attaques sont des exemples d'attaques de spécification.

L'attaque sur l'éditeur de liens (≪*linker*≫) [10] met en avant l'utilisation d'une faille dans l'implémentation des mécanismes défensifs embarqués. Certaines implémentations ne vérifient pas correctement que tous les décalages de branchement (≪*branching offset*≫) présents dans les méthodes correspondent à des repères (≪*token*≫) valides. Dans leur exploitation, les auteurs proposent d'utiliser cette défaillance pour exécuter un code normalement inaccessible et d'extraire des données dans la mémoire. La spécification est sûre, mais les contraintes imposées par les cartes à puces conduisent à des optimisations de l'implémentation. Ces optimisations conduisent parfois à des fautes de conception. L'attaque EMAN 1 [11] propose d'exploiter l'attaque sur l'éditeur de liens dans un applet (attaquant) pour modifier un autre applet (cible). Cette exploitation permet par exemple de supprimer une vérification de code PIN (≪*Personal Indentifier Number*≫, numéro d'identification personnel). Une autre exploitation de l'attaque sur l'éditeur de liens, EMAN 2 [12] propose de modifier l'entête de la pile d'exécution Java Card. Cela permet par exemple de changer la valeur du pointeur de programme Java (le *jpc*) et de pouvoir exécuter du code avec les droits d'exécution de l'appelant. Ces 3 attaques d'implémentation fonctionnent uniquement sur certaines cartes à puces.

2.1.3 Mécanismes de défense

Différents composants de sécurité permettent d'assurer la résistance aux attaques de la plateforme Java. La spécification définit un certain nombre de composants qui doivent être présents dans une carte à puce Java Card. Ces composants sont regroupés en deux

catégories, les vérifications statiques et les vérifications dynamiques.

2.1.3.1 Vérification statique

La *vérification statique* s'intéresse à vérifier le code avant qu'il ne soit exécuté par le système. Cette vérification représente la première ligne de défense. Le VCI est en charge de vérifier que les fichiers CAP (≪*Converted APplet*≫, format de fichier Java compilé pour cartes à puce Java Card) respectent la norme Java Card avant de pouvoir être exécutés sur une carte à puce. Ce composant est le seul mécanisme de vérification statique défini par la spécification. La figure 2.1 représente les différents comportements possibles. La

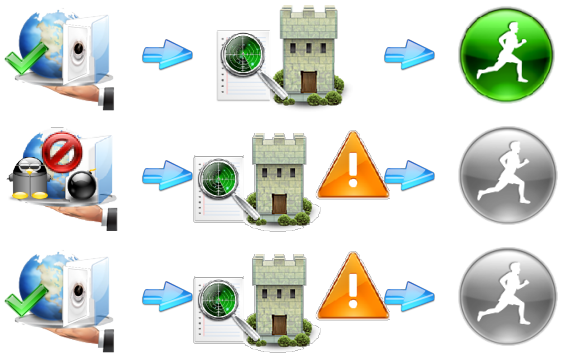

FIGURE 2.1 – Comportement d'un VCI

première ligne correspond à l'envoi d'un fichier CAP valide et autorisé par le VCI à être exécuté. La deuxième ligne correspond à l'envoi d'un fichier CAP invalide. Ce fichier est détecté par le VCI comme ne respectant pas la norme. Il n'est donc pas autorisé à être installé (il ne peut donc pas être exécuté). La dernière ligne correspond au rejet d'un fichier valide. Un fichier acceptable par la norme n'est pas nécessairement accepté par une implémentation particulière du VCI . La norme ne définit que ce qui doit absolument être rejeté. En effet, certaines contremesures peuvent rajouter des contraintes, réduisant l'ensemble des CAP valides à un sous-ensemble. Le processus de vérification est expliqué plus en détail dans la section 3.2.

Les cartes à puces ne disposant pas de grande puissance de calcul et de stockage, cette vérification est très complexe. Bien que les cartes soient de plus en plus puissantes, cette vérification n'est que partiellement embarquée dans les cartes à puce. Les opérations les plus complexes sont effectuées à l'extérieur de la carte par un tiers de confiance. Java Card possède donc un mécanisme de chargement d'applications permettant de fiabiliser l'installation d'applications.

2.1.3.2 Mécanisme de chargement d'application

La norme GlobalPlateform assure le chargement sécurisé d'application par des personnes autorisées. L'utilisation de protocoles cryptographiques permet de vérifier que la personne souhaitant charger une application possède les clefs de chargement. Ces personnes sont reconnues pour être des personnes de confiance. Cependant, avec l'arrivée des cartes ouvertes, le chargement peut être réalisé après délivrance et par des personnes potentiellement malveillantes. Les cartes ouvertes doivent donc assurer elles-mêmes leur défense.

2.1.3.3 Vérification dynamique

La *vérification dynamique* assure la sécurité durant l'exécution de l'application. Le pare-feu est le composant chargé de la vérification dynamique. Bien que l'intégration d'un VCI n'est pas imposée par la spécification Java Card, des portions du VCI sont de plus en plus effectuées dynamiquement. Parmi ces travaux, nous pouvons citer les machines virtuelles défensives [13] et l'utilisation d'automates de sécurité [14].

2.2 Sûreté de fonctionnement

La sûreté de fonctionnement est une notion générique regroupant un ensemble de définitions relatives aux défaillances des systèmes informatiques que nous utiliserons dans ce document. Jean-Claude Laprie, dans [15], établit la définition suivante : «La sûreté de fonctionnement d'un système informatique, est la propriété qui permet à un utilisateur de placer une confiance justifiée dans le service qu'il leur délivre».

Les notions de la sûreté de fonctionnement sont regroupées en trois classes : les entraves, les moyens et les attributs. Les entraves sont les comportements indésirables pouvant survenir. Elles sont décomposées en 3 classes : fautes, erreurs et défaillances. Les moyens permettent de continuer à assurer la sûreté malgré la présence d'entraves. Les attributs représentent les réactions du système attendues en réponse aux entraves et aux moyens.

2.2.1 Deux types de fautes

Dans ces recherches, nous utiliserons le mot faute pour désigner deux concepts. Les fautes que nous cherchons correspondent aux fautes qui ont été introduites dans le système et que nous souhaitons détecter. Les fautes que nous faisons correspondent aux actions que nous entreprenons sur le système pour découvrir des défaillances.

Nous cherchons à détecter des fautes d'implémentation du VCI . Dans la figure 2.2, reprise du livre précédemment cité [15], ces fautes correspondent aux fautes de conception. Elles sont dues à une faute accidentelle ou une faute intentionnelle sans volonté de nuire. N'ayant pas accès au code source du VCI dans nos recherches, ces fautes seront donc permanentes.

Nos tests, correspondant à des attaques, seront des applications données en entrée au VCI . Ces tests seront donc externes et constitués de fautes intentionnellement nuisibles. Nos tests correspondent donc à des intrusions.

2.2.2 Diagnostic

Lorsqu'une défaillance sera détectée, nous devrons procéder à un diagnostic afin de trouver la faute qui a conduit à celle-ci. Ce diagnostic est la seconde étape pour l'élimination des fautes. Dans ces recherches, nous travaillons avec des implémentations en boîte noire du VCI . Le diagnostic permettant de faire le lien entre les défaillances et les fautes recherchées ne sera donc pas traité. De plus, nos tests ne nous permettent pas de conclure à une faute de conception. Cela est notamment vrai dans le cas où le système sous tests subit un dysfonctionnement matériel.

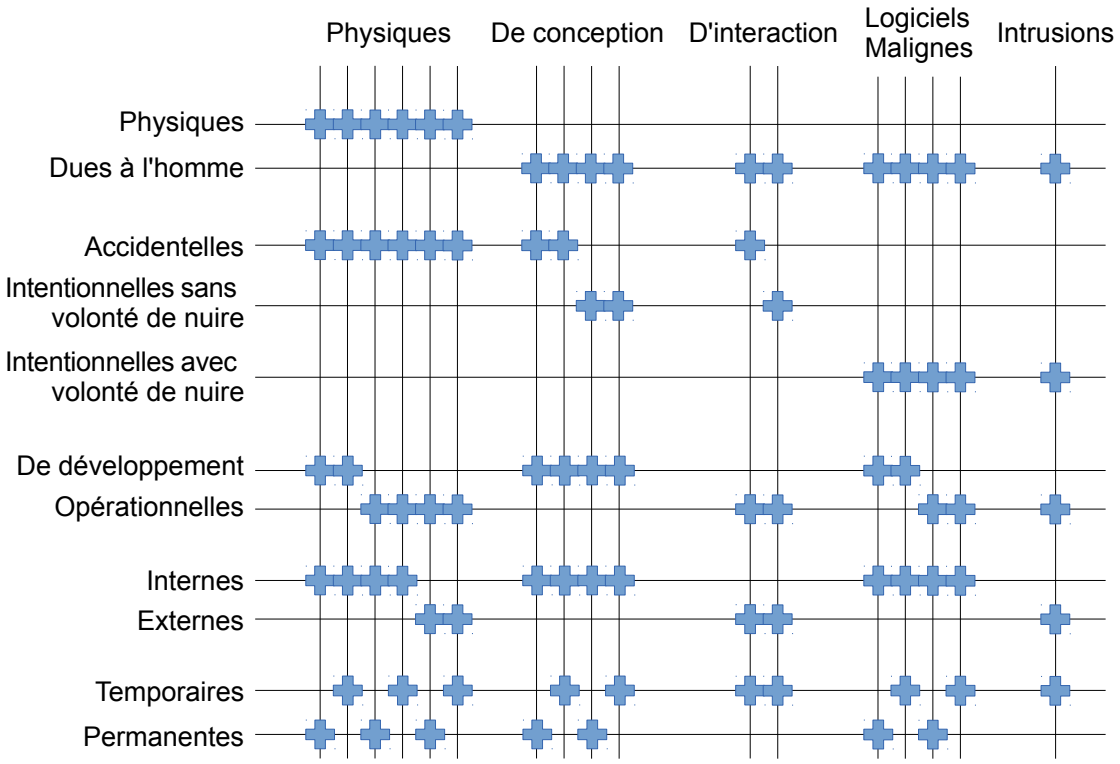

FIGURE 2.2 – Sûreté de fonctionnement, classification des fautes

2.2.3 Défaillances exploitables

Un système est en interaction avec d'autres systèmes (environnement). La défaillance d'un système est une faute pour son environnement. Une défaillance exploitable est une défaillance d'un système qui provoque une défaillance de son environnement. Par exemple, une défaillance d'une carte bancaire qui entraine une défaillance du système bancaire (détournement d'argent) est une défaillance exploitable. Dans ces recherches, nous nous consacrerons uniquement à la découverte de défaillances sans définir si elles sont exploitables ou non.

2.3 Vérification du vérifieur

Le VCI est un composant de sécurité complexe qui a été défini dans [16]. Une définition plus détaillée a été proposée dans [17] permettant de lever un certain nombre d'ambiguïtés. Pour rendre ce composant sécurisé, de multiples travaux ont été nécessaires. Le premier travail a consisté à supprimer les ambiguïtés de la spécification. Dans un second temps, des implémentations sûres ont été proposées. Les méthodes de vérification ont pour but de s'assurer que certains comportements sont respectés. Différentes méthodes de vérification ont permis d'assurer la sécurité et la sûreté d'un tel système.

2.3.1 Vérification formelle

Les méthodes formelles permettent de vérifier qu'un modèle respecte certaines propriétés. Ces modèles décrivent des systèmes à l'aide de notations mathématiques.

L'utilisation de telles notations garantit la non-ambiguïté. Cela permet d'automatiser les processus de raisonnement, tels que les preuves, et ainsi de garantir le processus de vérification.

La spécification d'un système décrit un ensemble de comportements attendus. La frontière caractérisant l'ensemble des comportements attendus peut-être décrite informellement ou formellement. Dans le cas d'une spécification informelle, certaines ambiguïtés rendent cette frontière floue. Nous distinguons donc 3 types de comportements : attendus, ambigus et non attendus. Dans le cas d'une spécification formelle, la frontière est définie à l'aide de formules mathématiques. Les comportements ambigus n'existent donc pas. Le système est ainsi l'ensemble des comportements spécifiés. Par opposition, le système n'est pas l'ensemble des comportements non spécifiés.

2.3.1.1 Formalisations de la spécification

La formalisation d'un système permet de garantir une très grande sécurité (sûreté dans le cas général). L'article [18] recense les principaux modèles relatifs à Java et à Java Card proposés avant l'année 2000. Les premiers modèles, tels que ceux proposés dans [19, 20, 21], ont permis de vérifier la spécification. Ces modèles ont permis de lever un certain nombre d'ambiguïtés et de corriger certains problèmes présents dans les premières versions de la spécification Java Card.

Nous avons vu dans la section 2.1.2 que malgré ses formalisations, certaines attaques contre la spécification sont toujours possibles. Une preuve formelle repose sur un modèle et un ensemble de propriétés. Les preuves permettent uniquement d'assurer que les propriétés sont vraies dans le modèle. La première source de problème provient d'un modèle incomplet qui ne représente pas fidèlement le système. Dans la section suivante, nous verrons un exemple de modèle qui a permis d'obtenir une implémentation prouvée permettant de résoudre en partie ce premier problème. Le deuxième problème provient d'une propriété non exprimée dans l'ensemble des propriétés que l'on doit prouver sur le modèle. L'attaque sur le mécanisme de transaction en est un bon exemple. Ces derniers problèmes sont généralement découverts de façon empirique.

2.3.1.2 Implémentation embarquée prouvée formellement

Le projet européen *MATISSE* [22] et les travaux de Ludovic Casset [23, 24, 25] proposent une implémentation prouvée du VCI embarqué. Le but était de proposer un vérifieur embarqué sûr permettant à une carte à puce d'assurer elle-même sa propre sécurité. Le modèle prouvé utilise le langage B, raffiné jusqu'au B0, puis compilé. La compilation du B0 permet de produire un code qui correspond exactement au modèle. Cette méthode permet donc d'éliminer la première source du problème bien que l'implémentation puisse toujours posséder des failles dues à des propriétés qui n'auraient pas été prouvées.

L'algorithme proposé utilise la méthode de vérification de preuves grâce au PCC (≪*Proof-Carring Code*≫). Les PCC reposent sur l'idée que vérifier une preuve est plus simple que de la construire. Les preuves sont donc générées hors de la carte puis l'application est chargée dans la carte accompagnée de sa preuve. La partie complexe, la génération de la preuve, est donc effectuée sur un ordinateur. La partie simple, la vérification de la preuve, est effectuée dans la carte. L'implémentation du vérifieur

embarqué repose donc toujours sur une opération effectuée par un ordinateur. Cependant, contrairement à un mécanisme de certification, le VCI vérifie l'application.

2.3.1.3 Implémentation basée sur un vérifieur de modèle

Le modèle proposé dans [26] a attiré notre attention, notamment pour la vérification de structure du VCI . Cette modélisation, écrite en Alloy, se base sur l'exploration de modèle (≪*model checking*≫). Elle n'a pas pour but d'être embarquée dans une carte à puce. Elle se base sur une modélisation en deux parties.

Le composant *Declarations* déclare un ensemble de variables représentant les champs du fichier *Class*, ainsi que des variables utiles à l'algorithme du VCI . Le composant *Body* renferme l'ensemble du mécanisme de vérification du VCI . Ces deux composants représentent la première partie de leur méthode. Cette partie est fixe et ne dépend pas de l'application que l'on veut vérifier. Le composant *Initialization* est vide par défaut. Il est rempli pour représenter une application à vérifier à l'aide du programme *Class2Alloy Translator*.

2.3.2 Test d'implémentation

Dans la section précédente, nous avons vu que l'utilisation de modèle formel permettait de produire des systèmes sûrs par construction. Cependant, notre problématique consiste à vérifier une implémentation et non à en proposer une. Les techniques de vérification décrites précédemment ne sont donc pas adaptées. De plus, même un système développé formellement peut contenir des fautes. En effet, les propriétés exprimées ont été prouvées, mais il est généralement impossible de savoir si l'ensemble des propriétés exprimées est suffisant ou complet. Dans ces recherches, nous souhaitons tester la résistance d'une implémentation relative à des tentatives d'intrusion telles que décrites dans la section 2.1.2.

La méthode classique de test consiste à construire des tests unitaires basés sur le code source de l'implémentation. Des outils tels que JUnit permettent d'automatiser les vérifications. Cependant, il est difficile de déterminer la qualité d'un tel ensemble de tests. Pour cela, la mutation de programme permet de mesurer la capacité d'un ensemble de tests à découvrir de potentielles fautes. L'idée consiste à introduire des fautes dans l'implémentation puis à vérifier si l'ensemble de tests est capable de détecter la défaillance. Cette technique a été appliquée à différents langages de programmation [27] dont : Fortran [28], C [29], ADA [30] et Java [31].

Bien qu'intéressante, cette technique possède un défaut majeur, la génération de mutants sémantiquement équivalents. En effet, certaines mutations produisent des programmes syntaxiquement différents, mais sémantiquement équivalents. Prenons par exemple un système tolérant aux fautes et composé de 3 sous-systèmes. Chacun des sous-systèmes effectue le même calcul, mais de trois façons différentes. Après calcul, le système compare les résultats. Si au moins 2 des sous-systèmes ont la même réponse, le système renvoie cette réponse. Dans ce système, si une faute est introduite par mutation dans l'un des sous-systèmes, le comportement du système n'est pas modifié. La mutation de programme n'est donc pas complètement adéquate pour déterminer la qualité d'un ensemble de tests pour des systèmes complexes.

Les techniques de vérification décrites précédemment nécessitent le code source ou

la version binaire du SUT (≪*System Under Test*≫, système sous test). Selon notre problématique, les implémentations sous test étant embarquées, nous ne possédons ni le code source ni le binaire. Le seul moyen de tester l'application consiste à interagir avec elle. Des techniques de test en boite noire doivent donc être employées. Parmi ces techniques, seuls des tests manuels ont été réalisés pour vérifier le VCI . Les travaux [32, 33] proposent deux attaques contre le VCI .

2.3.3 Test à base de modèle

Le MBT (≪*Model-Based Testing*≫, test à base de modèle), [34, 35] est une technique permettant d'extraire des tests à partir d'un modèle formel. Un modèle formel représente généralement une abstraction du système, c.-à-d. le modèle représente ce que doit faire le système, mais pas comment il doit le faire. Cette technique sera plus largement décrite dans la section 3.4.

Différents paradigmes permettent de construire des modèles formels. Le type de test que l'on peut extraire dépend grandement du paradigme choisi. Il est donc important d'adapter le modèle en fonction des tests que l'on souhaite générer. Bien que la plupart des modèles pour le MBT soient en UML [36] ou (E)FSM (≪*Extended Finite State Machine*≫), il existe des travaux traitants d'autres langages : Alloy et SAT [37], Timed Automata and Uppaal [38, 39] ou encore Event-B [40]. Une taxonomie [41] a été proposée et permet de choisir le paradigme de modélisation le plus adapté au SUT. Plus récemment, une taxonomie dédiée au test de sécurité a été proposée [42].

Dans le domaine des cartes à puces, différents travaux utilisent le test à base de modèle. La thèse [43] propose une méthode générale permettant de tester les applications Java Card. Le projet POSÉ [44] propose de tester le système de gestion de répertoire et de fichier pour carte à puce.

Le MBT est une technique permettant d'extraire un sous-ensemble de comportement du modèle. Pour extraire des tests relatifs aux intrusions, le modèle doit représenter ces intrusions. Cependant, il est généralement impossible de modéliser l'ensemble des intrusions. Le MBT ne permet donc pas de vérifier la résistance aux attaques des implémentations de VCI . Il est nécessaire de trouver une technique capable de générer automatiquement des modèles d'intrusion.

2.3.3.1 Outils de MBT

L'outil CertifyIt, dont une description est donnée dans [45], permet de guider la recherche de tests selon certains critères. Ce logiciel est développé par la société française Smartesting notamment spécialisée dans le test d'application pour carte à puce. Comme décrit dans leur article [46], leurs modèles se basent sur la notation UML (≪*Unified Modeling Language*≫) accompagnée de formules OCL (≪*Object Constraint Language*≫). Ces dernières permettent de formaliser les modèles UML. Pour sélectionner leurs critères, ils se basent sur une animation du modèle combinée avec des propriétés statiques et dynamiques.

ProB est un outil d'exploration de modèle basé sur Prolog développé par l'Université de Düsseldorf. Il a été partiellement adapté pour le MBT [47, 48, 49] et des portions de l'outil ProTest [48] y ont été intégrées. Cet outil a comme principal avantage de couvrir l'ensemble des utilisations dont nous avons besoin dans nos recherches. Il permet de vérifier

le modèle et de générer des tests. Bien que les algorithmes de génération de tests existants soient simples, ils nous ont permis de réaliser nos premières expérimentations. La mise à disposition d'une API ainsi que l'ensemble du code source nous ont permis de proposer nos propres algorithmes de MBT. Ces derniers sont décrits dans la section 4.5.

2.3.4 Mutation de spécification

La mutation de spécification permet de transformer automatiquement des modèles formels. Contrairement à la mutation de programme, certaines propriétés peuvent être vérifiées sur le modèle originel, sur les modèles mutants et sur le processus de mutation. Le premier travail a été réalisé sur des représentations formelles de programmes [50]. Les techniques de mutation ont ensuite été utilisées dans le contexte de tolérance aux fautes afin de vérifier si une spécification est sûre. Les techniques de mutation ont été appliquées sur différents langages : OCL [51], CSP [52] et l'utilisation combinée de SMV et de CLT [53].

Cette méthode a été utilisée par [54] pour vérifier certaines parties du VCI . Les auteurs s'intéressent au problème de la vérification des branchements. Ce travail est complémentaire à celui présenté dans cette thèse.

2.4 Positionnement général

Dans ces recherches, le moyen utilisé sera l'élimination des fautes par des méthodes de test. Ces tests, intrusions, correspondent aux attaques décrites en section 2.1. Le VCI , décrit dans la section 2.1.3, est un composant essentiel dans la sécurité des cartes à puces Java Card. Dans ces recherches, nous améliorons la sécurité relative à des tentatives d'intrusion d'implémentations du VCI . Comme nous l'avons vu dans la section 2.3.1, de nombreux travaux ont permis de vérifier la spécification et de produire des implémentations sûres. Ainsi, ces implémentations restent sûres face à des attaques logiques. Cependant, les implémentations embarquées ne sont pas toutes basées sur ces travaux. D'autres méthodes plus traditionnelles, ne comprenant pas de vérification systématique ou formelle, peuvent être utilisées pour produire des VCI . Des attaques d'implémentation peuvent rendre ces VCI non sûrs. Pour les vérifier, nous avons choisi d'utiliser des techniques de test en boite noire en nous inspirant des attaques logiques décrites dans la section 2.1.2. L'ensemble de tests généré contient des tests d'intrusion permettant de vérifier le bon fonctionnement d'implémentation du VCI face à des tentatives d'intrusion. De plus, cet ensemble de tests est générique et peut être appliqué à toute implémentation du VCI .

Les méthodes de tests d'intrusion en boite noire sont actuellement trop chères ou pas assez performantes. Dans le cas du VCI , les possibles intrusions étant trop nombreuses et diversifiées, elles ne peuvent pas toutes être modélisées formellement. Le test manuel, bien que très précis, possède un rapport (couverture/effort requis) peu intéressant. Le test à base de modèle décrit dans la section 2.3.3 est la méthode automatique la plus précise, mais nécessite d'avoir modélisé l'ensemble des intrusions possibles. Cet ensemble est généralement trop vaste et certaines intrusions ne sont pas forcément connues. Il est donc nécessaire de trouver une méthode permettant d'explorer des comportements non spécifiés.

Pour découvrir les modèles de fautes, les techniques de mutation peuvent être utilisées. Comme nous l'avons vu dans les sections 2.3.2 et 2.3.4, la mutation est utilisée pour le test de programme ou pour vérifier la tolérance aux fautes. Nous proposons d'adapter les techniques de mutation de spécification et de test à base de modèle, au problème de la vérification du VCI .

Afin de résoudre le problème de la vérification du VCI , la méthode de génération de tests de vulnérabilité proposée se base sur la mutation de modèle formel et le test à base de modèle. Cette méthode prenant un modèle en entrée, nous devons construire un modèle du VCI compatible avec celle-ci. Notre modèle s'inspire de modèles décrits dans la section 2.3.1. Cependant, leur but étant différent, notre modélisation du VCI doit être adaptée aux techniques utilisées. Nos modèles ne permettront pas de produire un vérifieur embarqué. De plus, notre modèle de la vérification de structure est inspiré de l'architecture du modèle Alloy. La séparation des déclarations de champs, des contraintes imposées par le VCI et des valeurs nous permettra de générer nos tests plus facilement. Après étude des différents langages disponibles, nous avons choisi d'utiliser le langage Event-B qui est décrit dans la section 3.3. La mutation de spécification n'a jamais été appliquée au langage Event-B. Pour effectuer ces mutations, les plateformes Rodin et ProB mettent à disposition des API permettant d'implémenter la méthode de mutation de spécification proposée. Le processus de mutation s'effectue sur des variables de type plus complexes que les travaux existants. De plus, de nouvelles règles de mutation ont été définies, permettant d'ouvrir le champ des mutations possibles. ProB est un outil d'exploration de modèle que nous avons adapté pour générer les tests. Peu de travaux traitent du test à base de modèle en utilisant ProB.

Chapitre 3 :

Fondements

Dans ce chapitre sont présentées les notions essentielles à la bonne compréhension du travail présenté. Les deux premières sections introduisent les notions relatives au cas d'étude, les cartes à puce. Les notions en rapport avec la méthode proposée sont décrites dans les deux sections suivantes.

3.1 Java Card

Les cartes à puces sont des systèmes largement utilisés : cartes bancaires, cartes SIM ou encore carte d'authentification. De par leur petite taille, leur puissance de calcul et de stockage est limitée. La spécification Java Card [55] intègre un sous-ensemble des technologies Java adapté pour les contraintes des cartes à puces. Les programmes sont généralement des «*applets*» bien que depuis la version 2.2.2 il soit possible de charger des «*servlets*».

Dû au peu de ressources disponibles sur les cartes à puce, la JCVM («*Java Card Virtual Machine*», machine virtuelle Java Card), que l'on peut trouver sur une station de travail a été séparée, comme on peut le voir sur la figure 3.1, en deux :

— une partie à l'extérieur de la carte, généralement sur un ordinateur. Cette partie comporte un convertisseur ainsi qu'un vérifieur de code intermédiaire.
— une partie dans la carte correspond à l'interpréteur de code qui se charge d'exécuter les applications.

Le cycle de vie d'un programme Java Card est représenté par la figure 3.1. La ligne en pointillé permet de distinguer les éléments en dehors de la carte des éléments internes à la carte. La première étape consiste à compiler le code source Java, puis le convertir en fichier CAP et enfin le vérifier à l'aide du VCI . L'application est ensuite transmise à la carte à l'aide de protocoles définis par la norme GlobalPlateform. Elle est ensuite installée et prête à être utilisée. Lors de l'installation, des vérifications peuvent être effectuées par la carte.

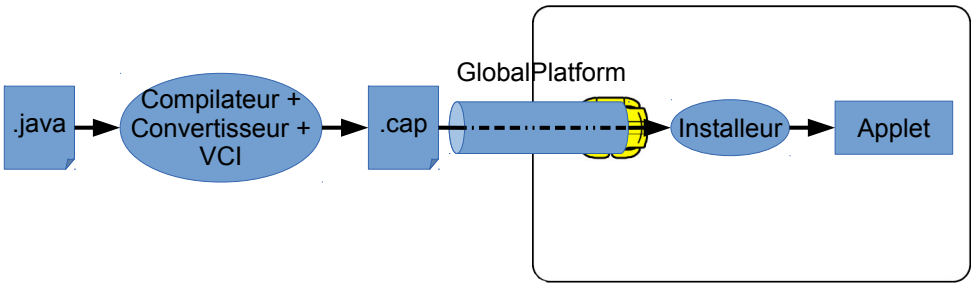

FIGURE 3.1 – Cycle de vie d'un programme Java Card

3.2 Vérifieur de code intermédiaire Java Card

Comme nous l'avons décrit dans la section 2.1.3, le VCI représente la partie statique de vérification des applications Java. Une *application acceptable* est une application qui respecte la spécification par opposition à une *application rejetable* qui ne la respecte pas. Une *application acceptée* est une application considérée comme valide par une implémentation du VCI par opposition à une *application rejetée*. Le rejet d'une

application acceptable est un comportement normal et ne représente pas une défaillance du VCI . En effet, la spécification définit un mécanisme de vérification qui est une approximation sûre et autorise à rejeter plus d'applications si nécessaire (dans le cas de contre-mesure par exemple). En revanche, l'acceptation d'un programme rejetable est problématique. Ce comportement, représenté par la figure 3.2, correspond à une défaillance susceptible de conduire à des failles de sécurité. Nous nous intéressons à ce dernier comportement.

FIGURE 3.2 – Comportement fautif d'un VCI

Afin de vérifier les applications, le VCI procède en deux étapes, la vérification de structure puis la vérification de type. Si la vérification de structure échoue, le VCI rejette l'application sans effectuer la vérification de type. Si elle réussit, l'application est ensuite envoyée au vérifieur de type. L'application est acceptée si le vérifieur de type l'accepte.

3.2.1 Vérification de structure

La structure des applications envoyées à la carte doit respecter un certain format de fichier que l'on appelle CAP. Ce fichier est constitué de douze composants. Chaque composant est constitué de plusieurs champs. En plus de ces contraintes structurelles, le contenu de chaque champ doit respecter un certain nombre de propriétés. Ces propriétés sont décomposées en deux catégories, les contraintes internes et les contraintes externes. Une propriété sur un ou plusieurs champs d'un même composant est une contrainte interne. Une propriété faisant intervenir des champs de composants différents est appelée contrainte externe. La vérification de structure du VCI correspond donc à vérifier que toutes ces contraintes sont respectées. Si une seule contrainte est invalide, la vérification s'arrête et le fichier CAP est rejeté.

3.2.2 Exemple de vérification de structure

Si nous prenons par exemple le composant *Applet*. Il est défini que le champ *size* doit représenter 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*. Nous avons besoin d'autres composants afin de valider ou invalider la propriété, ce qui correspond à une contrainte externe.

3.2.3 Vérification de type

La vérification de type analyse le comportement de l'application par une exécution symbolique. Pour cela, elle récupère l'ensemble des méthodes dans le *Method Component*. Chaque méthode est traitée individuellement par une analyse de son tableau d'octets (≪*bytecode*≫). Chaque instruction constituant ce dernier possède une précondition et

une post-condition définie dans la spécification Java Card [55]. Une application sera donc considérée comme valide si pour toutes les méthodes, toutes les préconditions sont satisfaites. Cependant, les informations présentes dans le fichier CAP sont insuffisantes pour vérifier que toutes les préconditions sont bien respectées. Chaque méthode est donc symboliquement exécutée afin de reconstituer les informations manquantes. Les préconditions ne portant que sur le type de la mémoire (et non les valeurs de celle-ci), cette exécution reconstitue uniquement le typage des structures de données. L'algorithme 1 décrit cette vérification, telle que spécifiée dans [16]. La variable p représente le programme à vérifier. La variable fs représente l'état de la pile d'opérandes et des variables locales pour chaque instruction du programme p. Cet état ne contient que le type des variables et des opérandes de la pile. La variable b est un vecteur de booléens : pour chaque chaque instruction i du programme, si $b(i) = TRUE$, alors la vérification de type pour i doit être faite (ou refaite si son instruction prédécesseur a été refaite). Au départ, seule la première instruction est à vérifier (ie, $b(10) = TRUE$). Ensuite, tous les successeurs de cette instruction seront visités, en mettant leur $b(i)$ à $TRUE$. Les branchements induiront des boucles, et les instructions des boucles seront visitées plusieurs fois. La ligne 12 ne calcule que les nouvelles valeurs des types des variables locales et de la pile, et propage ces valeurs aux instructions successeurs de l'instruction courante. Une instruction séquentielle n'a qu'un successeur, alors qu'un branchement peut en avoir plusieurs. Pour propager, on calcule le supremum ($post \bigvee fs(s)$) dans la hiérarchie de types entre le type calculé en résultat de l'instruction courante, et le type de chacun de ses successeurs. Au départ, on initialise le type d'une instruction avec le type spécial $\perp$ qui est l'élément minimal de la hiérarchie de types. Un programme est considéré comme invalide si une de ses instructions est appelée avec un type incorrect par rapport à sa spécification ou bien si le supremum est l'élément maximal de la hiérarchie de types, qui représente des types incompatibles.

La machine virtuelle Java utilise une trame (¡¡frame¿¿) pour représenter un appel de méthode. Une trame contient les variables locales de la méthode et la pile d'opérandes qui est utilisée pour évaluer les expressions apparaissant dans la méthode. Dans la suite, nous utiliserons simplement le terme pile pour désigner la pile d'opérandes.

3.2.4 Exemples de vérification de type

Pour illustrer ce processus, prenons quatre exemples d'applications, représentées par leurs tableaux d'octet. Les deux premiers sont des applications linéaires et les deux derniers comportent des points de branchement. Nous utiliserons les instructions suivantes :

- *aconst_null* : empile une référence nulle au sommet de la pile,
- *ifeq X* : modifie le flot de contrôle en fonction de la valeur du *short* au sommet de la pile [1],
- *pop* : supprime l'élément au sommet de la pile,
- *return* : retourne à l'appelant,
- *s2b* : tronque le *short* au sommet de la pile et étend son signe,
- *sadd* : additionne les deux éléments de type *short* au sommet de la pile,
- *sconst_N* : empile le *short* N (m1 représente -1).

1. Si l'élément au sommet de la pile est égal à 0, l'interprétation du tableau d'octet continue à l'adresse *adresseCourante* + *Offset*. Sinon l'exécution continue à l'adresse *adresseCourante* + 1.

Algorithm 1 Patron de l'algorithme du VCI

```
1: input p : 1..maxpc → JBC_INSTRUCTION
2: var fs : 1..maxpc → FRAME_STATE
3: var b : 1..maxpc → BOOLEAN
4:
5: procedure BYTECODEVERIFIER(p)
6:     fs := ({1} × {InitialStateFrame}) ∪ (2..maxpc × {⊥})
7:     b := ({1} × {true}) ∪ (2..maxpc × {false})
8:     while ∃i : i ∈ 1..maxpc · b(i) do
9:         pick some i such that b(i)
10:        b(i) := false
11:        if p(i) satisfies the precondition defined in [55] then
12:            post := frame state obtained by executing p(i) on fs(i)
13:            S := successor instructions of p(i) in p
14:            if ∀s ∈ S · post is type compatible with fs(s) then
15:                b := b ⩤ (λs · s ∈ S ∧ post ≠ fs(s) | true)
16:                fs := fs ⩤ (λs · s ∈ S | post ⋁ fs(s))
17:            else
18:                return program invalid
19:            end if
20:        else
21:            return program invalid
22:        end if
23:    end while
24:    return program valid
25: end procedure
```

L'exemple de la figure 3.3 est une application acceptable. Le tableau représente l'état de la pile avant exécution et après exécution de chaque instruction. Le deuxième exemple de la figure 3.4 est rejetable, car la précondition de l'instruction *sadd* n'est pas respectée. La précondition de l'instruction *sadd* impose que les deux éléments en sommet de pile soient de type short.

La figure 3.5 est un exemple de programme valide avec deux chemins d'exécution engendrés par le branchement conditionnel *ifeq* 3 à l'instruction 3. Dans le premier chemin, on retrouve l'entier 0 au sommet de la pile, ce qui fait brancher l'exécution du programme à l'instruction 6, *pop*, qui n'a pas de précondition sur la valeur au sommet de la pile. Le deuxième chemin d'exécution, qui consiste à ne pas brancher à l'instruction 6, et qui n'est pas accessible dans ce cas particulier, vu que le sommet contient l'entier 0, sera quand même analysé par le VCI , car le VCI ne simule pas le contenu de la pile, mais seulement son typage. Le deuxième chemin enlève le *short* restant du sommet de la pile et y ajoute une référence null. À l'instruction 6, dépendamment du chemin parcouru durant l'exécution, on retrouve soit un *short* au sommet de la pile, soit une référence. Comme ces deux types n'ont pas d'ancêtre commun dans la hiérarchie de typage de Java, nous utilisons le type spécial *Top* pour représenter cela dans la colonne Inférence, qui contient le type le plus général représentant le contenu de la pile à une instruction donnée. Puisque l'instruction no 6 est un *pop*, cela ne pose pas de problème de typage.

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

FIGURE 3.3 – Code intermédiaire valide du premier programme

PC	Instructions	Pile	
		avant	après
1	*aconst_null*	∅	*ref*
2	*sconst_1*	*ref*	*ref*; *short*
3	*sadd*	*ref*; *short*	rejet
4	*pop*	rejet	rejet
5	*return*	rejet	rejet

FIGURE 3.4 – Code intermédiaire invalide du deuxième programme

L'exemple de la figure 3.6 représente un programme invalide. Il est quasi identique à celui de la figure 3.5, sauf pour l'instruction no 6, qui est une instruction *s2b*, et qui requiert un élément de type *short* en sommet de pile. Les deux chemins produisent des types différents, représentés par le type *Top*. La précondition de l'instruction *s2b* est respectée pour le chemin 2, mais elle ne l'est pas pour le chemin 1. Comme l'un des chemins ne respecte pas la précondition de *s2b*, le programme est rejeté.

3.3 La méthode Event-B

La méthode Event-B [56] est une méthode formelle permettant de modéliser des systèmes abstraits. Elle se base sur une notation pré/post-condition ainsi que la théorie des ensembles. La représentation des différents niveaux d'abstraction du système est contrôlée

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*	∅		∅		

FIGURE 3.5 – Code intermédiaire valide du troisième programme

PC	Instructions	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	

FIGURE 3.6 – Code intermédiaire invalide du quatrième programme

à l'aide de raffinements prouvés.

Un modèle Event-B est un ensemble de contextes et de machines. Les contextes représentent la partie statique en définissant les constantes et les axiomes (contraintes sur ces constantes). Ces derniers sont mathématiquement désignés par le symbole *Axm*. Les machines représentent la partie dynamique en définissant les variables, les invariants (contraintes sur les variables) et les événements. Ils sont représentés respectivement par les symboles *Inv* et *Evt*.

Trois types de relations existent entre les composants d'un modèle. Un contexte peut étendre un ou plusieurs contextes. Cette relation permet d'inclure récursivement tous les éléments des contextes fils dans un contexte parent. Une machine peut voir les valeurs des constantes définies par un ou plusieurs contextes. Finalement le raffinement permet de définir une relation de raffinement Event-B entre une machine abstraite et une machine concrète.

Les événements sont composés de deux parties. La *garde* représente les conditions permettant de déterminer si l'événement peut survenir [2]. L'*action* représente le changement de l'état du système si l'événement est exécuté. Ces deux éléments sont respectivement représentés par *Grd* et *Act*.

Le raffinement d'un événement abstrait par plusieurs événements concrets permet d'étendre les caractéristiques abstraites. Toutes les informations abstraites, telles que les gardes ou les actions, sont conservées. La preuve de raffinement permet de s'assurer de la conservation des propriétés déjà prouvées ainsi que de la compatibilité des informations concrètes avec les informations abstraites. Les propriétés à vérifier peuvent ainsi être ajoutées dans chaque raffinement. Le dernier raffinement vérifiera toutes les propriétés abstraites introduites dans les raffinements précédents.

3.3.1 Vérification d'un modèle

Un certain nombre de vérifications doivent être effectuées afin de s'assurer que le modèle respecte les propriétés définies par la méthode Event-B. Il existe principalement deux types de méthodes applicables à un modèle en Event-B : la preuve et l'exploration de modèle. Dans le premier cas, des prouveurs de théorèmes basés sur des règles d'inférences sont utilisés. Différents prouveurs sont à notre disposition dans la plateforme Rodin. Bien

2. Si la garde n'est pas satisfaite, l'événement ne peut pas survenir.

que la preuve soit la méthode de vérification la plus complète, certaines propriétés restent très complexes à prouver et mal outillées à l'heure actuelle. Pour l'exploration de modèle, ProB permet d'animer et de vérifier le respect des contraintes. Des propriétés inaccessibles au prouveur, telles que des formules temporelles LTL ou CTL, peuvent ainsi être vérifiées.

3.4 Test à base de modèle

Le MBT [35] est une technique permettant d'extraire un sous-ensemble de comportement de modèles. Cette extraction est guidée par certains critères choisis en fonction des objectifs de test. Des algorithmes de recherche de solutions sont utilisés afin de rechercher des tests permettant de satisfaire les critères de tests. Lorsqu'une solution est trouvée, les informations permettant de constituer le test sont extraites. Étant basé sur une abstraction du système, l'ensemble de tests généré est abstrait et nécessite une concrétisation pour pouvoir être interprété par le SUT.

Les critères de couverture caractérisent les propriétés que l'on souhaite vérifier. Dans un bon ensemble de tests, tous les objectifs de test doivent être satisfaits. L'utilisation de modèles formels, ainsi que de ces critères, permet de vérifier formellement que l'ensemble de tests satisfait tous les objectifs de test. Il existe différents critères de tests qui peuvent être combinés pour satisfaire les objectifs de test. Chaque critère pourra être totalement respecté, non respecté, ou partiellement respecté. Ce dernier cas représente un objectif de test nécessitant la génération de plusieurs tests et où seuls certains tests ont été obtenus. Il existe principalement deux types de critères de couverture, exposés dans les sections suivantes. Pour simplifier les explications et la mise en relation avec les travaux existant dans le domaine du MBT, les termes ne sont pas traduits en français.

3.4.1 Critères de couverture des données

Les critères de couverture des données («*data coverage criteria*») peuvent être utilisés pour 2 objets : des données ou des contraintes. En Event-B, les données sont les constantes, les variables et les paramètres des événements. Les contraintes sont les axiomes et les gardes. Dans le cas des variables, le modèle est un prédicat. Pour les événements, cela représente les contraintes que doit satisfaire l'événement pour être conservé dans l'ensemble des tests.

Les valeurs conservées dans l'ensemble de tests final seront extraites de l'espace d'états atteignables généré par l'algorithme de recherche de solution. Pour certains modèles, l'espace d'état ne peut pas toujours être couvert. Si toutes les valeurs possibles pour la donnée ont été générées par l'algorithme de recherche de solution, le critère de test est satisfait.

3.4.1.1 Pour une seule donnée

Une donnée peut posséder plusieurs contraintes. Si au moins une contrainte est respectée, la valeur pour la donnée sera conservée. Contraintes de données :

- *One value* : la première valeur valide trouvée.
- *Min value* : la valeur la plus petite trouvée.
- *Max value* : la valeur la plus grande trouvée.
- *Explicite values (values)* : liste explicite de valeurs à tester.

— *Random values (nbr)* : *nbr* représente le nombre de valeurs valides prises au hasard (ce ne sont pas nécessairement les *nbr* premières trouvées).
— *Boundary values (boundWidth)* : les valeurs aux bornes, *boundWidth* représente la largeur des bornes.
— *Predicate value (pred)* : toute valeur qui satisfait le prédicat *pred*.
— *All values* : toutes les valeurs trouvées.

3.4.1.2 Pour une combinaison de données

Un groupe de données peut posséder plusieurs contraintes et pourra contenir une ou plusieurs données. Si au moins une contrainte de groupe est respectée et que toutes les données qui le composent ont au moins une contrainte respectée, les valeurs seront conservées. Contraintes de combinaison de données :

— *Pairwise dependances* : on considère les combinaisons de données deux à deux.
— *Nwise dependances (nbr)* : combinaisons de 2..*nbr* données.
— *All combinations coverage* : toutes les combinaisons possibles de données.

3.4.1.3 Pour un prédicat

Un prédicat peut posséder plusieurs contraintes. Si au moins une contrainte est respectée, la valeur pour la donnée sera conservée (un prédicat est composé de conditions). Contraintes de prédicat :

— *One value* : une valeur qui satisfait le prédicat.
— *Multidimensional boundaries coverage* : cherche indépendamment les bornes de chaque donnée.
— *All bondary values* : toutes les valeurs aux bornes.
— *Random values (nbr)* : *nbr* représente le nombre de valeurs valides prises au hasard (ce ne sont pas les nbr premières trouvées).
— *All values* : toute valeur qui satisfait le prédicat.

3.4.1.4 Pour une combinaison de prédicats

Un groupe de prédicats peut posséder plusieurs contraintes et pourra contenir un ou plusieurs prédicats. Si au moins une contrainte de groupe est respectée et que tous les prédicats qui le composent ont au moins une contrainte respectée, les valeurs seront conservées. Pour une combinaison de prédicats :

— *Pairwise dependances* : on considère les combinaisons des prédicats deux à deux.
— *Nwise dependances (nbr)* : combinaisons de 2..*nbr* prédicats.
— *All combinations coverage* : toutes les combinaisons de prédicats.

3.4.2 Critères de couverture du graphe de transition

Les Critères de couverture du graphe de transition (≪*structural model coverage criteria*≫) sont relatifs à la structure du modèle. Un événement Event-B correspond à une ou plusieurs transitions.

3.4.2.1 Critères sur les transitions et les événements

Dans le cas général, ces critères s'appliquent aux transitions. Dans le cas d'Event-B, une transition peut être interprétée comme un événement ou l'une des instanciations possibles d'un événement. Pour lever cette ambiguïté, nous parlons de transition pour l'instanciation d'un événement et d'événement pour l'ensemble des instanciations possibles d'un événement.

- *All-states coverage* : tous les états sont couverts : chaque état est visité au moins une fois par un test.
- *All-events coverage* : chaque événement est au moins testé une fois.
- *All-transitions coverage* : chaque transition est au moins testée une fois.
- *All-events-pairs coverage* : chaque paire d'événements est atteignable.
- *All-transition-pairs coverage* : chaque paire de transitions est atteignable.
- *All-loop-free-paths coverage* : tous les chemins, à l'exception des boucles.
- *All-events-round-trips coverage* : tous les événements, à l'exception des chemins contenant plus d'une boucle.
- *All-transition-round-trips coverage* : toutes les transitions, à l'exception des chemins contenant plus d'une boucle.
- *All-events-one-loop-paths coverage* : tous les chemins (au niveau des événements), à l'exception des chemins contenant plus d'une boucle.
- *All-transition-one-loop-paths coverage* : tous les chemins (au niveau des transitions), à l'exception des chemins contenant plus d'une boucle.
- *All-events-paths coverage* : tous les chemins (au niveau des événements).
- *All-transitions-paths coverage* : tous les chemins (au niveau des transitions).
- *All-configurations coverage* : s'applique à des systèmes parallèles. Assure que toutes les configurations sont testées.

3.4.2.2 Critères sur la modification du flot de contrôle

Une décision peut posséder plusieurs contraintes et est composée de conditions.

- *State coverage* : l'ensemble de tests doit couvrir tout l'espace d'état.
- *Decision coverage* : l'ensemble de tests doit vérifier que la décision a au moins une fois été testée vraie et une fois fausse. Il faut également assurer que state coverage est vraie.
- *Condition Coverage* : toutes les conditions doivent être testées vraie et faux.
- *Decision/condition coverage* : assure que Decision coverage et Condition Coverage sont testées, c.-à-d. toutes les décisions sont testées vraie et fausse et pour chaque décision, toutes les conditions ont été testées vraie et fausse.
- *Full predicate coverage* : toutes les conditions sont testées vraie et fausse, dans le cas où le résultat affecte l'état de sortie de la transition, c.-à-d. la condition affecte la décision.
- *Modified condition/decision coverage (MC/DC)* : toutes les conditions sont testées vraie ou fausse, mais pas les deux en même temps.
- *Multiple condition coverage (MCC)* : couvre toutes les combinaisons de condition vraie et fausse.
- *Path coverage* : tous les chemins sont dans l'ensemble de tests.
- *All events* : tous les événements doivent être couverts au moins une fois.

3.4.3 Algorithmes de recherche de solution

Différents algorithmes permettent d'extraire des tests. Leur performance est directement liée à la notation formelle du modèle ainsi qu'au type de critères à satisfaire. Ces algorithmes peuvent être classés en deux catégories. Certains algorithmes explorent le modèle dans le but de satisfaire les critères de couverture. Les critères permettent de guider la recherche de solution. D'autres algorithmes tentent de parcourir l'espace d'état du modèle puis d'en extraire des tests. Dans ce cas, l'exploration n'est pas guidée.

Dans les deux cas, nous avons besoin d'un algorithme de parcours de graphe. Dans le premier cas, l'algorithme sera guidé durant sa recherche, ce qui peut lui économiser certaines opérations. Dans le second cas, un graphe plus générique et donc potentiellement plus lourd doit être généré à l'avance. Cependant, ces algorithmes trouvent en général une solution plus rapidement.

Les algorithmes de parcours de graphes peuvent être utilisés. Parmi ces algorithmes nous pouvons citer [57] : le parcours en largeur, en profondeur, à coût uniforme, *best first*, A*, Dijkstra, Kruskal, Prim, Sollin ou encore le tour transitif. Bien que simples, ces algorithmes sont généralement très performants. Des algorithmes plus spécialisés aux problèmes de MBT ont été proposés [35] : D-method, W-method, Wp-method, U-method, T-method.

Dans le cas d'Event-B, un test peut être représenté de deux façons. Dans le cas d'un modèle constitué uniquement de contexte, la valeur des constantes est un test. Ces valeurs sont obtenues par résolution de contrainte. Il existe généralement plusieurs valeurs possibles. Le modèle doit simplement être satisfait, c.-à-d. les valeurs doivent satisfaire les axiomes. Dans le cas d'un modèle constitué de machines, un test peut être représenté soit par un état [3], soit par une trace. Dans le cas d'un état, les valeurs des constantes et des variables permettent de constituer le test de la même manière que pour les contextes. Pour les traces, chaque transition représente une partie du test. Ces transitions représentent généralement des interactions avec le SUT.

3.4.4 Concrétisation des tests

Les tests générés sont au même niveau d'abstraction que le modèle. Il est donc nécessaire de les rendre compréhensibles par le SUT. Ce processus, appelé concrétisation peut-être soit statique soit dynamique. Dans le cas où toutes les réponses du SUT peuvent être prédites, une *concrétisation statique* peut être utilisée. Un ensemble de tests concret est généré et peut être utilisé sur toute implémentation d'une même spécification. Cette technique a l'avantage de produire un ensemble de tests générique. Dans le cas contraire, la *concrétisation dynamique* s'effectue en même temps que l'exécution des tests. Un test étant généralement constitué d'étapes, chaque interaction avec le SUT permet de choisir l'étape suivante. Cela permet d'adapter un test en fonction des réponses du système sous test. Cependant, cette technique ne permet pas de produire un ensemble de tests générique.

3. Cela est réalisé à l'aide d'un modèle qui décrit un algorithme de test où l'état final représente le résultat de la génération du test.

3.4.5 Génération de tests d'intrusion

Pour extraire uniquement les tests requis, des contraintes supplémentaires permettent de guider l'extracteur de tests. Pour extraire des tests relatifs aux intrusions, le modèle doit représenter ces intrusions. Le modèle devra donc décrire le comportement du système en présence de toutes les intrusions. Cependant, il est nécessaire de connaître l'ensemble des intrusions. Cela est généralement impossible et leur découverte est un travail empirique. De plus, il est parfois impossible de modéliser toutes les intrusions connues. Dans le cas du VCI , le nombre d'intrusions potentielles est beaucoup trop grand pour envisager de toutes les modéliser. De plus, peu d'intrusions sont connues. Il est donc nécessaire de trouver une technique capable de générer automatiquement de potentielles intrusions.

3.5 Conclusion

Dans la première section, nous avons décrit l'importance des mécanismes de vérification des cartes à puce Java Card. Durant le chargement d'application, le VCI est en charge de la vérification statique du respect de la spécification de la JCVM. Ce composant de sécurité représente la première ligne défensive. Son fonctionnement a été détaillé dans la seconde section. Il se compose de deux sous-processus, la vérification de structure et la vérification de type. Ces deux processus correspondent à nos deux cas d'études. Certaines méthodes de vérification ont été présentées dans les sections suivantes. La représentation formelle du système permet de le vérifier à l'aide de preuves ou d'exploration de modèle. Ces méthodes ne sont pas adaptées à notre problématique de vérification dynamique. Cependant, une représentation formelle permet d'utiliser les techniques de test à base de modèle et ainsi d'améliorer le processus de génération de tests. Les concepts présentés dans ce chapitre ont été combinés afin de produire la méthode et les cas d'études.

Chapitre 4 :

Génération de tests de vulnérabilité pour Event-B

Pour améliorer la génération de tests d'intrusion, nous avons proposé une nouvelle méthode appelée VTG (« *Vulnerability Test Generation* », génération de tests de vulnérabilités). Celle-ci repose sur deux autres méthodes, la mutation de spécification et le test à base de modèle. Une spécification formelle du SUT représente les comportements sans faute. La première étape consiste à explorer un sous-ensemble des comportements non spécifiés par relâchement de contraintes dans le but de découvrir des modèles d'intrusions. Ce processus est décrit dans les quatre premières sections. La seconde étape consiste à extraire les tests des modèles d'intrusion. Un algorithme d'extraction statique de tests, basé sur un explorateur de modèle, est proposé dans la dernière section.

4.1 Schéma général

La mutation de spécification est utilisée pour transformer un modèle formel selon des règles permettant de garantir certaines propriétés. La flèche bleue de la figure 4.1 représente ce processus. Le demi-ovale vert (à gauche) correspond aux comportements fonctionnels du système (fonctionnement sans intrusion). Le demi-ovale en pointillé rouge (à droite) correspond aux comportements du système en présence d'intrusions. L'ovale composé des demi-ovales vert et rouge, correspond au SUT. Les tests d'intrusion permettent de vérifier que, pour les comportements situés dans le demi-ovale rouge, les comportements observés correspondent aux comportements attendus. Cependant, les comportements du demi-ovale rouge ne sont parfois pas spécifiables pour plusieurs raisons : l'effort requis pour les spécifier est trop important, par rapport aux bénéfices retirés, ou bien il est simplement trop difficile de les spécifier. Par exemple. il serait très difficile de spécifier une injection SQL dans un système. La mutation de spécification correspond, dans le cas de la génération de tests de vulnérabilité, à générer le demi-ovale rouge à partir du demi-ovale vert.

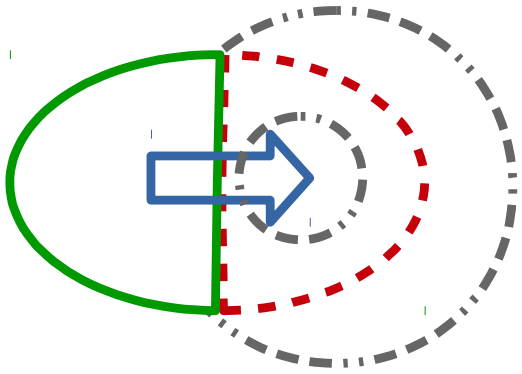

FIGURE 4.1 – Recherche de modèle de faute par mutation

Les deux problèmes de la mutation sont la surévaluation et la sous-évaluation représentées par les formes grises (pointillés externes et internes). La surévaluation correspondrait à la génération de systèmes mutants qui ne sont plus en rapport avec le SUT. Pour empêcher la surévaluation, l'utilisateur doit décider quelles parties du modèle il est pertinent de muter. Le modèle des comportements fonctionnels est ainsi constitué de deux parties : une partie à ne pas tester et une partie à tester. Ce modèle permet de définir la frontière du SUT. À l'inverse, ne générer qu'une sous-partie du modèle des

comportements en présence d'intrusion correspond à une sous-évaluation. Pour éviter cela, la frontière du SUT doit permettre l'exploration de tous les comportements en présence d'intrusion. c.-à-d. lus, le processus de mutation doit assurer d'être complet, c.-à-d. couvrir tous les comportements qui respectent la partie fonctionnelle du modèle et qui ne sont pas identiques aux comportements fonctionnels. Pour cela, la frontière du processus de transformation doit être vérifiée. Comme nous le verrons dans la section 4.3, les parties les plus importantes ont été formalisées. Le reste du processus a été vérifié empiriquement à l'aide de cas d'études.

Le VTG est composé de 3 processus : l'aplanissement du modèle originel, la mutation de spécification et le test à base de modèle. Le premier processus, décrit dans la section 4.2, adapte automatiquement le modèle originel aux problématiques de mutation. La mutation de spécification est décrite dans les sections 4.3 et 4.4. Le test à base de modèle est décrit dans la section 4.5. La figure 4.2 représente ces trois processus ainsi que l'intégration du VTG dans le processus de test et l'ensemble des processus le constituant.

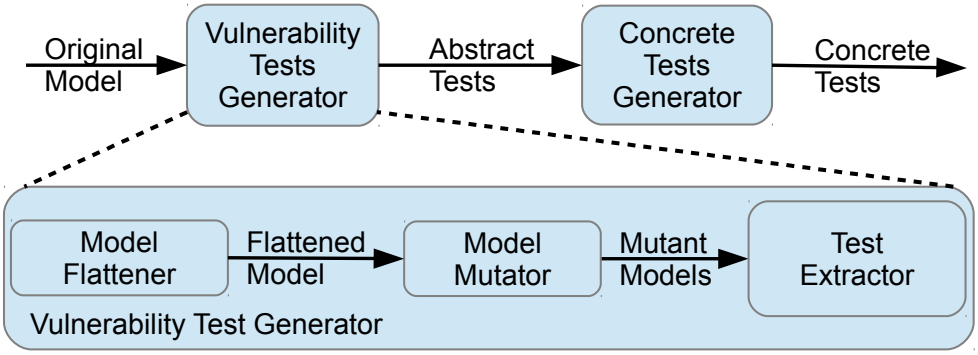

FIGURE 4.2 – Processus composant le VTG

Les tests d'intrusions générés par le VTG sont au même niveau d'abstraction que le modèle originel. Il convient donc de concrétiser ces tests afin de les exécuter sur le SUT. La concrétisation des tests étant différente pour chaque SUT, ce processus est donc indépendant du VTG et sera décrit pour chaque cas d'étude. Ces derniers seront présentés dans le chapitre 6.

Pour valider le concept de VTG, nous avons choisi de travailler avec la méthode Event-B. La grammaire du langage Event-B est suffisamment riche pour représenter des systèmes complexes et expérimenter rapidement. Le VTG permet de traiter un sous-ensemble des fonctionnalités de la méthode Event-B. La partie sous test contient l'ensemble des contraintes que l'on souhaite tester. La partie fonctionnelle décrit le fonctionnement du modèle.

4.2 Aplanissement du modèle originel

Lors de la mutation, si nous mutons une information concrète, seul l'événement concerné sera affecté. Cependant, si un événement abstrait est muté, tous les événements concrets sont affectés. La figure 4.3 représente un modèle constitué de 3 raffinements, du plus abstrait au plus concret : $R0$, $R1$ et $R2$. Les cercles représentent des événements et les flèches, des relations d'héritage. La mutation d'une garde d'un événement abstrait

entraîne la mutation de la garde de tous ses événements concrets, par héritage. Si $e1$ est muté, alors $e11$, $e12$, $e111$, $e112$ et $e121$ sont affectés. Cela a pour effet de limiter les combinaisons de mutations entre les gardes héritées par les événements les plus concrets.

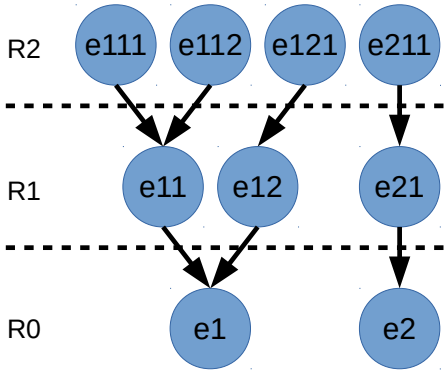

FIGURE 4.3 – Relation de raffinement d'événements Event-B

La mutation de machine n'est donc pas compatible avec le raffinement. Pour résoudre ce problème, l'aplanissement du modèle permet de rassembler l'ensemble des informations inductives dans une unique machine concrète. Cette machine est appelée machine aplanie et contient toutes les informations du modèle originel. Dans notre exemple, les informations contenues dans $R0$ sont transférées dans $R1$ pour former $R1'$ qui ne possède plus de liens avec le raffinement $R0$. La garde de $e1$ est par exemple recopiée dans $e11$ et $e12$ et l'information de raffinement est effacée. De la même manière, les informations de $R1'$ sont transférées dans $R2$ pour former $R2'$.

Du point de vue de l'exploration de modèle, le modèle aplani est équivalent au modèle originel. Les contextes ne possédant pas de notion de raffinement, ils ne sont pas affectés par cet aplanissement. La machine aplanie contient l'ensemble des contextes nécessaires à son fonctionnement. Le modèle aplani devient ensuite le modèle originel pour l'étape suivante, la mutation de modèle.

4.3 Mutation de formule

Considérons une formule F représentant le comportement attendu de notre système, c.-à-d. le modèle formel de notre système. Une règle de mutation est une règle de réécriture permettant de transformer la formule F en une autre formule G telle que G représente les comportements non attendus de notre système F, c.-à-d. G représente les comportements non modélisés. La règle générique de mutation est donc :

$$mutForm(F) \rightsquigarrow G$$

Le résultat de la mutation est ensuite utilisé pour la génération des tests. Il faut donc adapter la mutation aux objectifs de test. Les tests que nous souhaitons générer doivent permettre d'identifier rapidement la faute à l'origine de la défaillance dans le SUT. Pour cela, la décomposition de G en différents cas permet de remonter plus facilement à la source du problème. La mutation de F produit donc un ensemble de mutations, chacune de ces mutations correspondant à une faute particulière. Nous noterons G_n la n^{ieme} décomposition de la mutation. Une règle de mutation ne peut prendre en

compte que la formule courante et ses sous-formules. Afin de chercher le plus précisément possible l'origine d'une défaillance, nous souhaitons effectuer la faute la plus petite possible, ainsi que toutes ses combinaisons, pour une formule donnée. Pour cela, nous appliquons la mutation récursivement sur chacune des sous-formules. Dans la suite de ce document, nous noterons G_n une mutation non récursive, $mutForm(G_n)$ une mutation récursive et G'_n une mutation potentiellement récursive, correspondant soit à G_n soit à $mutForm(G_n)$. L'utilisation d'un prolongement point par point (*pointwise extension*) permet de constituer l'ensemble des réécritures. Ce prolongement est défini de la façon suivante :

$$mutForm(F1\ op\ F2) = ptw(mutForm(F1)\ op\ F2) \cup ptw(F1\ op\ mutForm(F2))$$

$$Soit\ mutForm(F) = \{G_1, \ldots, G_n\} :$$

$$ptw(mutForm(F1)\ op\ F2) = \{G_1\ op\ F2\} \cup \ldots \cup \{G_n\ op\ F2\}$$

La formule F peut être mutée dans deux cas : soit pour la rendre fausse, soit pour représenter les cas où elle n'est pas bien définie. Nous séparerons ces deux cas respectivement en *mutBool* et *mutWD*. Ces deux types de mutations sont décrits dans les sections 4.3.1 et 4.3.2. La mutation d'une formule est l'union de ces deux types de mutation :

$$mutForm(F) \rightsquigarrow mutBool(F) \cup mutWD(F)$$

Pour les règles de mutation définies plus bas, nous utilisons les symboles suivants :
— $i_1, i_2 \in \mathbb{N}$.
— $b, b_1, b_2 \in BOOL$.
— $F, F1, F2$ des formules.
— e un scalaire, peu importe le type.
— E, E_1, E_2 des ensembles.
— $\top$ et $\bot$ représente respectivement vraie et faux.

4.3.1 Mutation booléenne, *mutBool*

La mutation booléenne consiste à générer les formules représentant les cas pour lesquels la formule originelle est fausse du point de vue de la logique des prédicats. La mutation de la formule racine permet de récursivement générer toutes les mutations. Le résultat d'un appel récursif est ensuite combiné pour générer tous les cas. Les règles de mutation suivantes ont été proposées [1] :

1. $mutBool(\bot) \rightsquigarrow \{\top\}$
2. $mutBool(\top) \rightsquigarrow \{\bot\}$
3. $mutBool(i_1 > i_2) \rightsquigarrow \{i_1 = i_2\} \cup \{i_1 < i_2\}$
4. $mutBool(i_1 < i_2) \rightsquigarrow \{i_1 = i_2\} \cup \{i_1 > i_2\}$
5. $mutBool(i_1 \geq i_2) \rightsquigarrow \{i_1 < i_2\}$
6. $mutBool(i_1 \leq i_2) \rightsquigarrow \{i_1 > i_2\}$
7. $mutBool(i_1 = i_2) \rightsquigarrow \{i_1 < i_2\} \cup \{i_1 > i_2\}$

1. Ces règles traitent tous les opérateurs booléens du langage Event-B.

8. $mutBool(i_1 \neq i_2) \rightsquigarrow \{i_1 = i_2\}$
9. $mutBool(bool(F) = b) \rightsquigarrow \{bool(mutBool(F)) = b\}$
10. $mutBool(b = bool(F)) \rightsquigarrow \{b = bool(mutBool(F))\}$
11. $mutBool(bool(F1) = bool(F2)) \rightsquigarrow \{bool(mutBool(F1)) \neq bool(mutBool(F2))\}$
12. $mutBool(b_1 = b_2) \rightsquigarrow \{b_1 \neq b_2\}$
13. $mutBool(b_1 \neq b_2) \rightsquigarrow \{b_1 = b_2\}$
14. $mutBool(F1 \wedge F2) \rightsquigarrow \{mutBool(F1) \wedge F2\} \cup \{F1 \wedge mutBool(F2)\} \cup \{mutBool(F1) \wedge mutBool(F2)\}$
15. $mutBool(F1 \vee F2) \rightsquigarrow \{mutBool(F1) \wedge mutBool(F2)\}$
16. $mutBool(F1 \Rightarrow F2) \rightsquigarrow \{F1 \wedge mutBool(F2)\}$
17. $mutBool(F1 \Leftrightarrow F2) \rightsquigarrow \{mutBool(F1 \Rightarrow F2 \wedge F2 \Rightarrow F1)\}$
18. $mutBool(\neg(F)) \rightsquigarrow \{\neg(mutBool(F))\}$
19. $mutBool(\forall z \cdot F) \rightsquigarrow \{\exists z.mutBool(F)\}$
20. $mutBool(\exists z \cdot F) \rightsquigarrow \{\forall z.mutBool(F)\}$
21. $mutBool(e \in E) \rightsquigarrow \{e \notin E\}$
22. $mutBool(e \notin E) \rightsquigarrow \{e \in E\}$
23. $mutBool(E_1 \subseteq E_2) \rightsquigarrow \{E_2 \subset E_1\} \cup \{E_1 \cap E_2 = \varnothing\} \cup \{E_1 \cap E_2 \neq \varnothing \wedge E_1 \not\subseteq E_2\}$
24. $mutBool(E_1 \not\subseteq E_2) \rightsquigarrow \{E_1 = E_2\} \cup \{E_1 \subset E_2\}$
25. $mutBool(E_1 \subset E_2) \rightsquigarrow \{E_1 \neq \varnothing \wedge E_2 = \varnothing\} \cup \{E_1 = \varnothing \wedge E_2 = \varnothing\} \cup \{E_1 \neq \varnothing \wedge E_1 = E_2\} \cup \{E_2 \neq \varnothing \wedge E_2 \subset E_1\} \cup \{E_1 \neq \varnothing \wedge E_2 \neq \varnothing \wedge \exists x \cdot x \in E_1 \wedge x \in E_2\} \cup \{E_1 \neq \varnothing \wedge E_2 \neq \varnothing \wedge E_1 \cap E_2 = \varnothing\}$
26. $mutBool(E_1 \not\subset E_2) \rightsquigarrow \{E_1 \subset E_2\}$
27. $mutBool(E_1 = E_2) \rightsquigarrow (E_1 \neq E_2)$

Prenons comme exemple la mutation booléenne de la formule $x > 14 \wedge t = true$. La première étape génère les mutations suivantes :

$$\begin{aligned} mutBool(x > 14 \wedge t = true) \rightsquigarrow\ & \{mutBool(x > 14) \wedge \quad t = true\} \ \cup \\ & \{\quad x > 14 \ \wedge mutBool(t = true)\} \cup \\ & \{mutBool(x > 14) \wedge mutBool(t = true)\} \end{aligned}$$

La seconde étape génère les mutations suivantes, correspondant au résultat final :

$$\begin{aligned} mutBool(x > 14 \wedge t = true) \rightsquigarrow\ & \{(x < 14 \wedge t = true), (x = 14 \wedge t = true)\} \cup \\ & \{(x > 14 \wedge t \neq true)\} \quad \cup \\ & \{(x < 14 \wedge t \neq true), (x = 14 \wedge t \neq true)\} \end{aligned}$$

4.3.2 Mutation de bonne définition, *mutWD*

Le langage Event-B définit des règles de bonne définition. Ces règles peuvent être interprétées comme des préconditions aux formules afin de s'assurer qu'un opérateur partiel est utilisé seulement pour les cas où il est défini. Ces préconditions se comportent donc comme des conditions masquées derrière une formule. Pour générer l'ensemble des cas de test, il est nécessaire de considérer les cas où les formules ne sont pas bien définies. Ces règles sont appliquées récursivement. Les sous-formules concernées par la mutation sont remplacées. *mutWD*, pour les autres opérateurs du langage, les opérateurs qui ne sont pas partiels, ne fait que propager *mutWD* aux opérandes. La mutation de bonne définition d'une constante ou d'une variable retourne $\varnothing$. Dans le cas où le nombre d'éléments est fini, les règles suivantes sont utilisées :

1. $mutWD(i_1 \div i_2) \rightsquigarrow mutWD(i_1) \cup mutWD(i_2) \cup \{i_2 = 0\}$
2. $mutWD(min(E)) \rightsquigarrow mutWD(E) \cup \{E = \varnothing\}$
3. $mutWD(max(E)) \rightsquigarrow mutWD(E) \cup \{E = \varnothing\}$
4. $mutWD(inter(E)) \rightsquigarrow mutWD(E) \cup \{E = \varnothing\}$
5. $mutWD(card(E)) \rightsquigarrow mutWD(E) \cup \{\neg(finite(E))\}$
6. $mutWD(F1\ op\ F2) = mutWD(F1) \cup mutWD(F2)$
7. $mutWD(\neg F) = mutWD(F)$

Pour les fonctions d'ordre n, nous proposons la définition récursive suivante :

1. si $n = 1$ alors $mutWD(f(x_1)) \rightsquigarrow \{x_1 \notin dom(f)\} \cup mutWD(x1) \cup mutWD(f)$
2. si $n = 2$ alors $mutWD(f(x_1)(x_2)) \rightsquigarrow \{x_2 \notin dom(f(x_1)) \cup mutWD(f(x_1))\} \cup mutWD(f) \cup mutWD(x1) \cup mutWD(x2)$
3. si $n > 2$ alors $mutWD(f(x_1)...(x_n)) \rightsquigarrow \{x_n \notin dom(f(x_1)...(x_{n-1})) \cup mutWD(f(x_1)...(x_{n-1}))\}$

Pour les fonctions d'arité supérieure à un, la formule où X représente $x_1, x_2, ..., x_n$ s'applique de la façon suivante :

$$\begin{aligned} mutWD(f(X)) \quad = \quad & \{(X) \notin dom(f)\} \cup \\ & mutWD(f) \cup mutWD(x_1) \cup mutWD(x_2) \cup ... \cup mutWD(x_n) \end{aligned}$$

Considérons la formule $f(x) = y \wedge (4/x) > z$. La première étape consiste à générer les mutations suivantes :

$$mutWD(f(x) = y \wedge (4/x) > z) \rightsquigarrow \{mutWD(f(x) = y)\} \cup \{mutWD((4/x) > z)\}$$

La seconde étape génère les mutations suivantes :

$$mutWD(f(x) = y \wedge (4/x) > z) \rightsquigarrow \{x \notin dom(f)\} \cup \{x = 0\}$$

4.3.3 Propriétés des mutations

Pour garantir le mécanisme de génération de tests de vulnérabilité, un certain nombre de propriétés doivent être vérifiées. Ces dernières ont été formalisées, permettant ainsi de

prouver la bonne définition, la disjointure, la cohérence, la complétude et la minimalité d'un jeu de règles de mutations.

4.3.3.1 Propriété de bonne définition

La spécification du langage Event-B utilisée dans Rodin définit un ensemble de propriétés de bonne définition pour les formules. Les formules produites par la mutation doivent elles aussi respecter ces propriétés. Une preuve par induction assure que toutes les mutations produisent des formules bien définies.

4.3.3.2 Propriété de disjointure

La propriété de disjointure impose que, pour toute formule, il existe une unique règle de mutation applicable, c.-à-d. une formule F ne peut s'unifier à la fois avec une règle $R1$ et une règle $R2$. Cette propriété s'applique sur les mutations booléennes et de bonne définition séparément. On suppose pour cela que les formules à muter sont bien définie. Pour cela, nous devons prouver que toutes les parties gauches des règles sont bien deux à deux distinctes.

4.3.3.3 Propriété de cohérence et de complétude

La propriété de cohérence assure que nous ne générons pas de faux positif, c.-à-d. si une formule mutée est vraie, alors la formule à muter est fausse. Soit G une mutation d'une formule F, alors :

$$(G \Rightarrow \neg F)$$

Réciproquement, si la formule à muter est fausse, il faut s'assurer qu'au moins une des formules mutées couvre ce cas. Cela donne lieu à une propriété de complétude. Soit $G'_1, ..., G'_n$ les mutations de la formule F, alors :

$$\neg F \Rightarrow G'_1 \vee G'_2 \vee ... \vee G'_n$$

La cohérence et la complétude donnent lieu à une unique propriété :

$$\neg F \Leftrightarrow (G'_1 \vee G'_2 \vee ... \vee G'_n)$$

4.3.3.4 Propriété de minimalité

La propriété de minimalité assure de ne pas générer plusieurs fois les mêmes tests. Elle est optionnelle, mais assure une bonne qualité des mutations obtenues. Comme nous ne souhaitons pas devoir parcourir plusieurs fois les mêmes cas de tests, il faut que toutes les mutations d'une formule soient disjointes deux à deux. Soit $G'_1, ..., G'_n$ les mutations de la formule F. Alors :

$$\begin{array}{lllll} \neg(G'_1 \wedge G'_2) & \wedge & \neg(G'_1 \wedge G'_3) & \wedge ... \wedge & \neg(G'_1 \wedge G'_n) \quad \wedge \\ & & \neg(G'_2 \wedge G'_3) & \wedge ... \wedge & \neg(G'_2 \wedge G'_n) \quad \wedge \\ & & & ... & \\ & & & & \neg(G'_{n-1} \wedge G'_n) \end{array}$$

4.4 Mutation de modèle Event-B

La mutation de tous les éléments constituant le langage Event-B permettrait de représenter tous les modèles réalisables. Cependant, beaucoup de mutations ne correspondent pas aux objectifs de vérification, c.-à-d. sont des surévaluation du SUT. Seules certaines parties du langage représentent les propriétés que nous cherchons à tester. Ainsi, seule ces parties peuvent être mutées. Cette restriction constitue une première partie de la frontière du SUT. L'autre partie doit être fixée manuellement par la personne chargée de réaliser les tests. Dans la section 4.4.1 nous donnons le processus de mutation d'un modèle constitué uniquement de contextes. Dans la section 4.4.2, nous nous intéresserons à un modèle constitué de machines et de contextes, mais seules les machines seront mutées.

4.4.1 Mutation de contexte

Nous considérons dans cette section un modèle constitué d'un unique contexte ou d'un contexte qui étend l'ensemble des contextes du modèle. Les propriétés que nous cherchons à tester sont représentées par les axiomes. Par exemple, un contexte peut être utilisé pour représenter les contraintes structurelles d'un fichier CAP valide. Ces contraintes indiquent, par exemple, que la valeur du champ `magic` dans le composent `Header` est `0xDECAFED`[2]. La mutation de ces contraintes nous permet de générer des fichiers cap invalides, et ce de manière systématique. La mutation d'un axiome permet d'obtenir les axiomes mutants caractérisant les fautes que nous cherchons dans le SUT. Nous distinguons deux types d'axiomes, les axiomes à ne pas tester et les axiomes à tester. La fonction $axmToTest(origCtx)$ renvoie l'ensemble des axiomes à tester. Les axiomes à ne pas tester constituent la partie fonctionnelle du modèle et sont uniquement présents pour son bon fonctionnement. Dans l'implémentation proposée, l'utilisation du marqueur «_t» à la fin du label de l'axiome permet de désigner les axiomes à tester. Par extension, les axiomes dont le label ne se termine pas par le marqueur ne sont pas à tester. Ces axiomes servent, par exemple, à définir le type des constantes. Ils ne seront donc pas affectés par la mutation. Les axiomes à tester constituent la partie du modèle à tester et seront mutés.

Les axiomes sont les seuls éléments du modèle originel à être modifiés. Les contextes mutants sont obtenus par clonage du modèle originel et remplacement d'axiomes. L'algorithme 2 décrit cette transformation. La première partie, avant la ligne 11, représente la mutation des axiomes. Dans la seconde partie, les axiomes mutants obtenus sont utilisés pour la création des contextes mutants. Lorsque l'algorithme termine, la variable *MutCtx* contient tous les contextes mutants. Chaque contexte mutant caractérise une faute ou une combinaison de fautes. Pour chaque contexte mutant, les axiomes à tester sont séparés en deux groupes : ceux à muter et les autres. L'ensemble des sous-ensembles des axiomes à tester permet de décrire toutes les combinaisons d'axiomes à muter. Cet ensemble est noté *Attam* (*axiom to test and mutate*). Chaque élément de l'ensemble *Attam* est utilisé pour constituer des contextes mutants. Dans ces contextes mutants, les axiomes à muter sont remplacés par l'une de leurs mutations. La mutation de la formule constituant l'axiome est décrite dans la section 4.3. La fonction $mutForm(axm)$ représente la mutation de formule et renvoie un ensemble où chaque élément est une mutation possible de l'axiome donné en paramètre. Dans le cas où plusieurs axiomes doivent être mutés simultanément, la

2. Valeur définie dans la spécification Java Card.

combinaison des mutations de leur formule est représenté par la fonction *combineSets*(F). Elle est utilisé pour calculer toutes les mutations d'un contexte. Si un contexte comprend plusieurs axiomes, chaque axiome ayant plusieurs mutations, une mutation d'un contexte est constitué d'une combinaison des mutations des axiomes. La formule *combineSets*(F) retourne un ensemble d'ensembles de formules X où X contient exactement un élément de chaque S_i. Elle est définie de la façon suivante, où $S_1, \ldots, S_n$ sont des ensemble de formules :

$$combineSet(\{S_1, ..., S_n\}) = \{X | X \subseteq \bigcup(\{S_1, ..., S_n\}) \wedge \forall i \cdot i : 1..n : card(X \cap S_i) = 1\}$$

Algorithm 2 Mutation générale de contexte

```
1: procedure MUTATECONTEXT(origCtx)
2:     MutCtx := ∅
3:     for all Aum ∈ ℙ(axmToTest(origCtx)) do
4:         MutAut := ∅
5:         for all aum ∈ Aum do
6:             MutAxm ∪= {mutForm(aum)}
7:         end for
8:         for all MutAxmCase ∈ combineSets(MutAxm) do
9:             mutCtxTmp := origCtx
10:            mutCtxTmp.Axioms := mutCtxTmp.Axioms\Aum
11:            mutCtxTmp.Axioms ∪= MutAxmCase
12:            MutCtx ∪= {mutCtxTmp}
13:        end for
14:    end for
15:    return MutCtx
16: end procedure
```

Bien que cette solution soit complète, elle génère un grand nombre de contextes mutants. En fonction des objectifs de test, il est parfois préférable de sélectionner un sous-ensemble de ces mutations. Pour le cas d'étude présenté en section 5.2, nous considérons que chaque propriété à tester est représentée par un unique axiome et que chaque propriété doit être testée indépendamment. La génération des contextes mutants ne contenant qu'un unique axiome mutant est suffisante. Ainsi, la génération de l'ensemble des sous-ensembles des axiomes à muter n'est pas nécessaire. Seul l'ensemble des axiomes à muter est utilisé. L'algorithme 2 décrit la génération de contextes mutants contenant un unique axiome mutant.

4.4.2 Mutation de machine

Dans une machine, les propriétés de sécurité que nous cherchons à tester sont représentées par les gardes des événements. L'obtention des machines mutantes nécessite donc la génération d'événements mutants possédant une garde mutée. Dans un premier temps, nous allons décrire comment obtenir les différents événements mutants. L'assemblage des événements mutants et des événements originaux, permettant de générer les machines mutantes, sera décrit dans un second temps.

Algorithm 3 Mutation simplifiée de contexte

```
1: procedure MUTATECONTEXTSIMPLE(origCtx)
2:     MutCtx := ∅
3:     for all aum ∈ axmToTest(origCtx) do
4:         for all MutAxmCase ∈ mutForm(aum) do
5:             mutCtxTmp := origCtx
6:             mutCtxTmp.Axioms := mutCtxTmp.Axioms\{aum}
7:             mutCtxTmp.Axioms ∪= {MutAxmCase}
8:             MutCtx ∪= {mutCtxTmp}
9:         end for
10:    end for
11:    return MutCtx
12: end procedure
```

4.4.2.1 Génération des événements mutants

Les gardes des événements sont traitées de façon similaire aux axiomes. Chacun des événements à muter est transformé à l'aide de l'algorithme 4. Les gardes sont décomposées en deux groupes, celles à ne pas tester et celles à tester, correspondant respectivement aux propriétés fonctionnelles et aux propriétés à tester. Pour chaque mutation, un événement mutant est généré. Le marqueur «_t» à la fin du label de la garde permet de définir ci cette garde est à tester. La fonction $grdToTest(g)$ renvoie l'ensemble des gardes à tester. Les gardes à ne pas tester ainsi que les gardes à ne pas muter de l'événement originel sont conservées dans l'événement mutant. L'ensemble des sous-ensembles des gardes à tester permet de sélectionner celles à muter. Chaque garde à muter est remplacée par l'une de ses mutations. Le processus de mutation de la formule contenue dans les gardes est décrit dans la section 4.3. Si plusieurs gardes doivent être mutées, la combinaison de leurs mutations permet de représenter tous les cas possibles.

La garde de l'événement originel n'étant plus respectée, il n'est généralement pas possible de savoir quelle sera la postcondition. En effet, il n'est pas possible de connaître l'état du système s'il accepte d'exécuter une transition invalide. En fonction des objectifs de test, l'action de l'événement mutant est laissée soit inchangée, soit vide. Bien que cette solution soit incomplète, l'ensemble des mutations nécessaires pour notre cas d'étude a pu être générés. Les autres éléments de l'événement originel sont recopiés. La ligne 12 décrit le cas pour lequel l'action de l'événement mutant doit être supprimée. Si l'action doit être conservée, cette ligne n'existe pas.

4.4.2.2 Génération des modèles mutants

Les événements mutants peuvent être combinés de plusieurs façons avec les événements originaux pour obtenir les modèles mutants. Trois critères permettent de définir quel type de modèle mutant nous souhaitons générer. Ces critères sont choisis en fonction des objectifs de test et doivent être adaptés pour chaque SUT.

Le processus d'assemblage des événements mutants dans les machines mutantes est similaire aux gardes dans les événements et aux axiomes dans les contextes. L'ensemble des sous-ensembles des événements originels permet de sélectionner les événements à muter pour générer chaque machine mutante. Lorsque plusieurs événements sont à muter,

Algorithm 4 Mutation d'un événement

```
1: procedure MUTATEEVENT(origEvt)
2:     MutEvt := ∅
3:     for all Gum ∈ ℙ(grdToTest(origEvt)) do
4:         MutGrd := ∅
5:         for all gum ∈ Gum do
6:             MutGrd ∪= mutForm({gum})
7:         end for
8:         for all MutGrdCase ∈ combineSets(MutGrd) do
9:             mutEvtTmp := origEvt
10:            mutEvtTmp.Guards := mutEvtTmp.Guards\Gum
11:            mutEvtTmp.Guards ∪= MutGrdCase
12:            mutEvtTmp.Actions := ∅
13:            MutEvt ∪= {mutEvtTmp}
14:        end for
15:    end for
16:    return MutEvt
17: end procedure
```

la combinaison de leur mutation permet de générer l'ensemble des modèles mutants possibles. Le premier critère consiste à choisir le nombre d'événements mutants par machine mutante. Il existe 4 possibilités :

1. le modèle comporte une seule mutation pour un événement à muter,
2. le modèle comporte toutes les mutations pour un événement à muter,
3. le modèle comporte une seule mutation pour chaque événement à muter ou
4. le modèle comporte tous les événements mutants de l'ensemble des événements à muter

Le second critère consiste à choisir quels événements originels sont conservés dans la machine mutante. Le premier cas consiste à ne garder aucun événement originel et le second à tous les conserver. Dans le troisième cas, tous les événements originels sont conservés à l'exception des événements mutés. Soit EVT l'ensemble des événements originels, c.-à-d. l'ensemble contenant tous les événements de la machine originelle. L'ensemble $EvtToKeep$ définit, pour une machine composée de n événements, les événements à conserver dans la machine mutante :

1. $EvtToKeep := \varnothing$
2. $EvtToKeep := EVT$
3. $EvtToKeep := EVT \setminus EvtToMutate$

Après l'exécution d'un événement mutant, il n'est généralement pas possible de savoir dans quel état est le système. Il est donc préférable d'interdire tout événement après que l'événement mutant soit survenu. Cependant, cela empêche la génération d'un post-ambule. Le troisième critère permet de définir si les événements peuvent survenir après l'événement mutant, si seuls les événements originaux peuvent survenir ou si aucun ne peut survenir. Ce critère est contrôlé en ajoutant la variable booléenne $eutExecuted$. Cette variable est initialisée à $FALSE$ et devient égale à $TRUE$ dés qu'un événement mutant

est exécuté. Une action est ajoutée à chaque événement mutant. Si tous les événements peuvent survenir après l'exécution d'un événement, aucune modification supplémentaire n'est apportée au modèle mutant. Si seuls les événements originaux peuvent survenir, une garde est ajoutée dans chaque mutant. Cette garde contrôle qu'aucun événement mutant n'ait été exécuté en se référant à la valeur de la variable *eutExecuted*. Si aucun événement ne doit survenir après l'exécution d'un événement mutant, la garde définie précédemment est ajoutée à tous les événements des modèles mutants.

L'algorithme 5 est utilisé dans les cas d'études présentés dans les sections 5.2 et 5.3. Il décrit la génération des machines mutantes telles que les modèles mutants contiennent :

— l'ensemble des sous-ensembles des événements mutants,
— les événements originels sont conservés à l'exception des événements mutés et
— seuls les événements originaux peuvent survenir après l'événement mutant.

Algorithm 5 Génération des machines mutantes

```
1: procedure MUTATEMACHINE(origMch)
2:     MutMch := ∅
3:     for all Eum ∈ ℙ(evtToTest(origMch)) do
4:         MutEvt := ∅
5:         for all eum ∈ Eum do
6:             MutEvt ∪= mutateEvent(eum)
7:         end for
8:         for all MutEvtCase ∈ combineSets(MutEvt) do
9:             mutMchTmp := origMch
10:            mutMchTmp.Varialbes ∪= {eutExecuted}
11:            mutMchTmp.Invariants ∪= {eutExecuted ∈ BOOL}
12:            mutMchTmp.Events := mutMchTmp.Events\Eum
13:            for all mutEvt ∈ MutEvtCase do
14:                mutEvtTmp := mutEvt
15:                mutEvtTmp.Guard ∪= {eutExecuted = FALSE}
16:                mutEvtTmp.Actions ∪= {eutExecuted := TRUE}
17:            end for
18:            MutMch ∪= {mutEvtTmp}
19:        end for
20:    end for
21:    return MutMch
22: end procedure
```

4.5 Extraction des tests

Une fois les modèles de faute générés, il est nécessaire d'en extraire des tests. Pour cela, nous utilisons les techniques de test à base de modèle. L'extraction des tests que nous présentons s'applique à n'importe quel modèle, qu'il soit un modèle muté ou un modèle original. L'extraction des tests permet de parcourir un modèle et d'extraire les tests. Si l'extraction est appliqué à un modèle non muté, les tests générés sont des tests fonctionnels classiques. Si l'extraction est appliqué à un modèle muté, les tests générés sont

des tests de détection de vulnérabilité. Comme nous l'avons vu dans la section 3.4.4, la concrétisation peut être statique ou dynamique en fonction du SUT. Dans le cas du VCI , cette concrétisation est effectuée statiquement et sera décrite pour chaque cas d'étude. Les algorithmes de génération de tests à base de modèle proposés génèrent donc des tests abstraits qui seront concrétisés par la suite. Ils reposent sur ProB et y ont été intégrés. Ces deux algorithmes sont respectivement utilisés pour des modèles constitués uniquement de contextes et des modèles dont l'élément racine est une machine. Il sont présentés dans les sections 4.5.1 et 4.5.2. Bien que ProB soit conçu pour l'exploration de modèles [3], comme décrit dans la section 5.1, ces deux algorithmes se basent sur la résolution de contraintes. Cette dernière technique fonctionne bien avec des contraintes sur des espaces d'états vastes. De plus, différentes tactiques permettent d'améliorer les résultats en guidant la recherche de solution. Nous considérons ici que le modèle est prouvé. Le travail présenté dans cette section a été réalisé conjointement avec Michael Leuschel.

4.5.1 Contexte

Dans le cas des contextes, nous utilisons l'algorithme de résolution de contraintes fourni par ProB. Le résultat obtenu est un ensemble d'instanciations possibles, chacune représentant un test abstrait. Notre cas d'étude a permis d'apporter de nouveaux cas d'usages et d'améliorer ProB. Avant de démarrer la résolution de contraintes, un processus permet de restreindre l'espace de recherche. Tout d'abord, ce processus détermine si une variable est définie dans un espace fini ou non. Pour les variables finies, le domaine de définition est ensuite restreint en fonction des contraintes. Cet algorithme a été amélioré par le laboratoire de l'Université de Düsseldorf. Le cas d'étude présenté dans la section 5.2 utilise ce processus.

4.5.2 Machine

Dans le cas des machines, un test est représenté par une trace, généralement composée de 4 parties : le préambule, le corps, l'observation et le post-ambule. En event-B, une trace est une suite d'événements. Le préambule permet de conduire le système dans un état permettant d'effectuer une vérification. Le corps effectue l'action à vérifier. L'observation permet de vérifier que l'action du corps s'est effectuée correctement. Le post-ambule ramène le système dans un état où les prochains tests peuvent être exécutés. Pour limiter la complexité de la recherche de trace, trois processus permettent de restreindre l'espace de recherche : l'analyse de faisabilité, l'analyse de faisabilité de couple d'événements et l'analyse d'activabilité.

4.5.2.1 Analyse de faisabilité

Un modèle mutant peut contenir de multiples événements mutants. Les règles de mutation génèrent parfois des événements qui ne peuvent pas être exécutés. L'analyse de faisabilité permet d'éliminer ces événements pour la recherche de trace. Chaque événement est analysé indépendamment. Cette analyse se base sur une vérification statique qui cherche au moins une solution pour le prédicat composé des axiomes, des invariants et de la garde de l'événement analysé. Une limite de temps permet de garantir la terminaison

3. ProB est plus précisément conçu pour l'exploration du graphe de transition.

de cette vérification. Si cette limite est atteinte, l'événement analysé est étiqueté comme potentiellement faisable. Si l'analyse n'a pas trouvé de solution, l'événement est étiqueté comme infaisable et ne sera pas conservé dans l'algorithme de recherche de trace. Si une solution est trouvée, l'événement est étiqueté comme faisable. Dans ce dernier cas, l'événement n'est peut-être pas atteignable.

4.5.2.2 Analyse de faisabilité de couple d'événements

La seconde analyse permet de déterminer si deux événements e et f peuvent se succéder, c.-à-d. que f est exécutable après que e ait été éxécuté. Cette analyse utilise l'ensemble des événements faisables et potentiellement faisables. L'équation suivante est vérifiée pour chaque couple d'événements par résolution de contrainte, où BA_e représente le prédicat avant-après de l'événement e et Grd'_f la garde de l'évènement f après l'exécution de e :

$$Inv \wedge BA_e \wedge Grd'_f$$

Cette analyse est représentée dans l'algorithme par la fonction $feasibleAfter(e)$. De la même façon que précédemment, une limite de temps est définie. Si cette limite est atteinte, nous considérons que f appartient à $feasibleAfter(e)$.

4.5.2.3 Analyse d'activabilité

Finalement, l'analyse d'activabilité permet de déterminer si l'exécution de l'événement e a rendu l'événement f exécutable, c.-à-d. que f n'était pas exécutable avant que e ne soit exécuté et que f le devienne juste après l'exécution de e.

$$\neg Grd_f \wedge Inv \wedge BA_e \wedge Grd'_f$$

Cette analyse est noté $enable(e)$. Dans le cas où aucun dépassement de temps n'est atteint, nous pouvons constater que $enable(e) \subseteq feasibleAfter(e)$.

4.5.2.4 Algorithme proposé

L'algorithme proposé dans la figure 6 permet d'extraire des tests à partir d'un modèle Event-B. Basé sur un algorithme de parcours en largeur de graphe, il tente de trouver des traces permettant d'exécuter les événements sous test. Ces derniers sont représentés par l'ensemble *Targets*. L'algorithme termine lorsque tous les événements à tester ont été couverts ou que la profondeur maximum est atteinte. Les fonctions $feasibleAfter$ et *enable* ont été généralisées et peuvent prendre une trace en paramètre. Dans ce cas, le prédicat avant-après représente la trace. La variable *Paths* est une liste de traces. La notation [ ] représente une trace vide. La variable *depth* représente la profondeur courante. Elle permet de déterminer si la profondeur maximum a été atteinte et de définir la longueur des traces présentes dans *Path*. Les événements sous test n'ayant pas été inclus dans un test sont stockés dans la variable *remainingTargets*. Finalement, les tests sont conservés dans l'ensemble *Tests*. À chaque itération de l'algorithme de parcours, la trace est étendue. La boucle principale est représentée par la ligne 6. Chaque itération se compose de 3 étapes. La première, représenté par les lignes 7 à 14, permet d'étendre une trace en ajoutant des événements sous tests. Pour chaque trace, une liste d'événements activables à tester est générée. Pour chacune de ces nouvelles traces, ProB tente d'en

trouver une instanciation par résolution de contraintes. Si une solution est trouvée, cela correspond à un test. Ce dernier est conservé et l'événement traité est retiré de l'ensemble des événements à couvrir. La seconde étape, représenté par les lignes 16 à 18, vérifie qu'il reste toujours des objectifs de test à satisfaire. Si tous les objectifs de test ont été satisfaits, l'algorithme termine. La troisième étape, représentée par les lignes 20 à 29, étend les traces sans ajouter d'événement sous test. La liste des traces est mise à jour et le processus recommence avec des traces plus grandes. Un exemple est donné pour le cas d'étude sur la vérification de type en section 5.3.11.

Algorithm 6 Algorithme d'extraction des tests.

```
1: procedure testsExtraction(Targets, maxDepth)
2:     Paths := {[ ]}
3:     depth := 0
4:     remainingTargets := Targets
5:     Tests := ∅
6:     while depth ≤ maxDepth do
7:         for all p ∈ Paths do
8:             for all t ∈ remainingTargets ∩ enable(p) do
9:                 if solveConstraints(p ← t) then
10:                    Tests ∪= {p ← t}
11:                    remainingTargets \= {t}
12:                end if
13:            end for
14:        end for
15:
16:        if remainingTargets ≠ ∅ then
17:            break
18:        end if
19:
20:        Paths′ := ∅
21:        for all p ∈ Paths do
22:            for all e ∈ feasibleAfter(p) \ Targets do
23:                if solveConstraints(p ← e) then
24:                    Paths′ ∪= {p ← e}
25:                end if
26:            end for
27:        end for
28:        Paths := Paths′
29:        depth += 1
30:    end while
31: end procedure
```

4.6 Conclusion

Les méthodes existantes ne permettant pas de tester efficacement des implémentations embarquées de VCI , nous avons proposé une méthode de génération de tests de

vulnérabilité fondée sur la méthode Event-B et la mutation de modèle (contexte et machine). Cette méthode se compose de trois parties. La première partie, l'aplanissement de modèle, permet de rassembler les informations de tous les raffinements dans un unique niveau de raffinement. Les comportements du modèle produit sont identiques à ceux du modèle originel. Cette adaptation du modèle est nécessaire pour le rendre compatible avec l'algorithme de mutation. Dans la seconde partie, le modèle du SUT est muté pour représenter de potentielles fautes. Nous proposons deux algorithmes, l'un pour la mutation de contexte et l'autre pour la mutation de machine. Pour garantir le processus de mutation, les parties les plus importantes ont été vérifiées. Cela est notamment le cas pour les règles de mutation de formules. La dernière partie consiste à extraire des tests à partir des modèles de faute. Un algorithme de résolution de contraintes basé sur ProB est proposé.

Au cours de ces recherches, de multiples itérations ont permis de rendre la méthode de plus en plus générique. Des cas d'études non présentés dans le cadre de cette thèse ont permis de vérifier son fonctionnement sur des SUT de plus haut niveau. Le protocole de paiement EMV [58] a permis de montrer l'applicabilité du VTG sur des modèles UML.

Chapitre 5 :

Application au vérifieur de code intermédiaire

La méthode de VTG a été appliquée au problème de la vérification du VCI . Les tests que nous cherchons à produire sont des applications Java Card compilées et stockés dans des fichiers CAP. Pour cela, nous souhaitons générer des fichiers CAP à partir des contraintes du VCI . Pour simplifier les expérimentations, la vérification de structure et la vérification de type ont été étudiées indépendamment. Elles sont respectivement décrites dans les sections 5.2 et 5.3. Pour chacun de ces cas d'étude, différents modèles Event-B ont été proposés. Les différents cas d'utilisation de ces modèles sont décrits dans la section 5.1. Chaque modèle proposé représente un algorithme de génération de fichiers CAP valides. Le VTG est utilisé pour y introduire des fautes et générer des fichiers CAP invalides par mutation et résolution de contraintes.

5.1 Utilisation des modèles

L'utilisation des modèles proposés peut se faire dans plusieurs sens. En nous basant sur un algorithme de résolution de contraintes, tel que celui de ProB, il est possible de vérifier si une valeur satisfait les contraintes du modèle ou de trouver une valeur satisfaisant ces contraintes. Ainsi, nous pouvons vérifier un fichier CAP existant ou en construire un (ce dernier pouvant être valide ou non).

5.1.1 Vérification de fichiers CAP

Le modèle peut-être utilisé afin de vérifier un fichier CAP. Cette utilisation correspond à celui d'un VCI . Pour cela, le fichier CAP est traduit en modèle Event-B puis combiné avec le modèle du VCI . Cette traduction est effectuée automatiquement par l'outil CAP2Rodin décrit dans la section 6.2.1. Nous obtenons ainsi un modèle que nous appelons modèle prérempli. Dans le modèle prérempli, les valeurs des constantes et des variables sont ainsi fixées. L'algorithme de résolution de contraintes vérifie simplement que ces valeurs satisfont les contraintes du modèle.

En sortie, nous pourrons obtenir les résultats suivants : une validation du modèle, un contre-exemple ou, pas de solution. Dans le cas d'un modèle validé, cela signifie que le fichier CAP respecte la norme Java Card, c.-à-d. qu'un VCI peut accepter ce fichier. Dans le cas d'un contre-exemple, cela signifie qu'au moins une contrainte n'a pas été respectée, c.-à-d. qu'un VCI doit rejeter ce fichier. Dans le cas où nous n'avons pas trouvé de solution, cela signifie que nous n'avons pas pu trouver de contre-exemple. Dans une implémentation du VCI , il serait alors impératif de rejeter le fichier CAP sous peine d'accepter l'installation d'applications potentiellement malveillantes.

Malgré que cette utilisation ne réponde pas à notre objectif de test, cela nous a permis de vérifier la validité de notre modèle. Pour cela, nous avons utilisé un ensemble de fichiers CAP valides et de fichiers CAP invalides. Cette méthode a l'avantage d'être plus simple et plus rapide que la preuve complète du modèle.

5.1.2 Construction de fichiers CAP valides

À l'inverse, si nous voulons générer un fichier CAP à partir du modèle, il faut trouver des valeurs pour toutes les constantes et variables. La première solution serait de partir d'un modèle de valeurs totalement vides. Cependant, le nombre de possibilités

est généralement trop important. Pour guider la recherche de solutions, nous utilisons un modèle partiellement prérempli. Dans ce dernier, certaines constantes et variables ont des valeurs fixées, tandis que d'autres sont laissées libres. Les valeurs manquantes sont retrouvées par résolution de contraintes. Cette méthode peut-être utilisée afin de générer des fichiers CAP valides, permettant de tester la partie fonctionnelle d'implémentations de VCI .

5.1.3 Construction de fichiers CAP invalides

L'utilisation de la résolution de contraintes sur un modèle valide permet de générer des fichiers CAP valides. À l'inverse, si nous utilisons un modèle muté obtenu à l'aide du VTG, les fichiers CAP générés sont ainsi invalides. Cette utilisation correspond à celle souhaitée afin de générer des tests d'intrusion.

5.2 Test de la vérification de structure des fichiers CAP

Le mécanisme de vérification de structure, décrit dans la section 3.2.1, est un problème statique du point de vue de l'Event-B. Le modèle est donc constitué uniquement de contextes. Afin d'offrir une meilleure traçabilité entre la spécification Java Card et le modèle Event-B de cette spécification, une architecture et des conventions encadrent ce modèle. Ces dernières, respectivement décrites dans les sections 5.2.1 et 5.2.2, ont été obtenues empiriquement pour rendre le modèle compatible avec la méthode de VTG. Tous les champs des fichiers CAP qui sont chargés dans une carte ont été modélisés. 130 contextes permettant de décrire les champs d'un fichier CAP. 250 constantes permettent de représenter les différents champs du fichier CAP. Les contraintes fonctionnelles sont définies à l'aide d'environ 400 axiomes. La vérification de structure étant un problème vaste, seules les contraintes en rapport avec le mécanisme de vérification des relations d'héritage ont été testées. Ce mécanisme nécessite 24 axiomes pour être définis. Leur modèle est donné dans la section 5.2.3. L'utilisation du VTG, pour obtenir ces tests, est décrite dans la section 5.2.4. Ce cas d'étude a été réalisé conjointement avec Mathieu Lassale dans le cadre de sa maitrise [59].

5.2.1 Architecture et conventions

Un fichier CAP est composé de douze composants. Ces composants sont modélisés à l'aide de groupes de contextes, qui respectent la même architecture. La figure 5.1 représente deux composants, représentés par les rectangles à bord arrondi. Les boites à bord droit sont des groupes de contextes Chaque ovale représente un contexte possédant une fonction particulière. Une flèche d'un contexte A vers un contexte B indique que A est une extension de B.

Quatre contextes sont communs à l'ensemble des composants. Le contexte *Primitive Types* contient les types de base définis par la spécification tels que les octets. Le contexte *Common Structure* contient des structures de données plus complexes. Le contexte *Constants* contient les valeurs statiques définies dans la spécification telle que *DECAFED*.

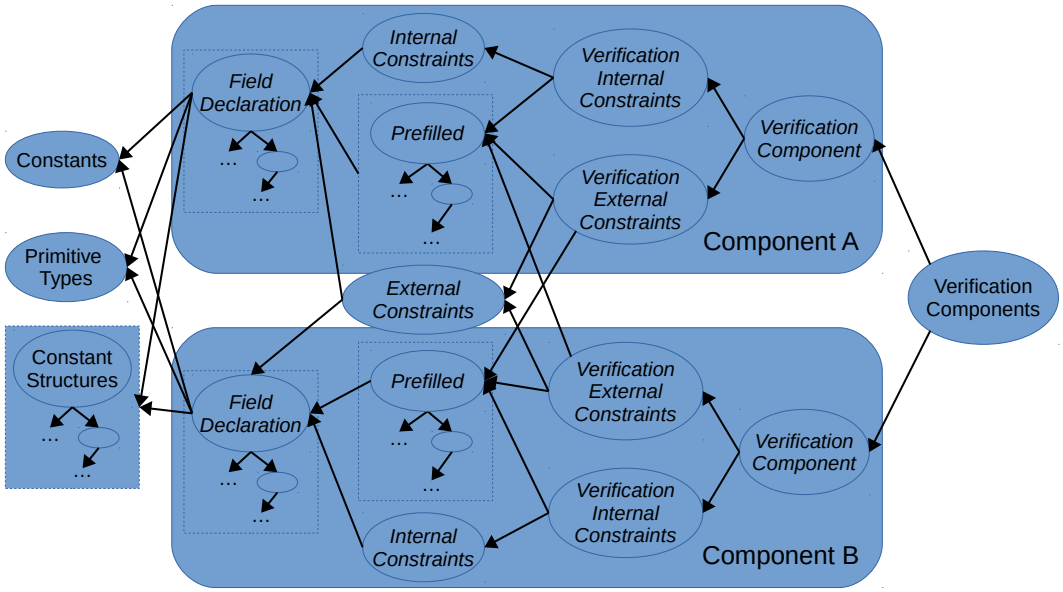

FIGURE 5.1 – Architecture du modèle de la vérification de structure

Finalement, le contexte *Components Verification* est utilisé pour faire l'union de tous les contextes de la spécification, c.-à-d. qu'il étend tous les autres contextes.

Chaque composant possède son propre ensemble de contextes. Les contextes *Structure* représentent les champs ainsi que leur occupation mémoire. Les contextes *Prefilled* contiennent les valeurs des champs dans le cas d'un modèle prérempli. Les contextes *Internal Constraints* représentent les contraintes internes. Les contextes *External Constraints* permettent de définir les contraintes entre plusieurs composants. Pour ne pas dupliquer l'expression de contraintes externes, ces contextes sont externes aux composants. Les contextes de vérification, *Internal Constraints Verification*, *External Constraints Verification*, *Component Verification* et *Components Verification* permettent de vérifier les contraintes auxquelles ils se rapportent. La partie à tester du modèle est représentée par les contextes des contraintes internes et les contextes des contraintes externes. Les autres contextes correspondent à la partie fonctionnelle du modèle.

5.2.2 Représentation de données d'un fichier CAP

Trois types de données existent [1] : les types primitifs, les tableaux et les structures.

1. Des cas particuliers sont détaillés dans le mémoire de maitrise de Mathieu Lassale [59].

5.2.2.1 Types primitifs

Il y a quatre types primitifs dans le fichier CAP : *bit_4*, *u1*, *u2* et *u4*. Ils sont modélisés à l'aide de 4 constantes et des 4 axiomes suivants :

$$\begin{aligned} bit\_4 &\in 0 \mathinner{..} 127 \\ u1 &\in 0 \mathinner{..} 255 \\ u2 &\in 0 \mathinner{..} 65535 \\ u4 &\in 0 \mathinner{..} 4294967295 \end{aligned}$$

Dans la suite de ce document, la notation u correspond à *bit_4*, *u1*, *u2* ou *u4*. Un champ Chp de type primitif est donc noté : $Chp \in u$.

5.2.2.2 Tableaux de types primitifs

Un tableau est défini à l'aide de deux constantes qui représentent respectivement sa taille et son contenu. Tous les indices d'un tableau devant correspondre à un unique élément du tableau, son contenu est modélisé par une fonction totale. Un tableau Tab de type u est modélisé à l'aide de 2 constantes et de deux axiomes :

$$\begin{aligned} TabLength &\in u \\ Tab &\in 1 \mathinner{..} TabLength \rightarrow u \end{aligned}$$

5.2.2.3 Structures

Une structure est un ensemble de champs. Chaque champ peut être un type primitif, un tableau de types primitifs, une structure ou un tableau de structures. Une structure est donc une composition de champs représentée par un ensemble porteur reliant les champs qui la constitue. Une fois définie, elle peut s'utiliser de la même façon qu'un type primitif. Prenons par exemple une structure Str contenant un champ de type primitif Chp et un tableau de type primitif Tab. Cette structure est définie par les axiomes suivants :

$$\begin{gathered} finite(Str) \\ Chp \in Str \rightarrow u \\ TabLength \in Str \rightarrow u \\ Tab \in Str \rightarrow (u \nrightarrow u) \\ \forall s \cdot \quad s \in Str \Rightarrow dom(Tab(s)) = 1 \mathinner{..} TabLength(s) \end{gathered}$$

Chaque instance de la structure est représentée par un élément de l'ensemble porteur Str. Pour pouvoir être concrétisée, l'ensemble porteur Str est de taille finie. Les champs Chp et $TabLength$ sont chacun représentés par une fonction qui associe à chaque instance de la structure une valeur particulière. Pour le tableau, le domaine étant différent pour chaque instance de la structure, il n'est pas possible de le définir à l'aide d'une fonction totale. Pour cela, une contrainte assure que pour chaque instance de la structure, la taille

de son tableau est cohérente avec la taille définie par la variable *TabLength* pour cette instance.

Prenons maintenant un exemple d'instanciation de ce modèle. Prenons deux structures *Str0* et *Str1* toutes deux composées d'un champ et d'un tableau. Le champ de *Str0* vaut 16 et celui de *Str1* vaux 42. La structure *Str0* contient un tableau vide. La structure *Str1* contient un tableau qui contient 3 éléments : 2, 4 et 8.

$$Chp = \{Str0 \mapsto 16, Str1 \mapsto 42\}$$
$$TabLength = \{Str0 \mapsto 0, Str2 \mapsto 3\}$$
$$Tab = \{Str0 \mapsto \{\}, Str1 \mapsto \{(0 \mapsto 2), (1 \mapsto 4), (2 \mapsto 8)\}\}$$

5.2.2.4 Example de l'héritage des interfaces

Dans le composant `class`, les informations sur les interfaces sont stockées dans la structure `interface_info`. Dans cette structure, deux champs sont impliqués dans l'héritage : `cc_ii_bitfield_interface_count` et `cc_ii_superinterfaces`. Pour la structure `interface_info`, nous avons ajouté un champ `offset`. Java supportant l'héritage multiple pour les interfaces, il est donc stocké sous la forme d'un tableau. Ce tableau contient tous les parents directs et indirects d'une interface. Sa taille est contenue dans le champ `cc_ii_bitfield_interface_count`. Le tableau est défini dans le champ `cc_ii_superinterfaces`. Il peut être vide si l'interface n'hérite que de `Object`. Ces propriétés structurelles sont exprimées à l'aide des axiomes suivants :

$$\begin{array}{l}
\texttt{finite(interface_info)} \\
\texttt{cc_ii_offset} \in \texttt{interface_info} \rightarrow \texttt{u2} \\
\texttt{cc_ii_bitfield_interface_count} \in \texttt{interface_info} \rightarrow \texttt{bit_4} \\
\texttt{cc_ii_superinterfaces} \in \texttt{interface_info} \rightarrow (\texttt{bit_4} \nrightarrow \texttt{u2}) \\
\forall \texttt{interface} \cdot (\texttt{interface} \in \texttt{interface_info} \\
\quad \Rightarrow \texttt{dom(cc_ii_superinterfaces(interface))} = \\
\qquad \texttt{1} \mathinner{..} \texttt{cc_ii_bitfield_interface_count(interface))}
\end{array} \tag{5.1}$$

5.2.3 Modélisation des contraintes d'héritage

La relation d'héritage sur les interfaces forme un treillis. Cet ordre s'applique sur les offsets des interfaces définies dans l'application. Chaque interface doit avoir un offset plus grand que celui de ses parents. Cette caractéristique est représentée par l'axiome suivant :

$$\begin{array}{l}
\forall \texttt{interface} \cdot (\\
\quad \texttt{interface} \in \texttt{interface_info} \\
\quad \Rightarrow (\forall \texttt{sup} \cdot (\\
\qquad \texttt{sup} \in \texttt{ran(cc_ii_superinterfaces(interface))} \\
\qquad \Rightarrow \texttt{cc_ii_offset(interface)} > \texttt{sup} \\
)))
\end{array} \tag{5.2}$$

En Java une interface ne peut pas hériter deux fois de la même interface. L'axiome

suivant permet de garantir cette propriété :

$$\begin{array}{l}\forall \texttt{interface}\cdot(\texttt{interface} \in \texttt{interface\_info}\\ \quad \wedge \texttt{cc\_ii\_bitfield\_interface\_count}(\texttt{interface}) > 1\\ \quad \Rightarrow(\forall \texttt{super\_interface1}, \texttt{super\_interface2}\cdot(\\ \qquad \texttt{super\_interface1} \in \texttt{dom}(\texttt{cc\_ii\_superinterfaces}(\texttt{interface}))\\ \qquad \wedge \texttt{super\_interface2} \in \texttt{dom}(\texttt{cc\_ii\_superinterfaces}(\texttt{interface}))\\ \qquad \wedge \texttt{super\_interface1} \neq \texttt{super\_interface2}\\ \qquad \Rightarrow \texttt{cc\_ii\_superinterfaces}(\texttt{interface})(\texttt{super\_interface1}) \neq\\ \qquad\quad \texttt{cc\_ii\_superinterfaces}(\texttt{interface})(\texttt{super\_interface2})\\ )))\end{array} \tag{5.3}$$

5.2.4 Test des contraintes d'héritage

La première étape du VTG est l'aplanissement du modèle. Ce dernier étant constitué uniquement de contextes, il n'est pas nécessaire de l'aplanir. La seconde étape est la mutation. L'algorithme 3 permet de générer les modèles mutants. Finalement il est nécessaire d'en extraire des tests.

5.2.4.1 Critères de couverture.

Pour nous assurer que les tests couvrent toutes les propriétés que nous cherchons à vérifier, nous avons utilisé les critères de couverture des données. Partant de l'hypothèse que tous les tests sont équivalents, le critère *One Value* pour chaque mutant est suffisant.

5.2.4.2 Les mutations obtenues

Nous présentons ici un exemple de mutation pour chaque contrainte. La mutation de $\forall$ a été modifiée, car elle entraîne la création d'un $\exists$ selon les règles définies dans la section 4.3. Or ce quantificateur ne concerne qu'un élément de l'ensemble concerné, laissant tous les autres éléments libres de toutes contraintes. Pour éviter la génération de fautes multiples, le quantificateur $\exists$ de la mutation contient la formule originale. Grâce à cette règle, les éléments, qui ne sont pas concernés par ce quantificateur, sont contraints en respectant la spécification. Les équations 5.4 et 5.5 sont respectivement les mutations des équations 5.2 et 5.3.

$$\begin{array}{l}\exists \texttt{interface}, \texttt{sup}\cdot(\texttt{interface} \in \texttt{interface\_info}\\ \quad \wedge \texttt{sup} \in \texttt{ran}(\texttt{cc\_ii\_superinterfaces}(\texttt{interface}))\\ \quad \wedge \texttt{sup} \in \texttt{ran}(\texttt{cc\_ii\_offset})\\ \quad \wedge \mathbf{ic\_class\_ref(cc\_ii\_offset(interface)) = ic\_class\_ref(sup)}\\ \quad \wedge \forall \texttt{int}\cdot(\texttt{int} \in \texttt{interface\_info}\\ \qquad \wedge \texttt{interface} \neq \texttt{int}\\ \qquad \Rightarrow \forall \texttt{sup1}\cdot(\texttt{sup1} \in \texttt{ran}(\texttt{cc\_ii\_superinterfaces}(\texttt{int}))\\ \qquad\quad \wedge \texttt{sup1} \in \texttt{ran}(\texttt{cc\_ii\_offset})\\ \qquad\quad \Rightarrow \texttt{ic\_class\_ref}(\texttt{cc\_ii\_offset}(\texttt{int})) = \texttt{ic\_class\_ref}(\texttt{sup1})\\ )))\end{array} \tag{5.4}$$

$$
\begin{array}{l}
\exists \texttt{interface}, \texttt{sup1}, \texttt{sup2} \cdot (\texttt{interface} \in \texttt{interface\_info} \\
\quad \wedge \texttt{sup1} \in \texttt{dom}(\texttt{cc\_ii\_superinterfaces}(\texttt{interface})) \\
\quad \wedge \texttt{sup2} \in \texttt{dom}(\texttt{cc\_ii\_superinterfaces}(\texttt{interface})) \\
\quad \wedge \texttt{sup1} \neq \texttt{sup2} \\
\quad \wedge \mathbf{cc\_ii\_superinterfaces(interface)(sup1)} = \\
\qquad \mathbf{cc\_ii\_superinterfaces(interface)(sup2)} \\
\quad \wedge \forall \texttt{int} \cdot (\texttt{int} \in \texttt{interface\_info} \\
\qquad \wedge \texttt{interface} \neq \texttt{int} \\
\qquad\quad \Rightarrow (\forall \texttt{supi1}, \texttt{supi2} \cdot (\texttt{sup1} \in \texttt{dom}(\texttt{cc\_ii\_superinterfaces}(\texttt{interface})) \\
\qquad\qquad \wedge \texttt{sup2} \in \texttt{dom}(\texttt{cc\_ii\_superinterfaces}(\texttt{interface})) \\
\qquad\qquad \wedge \texttt{sup1} \neq \texttt{sup2} \\
\qquad\qquad \Rightarrow \texttt{cc\_ii\_superinterfaces}(\texttt{interface})(\texttt{supi1}) \neq \\
\qquad\qquad\quad \texttt{cc\_ii\_superinterfaces}(\texttt{interface})(\texttt{supi2}) \\
\quad))))
\end{array}
\tag{5.5}
$$

5.2.4.3 Extraction des tests

La résolution de contraintes est ensuite utilisée sur les modèles mutants pour extraire les tests. Ce problème entraîne une explosion combinatoire dans ProB. Il est donc nécessaire d'utiliser un modèle prérempli qui a été conçu manuellement en s'inspirant d'un fichier CAP existant. Pour notre exemple, nous utilisons un modèle préremplis contenant les 2 interfaces définies de la façon suivante :

$$partition(interface\_info, \{interface\_info1\}, \{interface\_info2\})$$

L'instantiation de la mutation 5.4, qui modifie l'ordre associé à l'héritage, est représentée par la figure 5.2. Les valeurs suivantes, où les effets de la mutation sont en rouge, sont obtenues par résolution de contraintes :

$$
\begin{array}{l}
cc\_ii\_offset = \{interface\_info1 \mapsto 0, interface\_info2 \mapsto 1\} \\
cc\_ii\_bitfield\_interface\_count = \{interface\_info1 \mapsto 0, interface\_info2 \mapsto 2\} \\
cc\_ii\_superinterfaces = \{interface\_info1 \mapsto \{\}, ..., \\
\qquad\qquad\qquad\qquad interface\_info2 \mapsto \{(1 \mapsto interface\_info2)\}\}
\end{array}
$$

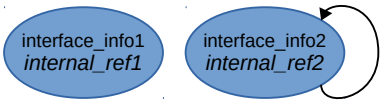

FIGURE 5.2 – Héritage cyclique

L'instantiation de la mutation 5.5, illustrée en figure 5.3, représente les cas où `interface_info2` hérite deux fois de la même interface. Elle est obtenue à partir de

la mutation de la seconde contrainte.

$$cc\_ii\_offset = \{interface\_info1 \mapsto 0, interface\_info2 \mapsto 1\}$$
$$cc\_ii\_bitfield\_interface\_count = \{interface\_info1 \mapsto 0, interface\_info2 \mapsto 2\}$$
$$cc\_ii\_superinterfaces = \{interface\_info1 \mapsto \{\},$$
$$interface\_info2 \mapsto \{(1 \mapsto interface\_info1), (2 \mapsto interface\_info1)\}\}$$

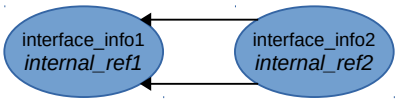

FIGURE 5.3 – Duplication de la relation d'héritage

5.2.4.4 Résultats expérimentaux

Le VTG a permis de générer 5 modèles mutants. L'un de ces modèles n'était pas satisfaisable. De ces mutants, 4 tests ont été générés. Ces derniers ont été utilisés pour vérifier le VCI d'Oracle et n'ont révélés aucune défaillance.

5.3 Test de la vérification de type des fichiers CAP

Les tests d'intrusions ont pour but de détecter des défaillances tout en simplifiant leur correction. Comme nous l'avons vu dans la section 3.2.3, la vérification de type s'effectue indépendamment pour chaque méthode. Pour remonter plus simplement à l'origine de la défaillance, une seule méthode contient la charge active du test. Dans ce cas d'étude, le modèle s'intéresse donc à produire une unique méthode. Du point de vue de l'Event-B, il est plus simple de représenter ce processus à l'aide d'un modèle dynamique, c.-à-d. à l'aide de machines. Le modèle complet est donné en annexe 11. Les tests abstraits obtenus à partir du modèle proposé représentent le tableau d'octets de cette méthode, ainsi que certaines informations de son entête, tout en assurant le respect de la spécification de la JCVM. La mutation du modèle permet de produire des instructions dont leurs préconditions ne respectent pas la spécification. Les tests qui en sont extraits permettent de produire des méthodes dont l'une des instructions ne respecte pas la spécification. Pour la concrétisation des tests, un fichier CAP prérempli possède une méthode prévue pour contenir le test. Dans nos expérimentations, nous utilisons la méthode 6 du fichier donné en annexe 10. Chaque instruction de Java Card est représentée par un ou plusieurs évènements dans la machine. L'exécution de l'événement représente l'occurrence de l'instruction correspondante dans le tableau d'octets de la méthode générée. Une trace d'exécution de la machine représente donc le code d'une méthode, et constitue un test. La garde d'un événement représente les vérifications de type pour la pile et les variables locales de la méthode. Il possède également un certain nombre d'éléments, tel que des classes, utilisées à nos tests.

5.3.1 Architecture et périmètre du modèle

Le modèle est décomposé en 7 raffinements introduisant successivement les mécanismes nécessaires à la vérification de type. Les événements abstraits caractérisent des ensembles d'instructions et sont décomposés en différents événements concrets pour représenter des sous-ensembles plus précis d'instructions. Le modèle B classique, donné en annexe 9, définit ces ensembles. Chaque ensemble représente des caractéristiques pour les instructions. L'ensemble *ASDrThInst* représente les instructions prises en compte dans ce travail. Le modèle est composé de 66 événements représentant 62 [2] instructions Java Card. Le premier niveau de spécification, décrit dans la section 5.3.2, représente les contraintes relatives au flot de contrôle. Les structures de données sont successivement introduites dans les niveaux de spécification subséquents. L'ordre dans lequel celles-ci peuvent être modélisées n'est pas contraint. Dans notre modèle, nous avons choisi de les introduire dans l'ordre suivant : la pile (sections 5.3.3, 5.3.4 et 5.3.5), puis les variables locales (section 5.3.6) et enfin le tas (section 5.3.7). Dans le dernier niveau de raffinement, présenté dans la section 5.3.8, chaque événement représente une instruction. Les contraintes à tester sont modélisées par 164 gardes réparties dans les différents événements et raffinements. Ce modèle étant destiné à être utilisé pour générer des tests de vulnérabilité, certaines adaptations ont été nécessaires et seront décrites pour chaque raffinement.

Notre modèle ne prend actuellement pas en compte les mécanismes d'exceptions, l'accès aux champs des objets et les appels de fonctions. Une modélisation plus complète de la structure des fichiers CAP est nécessaire. Malgré cela, l'instruction `invokespecial` et le composant `Constant Pool`, permettant la création d'objets initialisés, ont été partiellement implémentés et seront décrits dans la section 5.3.7. De plus, la gestion du flot de contrôle est partiellement modélisée.

5.3.2 Flot de contrôle

Le modèle le plus abstrait représente la vérification du flot de contrôle. Dans ce modèle, donné en figure 5.4, chaque événement représente respectivement les sous-ensembles d'instructions *returnIntructions* et *noReturnIntructions*. À chaque étape d'exploration, un événement, dont la garde est respectée, est sélectionné. Dans le cas d'un programme linéaire, c.-à-d. sans branchement, la terminaison de la génération du tableau d'octets survient lorsqu'une instruction de retour est ajoutée. Après l'exécution de l'événement *returnInstructions*, aucune autre instruction ne peut survenir dans notre modèle. Cet événement conduit donc à une impasse et stoppe la recherche de solution. Il n'est pas nécessaire d'utiliser une variable pour construire le tableau d'octets. Chaque transition correspond à une portion du tableau d'octet. Cela permet de rendre l'exploration de modèle beaucoup plus rapide.

5.3.3 Taille de la pile

Le premier niveau de raffinement représente les modifications de la taille de la pile. Chaque événement du raffinement précédent peut être décomposé en 6 nouveaux événements. Dans notre modèle, les instructions Java Card peuvent : ne pas changer la

2. 34% des instructions Java Card sont ainsi modélisées.

Event *noReturnEvent_R01* $\widehat{=}$
  when
    grd1 : programRunning = TRUE
  then
    skip
  end
Event *returnEvents_R01* $\widehat{=}$
  when
    grd1 : programRunning = TRUE
  then
    act1 : programRunning := FALSE
  end

FIGURE 5.4 – Événements de contrôle du flot d'exécution d'un programme linéaire

taille de la pile, incrémenter la taille de la pile d'une ou deux unités, décrémenter la taille de la pile d'une, deux ou trois unités.

La constante *MaxStackSize* définit la taille maximum que peut prendre la pile. Cette constante correspond à la valeur de la taille maximum de la pile que l'on retrouve dans l'entête des méthodes. À l'initialisation du contexte, la valeur de cette constante est choisie de façon non déterministe. Pour borner la recherche de solution, nous avons limité sa borne supérieure à 9. Cette limite a été obtenue expérimentalement et permet de générer tous les tests appropriés. La variable *stackSize* représente la taille de la pile. À l'initialisation, la pile étant vide, cette variable est initialisée à 0.

Dans ce raffinement, les gardes des événements garantissent que la taille de la pile est toujours positive et, inférieure ou égale à sa taille maximum. Ces gardes représentent deux objectifs de test et seront utilisées lors de la mutation. Selon les règles de mutation que nous avons définies, l'utilisation des symboles $<$ et $>$ disperse mieux[3] les modèles mutants que les symboles $\leq$ et $\geq$. Les gardes utilisent dont des symboles $<$ et $>$.

L'événement donné en figure 5.5 représente les instructions qui incrémentent d'une unité la taille de la pile. Après raffinement, cet événement est, par exemple, utilisé pour représenter l'instruction `sconst_3`.

5.3.4 Nombre d'éléments manipulés sur la pile

Le deuxième niveau de raffinement de la pile représente les éléments de la pile en entrée et les sorties des instructions. Les éléments en entrée sont dépilés, puis l'instruction commence son traitement. Après l'exécution de l'instruction, des éléments peuvent être empilés. Ce modèle assure que la pile contient au moins autant d'éléments que d'entrées nécessaires pour l'exécution d'une instruction. Le modèle assure aussi le respect des modifications de taille définie dans le raffinement précédent. Aucune nouvelles constante ou variable n'est ajoutée, seule la garde des événements est raffinée par renforcement.

3. Cela permet un échantillonnage sur des espaces plus précis.

Event *stackSizeIncrease1_R02* $\widehat{=}$
extends *noReturnEvent_R01*

when

grd1 : $programRunning = TRUE$

grd2_t : $stackSize < MaxStackSize$

then

act1 : $stackSize := stackSize + 1$

end

FIGURE 5.5 – Événement d'incrément de la taille de la pile

Event *2stackElementInput_1stackElementOutput_R03* $\widehat{=}$
extends *stackSizeDecrease1_R02*

when

grd1 : $programRunning = TRUE$

grd2_t : $stackSize > 0$

grd3_t : $stackSize > 1$

then

act1 : $stackSize := stackSize - 1$

end

FIGURE 5.6 – Éléments de la pile en entrée et en sortie d'instructions

Pour la même raison que pour le raffinement précédent, les gardes utilisent des relations d'ordre strictes.

La figure 5.6 définit un événement prenant en entrée deux éléments sur la pile et empile un élément en sortie. La garde grd3_t est ajoutée afin d'assurer que suffisamment d'éléments se trouvent sur la pile. L'instruction `sadd` raffine cet événement.

5.3.5 Types et arbre d'héritage

Le troisième niveau de raffinement de la pile introduit les types Java Card. Ce niveau est composé d'un contexte définissant les types et d'une machine les utilisant.

Le contexte représente les types et leur organisation selon l'arbre d'héritage Java Card[4]. La figure 5.7 représente ce treillis de type. Les éléments clairs de la figure représentent les types que peut prendre la mémoire. Les éléments sombres sont utilisés pour l'inférence de type et ne peuvent être affectés à un espace mémoire. Pour vérifier la compatibilité de deux éléments, il est nécessaire de modéliser le treillis de type Java Card. En Event-B, cette structure de donnée peut-être calculée, mais nécessite l'utilisation d'une fermeture transitive trop coûteuse en temps de calcul. L'arbre d'héritage étant statique, il peut-être généré à l'avance et stocké dans une relation contenant les informations de

4. Le type *short* est noté *shart* pour éviter les problèmes de décomposition de mot en Rodin, *short* est parfois interprété comme la formule $sh \vee t$.

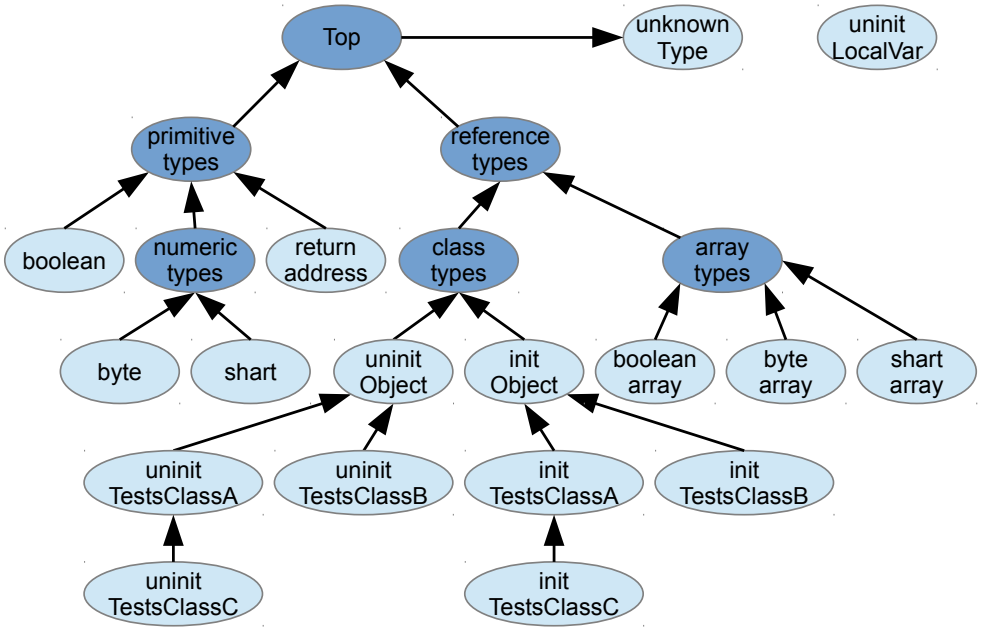

FIGURE 5.7 – Arbre d'héritage Java Card modélisé

compatibilité entre deux types. Pour effectuer ce calcul, nous avons produit un modèle en B Classique. Le résultat est ensuite intégré au modèle Event-B. Le type, *uninitLocalVar*, représente des éléments non initialisés dans les variables locales. Ces variables locales doivent tout d'abord être écrites avant de pouvoir être lues. Le type, *unknownType*, représente un type non connu. Les éléments au-dessus de la pile sont par exemple affectés à ce type. L'ajout de cet élément dans le modèle permet de rendre possibles certains tests de vulnérabilité.

Le type de chacun des éléments de la pile est représenté par la nouvelle variable `stackTypes` :

$$stackTypes \in -4 \mathinner{..} MaxStackSize - 1 \nrightarrow TYPES$$

Le domaine de cette relation représente les index des éléments dans la pile et son codomaine, représente le type associé. Pour les besoins de la mutation, nous avons dû instancier des éléments à des valeurs négatives. Les éléments possédant un index négatif représentent des éléments en-dessous de la pile. Leur type sera défini à *unknownType*. Nous avons déterminé expérimentalement que 4 éléments étaient nécessaires afin de générer l'ensemble des tests pour les attaques de débordement de pile vers le bas.

Pour généraliser la décomposition d'événements, la vérification du type n'est pas effectuée à ce niveau de raffinement. Les éléments de la pile en entrée et en sortie de chaque événement sont représentés par les paramètres des événements. Ces paramètres sont des types génériques et sont utilisés ultérieurement pour définir quel type doit être utilisé pour chaque instruction. Les types en entrée dépendent des éléments au sommet de la pile. Les modifications de la pile seront effectuées de façon générique.

L'événement de la figure 5.8 représente des instructions modifiant le type de l'élément au sommet de la pile. La variable représentant les types de la pile est modifiée sans prendre en compte le type des éléments manipulés. Les instructions `checkcast` et `newarray` sont représentées par cet événement.

Event *1stackElementInput_1stackElementOutput_R04* $\widehat{=}$
extends *1stackElementInput_1stackElementOutput_R03*

any

`firstStackInputType`

`firstStackOutputType`

where

grd1 : $programRunning = TRUE$

grd2_t : $stackSize > 0$

grd3 : $\texttt{firstStackInputType} \in \texttt{TYPES}$

grd4 : $\texttt{firstStackOutputType} \in \texttt{TYPES}$

grd5 : $\texttt{firstStackInputType} = \texttt{stackTypes}(\texttt{stackSize} - 1)$

then

act1 : $\texttt{stackTypes} := \texttt{stackTypes} \lhd \{\texttt{stackSize} - 1 \mapsto \texttt{firstStackOutputType}\}$

end

FIGURE 5.8 – Manipulation d'éléments typés sur la pile

5.3.6 Variables locales

La deuxième structure de donnée introduite représente les variables locales. Pour simplifier notre modélisation, nous avons choisi de ne pas représenter les éléments de type `int`. La valeur initiale des variables locales est définie par un contexte. Comme nous l'avons vu précédemment, nous utilisons un fichier CAP existant. Les arguments de notre méthode sont donc définis par l'entête de la méthode que nous utilisons pour générer les tests. Le tableau représentant les variables locales est généralement décomposé en deux parties : les arguments de la méthode et les autres variables locales. Elles sont respectivement représentées par les contraintes suivantes :

$$ArgumentTypes = \{0 \mapsto byte, 1 \mapsto shart, 2 \mapsto referenceTypes\}$$

$$FreeLocVar = (MaxArgumentIndex + 1 \mathinner{..} MaxLocVarIndex) \times \{uninitLocalVar\}$$

Une troisième partie, représentant des éléments en dehors de ce tableau, a été ajoutée dans le but de générer les tests de vulnérabilité. Cette caractéristique est représentée par la contrainte suivante :

$$OutLocVar = (MaxLocVarIndex + 1 \mathinner{..} MaxLocVarIndex + nbOutOfBoundIndex) \times \{unknownType\}$$

Event *0stackElemIn_1stackElemOut_load1LocalVariableOnTOS_R05* $\widehat{=}$
extends *0stackElementInput_1stackElementOutput_R04*

any

firstStackOutputType

`prm_index`

where

`grd1` : $programRunning = TRUE$

`grd2_t` : $stackSize < MaxStackSize$

`grd3` : $firstStackOutputType \in TYPES$

`grd4` : $\texttt{prm\_index} \in \texttt{dom}(\texttt{localVariablesTypes})$

`grd5_t` : $\texttt{prm\_index} \leq \texttt{MaxLocalVariablesIndex}$

`grd6` : $\texttt{firstStackOutputType} = \texttt{localVariablesTypes}(\texttt{prm\_index})$

then

`act1` : $stackSize := stackSize + 1$

`act2` : $stackTypes := stackTypes \cup \{stackSize \mapsto firstStackOutputType\}$

end

FIGURE 5.9 – Manipulation des variables locales

À l'initialisation du modèle, les variables locales sont fixées aux valeurs suivantes [5] :

$$initLocVar \subseteq ArgumentTypes \cup FreeLocVar \cup OutLocVar$$

L'événement de la figure 5.9 représente le chargement du contenu d'une variable locale au sommet de la pile. Le type de l'élément en sortie est contraint par le type lu dans les variables locales. L'instruction `sload_2` utilisera cet événement.

5.3.7 Initialisation des Objets

Le langage Java n'autorise pas un objet à être utilisé tant qu'il n'a pas été initialisé. La création d'un objet initialisé est une procédure composée de trois ou quatre étapes respectivement pour des objets statiques et non statiques. La première étape consiste à mettre au sommet de la pile un objet [6] non initialisé à l'aide de l'instruction `new`. Cette référence est ensuite dupliquée. Dans le cas d'une classe non statique, la référence vers la classe de l'appelant est poussée au sommet de la pile. Finalement l'instruction `invokespecial` initialise l'objet. Après cette procédure, l'élément au sommet de la pile est la référence vers l'objet initialisé.

Le modèle comporte l'ensemble des instructions permettant de réaliser ce processus. Cependant, la génération de trace de longueur trois et quatre requiert beaucoup de temps de calcul. Pour simplifier cette tâche, nous avons ajouté deux événements représentant la création d'un objet statique initialisé et d'un objet non statique initialisé. L'animation de

5. Le nom des variables est contracté : *LocVar* pour *LocalVariable*

6. Notre modèle ne prenant pas en compte les références, nous plaçons directement l'objet dans la pile.

Event *creatInitObject_R06* $\widehat{=}$
extends *0stackElemIn_1stackElemOut_read1LocalVariable_R05*

any

firstStackOutputType
prm_localVariables_index
`prm_constantPool_index`

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize < MaxStackSize$
grd3 : $firstStackOutputType \in TYPES$
grd4 : $prm\_localVariables\_index = 0$
grd5 : `prm_constantPool_index` $\in$ `dom(ConstantPool)`
grd6 : `firstStackOutputType` = `ConstantPool(prm_constantPool_index)`

then

act1 : $stackSize := stackSize + 1$
act2 : $stackTypes := stackTypes \cup \{stackSize \mapsto firstStackOutputType\}$

end

FIGURE 5.10 – Événement de création d'un objet non statique initialisé

la création d'objets initialisés est réalisée en une seule transition. La figure 5.10 représente l'initialisation d'un objet non statique.

Pour initialiser un objet, le modèle doit contenir les informations relatives aux références des classes dans le fichier CAP ainsi qu'une représentation des adresses. Le composant `Constant_Pool` est utilisé pour définir les références. Cependant, nous ne l'avons pas entièrement modélisé. Seules les références des classes sont modélisées. L'utilisation d'adresses requiert une modélisation du tas. Un modèle complet du tas étant trop complexe à animer, nous en avons proposé une version simplifiée en interdisant la duplication de référence.

5.3.8 Représentation des instructions

Le dernier niveau de raffinement introduit les instructions Java Card. Dans les raffinements précédents, le type des éléments en entrée des événements n'est pas contraint. L'ajout de gardes dans chaque événement permet de spécifier les types acceptés pour chaque instruction. Les indexes des variables locales sont pour certaines instructions fixes. Cette caractéristique est représentée par l'ajout d'une garde définissant la valeur exacte de l'index à utiliser. Certaines instructions possèdent des arguments n'intervenant pas dans le typage de la mémoire. Pour simplifier le modèle, nous ne les avons pas représentés. Ces valeurs seront choisies arbitrairement lors de la concrétisation des tests. L'instruction `saload` est modélisée par l'événement de la figure 5.11.

Event $saload\_R07 \mathrel{\widehat{=}}$
extends $2stackElemIn\_1stackElemOut\_R06$

any

$firstStackInputType$
$secondStackInputType$
$firstStackOutputType$

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize > 0$
grd3_t : $stackSize > 1$
grd4 : $firstStackInputType \in TYPES$
grd5 : $secondStackInputType \in TYPES$
grd6 : $firstStackOutputType \in TYPES$
grd7 : $firstStackInputType = stackTypes(stackSize - 1)$
grd8 : $secondStackInputType = stackTypes(stackSize - 2)$
grd9_t : `firstStackInputType` = `shart`
grd10_t : `secondStackInputType` = `shartArray`
grd11 : `firstStackOutputType` = `shart`

then

act1 : $stackSize := stackSize - 1$
act2 : $stackTypes := (\{stackSize-1, stackSize-2\} \lhd stackTypes) \cup \{stackSize-2 \mapsto firstStackOutputType\}$

end

FIGURE 5.11 – Événement `saload`

5.3.9 Exemple de génération de tests fonctionnels

Pour bien comprendre le fonctionnement du modèle proposé, nous allons donner quelques exemples de génération de tests fonctionnels. Ces tests sont des fichiers CAP acceptables par des VCI . Pour commencer, reprenons l'exemple du tableau d'octets donné dans la figure 3.3 du chapitre 3. Pour ce premier exemple, chaque étape de l'exploration du modèle Event-B est effectuée manuellement. La première étape consiste à initialiser le contexte, c.-à-d. fixer une valeur à toutes les constantes. Parmi ces constantes, certaines sont déterministes et d'autres peuvent prendre différentes valeurs. Pour les constantes non déterministes, le choix d'une valeur doit permettre de trouver la trace souhaitée. Dans notre modèle, une seule constante est non déterministe : `MaxStackSize`. Pour notre exemple, la taille maximum de la pile doit être supérieure ou égale à 2. La seconde étape de l'exploration est l'initialisation de la machine. Dans le modèle proposé, cette étape est déterministe. Après l'initialisation du contexte et de la machine, la taille de la pile est égale à 0 et est définie par la formule suivante :

$$\begin{aligned} stackTypes = \{ \quad & (-4 \mapsto unknownType), (-3 \mapsto unknownType), \\ & (-2 \mapsto unknownType), (-1 \mapsto unknownType) \ \} \end{aligned}$$

À ce stade, aucune instruction n'a été exécutée. Pour chacune des étapes d'animation, un événement va être exécuté. Chacune de ces exécutions représentera l'exécution d'une instruction. La garde de l'événement `sconst_3` impose que le programme ne soit pas fini et que la pile ne soit pas pleine. Aucune instruction de retour n'ayant été généré pour le moment, le programme est toujours en génération. La pile étant vide et pouvant contenir au maximum 2 éléments, il y a suffisamment de place pour y empiler un *short*. L'argument de l'événement représentant le type de l'élément à pousser au sommet de la pile est déterministe. Cet événement n'a donc qu'une instanciation possible. Pour les événements suivants, le même raisonnement permet de constater que ces instructions peuvent être exécutées. La trace obtenue, ainsi que les informations de l'entête de la méthode, représentent un test abstrait. Le fichier CAP prévu pour contenir ce test est le même que celui utilisé pour les tests de vulnérabilité.

Dans ce premier exemple, la trace était l'objectif de test. Pour le second exemple, notre objectif sera de trouver une trace permettant de tester l'instruction `sinc`. La recherche de trace est effectuée automatiquement à l'aide de l'algorithme 6. La trace suivante est obtenue :

[`SETUP_CONSTANTS`, `INITIALISATION`, `sinc(1)`, `return`]

5.3.10 Objectifs de test et critères de couverture

Le modèle mutant contient tous les événements mutants. Comme nous l'avons dit précédemment, un test est représenté par une trace. Chaque trace est composée d'un préambule et d'un corps. Le post-ambule est simplement constitué de l'instruction `return`. De plus, nous considérons que tous les préambules permettant d'activer un mutant sont équivalents, dans le but de réduire le nombre de tests. Nous utilisons les critères de couverture du graphe de transition. Le critère *All-events coverage* est appliqué à l'ensemble contenant tous les événements mutants.

Event *sload_R07* $\widehat{=}$
any
firstStackOutputType
prm_index
where
grd1 : programRunning = TRUE
grd2_t : stackSize < MaxStackSize
grd3 : firstStackOutputType ∈ TYPES
grd4 : prm_index ≥ 0
grd5_t : prm_index < MaxLocalVariablesIndex
grd6 : firstStackOutputType = localVariablesTypes(prm_index)
grd6_t : firstStackOutputType = shart
then
act1 : stackSize := stackSize + 1
act2 : stackTypes := stackTypes ∪ {stackSize ↦ firstStackOutputType}
end

FIGURE 5.12 – Événement `sload_R07`

5.3.11 Exemple de génération de tests de vulnérabilité

L'instruction *sload* est utilisée dans cette section pour illustrer le principe de génération de tests de vulnérabilité pour la vérification de type. Cette instruction charge au sommet de la pile le contenu d'une variable locale de type *short*. Elle possède 3 contraintes à tester :

1. la pile ne doit pas être pleine,
2. son paramètre *index* représente une position appartenant au domaine des variables locales,
3. la variable locale à la positon *index* doit être de type *short*.

Le modèle de cette instruction est donné dans la figure 5.12.

La première de la génération de tests de vulnérabilité étape consiste à générer le modèle aplani. Les modèles mutants sont ensuite obtenus à l'aide du modèle aplani. Les modèles mutants pour la vérification de type sont obtenus à l'aide des algorithmes 4 et 5. Les figures 5.13 et 5.14 représentent deux mutations parmi les 11 mutations possibles pour cette instruction. Les parties de garde mutées sont en rouge pour simplifier la lecture. Dans la première figure, l'index correspond à une position en dehors du tableau des variables locales. Dans la seconde figure, une combinaison de fautes est utilisée. Cet événement mutant permet de tester le comportement du système lorsque la pile est pleine et que l'élément à récupérer dans les variables locales n'est pas de type *short*.

Le modèle mutant est constitué d'une machine contenant tous les événements originaux ainsi que tous les événements mutants. L'algorithme 6 permet de chercher une trace pour satisfaire chacun de ces événements. Dans le cas de la figure 5.13, la trace trouvée est très courte. En effet, dès l'initialisation de la machine cet événement est exécutable. La trace

Event *sload_R07_EUT_615* $\widehat{=}$

any

firstStackOutputType

prm_index

where

grd1 : programRunning = TRUE

grd2_t : stackSize < MaxStackSize

grd3 : firstStackOutputType ∈ TYPES

grd4 : prm_index ≥ 0

grd5_t : prm_index > MaxLocalVariablesIndex

grd6 : firstStackOutputType = localVariablesTypes(prm_index)

grd6_t : firstStackOutputType = shart

then

act1 : stackSize := stackSize + 1

act2 : stackTypes := stackTypes ∪ {stackSize ↦ firstStackOutputType}

end

FIGURE 5.13 – Événement sload_R07_EUT_615

Event *sload_R07_EUT_620* $\widehat{=}$

any

firstStackOutputType

prm_index

where

grd1 : programRunning = TRUE

grd2_t : stackSize = MaxStackSize

grd3 : firstStackOutputType ∈ TYPES

grd4 : prm_index ≥ 0

grd5_t : prm_index ≤ MaxLocalVariablesIndex

grd6 : firstStackOutputType = localVariablesTypes(prm_index)

grd6_t : ¬(firstStackOutputType = shart)

then

act1 : stackSize := stackSize + 1

act2 : stackTypes := stackTypes ∪ {stackSize ↦ firstStackOutputType}

end

FIGURE 5.14 – Événement sload_R07_EUT_620

suivante est extraite :

$$[INITIALISATION, sload\_R07\_EUT\_615(3)]$$

Pour la figure 5.14, la trace doit être plus grande. En effet, l'un des gardes impose que la pile soit pleine. Le préambule consiste donc à trouver une trace permettant de remplir la pile. La trace suivante est extraite :

$$[INITIALISATION, aconst\_null, aconst\_null, aconst\_null, aconst\_null,$$
$$aconst\_null, aconst\_null, aconst\_null, aconst\_null, aconst\_null,$$
$$sload\_R07\_EUT\_615(1)]$$

Pour concrétiser ces tests, un fichier CAP est utilisé. Il contient une méthode vide que l'on remplit avec le contenu d'un test. Pour chaque test abstrait, un fichier CAP est généré. Chaque fichier CAP est mis en relation avec un événement mutant et une trace. Cela permet, dans le cas où un test met en avant une défaillance, de remonter plus facilement à la faute à l'origine du comportement observé.

5.3.12 Résultats expérimentaux

Ce cas d'étude est en cours de réalisation depuis le début de cette thèse. Différentes versions du modèle ainsi que de la méthode et des outils ont été proposés. Deux ensembles de tests ont été produits. Dans ce document, seule la dernière version du modèle, de la méthode et des outils développés est présentée. Bien que le second ensemble de tests [60] corresponde davantage à nos objectifs de test, le premier ensemble de tests [61] possède certaines caractéristiques intéressantes.

Le modèle du premier ensemble de tests représente certains types primitifs, la pile et les variables locales. Il est composé de 12 événements représentant chacun une instruction. Pour résoudre nos objectifs de test, deux solutions étaient envisagées. Comme nous l'avons vu dans la section 4.5 les modèles mutants ne permettent pas toujours de générer les tests souhaités. La première solution consiste à trier manuellement les mutants puis à démarrer le programme d'extraction sur les mutants restants en choisissant les bons paramètres pour chacun d'entre eux. La seconde solution consiste à démarrer le programme d'extraction sans tri préalable et avec des paramètres fixes. Dans le premier cas, le VTG produisait plus de 7300 tests en 1h05. Dans le second cas, plus de 10000 tests étaient produits en 2h30. Chacun de ces ensembles de tests permettait de vérifier tous les mécanismes du VCI modélisé.

Suite à ces premiers résultats, le modèle a été amélioré pour produire le modèle présenté dans ce document. Le nouveau modèle comporte 66 événements représentant 62 instructions Java Card [7]. La nouvelle version du modèle étant beaucoup plus complexe, la méthode et les outils n'étaient plus efficaces. Selon une estimation, cela aurait nécessité environ 2 années de calcul pour résoudre l'ensemble des objectifs de test. De plus, le temps d'exécution d'un test sur une carte à puce étant relativement long, cette stratégie n'était pas envisageable avec des ensembles de tests trop grands. Une nouvelle version de la méthode, présentée dans le chapitre 4, a ainsi été proposée. Les outils, détaillés dans le chapitre 6, ont été entièrement refaits. La première étape du VTG, consistant à

7. Certaines instructions sont représentées par plusieurs événements.

aplanir le modèle, n'existait pas dans la version précédente, mais permet désormais de traiter des modèles plus complexes. Ce processus nécessite quelques secondes. À partir de ce modèle aplani, 921 événements mutants sont produits. La longueur maximum des traces obtenues est de 5. Avec la version précédente, la mutation nécessitait plus de 24h de calcul. Cette étape nécessite maintenant environ une minute. Parmi les mutants, seuls 223 peuvent produire des tests. L'algorithme décrit dans la section 4.5 permet de trier automatiquement ces mutants en moins d'une minute. Finalement, la recherche de solution pour les mutants restants est largement améliorée et nécessite 45min de calcul. Un ensemble composé de 223 tests, un test par événement mutant, est produit. Cet ensemble, bien que plus petit que ceux obtenus précédemment, permet de vérifier de façon concise et précise l'ensemble des mécanismes étudiés.

Bien que permettant de générer plus de tests, le premier modèle ne permet de vérifier que $6,5\%$ des instructions du langage Java Card. Le second modèle permet d'étendre cette vérification à 33% des instructions. Pour les instructions considérées, 91% des contraintes ont été testées. La non-modélisation du type *int* est à l'origine des 9% restants, c.-à-d. que si le modèle prenait en compte les *int*, tous les mécanismes seraient vérifiés.

5.3.12.1 Analyse des résultats

Les ensembles de tests que nous avons produits ont été utilisés afin de vérifier l'implémentation du VCI d'Oracle [8]. Ce dernier a rejeté tous les tests du premier ensemble de tests, ne relevant ainsi aucune défaillance. Pour le second ensemble de tests, certains tests ont été acceptés. Une analyse approfondie de ces résultats a été nécessaire. En effet, la faute peut être attribuée : à l'implémentation, à la spécification ou aux tests. Dans ce dernier cas, où les tests sont caractérisés de faux positifs, la faute peut provenir d'une étape ou d'une combinaison d'étapes de la génération des tests. Le modèle et tous les composants du VTG, tant théoriques que leurs implémentations, sont ainsi remis en question. Après analyse, il semble que le VTG ne soit pas à remettre en cause. La faute se situe donc : dans l'implémentation du VCI , dans sa spécification ou dans notre modèle.

Le modèle a été construit en prenant en compte seulement la spécification des instructions. Les tests produits par le VTG ont conduit à la nécessité d'analyser trois instructions suivantes : `sinc`, `sinc_w` et `sload_0`. La figure 5.15 est extraite de la spécification Java Card. Il est défini que l'instruction `sinc` incrémente l'élément à l'index `index` et que cet élément doit être de type `short`. L'un des tests produits par le VTG consiste à utiliser cette instruction avec un élément de type non compatible avec le type `short`. Le test donné dans la figure 5.16 consiste à manipuler l'élément à l'index 0 des variables locales, qui est de type `byte`. Cette figure représente uniquement les parties importantes à la compréhension du test. Les informations importantes se trouvent notamment dans le `method_component` et le `descriptor_component` respectivement représenté par les lignes 1 à 6 et 7 à 13. Dans ce fichier CAP, la méthode référencée par l'offset @0x60 contient la charge active du test. Les lignes 2 à 5 représentent le contenu de la méthode, où le tableau d'octet est donné aux lignes 3 et 4. La charge active du test, l'instruction `sinc` qui charge la première variable locale, est à la ligne 3. La ligne 9 définit que cette méthode est statique, c.-à-d. que la première variable locale est le premier argument de la méthode. Les types des arguments de la méthode sont définis à la ligne 11 dans l'ordre suivant : `byte`, `short`, `reference`. La reference `L00.14` représente un référence

8. La version 2.2.1 du VCI d'Oracle a été utilisée.

7.5.89 sinc

Increment local short variable by constant

Format

sinc

index

const

Forms

sinc = 89 (0x59)

Stack

No change

Description

The index is an unsigned byte that must be a valid index into the local variable of the current frame (Section 3.5, "Frames"). The const is an immediate signed byte. The local variable at index must contain a short. The value const is first sign-extended to a short, then the local variable at index is incremented by that amount.

FIGURE 5.15 – Spécification de l'instruction `sinc`.

```
[...]
1  method_component{ [...]
2   method_info[8] // @0x60= { [...]
3   /*0062*/ L0:   sinc            0x00, 0x00
4   /*0065*/       return
5   }
6  }
7  descriptor_component{ [...]
8   method[5] = { [...]
9    access_flag    : (a) ACC_STATIC ACC_PRIVATE
10   method_offset  : @0x60
11   type_offset    : /* 63*/ ( B; S; L00.14; ) V
12  }
13 }
[...]
```

FIGURE 5.16 – Test `cbc_test_52.cap`.

vers un objet dont le type est représenté par l'élément 14 du `constant_coop_component`. Nous pouvons ainsi constater que l'instruction à la ligne 3 charge un élément de type `byte`. Notre travail n'étant basé que sur la spécification des instructions, le raisonnement suivant en découle. L'instruction `sinc` s'attend à lire un `short`, alors que la variable locale lue est de type `byte`. Ces deux types ne sont pas compatibles, comme le montre la figure 5.7, le type `byte` n'étant pas un sous-type du type `short`. La précondition de l'instruction `sinc` n'est donc pas respectée. Cette instruction, et par extension ce test, devrait être rejeté par un VCI . Par la suite, des expérimentations effectuées manuellement ont permis de trouver 4 autres instructions produisant le même comportement : `sload`, `sload_1`, `sload_2` et `sload_3`. En analysant toutes les instructions, il apparaît qu'aucune ne permet de manipuler une variable locale de type `byte`. Selon les résultats obtenus, il semblerait que les variables locales, de type `byte` sont assimilées au type `short`, c.-à-d. qu'un transtypage implicite ((«*implicit cast*») soit effectué. De plus, le processus d'appel de méthode à l'aide des instructions `invokeinterface`, `invokespecial`, `invokestatic` et `invokevirtual` ne semble pas permettre d'invoquer une méthode possédant des arguments de type `byte`. En effet, pour invoquer une méthode, ses arguments doivent être poussés sur la pile, or il est impossible de mettre des éléments de type `byte` dans la pile.

Il a été porté à notre attention que notre interprétation de la spécification Java Card est trop restrictive et que des informations complémentaires relatives aux variables locales de type `byte` peuvent y être trouvées. Le tableau 3-2 de la spécification distingue les *Java (Storage) types* des *Computational Types*. On peut y constater le changement de type qui nous intéresse. Cependant, le terme *Computational Types* n'est jamais défini. De plus, sans une définition précise, on pourrait, par exemple, en conclure qu'il faut représenter impérativement toutes les expressions de type `byte` en `short`, ce qui n'est notamment pas le cas pour les tableaux et les champs des objets. Notre interprétation ne tient pas non plus compte des transtypages implicites décrits dans la section relative à la structure des fichiers CAP. En effet, à la page 6-32 est indiqué que les variables et les paramètres de méthode de type `byte` sont stockés dans un mot (typiquement 16 bits), et donc, cela entraine que les variables locales et les paramètres de méthode de type `byte` sont représentés par des `short`. En conséquence, il n'y a pas d'erreur de typage pour ces instructions.

Pour toutes les instructions ne manipulant pas des variables locales, par exemple `s2b`, `bspush`, `baload` et `getfield_b`, la spécification Java Card indique explicitement une conversion du type `byte` au type `short`. Cela n'est pas le cas pour les instructions que nous avons identifiées (`sinc`, `sinc_w`, `sload`, `sload_0`, `sload_1`, `sload_2`, `sload_3`). La spécification porte à confusion et peut facilement être mal interprétée. La spécification de la machine virtuelle Java [62] est plus claire et précise ; elle indique que les variables locales contiennent des `int` (et donc, que les variables locales de types `byte` et `short` sont représentées sur 32 bits). Notre interprétation stricte de la spécification des instructions a conduit au modèle présenté dans ce document. La seconde interprétation pourrait être facilement incluse dans le modèle en spécifiant le transtypage implicite. Cette interprétation correcte de la spécification entrainerait la suppression des faux positifs. Notons que cela démontre indirectement l'efficacité de notre approche. En effet, si les variables locales n'étaient pas stockées sur 16 bits, mais plutôt sur 8 bits, alors nos tests auraient permis d'identifier des incohérences, car la lecture d'un mot de 16 bits pour une variable stockée sur 8 bits est effectivement une lecture erronée engendrant des erreurs de calcul, à cause des 8 bits lus en trop. Nous avons choisi de conserver l'interprétation initiale de la spécification pour mettre en évidence le fait que la spécification porte à

confusion et peut facilement être mal interprétée.

5.4 Conclusions

Les mécanismes composant le VCI ont été successivement introduits. Ce cas d'étude a été découpé en deux sous-problèmes : la vérification de structure et la vérification de type. Deux modèles Event-B de nature différente ont été produits. Pour la vérification de structure, le modèle est uniquement composé de contextes. Pour la vérification de type, le modèle est constitué de contextes et de machines, mais seules les machines sont utilisées pour représenter les contraintes à tester de la spécification.

En utilisant ces deux modèles, nous avons généré des tests d'intrusions. Pour la vérification de structure, seuls des tests abstraits ont été générés automatiquement. Leur concrétisation a été réalisée manuellement. Ces tests n'ont pas permis de découvrir de vulnérabilité. Pour la vérification de type, la concrétisation et l'exécution des tests ont été automatisées. Ces derniers ont permis de mettre en évidence une ambiguité de la spécification sur l'interprétation du type `short` dans les variables locales.

Nous avons effectué la modélisation de toutes les composantes d'un fichier CAP en déclarant toutes les structures nécessaires. Cela représente un travail de modélisation très important (130 contextes, 250 constantes et 400 axiomes) qui nous a permis de proposer des patrons de modélisation compatibles avec le VCI et que ProB peut explorer efficacement. Ces patrons ont été dérivés après comparaisons de plusieurs variantes de modélisation qu'il aurait été trop long d'inclure dans la thèse. Néanmoins, cela indique qu'il faut bien maîtriser le langage Event-B et la complexité algorithmique de l'exploration des modèles des formules du langage B. On peut optimiser l'évaluation des formules en utilisant certaines formes plutôt que d'autres, un peu comme on peut optimiser des énoncés SELECT du langage SQL en utilisant certaines formes particulières dans certains cas.

Nous avons démontré la faisabilité de l'approche en modélisant les contraintes de l'héritage entre les classes. C'est une expérimentation minimaliste, mais qui permet de montrer que ProB peut trouver des modèles pour une spécification complexe de très grande taille. L'utilisation de fichiers CAP pré-remplis est essentielle pour permettre de trouver des modèles, sinon ProB n'arrive pas à générer un modèle pour une spécification de si grande taille. La recherche d'un test n'est donc pas entièrement automatique, et elle nécessite l'intervention du spécifieur, même après avoir construit le modèle. Par contre, nous croyons qu'il y a un avantage important à utiliser une approche MBT pour la vérification de structure, car elle permet d'explorer systématiquement l'espace de test. Une génération manuelle de tests par un humain, avec un si grand nombre de contraintes à vérifier, devient vite trop complexe, et il est illusoire de penser qu'un humain peut générer un ensemble de tests avec une excellente couverture. Il reste encore beaucoup de travail à faire avant de conclure que l'approche est viable économiquement. Il reste à déterminer comment gérer la génération modulaire de tests, afin de combattre l'explosion combinatoire. Par exemple, les questions suivantes demeurent ouvertes. Combien de contraintes peuvent être prises en compte simultanément pour la génération de tests de structure ? Comment choisir les fichiers CAP pré-remplis ? Comment échantillonner les ensembles de valeurs permises pour les constantes dans ProB ?

Pour la vérification de types, les tests sont générés par un parcours en largeur du graphe de transition, afin de réduire le temps de génération des tests et minimiser la

longueur des tests. L'analyse de faisabilité deux-à-deux a permis de grandement élaguer l'espace de recherche. Il serait opportun de vérifier si des tests plus longs seraient nécessaires pour trouver certaines failles de sécurité. Comme le temps de calcul augmente exponentiellement avec la profondeur du graphe de transition, il faudrait déterminer des stratégies de recherche en profondeur avec certaines techniques d'élagage, comme par exemple un langage qui permettrait de guider la recherche en exprimant des objectifs intermédiaires de test, ou bien en procédant avec un parcours non-déterministe du graphe, ou bien en définissant des sous-ensembles d'instructions qui permettraient de réduire le nombre d'événements à considérer dans la recherche d'un test de longueur n.

La majorité des mécanismes du vérifieur de type ont été étudiés : la pile, les variables locales, l'arbre d'héritage et le tas (partiellement). La prise en compte des boucles nécessitera de modifier le modèle pour représenter un test par une variable d'état, plutôt que par la trace d'exécution de la machine. En effet, si on veut effectuer un branchement en arrière, on doit savoir quel était l'état de la pile et des variables à cet endroit dans le programme, afin de brancher vers un endroit compatible avec l'état de la pile et des variables après l'instruction de branchement. Il faudra donc stocker la pile et les variables locales pour chaque position du programme, ce qui complexifie le modèle, mais qui est tout-à-fait envisageable. De plus, il faut stocker le programme byte code qui représente le test dans une variable au fur et à mesure que les instructions sont exécutées. Dans le cas d'un branchement arrière vers un endroit avec une pile et des variables locales compatibles avec les valeurs courantes au moment du branchement, il faut effectuer une vérification en cascade, un peu comme l'algorithme de vérification du VCI donné à la figure 1, afin de s'assurer que le suprémum dans la hiérarchie de types entre les deux piles et variables locales est compatible avec les instructions qui suivent le point de branchement. Il faudra donc des événements de vérification du programme construit avec des branchements. Des problèmes similaires surgissent avec des branchements en avant ou des branchements conditionnels. Une première étude du problème, non documentée dans cette thèse, nous permet de poser comme hypothèse que cette piste est viable. Un travail de maîtrise est en cours pour valider cette hypothèse. Naturellement, le modèle devient beaucoup plus complexe, tant du point de vue de son espace d'états que du nombre d'événements à gérer. De plus, il est intéressant de voir que le modèle proposé dans cette thèse est adéquat pour tester tous les types d'instructions, sauf les branchements, et que l'ajout des branchements nous force à changer radicalement la structure du modèle, comme quoi il n'est pas facile de construire un modèle de manière incrémentale, en prenant en compte les aspects d'un problème un par un. Ceci étant dit, nous croyons que l'approche incrémentale de construction du modèle fut bénéfique, car elle nous a permis de décomposer la complexité du problème. Il n'y a pas de processus miraculeux dans la recherche d'une solution à un problème.

Chapitre 6 :

Outils développés

Afin de vérifier le bon fonctionnement de la méthode de génération de tests de vulnérabilité sur le cas d'étude du VCI , un ensemble d'outils ont été réalisés. Plusieurs itérations pour développer la méthode et les outils ont été nécessaires. De nombreuses versions majeures ont ainsi été proposées pour certains d'entre eux. Les premières étaient basées sur l'utilisation de l'API de Rodin. Cependant, cette API étant complexe et peu optimisée pour nos besoins, nous avons préféré utiliser l'API de ProB. Ces outils se décomposent en deux catégories : les outils pour la méthode et les outils pour les cas d'études.

6.1 Outils pour la méthode

Les outils pour la méthode sont génériques et peuvent être utilisés pour tout modèle ou SUT. Chacun de ces outils est l'implémentation d'un composant du VTG.

6.1.1 Aplanissement de modèles Event-B

L'aplanissement du modèle, décrit dans la section 4.2, est réalisé en Java. Il se base sur l'API de ProB pour la représentation objet et la manipulation de modèles Event-B. Ce modèle est ensuite transmis au module de mutation de modèle. Une seule version composée d'environ 500 lignes de code a été nécessaire et son développement a nécessité quelques heures.

6.1.2 Mutation de modèles Event-B

La mutation de modèles Event-B est décomposée en trois étapes. Dans un premier temps, les formules à muter sont extraites. Ces formules sont ensuite mutées selon les règles décrites dans la section 4.4. Finalement les résultats de la mutation précédente permettent de constituer les contextes mutants, événements mutants et machines mutantes comme décrits dans la section 4.4.2. Les mutations se font à travers la représentation objet des modèles dans ProB.

La mutation de formules se base sur un analyseur grammatical et syntaxique de formules Event-B. L'analyseur de Rodin a été réutilisé pour cette tâche. Il se base sur l'API TOM [63] développé par le LORIA. L'analyse permet de déterminer quelle règle de mutation doit être appliquée. Les règles de réécriture sont implémentées en Java. Un programme Java de 2700 lignes de code permet de réaliser la mutation de machines, de contextes ainsi que de formules sans prendre en compte les paramètres décrits dans la section 4.4.

Cette partie du VTG est la plus complexe et a subi de nombreuses modifications. Trois versions majeures ont été produites. La première se base sur une ancienne version de Rodin. Dû à de nombreux problèmes de stabilité de Rodin, le VTG n'était pas stable. Lorsqu'une version stable de la plateforme fut disponible, le VTG fut porté. De nombreuses fonctionnalités ayant disparu, une adaptation de nos logiciels a dû être réalisée. De plus, nous avons utilisé l'analyseur syntaxique et grammatical TOM. Cette version du programme était composée de plus de 10000 lignes de code et prenait en compte les paramètres de mutation. Pour la dernière version du VTG, Rodin a été abandonné au profit de ProB. Rodin étant destiné à éditer et prouver des modèles Event-B, la

représentation interne de ses modèles ne correspondait pas à nos objectifs. ProB ayant mis à disposition, durant la dernière année de doctorat, une API permettant de manipuler les modèles correspondant à nos besoins, nous avons redéveloppé le VTG. La production de la mutation de modèles a nécessité environ une année au total.

6.1.3 Extraction de tests

Cet outil est réalisé en partenariat avec Michael Leuschel. Il fait maintenant partie de l'ensemble des outils de test à base de modèle proposé par ProB. Seule l'extraction statique des tests est implémentée. Il est développé en Prolog. Deux mois nous ont été nécessaires. L'ensemble de tests abstrait est enregistré dans un fichier XML.

6.2 Outils pour le cas d'étude

Les outils pour le cas d'étude ne sont utilisables que pour le VCI . Cependant, ils sont des exemples d'outils qu'il serait nécessaire de développer pour d'autres SUT.

6.2.1 CAP2Rodin

Comme nous l'avons vu dans le chapitre précédent, nous utilisons un modèle prérempli. Le nombre de champs d'un fichier CAP étant très grand, nous ne pouvons pas les traduire manuellement dans un modèle Rodin. L'outil CAP2Rodin prend en entrée un fichier CAP et crée les modèles nécessaires à sa représentation en Rodin. Cet outil est développé en Java.

Il a été réalisé par 3 groupes d'étudiants de licence et de master de l'Université de Limoges. La version actuelle est complète et aurait nécessité environ 8 mois de travail pour une personne. Elle est composée de 2000 lignes de code.

6.2.2 XML2CAP

La concrétisation statique des tests est réalisée à l'aide de l'outil XML2CAP. Le VTG produit des tests abstraits sous forme de XML qui sont ensuite concrétisés en fichier CAP. Cet outil n'est pour le moment utilisable que pour la vérification de type. Il est développé en Java et se base sur l'API CAPMAP [64]. Ce développement a été réalisé conjointement avec des étudiants de l'Université de Limoges. Le modèle influant grandement sur le type de tests extraits, cet outil a été fréquemment mis à jour. La dernière version, composée de 600 lignes de programmation, a été produite en 2 jours.

6.2.3 TestOnPC et TestOnCard

Ces deux outils permettent d'exécuter les tests en envoyant les fichiers CAP au SUT. De plus, ils permettent de déterminer si le résultat d'un test correspond au résultat attendu. Ces deux implémentations permettent respectivement d'exécuter les tests sur l'implémentation du VCI de Oracle et sur des implémentations embarquées. Ces outils composés respectivement de 150 et 200 lignes de Java ont été développés en quelques heures, mais possèdent encore de nombreux problèmes hérités d'instabilités dans les drivers et API utilisés.

Chapitre 7 :

Publications

Au début de ce doctorat, la participation au workshop Deploy [65] a permis de présenter les grandes idées de cette thèse à la communauté Event-B. Ce workshop a été l'occasion de vérifier la maturité des outils gravitant autour de ce langage et ainsi de nous assurer que l'implémentation de la méthode proposée serait possible.

Après une année de travail, une première version de la méthode a été proposée. Elle a été appliquée à une sous-partie de la spécification du VCI . Le modèle ainsi que l'ensemble des outils permettant de valider l'approche ont aussi été proposés. Ces premiers résultats expérimentaux ont fait l'objet d'une première publication à la conférence internationale iFM [61]. Ils ont aussi été présentés lors de la conférence nationale de vulgarisation scientifique Colloque TI au Canada [66].

Suite à ces premiers résultats expérimentaux, le modèle de la vérification de type a été étendu. Cependant, la méthode ainsi que les outils n'étaient plus suffisamment efficaces pour fonctionner avec un modèle aussi gros. De plus, Mathieu Lassale a commencé son travail de recherche à la maîtrise sur le problème de la vérification de structure. La méthode a ainsi été mise à jour pour devenir plus générique et plus efficace afin de fonctionner avec les nouvelles expérimentations. La participation au workshop Rodin [67] a permis de commencer un travail sur l'adaptation des outils qui a conduit à un stage en Allemagne.

Ces améliorations de la méthode ainsi que sur le modèle de la vérification de type ont été présentées lors de la conférence SEFM [60]. Cet article a reçu l'une des récompenses de meilleur article de la conférence.

Le travail réalisé sur la vérification de structure est présenté en détail dans le mémoire de maîtrise de Mathieu Lassale [59]. Il fera prochainement l'objet d'une publication.

Chapitre 8 :

Conclusion

Les systèmes informatiques ont permis d'automatiser de nombreuses tâches et ainsi de travailler plus efficacement. Cependant, lorsqu'une défaillance survient, l'énergie nécessaire pour rendre le système de nouveau opérationnel est parfois très importante. Par opposition à la vérification de leurs propriétés fonctionnelles, qui est un domaine largement étudié, nous souhaitons vérifier leurs propriétés de sécurité. Peu de méthodes s'intéressent à la génération de tels tests et aucune méthode ne s'est avérée suffisamment efficace. L'objectif de cette thèse est de proposer une méthode de génération de tests, en boite noire, de vulnérabilité permettant d'étudier le comportement du système face à des tentatives d'intrusion. Pour valider cette méthode, elle est appliquée sur un cas d'étude, le VCI .

La méthode proposée, nommée VTG, se base sur une représentation du SUT en Event-B. L'utilisation de modèles formels permet de simplifier l'automatisation et la vérification d'une méthode. Cette méthode repose sur l'hypothèse que le coût de construction d'un modèle formel du système à tester est plus faible que le coût de construction des tests par un humain, tout en fournissant des tests qui offrent une meilleure couverture de l'espace d'état, ou bien que le coût supplémentaire de construction d'un modèle formel est compensé par la qualité des tests produits, et la facilité d'adapter le modèle et de re-générer les tests pour suivre l'évolution du SUT durant son cycle de vie. Cette hypothèse est bien sûr difficile à valider dans le cadre d'une thèse de doctorat. Il faudra que notre méthode soit expérimentée par plusieurs personnes sur une variété de systèmes pour vérifier cette hypothèse. Les propriétés formelles caractéristiques des parties les plus importantes du processus de mutation ont été vérifiées. Les algorithmes de mutation proposés ont permis de transformer le modèle des comportements acceptables afin de générer des modèles d'intrusion. De ces modèles mutants sont extraits des tests à l'aide d'algorithmes de MBT proposés. Pour cela, de nouvelles stratégies de mutation ainsi que des algorithmes de résolution de contraintes ont été proposés. Afin de rendre cette méthode utilisable, une implémentation est proposée. La réalisation de ce travail a nécessité un effort plus important que prévu. Cet effort comprend à la fois la conception de la méthode de test, l'apprentissage des plateformes Java Card, Rodin et ProB, ainsi qu'un effort de programmation important pour développer les outils supportant la méthode. L'application de la méthode à un nouveau problème devrait demander un effort beaucoup moins important. Les outils portant sur les tests abstraits sont entièrement réutilisables. Les outils de concrétisation des tests sont à développer pour chaque type de système ; cela ne représente toutefois pas une tâche très complexe à réaliser.

Le VCI étant un système vaste et complexe, seule une partie de la spécification a été étudiée afin de valider la méthode. De plus, il a été décomposé en deux sous-cas, la vérification de structure et la vérification de type. À l'aide de la méthode, un ensemble de tests abstraits est extrait puis concrétisé sous format de fichier CAP. Ces tests sont ensuite envoyés à des implémentations embarquées de VCI . Pour la vérification de structure, tous les champs d'un fichier CAP sont modélisés, mais seules les contraintes liées aux relations d'héritage sont étudiées. Pour la vérification de type, un sous-ensemble des mécanismes et des instructions a été étudié. Malgré ces simplifications, ce travail a permis de découvrir une défaillance, non exploitable, dans le VCI de Oracle. Ces tests génériques peuvent être utilisés pour vérifier toute implémentation de VCI . Cette caractéristique générique est possible grâce à une représentation du comportement du système et non de son implémentation.

Limites et perspectives

Les cas d'étude ont guidé ce travail et ont permis de repousser progressivement les limites de la méthode. Cependant, la génération de tests de vulnérabilité est un travail vaste dont certaines parties n'ont pour le moment pas été étudiées. La première étape, la mutation de spécification, a été appliquée à un sous-ensemble de la grammaire Event-B. L'ajout de processus de mutation pour les autres parties de la grammaire rendrait possible la génération d'autres types de mutants.

La façon de modéliser un système dépend de son utilisation. L'aplanissement est une première étape dans l'adaptation automatique du modèle originel aux contraintes imposées par la méthode proposée. Le VTG étant basé sur le MBT, l'ensemble des bonnes pratiques de modélisation de ce dernier doivent être appliquées. Cependant, d'autres règles de modélisation semblent nécessaires. La complexité du VCI nous a permis d'explorer plusieurs aspects du problème de la génération de tests de vulnérabilité, comme par exemple, de réaliser qu'il n'est pas nécessaire et souhaitable de nier chaque garde d'un événement, qu'il faut ajouter des variables de contrôle pour contrôler la génération des tests, que le type des variables doit parfois être élargi pour permettre de tester des vulnérabilités. On ne pourra déterminer la généralité de ces principes qu'en appliquant la méthode à plusieurs autres types de système. De plus, le modèle du système contient plus qu'une spécification du système ; il contient aussi des éléments qui permettent de contrôler la génération des tests, afin d'appliquer certains critères de test particuliers. Les critères et les objectifs de test sont donc répartis entre le modèle et l'outil ProB. Par exemple, le modèle indique quand terminer un test, à quel moment inclure une instruction, en utilisant des variables de contrôle et des gardes appropriées. En quelque sorte, on spécifie un critère de test là où cela est le plus simple : soit dans le modèle ou bien soit dans l'outil ProB qui est chargé de parcourir de modèle. Les critères généraux et ré-utilisables sont implémentés dans ProB (ex : recherche en largeur, chaque mutant inclu une seule fois, etc) ; les critères spécifiques au SUT sont pris en compte dans le modèle du SUT.

Concernant la mutation de modèle, il serait intéressant de proposer un algorithme unifié permettant de muter simultanément les contextes et les machines d'un modèle. De plus, seul un sous-ensemble de la grammaire Event-B (les axiomes, les gardes et les événements) est pour le moment muté. Pour rendre la solution plus générique, il est nécessaire de pouvoir muter l'ensemble de celle-ci. Entre autres, on pourrait aussi étudier des mutations aux actions d'un événement. Il faudrait alors comparer les taux de couverture obtenu par mutations de garde et par mutation d'actions. A priori, il est difficile d'estimer si l'une est supérieure à l'autre. La mutation des actions semble plus vaste, mais elle peut aussi s'exprimer par mutation de prédicat, en transformant les actions sous leur forme normale en Event-B, qui est constituée d'une formule entre les états avant et après. Nos travaux pourraient donc aussi s'appliquer à la mutation d'actions en mutant le prédicat des actions réécrites sous leur forme normale. Il serait intéressant de vérifier si ProB peut aussi bien gérer les mutations d'actions en passant par leur forme normale.

La plateforme Rodin est bien adaptée pour la construction de modèles Event-B "usuels". Elle est moins adaptée pour la mutation de modèles nécessaires à la génération de tests de vulnérabilité. Le passage à ProB nous a grandement facilité la tâche, en offrant une API mieux adaptée et une structure plus légère pour la représentation des mutants. La plateforme Rodin contient beaucoup d'information, comme la gestion des preuves et la représentation des modèles dans une base de données, qui ne sont pas pertinentes

dans un contexte de mutation. ProB nous permettra aussi de supporter plus facilement le B classique pour la mutation. Le choix du langage Event-B a permis de construire le modèle du VCI de manière incrémentale. Cependant, il apparait que pour la modélisation des branchements, l'absence de conditionnelle IF-THEN-ELSE dans le langage entraînera une explosion du nombre d'événements pour la validation des branchements dans le VCI . Le langage B serait plus approprié pour cet aspect. Notre approche pourrait s'appliquer très facilement au langage B. Le langage B serait utile pour spécifier des éléments de contrôle dans la génération des tests. L'outil ProB permet de combiner des instructions d'implémentation du langage B comme le WHILE-DO et le IF-THEN-ELSE avec des substituions abstraites, ce qui en fait un puissant outil d'implémentation de critères de contrôle de la génération de tests.

En ce qui concerne la nature du SUT, le travail réalisé sur le VCI correspond à un système déterministe, ce qui n'est pas surprenant, car les systèmes non déterministes posent des défis supplémentaires. En effet, on ne peut construire à l'avance les tests avant leur exécution sur un SUT non-déterministe. Par définition, il y a plusieurs sorties possibles pour une entrée donnée d'un système non-déterministe. L'entrée suivante peut dépendre des réponses produites pour les entrées précédentes. Il faut donc calculer le test en temps réel durant son exécution, en fonction des sorties produites par le SUT. Dans le début de travail réalisé sur le protocole EMV, il a été constaté que la méthode possédait quelques difficultés à tester des systèmes non-déterministes. Une adaptation de l'algorithme de MBT aux systèmes non déterministes est nécessaire pour prendre en compte de tels modèles. Bien que notre modèle ne couvre qu'une partie de la spécification du VCI , cela a permis de montrer le potentiel de la méthode. Une extension de ce cas d'étude aurait un impact plus complet sur la capacité de caractérisation de l'ensemble de tests produits. Les tests nécessaires à la vérification du VCI nécessitent une concrétisation statique. Cependant, de nombreux systèmes nécessitent des mécanismes de concrétisation dynamique. Dans le cadre du cas d'étude sur EMV, la concrétisation dynamique est nécessaire. Bien que non publiée, une preuve de concept outillée, basée sur une amélioration de ProB, a été réalisée.

Chapitre 9 :

Modèle des instructions Java Card

Le modèle B présenté dans cette annexe permet de classer les instructions Java Card dans différents ensembles représentant chacun une caractéristique particulière.

MACHINE Instructions

SETS
INSTRUCTIONS = {aaload, aastore, aconst_null, aload, aload_0, aload_1, aload_2, aload_3, anewarray, areturn, arraylength, astore, astore_0, astore_1, astore_2, astore_3, athrow, baload, bastore, bipush, bspush, checkcast, dup, dup_x, dup2, getfield_a, getfield_b, getfield_s, getfield_i, getfield_a_this, getfield_b_this, getfield_s_this, getfield_i_this, getfield_a_w, getfield_b_w, getfield_s_w, getfield_i_w, getstatic_a, getstatic_b, getstatic_s, getstatic_i, goto, goto_w, i2b, i2s, iadd, iaload, iand, iastore, icmp, iconst_m1, iconst_0, iconst_1, iconst_2, iconst_3, iconst_4, iconst_5, idiv, if_acmpeq, if_acmpne, if_acmpeq_w, if_acmpne_w, if_scmpeq, if_scmpne, if_scmplt, if_scmpge, if_scmpgt, if_scmple, if_scmpeq_w, if_scmpne_w, if_scmplt_w, if_scmpge_w, if_scmpgt_w, if_scmple_w, ifeq, ifne, iflt, ifge, ifgt, ifle, ifeq_w, ifne_w, iflt_w, ifge_w, ifgt_w, ifle_w, ifnonnull, ifnonnull_w, ifnull, ifnull_w, iinc, iinc_w, iipush, iload, iload_0, iload_1, iload_2, iload_3, ilookupswitch, imul, ineg, instanceof, invokeinterface, invokespecial, invokestatic, invokevirtual, ior, irem, ireturn, ishl, ishr, istore, istore_0, istore_1, istore_2, istore_3, isub, itableswitch, iushr, ixor, jsr, new, newarray, nop, pop, pop2, putfield_a, putfield_b, putfield_s, putfield_i, putfield_a_this, putfield_b_this, putfield_s_this, putfield_i_this, putfield_a_w, putfield_b_w, putfield_s_w, putfield_i_w, putstatic_a, putstatic_b, putstatic_s, putstatic_i, ret, return, s2b, s2i, sadd, saload, sand, sastore, sconst_m1, sconst_0, sconst_1, sconst_2, sconst_3, sconst_4, sconst_5, sdiv, sinc, sinc_w, sipush, sload, sload_0, sload_1, sload_2, sload_3, slookupswitch, smul, sneg, sor, srem, sreturn, sshl, sshr, sspush, sstore, sstore_0, sstore_1, sstore_2, sstore_3, ssub, stableswitch, sushr, swap_x, sxor}

CONSTANTS
instManipStack, instManipLocVar, instManipHeap, instReturn, instBranchSimple, instBranchDouble, instBranchMult, instBranch, instNoBranch, instManipInt, instManipArray, instVarElem, instManipCPool, ASDrThInst
PROPERTIES

/* data structures */
instManipStack = {aaload, aastore, aconst_null, aload, aload_0, aload_1, aload_2, aload_3, anewarray, areturn, arraylength, astore, astore_0, astore_1, astore_2, astore_3, athrow, baload, bastore, bipush, bspush, checkcast, dup, dup_x, dup2, getfield_a, getfield_b, getfield_s, getfield_i, getfield_a_this, getfield_b_this, getfield_s_this, getfield_i_this, getfield_a_w, getfield_b_w, getfield_s_w, getfield_i_w, getstatic_a, getstatic_b, getstatic_s, getstatic_i, i2b, i2s, iadd, iaload, iand, iastore, icmp, iconst_m1, iconst_0, iconst_1, iconst_2, iconst_3, iconst_4, iconst_5, idiv, if_acmpeq, if_acmpne, if_acmpeq_w, if_acmpne_w, if_scmpeq, if_scmpne, if_scmplt, if_scmpge, if_scmpgt, if_scmple, if_scmpeq_w, if_scmpne_w, if_scmplt_w, if_scmpge_w, if_scmpgt_w, if_scmple_w, ifeq, ifne, iflt, ifge, ifgt, ifle, ifeq_w, ifne_w, iflt_w, ifge_w, ifgt_w, ifle_w, ifnonnull, ifnonnull_w, ifnull, ifnull_w, iipush, iload, iload_0, iload_1, iload_2, iload_3, ilookupswitch, imul, ineg, instanceof, invokeinterface, invokespecial, invokestatic, invokevirtual, ior, irem, ireturn, ishl, ishr, istore, istore_0, istore_1, istore_2, istore_3, isub, itableswitch, iushr, ixor, jsr, new, newarray, pop,

pop2, putfield_a, putfield_b, putfield_s, putfield_i, putfield_a_this, putfield_b_this, putfield_s_this, putfield_i_this, putfield_a_w, putfield_b_w, putfield_s_w, putfield_i_w, putstatic_a, putstatic_b, putstatic_s, putstatic_i, s2b, s2i, sadd, saload, sand, sastore, sconst_m1, sconst_0, sconst_1, sconst_2, sconst_3, sconst_4, sconst_5, sdiv, sipush, sload, sload_0, sload_1, sload_2, sload_3, slookupswitch, smul, sneg, sor, srem, sreturn, sshl, sshr, sspush, sstore, sstore_0, sstore_1, sstore_2, sstore_3, ssub, stableswitch, sushr, swap_x, sxor}

& instManipLocVar = {aload, aload_0, aload_1, aload_2, aload_3, astore, astore_0, astore_1, astore_2, astore_3, getfield_a_this, getfield_b_this, getfield_s_this, getfield_i_this, iinc, iinc_w, iload, iload_0, iload_1, iload_2, iload_3, istore, istore_0, istore_1, istore_2, istore_3, ret, sinc, sinc_w, sload, sload_0, sload_1, sload_2, sload_3, sstore, sstore_0, sstore_1, sstore_2, sstore_3}

/* Controle flaw */
& instReturn = {return, sreturn, areturn, athrow, ireturn}
& instBranchSimple = {jsr, goto, goto_w, ret}
& instBranchDouble = {checkcast, if_acmpeq, if_acmpne, if_acmpeq_w, if_acmpne_w, if_scmpeq, if_scmpne, if_scmplt, if_scmpge, if_scmpgt, if_scmple, if_scmpeq_w, if_scmpne_w, if_scmplt_w, if_scmpge_w, if_scmpgt_w, if_scmple_w, ifeq, ifne, iflt, ifge, ifgt, ifle, ifeq_w, ifne_w, iflt_w, ifge_w, ifgt_w, ifle_w, ifnonnull, ifnonnull_w, ifnull, ifnull_w, instanceof}
& instBranchMult = {slookupswitch, stableswitch, ilookupswitch, itableswitch}
& instBranch = instBranchSimple instBranchDouble instBranchMult
& instNoBranch = INSTRUCTIONS \ instBranch

/* Data types */
& instManipInt = {bipush, getfield_i, getfield_i_this, getfield_i_w, getstatic_i, i2b, i2s, iadd, iaload, iand, iastore, icmp, iconst_m1, iconst_0, iconst_1, iconst_2, iconst_3, iconst_4, iconst_5, idiv, iinc, iinc_w, iipush, iload, iload_0, iload_1, iload_2, iload_3, ilookupswitch, imul, ineg, ior, irem, ireturn, ishl, ishr, istore, istore_0, istore_1, istore_2, istore_3, isub, itableswitch, iushr, ixor, s2i, sipush, getfield_i, getfield_i_this, getfield_i_w, getstatic_i, putfield_i, putfield_i_this, putfield_i_w, putstatic_i, s2i, sipush}

& instManipArray = {aaload, aastore, anewarray, arraylength, baload, bastore, checkcast, iaload, iastore, instanceof, newarray, saload, sastore}

/* Others */
& instManipCPool = {anewarray, checkcast, getfield_a, getfield_b, getfield_s, getfield_i, getfield_a_this, getfield_b_this, getfield_s_this, getfield_i_this, getfield_a_w, getfield_b_w, getfield_s_w, getfield_i_w, getstatic_a, getstatic_b, getstatic_s, getstatic_i, instanceof, invokeinterface, invokespecial, invokestatic, invokevirtual, new, putfield_a, putfield_b, putfield_s, putfield_i, putfield_a_this, putfield_b_this, putfield_s_this, putfield_i_this, putfield_a_w, putfield_b_w, putfield_s_w, putfield_i_w, putstatic_a, putstatic_b, putstatic_s, putstatic_i}

& instVarElem = {dup_x, invokeinterface, invokespecial, invokestatic, invokevirtual}

/* Aymerick Savary doctoral theses modeled instructions */

& ASDrThInst = INSTRUCTIONS \ instBranch \ instManipInt \ instManipCPool \ instVarElem \ {athrow} ∪ {new}

END

Chapitre 10 :

Application Java Card

```
.header = {
magic              : decaffed
minor_version      : 1
major_version      : 2
flags              : 4
pkg_minor_version  : 0
pkg_major_version  : 1
pkg_AID_length     : a
pkg_AID            : 46.56.55.4c.4e.54.45.53.54.53
}

.DirectoryComponent = {
component_sizes = {
Header           : 20
Directory        : 31
Applet           : 17
Import           : 21
ConstantPool     : 82
Class            : 56
Method           : 144
StaticField      : 10
ReferenceLocation: 29
Export           : 0
Descriptor       : 285
}
static_field_size_info = {
image_size       : 0
array_init_count : 0
array_init_size  : 0
}
import_count : 2
applet_count : 1
custom_count : 0
}

.Applets = {
AID: 46.56.55.4c.4e.54.45.53.54.53.41.70.70
install_method_offset: @003e
}



.ImportComponent = {
count : 2
package_info = {
minor_version : 0
major_version : 1
```

```
AID_length    : 7
AID           : a0.00.00.00.62.00.01
}
package_info = {
minor_version : 2
major_version : 1
AID_length    : 7
AID           : a0.00.00.00.62.01.01
}
}

.ConstantPool = {
/* 0000,   0 */CONSTANT_InstanceFieldRef : field 0 of class 0x000a
/* 0004,   1 */CONSTANT_InstanceFieldRef : field 0 of class 0x0014
/* 0008,   2 */CONSTANT_InstanceFieldRef : field 1 of class 0x0014
/* 000c,   3 */CONSTANT_InstanceFieldRef : field 0 of class 0x001e
/* 0010,   4 */CONSTANT_InstanceFieldRef : field 1 of class 0x001e
/* 0014,   5 */CONSTANT_StaticMethodRef  : external: 0x80,0x0,0x0
/* 0018,   6 */CONSTANT_StaticMethodRef  : 0x0008
/* 001c,   7 */CONSTANT_StaticMethodRef  : external: 0x81,0x3,0x0
/* 0020,   8 */CONSTANT_VirtualMethodRef : method 1 of class external: 0x81,0x3
/* 0024,   9 */CONSTANT_ClassRef         : 0x0028
/* 0028,  10 */CONSTANT_StaticMethodRef  : 0x0033
/* 002c,  11 */CONSTANT_VirtualMethodRef : method 1 of class external: 0x81,0xa
/* 0030,  12 */CONSTANT_ClassRef         : 0x001e
/* 0034,  13 */CONSTANT_StaticMethodRef  : 0x0023
/* 0038,  14 */CONSTANT_ClassRef         : external: 0x81,0x1
/* 003c,  15 */CONSTANT_ClassRef         : 0x0000
/* 0040,  16 */CONSTANT_StaticMethodRef  : 0x0001
/* 0044,  17 */CONSTANT_ClassRef         : 0x000a
/* 0048,  18 */CONSTANT_ClassRef         : 0x0014
/* 004c,  19 */CONSTANT_StaticMethodRef  : 0x0013
}

.class { // @0000
classes [0]: { //@0000
flags = (0x0)
interface_count: 0
super_class_ref: external class 0x0 of package 0x80
declared_instance_size     :0
first_reference_index      :-1
reference_count            :0
public_method_table_base   :1
public_method_table_count  :0
public_methods = {
}
package_method_table_base  :0
```

```
package_method_table_count :0
package_methods = {
}
interfaces = {
}

}
classes [1]: { //@000a
flags = (0x0)
interface_count: 0
super_class_ref: external class 0x0 of package 0x80
declared_instance_size     :1
first_reference_index      :0
reference_count            :1
public_method_table_base   :1
public_method_table_count  :0
public_methods = {
}
package_method_table_base  :0
package_method_table_count :0
package_methods = {
}
interfaces = {
}

}
classes [2]: { //@0014
flags = (0x0)
interface_count: 0
super_class_ref: external class 0x0 of package 0x80
declared_instance_size     :2
first_reference_index      :0
reference_count            :1
public_method_table_base   :1
public_method_table_count  :0
public_methods = {
}
package_method_table_base  :0
package_method_table_count :0
package_methods = {
}
interfaces = {
}

}
classes [3]: { //@001e
flags = (0x0)
```

```
interface_count: 0
super_class_ref: internal class reference (offset = 000a)
declared_instance_size     :2
first_reference_index      :0
reference_count            :1
public_method_table_base   :1
public_method_table_count  :0
public_methods = {
}
package_method_table_base  :0
package_method_table_count :0
package_methods = {
}
interfaces = {
}

}
classes [4]: { //@0028
flags = (0x0)
interface_count: 0
super_class_ref: external class 0x3 of package 0x81
declared_instance_size     :0
first_reference_index      :-1
reference_count            :0
public_method_table_base   :7
public_method_table_count  :2
public_methods = {
public method @0049
public method @005d
}
package_method_table_base  :0
package_method_table_count :1
package_methods = {

public method @005a
}
interfaces = {
}
}
}

.method {
method_info[0] // @0001= {
// flags     : 0
// max_stack : 1
// nargs     : 1
// max_locals: 0
```

```
/*0003*/ L0:   aload_0
/*0004*/       invokespecial   0x0005
/*0007*/       return
}

method_info[1] // @0008= {
// flags     : 0
// max_stack : 2
// nargs     : 2
// max_locals: 0
/*000a*/ L0:   aload_0
/*000b*/       invokespecial   0x0005
/*000e*/       aload_0
/*000f*/       aload_1
/*0010*/       putfield_a      0x00
/*0012*/       return
}

method_info[2] // @0013= {
// flags     : 0
// max_stack : 2
// nargs     : 2
// max_locals: 0
/*0015*/ L0:   aload_0
/*0016*/       invokespecial   0x0005
/*0019*/       aload_0
/*001a*/       aload_1
/*001b*/       putfield_a      0x01
/*001d*/       aload_0
/*001e*/       bspush          0x2a
/*0020*/       putfield_b      0x02
/*0022*/       return
}

method_info[3] // @0023= {
// flags     : 0
// max_stack : 2
// nargs     : 2
// max_locals: 0
/*0025*/ L0:   aload_0
/*0026*/       aload_1
/*0027*/       invokespecial   0x0006
/*002a*/       aload_0
/*002b*/       aload_1
/*002c*/       putfield_a      0x03
/*002e*/       aload_0
/*002f*/       sconst_0
```

```
/*0030*/       putfield_b      0x04
/*0032*/       return
}

method_info[4] // @0033= {
// flags     : 0
// max_stack : 1
// nargs     : 1
// max_locals: 0
/*0035*/ L0:   aload_0
/*0036*/       invokespecial   0x0007
/*0039*/       aload_0
/*003a*/       invokevirtual   0x0008
/*003d*/       return
}

method_info[5] // @003e= {
// flags     : 0
// max_stack : 2
// nargs     : 3
// max_locals: 0
/*0040*/ L0:   new             0x0009
/*0043*/       dup
/*0044*/       invokespecial   0x000a
/*0047*/       pop
/*0048*/       return
}

method_info[6] // @0049= {
// flags     : 0
// max_stack : 3
// nargs     : 2
// max_locals: 2
/*004b*/ L0:   aload_1
/*004c*/       invokevirtual   0x000b
/*004f*/       astore_2
/*0050*/       new             0x000c
/*0053*/       dup
/*0054*/       aload_0
/*0055*/       invokespecial   0x000d
/*0058*/       astore_3
/*0059*/       return
}

method_info[7] // @005a= {
// flags     : 0
// max_stack : 0
```

```
// nargs     : 2
// max_locals: 0
/*005c*/ L0:   return
}

method_info[8] // @005d= {
// flags     : 0
// max_stack : 1
// nargs     : 2
// max_locals: 0
/*005f*/ L0:   aload_1
/*0060*/       invokeinterface  nargs : 0x01,index : 0 const: 0x000e,method: 0x02
/*0065*/       pop
/*0066*/       return
}

method_info[9] // @0067= {
// flags     : 0
// max_stack : 2
// nargs     : 3
// max_locals: 1
/*0069*/ L0:   new             0x000f
/*006c*/       dup
/*006d*/       invokespecial   0x0010
/*0070*/       astore_3
/*0071*/       return
}

method_info[10] // @0072= {
// flags     : 0
// max_stack : 3
// nargs     : 1
// max_locals: 3
/*0074*/ L0:   new             0x0011
/*0077*/       dup
/*0078*/       aload_0
/*0079*/       invokespecial   0x0006
/*007c*/       astore_1
/*007d*/       new             0x0012
/*0080*/       dup
/*0081*/       aload_0
/*0082*/       invokespecial   0x0013
/*0085*/       astore_2
/*0086*/       new             0x000c
/*0089*/       dup
/*008a*/       aload_0
/*008b*/       invokespecial   0x000d
```

```
/*008e*/       astore_3
/*008f*/       return
}
}

.StaticFieldComponent = {
image_size              : 0
reference_count         : 0
array_init_count        : 0
array_init = {
}
default_value_count     : 0
non_default_value_count : 0
non_default_values = {
}
}

.ReferenceLocationComponent = {
offsets_to_byte_indices = {
  @0011  @001c  @0021  @002d  @0031
}
offsets_to_byte2_indices = {
  @0005  @000c  @0017  @0028  @0037  @003b  @0041  @0045
  @004d  @0051  @0056  @0062  @006a  @006e  @0075  @007a
  @007e  @0083  @0087  @008c
}
}

.DescriptorComponent = {
count : 5
class_descriptor_info[0] = {
token        : 0
access_flag  : ACC_PUBLIC
this_class_ref : internal class reference (offset = 0000)
interface_count  : 0
field_count  : 0
method_count  : 1
method[0] = {
token          : 0
access_flag    : (81) ACC_INIT ACC_PUBLIC
method_offset  : 0x1
type_offset    : /* 48*/ ( ) V
bytecode_count : 5
exception_handler_count : 0
exception_handler_index : 0
}
}
```

```
class_descriptor_info[1] = {
token        : 1
access_flag  : ACC_PUBLIC
this_class_ref : internal class reference (offset = 000a)
interface_count  : 0
field_count  : 1
method_count  : 1
fiels[0] = {
token          : 0
access_flag    : 18
field_ref      : 00 0a 00
type           : 42
}
method[0] = {
token          : 0
access_flag    : (81) ACC_INIT ACC_PUBLIC
method_offset  : 0x8
type_offset    : /* 50*/ ( L00.28; ) V
bytecode_count : 9
exception_handler_count : 0
exception_handler_index : 0
}
}
class_descriptor_info[2] = {
token        : 2
access_flag  : ACC_PUBLIC
this_class_ref : internal class reference (offset = 0014)
interface_count  : 0
field_count  : 2
method_count  : 1
fiels[0] = {
token          : 0
access_flag    : 18
field_ref      : 00 14 00
type           : 42
}
fiels[1] = {
token          : 1
access_flag    : 2
field_ref      : 00 14 01
type           : 32771
}
method[0] = {
token          : 0
access_flag    : (81) ACC_INIT ACC_PUBLIC
method_offset  : 0x13
type_offset    : /* 50*/ ( L00.28; ) V
```

```
bytecode_count : 14
exception_handler_count : 0
exception_handler_index : 0
}
}
class_descriptor_info[3] = {
token        : 3
access_flag  : ACC_PUBLIC
this_class_ref : internal class reference (offset = 001e)
interface_count  : 0
field_count  : 2
method_count  : 1
fiels[0] = {
token          : 0
access_flag    : 18
field_ref      : 00 1e 00
type           : 42
}
fiels[1] = {
token          : 1
access_flag    : 2
field_ref      : 00 1e 01
type           : 32771
}
method[0] = {
token          : 0
access_flag    : (81) ACC_INIT ACC_PUBLIC
method_offset  : 0x23
type_offset    : /* 50*/ ( L00.28; ) V
bytecode_count : 14
exception_handler_count : 0
exception_handler_index : 0
}
}
class_descriptor_info[4] = {
token        : 4
access_flag  : ACC_PUBLIC
this_class_ref : internal class reference (offset = 0028)
interface_count  : 0
field_count  : 0
method_count  : 7
method[0] = {
token          : 0
access_flag    : (84) ACC_INIT ACC_PROTECTED
method_offset  : 0x33
type_offset    : /* 48*/ ( ) V
bytecode_count : 9
```

```
exception_handler_count : 0
exception_handler_index : 0
}
method[1] = {
token          : 1
access_flag    : (9) ACC_STATIC ACC_PUBLIC
method_offset  : 0x3e
type_offset    : /* 56*/ ( [B; S; B; ) V
bytecode_count : 9
exception_handler_count : 0
exception_handler_index : 0
}
method[2] = {
token          : 7
access_flag    : (1) ACC_PUBLIC
method_offset  : 0x49
type_offset    : /* 59*/ ( L81.0a; ) V
bytecode_count : 15
exception_handler_count : 0
exception_handler_index : 0
}
method[3] = {
token          : 128
access_flag    : (0)
method_offset  : 0x5a
type_offset    : /* 59*/ ( L81.0a; ) V
bytecode_count : 1
exception_handler_count : 0
exception_handler_index : 0
}
method[4] = {
token          : 8
access_flag    : (1) ACC_PUBLIC
method_offset  : 0x5d
type_offset    : /* 63*/ ( L81.01; ) V
bytecode_count : 8
exception_handler_count : 0
exception_handler_index : 0
}
method[5] = {
token          : 255
access_flag    : (a) ACC_STATIC ACC_PRIVATE
method_offset  : 0x67
type_offset    : /* 67*/ ( B; S; L00.1e; ) V
bytecode_count : 9
exception_handler_count : 0
exception_handler_index : 0
```

```
}
method[6] = {
token          : 255
access_flag    : (2) ACC_PRIVATE
method_offset  : 0x72
type_offset    : /* 48*/ ( ) V
bytecode_count : 28
exception_handler_count : 0
exception_handler_index : 0
}
}
type_descriptor_info = {
Initial offset value = 42
constant_pool_type : 2a 2a 2e 2a 2e 30 32 30 30 ffff 30 36 ffff 32 ffff ffff
                30 ffff ffff 32
type_desc[0] = (offset = 42)  L  (PACKAGE_TOKEN= 00, CLASS_TOKEN= 28) PADDING
type_desc[1] = (offset = 46)  B PADDING
type_desc[2] = (offset = 48)  V PADDING
type_desc[3] = (offset = 50)  L  (PACKAGE_TOKEN= 00, CLASS_TOKEN= 28) V
type_desc[4] = (offset = 54)  [B PADDING
type_desc[5] = (offset = 56)  [B S B V
type_desc[6] = (offset = 59)  L  (PACKAGE_TOKEN= 81, CLASS_TOKEN= 0a) V
type_desc[7] = (offset = 63)  L  (PACKAGE_TOKEN= 81, CLASS_TOKEN= 01) V
type_desc[8] = (offset = 67)  B S L  (PACKAGE_TOKEN= 00, CLASS_TOKEN= 1e) V
}
}
```

Chapitre 11 :

Model de la vérification de type

CONTEXT R00_JCConstants
CONSTANTS

bit_4
u1
u2
u4

AXIOMS

axm1_R0 : bit_4 = 0 .. 127
axm2_R0 : u1 = 0 .. 255
axm3_R0 : u2 = 0 .. 65535
axm4_R0 : u4 = 0 .. 4294967296

END

CONTEXT R02_StackSizeConstaints
EXTENDS R00_JCConstants
CONSTANTS

MaxStackSize

AXIOMS

axm1_R02 : MaxStackSize ∈ bit_4
axm2_R02 : MaxStackSize ∈ 0 .. 9

END

CONTEXT R04_Lattice
EXTENDS R02_StackSizeConstaints
SETS

LATICE_ELEMENTS
TYPES

CONSTANTS

unknownType
uninitLocalVar
Top
primitiveTypes
boolean
numericTypes
byte
shart
returnAddress
referenceTypes
classTypes
Object_uninit
Object_init
TestClassA_init

TestClassB_init
TestClassC_init
TestClassA_uninit
TestClassB_uninit
TestClassC_uninit
arrayTypes
booleanArray
byteArray
shartArray
Lattice

AXIOMS

axm0_2_R04 : TYPES = {uninitLocalVar, unknownType, Top, primitiveTypes, boolean, numeri

axm0_3_R04 : uninitLocalVar ≠ unknownType ∧ uninitLocalVar ≠ Top ∧ uninitLocalVar ≠ primitiveTypes ∧ uninitLocalVar ≠ boolean ∧ uninitLocalVar ≠ numericTypes ∧ uninitLocalVar ≠ byte ∧ uninitLocalVar ≠ shart ∧ uninitLocalVar ≠ returnAddress ∧ uninitLocalVar ≠ referenceTypes ∧ uninitLocalVar ≠ classTypes ∧ uninitLocalVar ≠ Object_uninit∧uninitLocalVar ≠ TestClassA_uninit∧ uninitLocalVar ≠ TestClassC_uninit ∧ uninitLocalVar ≠ TestClassB_uninit ∧ uninitLocalVar ≠ Object_init ∧ uninitLocalVar ≠ TestClassA_init∧uninitLocalVar ≠ TestClassC_init∧uninitLocalVar ≠ TestClassB_init ∧ uninitLocalVar ≠ arrayTypes ∧ uninitLocalVar ≠ booleanArray ∧ uninitLocalVar ≠ byteArray ∧ uninitLocalVar ≠ shartArray ∧ unknownType ≠ Top ∧ unknownType ≠ primitiveTypes ∧ unknownType ≠ boolean ∧ unknownType ≠ numericTypes ∧ unknownType ≠ byte ∧ unknownType ≠ shart ∧ unknownType ≠ returnAddress ∧ unknownType ≠ referenceTypes ∧ unknownType ≠ classTypes ∧ unknownType ≠ Object_uninit ∧ unknownType ≠ TestClassA_uninit ∧ unknownType ≠ TestClassC_uninit ∧ unknownType ≠ TestClassB_uninit ∧ unknownType ≠ Object_init ∧ unknownType ≠ TestClassA_init ∧ unknownType ≠ TestClassC_init ∧ unknownType ≠ TestClassB_init ∧ unknownType ≠ arrayTypes ∧ unknownType ≠ booleanArray ∧ unknownType ≠ byteArray ∧ unknownType ≠ shartArray ∧ Top ≠ primitiveTypes ∧ Top ≠ boolean ∧ Top ≠ numericTypes ∧ Top ≠ byte ∧ Top ≠ shart ∧ Top ≠ returnAddress ∧ Top ≠ referenceTypes ∧ Top ≠ classTypes ∧ Top ≠ Object_uninit ∧ Top ≠ TestClassA_uninit ∧ Top ≠ TestClassC_uninit ∧ Top ≠ TestClassB_uninit ∧ Top ≠ Object_init ∧ Top ≠ TestClassA_init ∧ Top ≠ TestClassC_init ∧ Top ≠ TestClassB_init ∧ Top ≠ arrayTypes ∧ Top ≠ booleanArray ∧ Top ≠ byteArray ∧ Top ≠ shartArray ∧ primitiveTypes ≠ boolean ∧ primitiveTypes ≠ numericTypes ∧ primitiveTypes ≠ byte ∧ primitiveTypes ≠ shart ∧ primitiveTypes ≠ returnAddress ∧ primitiveTypes ≠ referenceTypes ∧ primitiveTypes ≠ classTypes ∧ primitiveTypes ≠ Object_uninit∧primitiveTypes ≠ TestClassA_uninit∧ primitiveTypes ≠ TestClassC_uninit ∧ primitiveTypes ≠

$$\texttt{TestClassB\_uninit} \wedge \texttt{primitiveTypes} \neq \texttt{Object\_init} \wedge \texttt{primitiveTypes} \neq \texttt{TestClassA\_init} \wedge \texttt{primitiveTypes} \neq \texttt{TestClassC\_init} \wedge \texttt{primitiveTypes} \neq \texttt{TestClassB\_init} \wedge \texttt{primitiveTypes} \neq \texttt{arrayTypes} \wedge \texttt{primitiveTypes} \neq \texttt{booleanArray} \wedge \texttt{primitiveTypes} \neq \texttt{byteArray} \wedge \texttt{primitiveTypes} \neq \texttt{shartArray} \wedge \texttt{boolean} \neq \texttt{numericTypes} \wedge \texttt{boolean} \neq \texttt{byte} \wedge \texttt{boolean} \neq \texttt{shart} \wedge \texttt{boolean} \neq \texttt{returnAddress} \wedge \texttt{boolean} \neq \texttt{referenceTypes} \wedge \texttt{boolean} \neq \texttt{classTypes} \wedge \texttt{boolean} \neq \texttt{Object\_uninit} \wedge \texttt{boolean} \neq \texttt{TestClassA\_uninit} \wedge \texttt{boolean} \neq \texttt{TestClassC\_uninit} \wedge \texttt{boolean} \neq \texttt{TestClassB\_uninit} \wedge \texttt{boolean} \neq \texttt{Object\_init} \wedge \texttt{boolean} \neq \texttt{TestClassA\_init} \wedge \texttt{boolean} \neq \texttt{TestClassC\_init} \wedge \texttt{boolean} \neq \texttt{TestClassB\_init} \wedge \texttt{boolean} \neq \texttt{arrayTypes} \wedge \texttt{boolean} \neq \texttt{booleanArray} \wedge \texttt{boolean} \neq \texttt{byteArray} \wedge \texttt{boolean} \neq \texttt{shartArray} \wedge \texttt{numericTypes} \neq \texttt{byte} \wedge \texttt{numericTypes} \neq \texttt{shart} \wedge \texttt{numericTypes} \neq \texttt{returnAddress} \wedge \texttt{numericTypes} \neq \texttt{referenceTypes} \wedge \texttt{numericTypes} \neq \texttt{classTypes} \wedge \texttt{numericTypes} \neq \texttt{Object\_uninit} \wedge \texttt{numericTypes} \neq \texttt{TestClassA\_uninit} \wedge \texttt{numericTypes} \neq \texttt{TestClassC\_uninit} \wedge \texttt{numericTypes} \neq \texttt{TestClassB\_uninit} \wedge \texttt{numericTypes} \neq \texttt{Object\_init} \wedge \texttt{numericTypes} \neq \texttt{TestClassA\_init} \wedge \texttt{numericTypes} \neq \texttt{TestClassC\_init} \wedge \texttt{numericTypes} \neq \texttt{TestClassB\_init} \wedge \texttt{numericTypes} \neq \texttt{arrayTypes} \wedge \texttt{numericTypes} \neq \texttt{booleanArray} \wedge \texttt{numericTypes} \neq \texttt{byteArray} \wedge \texttt{numericTypes} \neq \texttt{shartArray} \wedge \texttt{byte} \neq \texttt{shart} \wedge \texttt{byte} \neq \texttt{returnAddress} \wedge \texttt{byte} \neq \texttt{referenceTypes} \wedge \texttt{byte} \neq \texttt{classTypes} \wedge \texttt{byte} \neq \texttt{Object\_uninit} \wedge \texttt{byte} \neq \texttt{TestClassA\_uninit} \wedge \texttt{byte} \neq \texttt{TestClassC\_uninit} \wedge \texttt{byte} \neq \texttt{TestClassB\_uninit} \wedge \texttt{byte} \neq \texttt{Object\_init} \wedge \texttt{byte} \neq \texttt{TestClassA\_init} \wedge \texttt{byte} \neq \texttt{TestClassC\_init} \wedge \texttt{byte} \neq \texttt{TestClassB\_init} \wedge \texttt{byte} \neq \texttt{arrayTypes} \wedge \texttt{byte} \neq \texttt{booleanArray} \wedge \texttt{byte} \neq \texttt{byteArray} \wedge \texttt{byte} \neq \texttt{shartArray} \wedge \texttt{shart} \neq \texttt{returnAddress} \wedge \texttt{shart} \neq \texttt{referenceTypes} \wedge \texttt{shart} \neq \texttt{classTypes} \wedge \texttt{shart} \neq \texttt{Object\_uninit} \wedge \texttt{shart} \neq \texttt{TestClassA\_uninit} \wedge \texttt{shart} \neq \texttt{TestClassC\_uninit} \wedge \texttt{shart} \neq \texttt{TestClassB\_uninit} \wedge \texttt{shart} \neq \texttt{Object\_init} \wedge \texttt{shart} \neq \texttt{TestClassA\_init} \wedge \texttt{shart} \neq \texttt{TestClassC\_init} \wedge \texttt{shart} \neq \texttt{TestClassB\_init} \wedge \texttt{shart} \neq \texttt{arrayTypes} \wedge \texttt{shart} \neq \texttt{booleanArray} \wedge \texttt{shart} \neq \texttt{byteArray} \wedge \texttt{shart} \neq \texttt{shartArray} \wedge \texttt{returnAddress} \neq \texttt{referenceTypes} \wedge \texttt{returnAddress} \neq \texttt{classTypes} \wedge \texttt{returnAddress} \neq \texttt{Object\_uninit} \wedge \texttt{returnAddress} \neq \texttt{TestClassA\_uninit} \wedge \texttt{returnAddress} \neq \texttt{TestClassC\_uninit} \wedge \texttt{returnAddress} \neq \texttt{TestClassB\_uninit} \wedge \texttt{returnAddress} \neq \texttt{Object\_init} \wedge \texttt{returnAddress} \neq \texttt{TestClassA\_init} \wedge \texttt{returnAddress} \neq \texttt{TestClassC\_init} \wedge \texttt{returnAddress} \neq \texttt{TestClassB\_init} \wedge \texttt{returnAddress} \neq \texttt{arrayTypes} \wedge \texttt{returnAddress} \neq \texttt{booleanArray} \wedge \texttt{returnAddress} \neq \texttt{byteArray} \wedge \texttt{returnAddress} \neq \texttt{shartArray} \wedge \texttt{referenceTypes} \neq \texttt{classTypes} \wedge \texttt{referenceTypes} \neq \texttt{Object\_uninit} \wedge \texttt{referenceTypes} \neq \texttt{TestClassA\_uninit} \wedge \texttt{referenceTypes} \neq \texttt{TestClassC\_uninit} \wedge \texttt{referenceTypes} \neq \texttt{TestClassB\_uninit} \wedge \texttt{referenceTypes} \neq \texttt{Object\_init} \wedge \texttt{referenceTypes} \neq \texttt{TestClassA\_init} \wedge \texttt{referenceTypes} \neq \texttt{TestClassC\_init} \wedge \texttt{referenceTypes} \neq \texttt{TestClassB\_init} \wedge \texttt{referenceTypes} \neq \texttt{arrayTypes} \wedge \texttt{referenceTypes} \neq \texttt{booleanArray} \wedge \texttt{referenceTypes} \neq \texttt{byteArray} \wedge \texttt{referenceTypes} \neq \texttt{shartArray} \wedge \texttt{classTypes} \neq \texttt{Object\_uninit} \wedge \texttt{classTypes} \neq \texttt{TestClassA\_uninit} \wedge \texttt{classTypes} \neq \texttt{TestClassC\_uninit} \wedge \texttt{classTypes} \neq$$

$$\begin{aligned}
&\texttt{Object_init} \wedge \texttt{classTypes} \neq \texttt{TestClassA_init} \wedge \texttt{classTypes} \neq \\
&\texttt{TestClassC_init} \wedge \texttt{classTypes} \neq \texttt{TestClassB_init} \wedge \texttt{classTypes} \neq \\
&\texttt{arrayTypes} \wedge \texttt{classTypes} \neq \texttt{booleanArray} \wedge \texttt{classTypes} \neq \\
&\texttt{byteArray} \wedge \texttt{classTypes} \neq \texttt{shartArray} \wedge \texttt{Object_uninit} \neq \\
&\texttt{TestClassA_uninit} \wedge \texttt{Object_uninit} \neq \texttt{TestClassC_uninit} \wedge \\
&\texttt{Object_uninit} \neq \texttt{TestClassB_uninit} \wedge \texttt{Object_uninit} \neq \\
&\texttt{Object_init} \wedge \texttt{Object_uninit} \neq \texttt{TestClassA_init} \wedge \texttt{Object_uninit} \neq \\
&\texttt{TestClassC_init} \wedge \texttt{Object_uninit} \neq \texttt{TestClassB_init} \wedge \texttt{Object_uninit} \neq \\
&\texttt{arrayTypes} \wedge \texttt{Object_uninit} \neq \texttt{booleanArray} \wedge \texttt{Object_uninit} \neq \\
&\texttt{byteArray} \wedge \texttt{Object_uninit} \neq \texttt{shartArray} \wedge \texttt{TestClassA_uninit} \neq \\
&\texttt{TestClassC_uninit} \wedge \texttt{TestClassA_uninit} \neq \texttt{TestClassB_uninit} \wedge \\
&\texttt{TestClassA_uninit} \neq \texttt{Object_init} \wedge \texttt{TestClassA_uninit} \neq \\
&\texttt{TestClassA_init} \wedge \texttt{TestClassA_uninit} \neq \texttt{TestClassC_init} \wedge \\
&\texttt{TestClassA_uninit} \neq \texttt{TestClassB_init} \wedge \texttt{TestClassA_uninit} \neq \\
&\texttt{arrayTypes} \wedge \texttt{TestClassA_uninit} \neq \texttt{booleanArray} \wedge \texttt{TestClassA_uninit} \neq \\
&\texttt{byteArray} \wedge \texttt{TestClassA_uninit} \neq \texttt{shartArray} \wedge \texttt{TestClassC_uninit} \neq \\
&\texttt{TestClassB_uninit} \wedge \texttt{TestClassC_uninit} \neq \texttt{Object_init} \wedge \\
&\texttt{TestClassC_uninit} \neq \texttt{TestClassA_init} \wedge \texttt{TestClassC_uninit} \neq \\
&\texttt{TestClassC_init} \wedge \texttt{TestClassC_uninit} \neq \texttt{TestClassB_init} \wedge \\
&\texttt{TestClassC_uninit} \neq \texttt{arrayTypes} \wedge \texttt{TestClassC_uninit} \neq \\
&\texttt{booleanArray} \wedge \texttt{TestClassC_uninit} \neq \texttt{byteArray} \wedge \texttt{TestClassC_uninit} \neq \\
&\texttt{shartArray} \wedge \texttt{TestClassB_uninit} \neq \texttt{Object_init} \wedge \texttt{TestClassB_uninit} \neq \\
&\texttt{TestClassA_init} \wedge \texttt{TestClassB_uninit} \neq \texttt{TestClassC_init} \wedge \\
&\texttt{TestClassB_uninit} \neq \texttt{TestClassB_init} \wedge \texttt{TestClassB_uninit} \neq \\
&\texttt{arrayTypes} \wedge \texttt{TestClassB_uninit} \neq \texttt{booleanArray} \wedge \texttt{TestClassB_uninit} \neq \\
&\texttt{byteArray} \wedge \texttt{TestClassB_uninit} \neq \texttt{shartArray} \wedge \texttt{Object_init} \neq \\
&\texttt{TestClassA_init} \wedge \texttt{Object_init} \neq \texttt{TestClassC_init} \wedge \texttt{Object_init} \neq \\
&\texttt{TestClassB_init} \wedge \texttt{Object_init} \neq \texttt{arrayTypes} \wedge \texttt{Object_init} \neq \\
&\texttt{booleanArray} \wedge \texttt{Object_init} \neq \texttt{byteArray} \wedge \texttt{Object_init} \neq \\
&\texttt{shartArray} \wedge \texttt{TestClassA_init} \neq \texttt{TestClassC_init} \wedge \texttt{TestClassA_init} \neq \\
&\texttt{TestClassB_init} \wedge \texttt{TestClassA_init} \neq \texttt{arrayTypes} \wedge \texttt{TestClassA_init} \neq \\
&\texttt{booleanArray} \wedge \texttt{TestClassA_init} \neq \texttt{byteArray} \wedge \texttt{TestClassA_init} \neq \\
&\texttt{shartArray} \wedge \texttt{TestClassC_init} \neq \texttt{TestClassB_init} \wedge \texttt{TestClassC_init} \neq \\
&\texttt{arrayTypes} \wedge \texttt{TestClassC_init} \neq \texttt{booleanArray} \wedge \texttt{TestClassC_init} \neq \\
&\texttt{byteArray} \wedge \texttt{TestClassC_init} \neq \texttt{shartArray} \wedge \texttt{TestClassB_init} \neq \\
&\texttt{arrayTypes} \wedge \texttt{TestClassB_init} \neq \texttt{booleanArray} \wedge \texttt{TestClassB_init} \neq \\
&\texttt{byteArray} \wedge \texttt{TestClassB_init} \neq \texttt{shartArray} \wedge \texttt{arrayTypes} \neq \texttt{booleanArray} \wedge \\
&\texttt{arrayTypes} \neq \texttt{byteArray} \wedge \texttt{arrayTypes} \neq \texttt{shartArray} \wedge \texttt{booleanArray} \neq \\
&\texttt{byteArray} \wedge \texttt{booleanArray} \neq \texttt{shartArray} \wedge \texttt{byteArray} \neq \texttt{shartArray}
\end{aligned}$$

$$\begin{aligned}
&\texttt{axm1_1_R04} : \texttt{Lattice} = \{(\texttt{shartArray} \mapsto \texttt{shartArray}), (\texttt{byteArray} \mapsto \\
&\texttt{byteArray}), (\texttt{booleanArray} \mapsto \texttt{booleanArray}), (\texttt{arrayTypes} \mapsto \\
&\texttt{shartArray}), (\texttt{arrayTypes} \mapsto \texttt{byteArray}), (\texttt{arrayTypes} \mapsto \\
&\texttt{booleanArray}), (\texttt{arrayTypes} \mapsto \texttt{arrayTypes}), (\texttt{TestClassB_init} \mapsto \\
&\texttt{TestClassB_init}), (\texttt{TestClassC_init} \mapsto \texttt{TestClassC_init}), (\texttt{TestClassA_init} \mapsto \\
&\texttt{TestClassC_init}), (\texttt{TestClassA_init} \mapsto \texttt{TestClassA_init}), (\texttt{Object_init} \mapsto \\
&\texttt{TestClassB_init}), (\texttt{Object_init} \mapsto \texttt{TestClassC_init}), (\texttt{Object_init} \mapsto \\
&\texttt{TestClassA_init}), (\texttt{Object_init} \mapsto \texttt{Object_init}), (\texttt{TestClassB_uninit} \mapsto
\end{aligned}$$

TestClassB_uninit), (TestClassC_uninit ↦ TestClassC_uninit), (TestClassA_uninit ↦ TestClassC_uninit), (TestClassA_uninit ↦ TestClassA_uninit), (Object_uninit ↦ TestClassB_uninit), (Object_uninit ↦ TestClassC_uninit), (Object_uninit ↦ TestClassA_uninit), (Object_uninit ↦ Object_uninit), (classTypes ↦ TestClassB_init), (classTypes ↦ TestClassC_init), (classTypes ↦ TestClassA_init), (classTypes ↦ Object_init), (classTypes ↦ TestClassB_uninit), (classTypes ↦ TestClassC_uninit), (classTypes ↦ TestClassA_uninit), (classTypes ↦ Object_uninit), (classTypes ↦ classTypes), (referenceTypes ↦ shartArray), (referenceTypes ↦ byteArray), (referenceTypes ↦ booleanArray), (referenceTypes ↦ arrayTypes), (referenceTypes ↦ TestClassB_init), (referenceTypes ↦ TestClassC_init), (referenceTypes ↦ TestClassA_init), (referenceTypes ↦ Object_init), (referenceTypes ↦ TestClassB_uninit), (referenceTypes ↦ TestClassC_uninit), (referenceTypes ↦ TestClassA_uninit), (referenceTypes ↦ Object_uninit), (referenceTypes ↦ classTypes), (referenceTypes ↦ referenceTypes), (returnAddress ↦ returnAddress), (shart ↦ shart), (byte ↦ byte), (numericTypes ↦ shart), (numericTypes ↦ byte), (numericTypes ↦ numericTypes), (boolean ↦ boolean), (primitiveTypes ↦ returnAddress), (primitiveTypes ↦ shart), (primitiveTypes ↦ byte), (primitiveTypes ↦ numericTypes), (primitiveTypes ↦ boolean), (primitiveTypes ↦ primitiveTypes), (Top ↦ shartArray), (Top ↦ byteArray), (Top ↦ booleanArray), (Top ↦ arrayTypes), (Top ↦ TestClassB_init), (Top ↦ TestClassC_init), (Top ↦ TestClassA_init), (Top ↦ Object_init), (Top ↦ TestClassB_uninit), (Top ↦ TestClassC_uninit), (Top ↦ TestClassA_uninit), (Top ↦ Object_uninit), (Top ↦ classTypes), (Top ↦ referenceTypes), (Top ↦ returnAddress), (Top ↦ shart), (Top ↦ byte), (Top ↦ numericTypes), (Top ↦ boolean), (Top ↦ primitiveTypes), (Top ↦ Top), (unknownType ↦ unknownType), (uninitLocalVar ↦ uninitLocalVar)}

END

CONTEXT R05_LocalVariablesConstraints
EXTENDS R04_Lattice
CONSTANTS

MaxArgumentIndex
ArgumentTypes
MaxLocalVariablesIndex
nbOutOfBoundIndex
initLocalVariables

AXIOMS

axm1_R05 : MaxArgumentIndex ∈ ℕ
axm2_R05 : ArgumentTypes ∈ 0 .. MaxArgumentIndex → TYPES
axm3_R05 : MaxLocalVariablesIndex ∈ ℕ

axm4_R05 : MaxLocalVariablesIndex $\geq$ MaxArgumentIndex

axm5_R05 : nbOutOfBoundIndex $\in \mathbb{N}$

axm6_R05 : initLocalVariables $\in$ 0 .. MaxLocalVariablesIndex + nbOutOfBoundIndex $\rightarrow$ TYPES

axm7_R05 : MaxArgumentIndex $= 2$

axm8_R05 : ArgumentTypes $= \{0 \mapsto \text{byte}, 1 \mapsto \text{shart}, 2 \mapsto \text{referenceTypes}\}$

axm9_R05 : MaxLocalVariablesIndex $= 4$

axm10_R05 : nbOutOfBoundIndex $= 5$

axm11_R05 : initLocalVariables $\subseteq$ ArgumentTypes $\cup$ (MaxArgumentIndex + 1 .. MaxLocalVariablesIndex $\times$ {uninitLocalVar}) $\cup$ (MaxLocalVariablesIndex + 1 .. MaxLocalVariablesIndex + nbOutOfBoundIndex $\times$ {unknownType})

END

CONTEXT R06_ContantPool
EXTENDS R05_LocalVariablesConstraints
CONSTANTS

MaxConstantPoolIndex

ConstantPool

InitialisationMethods

AXIOMS

axm1 : MaxConstantPoolIndex $\in 1 .. 20$

axm2 : ConstantPool $\in 0 .. \text{MaxConstantPoolIndex} \nrightarrow$ TYPES

axm3 : InitialisationMethods $\in \mathbb{N} \nrightarrow \mathbb{N}$

axm4_R06 : ConstantPool $= \{14 \mapsto \text{TestClassC\_uninit}, 19 \mapsto \text{TestClassA\_uninit}, 20 \mapsto \text{TestClassB\_uninit}\}$

axm5_R06 : InitialisationMethods $= \{14 \mapsto 15, 19 \mapsto 6, 20 \mapsto 21\}$

END

CONTEXT R07_InstructionParameters
EXTENDS R06_ContantPool
SETS

ATYPES_E

CONSTANTS

ATYPES

T_BOOLEAN

T_BYTE

T_SHART

T_INT

AXIOMS

axm1_R07 : ATYPES $\subseteq$ ATYPES_E

axm2_R07 : ATYPES $= \{\text{T\_BOOLEAN}, \text{T\_BYTE}, \text{T\_SHART}, \text{T\_INT}\}$

axm3_R07 : T_BOOLEAN ≠ T_BYTE ∧ T_BOOLEAN ≠ T_SHART ∧ T_BOOLEAN ≠ T_INT ∧ T_BYTE ≠ T_SHART ∧ T_BYTE ≠ T_INT ∧ T_SHART ≠ T_INT

END

MACHINE R08_ArgumentsAndForms
REFINES R07_JavaCardInstructionsStackAndLocalModification
SEES R06_ContantPool
VARIABLES

programRunning
stackSize
stackTypes
localVariablesTypes

EVENTS
Initialisation
extended
begin

act1 : $programRunning := TRUE$
act2 : $stackSize := 0$
act3 : $stackTypes := -4 \mathinner{..} -1 \times \{unknownType\}$
act4 : $localVariablesTypes := initLocalVariables$

end
Event *aconst_null_R08* $\widehat{=}$
extends *aconst_null_R07*
any

firstStackOutputType
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize < MaxStackSize$
grd3 : $firstStackOutputType \in TYPES$
grd4 : $firstStackOutputType = referenceTypes$
grd5 : form = 1

then

act1 : $stackSize := stackSize + 1$
act2 : $stackTypes := stackTypes \cup \{stackSize \mapsto firstStackOutputType\}$

end
Event *aload_R08* $\widehat{=}$
extends *aload_R07*
any

firstStackOutputType
prm_index

form

where

grd1 : $programRunning = TRUE$

grd2_t : $stackSize < MaxStackSize$

grd3 : $firstStackOutputType \in TYPES$

grd4 : $prm\_index \in dom(localVariablesTypes)$

grd5_t : $prm\_index \leq MaxLocalVariablesIndex$

grd6 : $firstStackOutputType = localVariablesTypes(prm\_index)$

grd6_t : $referenceTypes \mapsto firstStackOutputType \in Lattice$

grd7 : $form = 21$

then

act1 : $stackSize := stackSize + 1$

act2 : $stackTypes := stackTypes \cup \{stackSize \mapsto firstStackOutputType\}$

end

Event *aload_0_R08* $\widehat{=}$
extends *aload_0_R07*

any

firstStackOutputType

prm_index

form

where

grd1 : $programRunning = TRUE$

grd2_t : $stackSize < MaxStackSize$

grd3 : $firstStackOutputType \in TYPES$

grd4 : $prm\_index \in dom(localVariablesTypes)$

grd5_t : $prm\_index \leq MaxLocalVariablesIndex$

grd6 : $firstStackOutputType = localVariablesTypes(prm\_index)$

grd7 : $prm\_index = 0$

grd8_t : $referenceTypes \mapsto firstStackOutputType \in Lattice$

grd9 : $form = 24$

then

act1 : $stackSize := stackSize + 1$

act2 : $stackTypes := stackTypes \cup \{stackSize \mapsto firstStackOutputType\}$

end

Event *aload_1_R08* $\widehat{=}$
extends *aload_1_R07*

any

firstStackOutputType

prm_index

form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize < MaxStackSize$
grd3 : $firstStackOutputType \in TYPES$
grd4 : $prm\_index \in dom(localVariablesTypes)$
grd5_t : $prm\_index \leq MaxLocalVariablesIndex$
grd6 : $firstStackOutputType = localVariablesTypes(prm\_index)$
grd7 : $prm\_index = 1$
grd8_t : $referenceTypes \mapsto firstStackOutputType \in Lattice$
grd9 : $\texttt{form} = 25$

then

act1 : $stackSize := stackSize + 1$
act2 : $stackTypes := stackTypes \cup \{stackSize \mapsto firstStackOutputType\}$

end

Event *aload_2_R08* $\widehat{=}$
extends *aload_2_R07*

any

firstStackOutputType
prm_index
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize < MaxStackSize$
grd3 : $firstStackOutputType \in TYPES$
grd4 : $prm\_index \in dom(localVariablesTypes)$
grd5_t : $prm\_index \leq MaxLocalVariablesIndex$
grd6 : $firstStackOutputType = localVariablesTypes(prm\_index)$
grd7 : $prm\_index = 2$
grd8_t : $referenceTypes \mapsto firstStackOutputType \in Lattice$
grd9 : $\texttt{form} = 26$

then

act1 : $stackSize := stackSize + 1$
act2 : $stackTypes := stackTypes \cup \{stackSize \mapsto firstStackOutputType\}$

end

Event *aload_3_R08* $\widehat{=}$
extends *aload_3_R07*

any

firstStackOutputType
prm_index
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize < MaxStackSize$
grd3 : $firstStackOutputType \in TYPES$
grd4 : $prm\_index \in dom(localVariablesTypes)$
grd5_t : $prm\_index \leq MaxLocalVariablesIndex$
grd6 : $firstStackOutputType = localVariablesTypes(prm\_index)$
grd7 : $prm\_index = 3$
grd8_t : $referenceTypes \mapsto firstStackOutputType \in Lattice$
grd9 : $\texttt{form} = 27$

then

act1 : $stackSize := stackSize + 1$
act2 : $stackTypes := stackTypes \cup \{stackSize \mapsto firstStackOutputType\}$

end

Event *areturn_R08* $\widehat{=}$
extends *areturn_R07*

any

firstStackInputType
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize > 0$
grd3 : $firstStackInputType \in TYPES$
grd4 : $firstStackInputType = stackTypes(stackSize - 1)$
grd5_t : $referenceTypes \mapsto firstStackInputType \in Lattice$
grd6 : $\texttt{form} = 119$

then

act1 : $programRunning := FALSE$

end

Event *arraylength_R08* $\widehat{=}$
extends *arraylength_R07*

any

firstStackInputType
firstStackOutputType
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize > 0$
grd3 : $firstStackInputType \in TYPES$
grd4 : $firstStackOutputType \in TYPES$

grd5 : $firstStackInputType = stackTypes(stackSize - 1)$
grd6_t : $arrayTypes \mapsto firstStackInputType \in Lattice$
grd7 : $firstStackOutputType = shart$
grd8 : $\texttt{form} = 146$

then

act1 : $stackTypes := stackTypes \lhd\!\!\!- \{stackSize - 1 \mapsto firstStackOutputType\}$

end

Event *astore_R08* $\widehat{=}$
extends *astore_R07*

any

$firstStackInputType$
prm_index
$\texttt{form}$

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize > 0$
grd3 : $firstStackInputType \in TYPES$
grd4 : $firstStackInputType = stackTypes(stackSize - 1)$
grd5 : $prm\_index \in dom(localVariablesTypes)$
grd6_t : $prm\_index \leq MaxLocalVariablesIndex$
grd7_t : $referenceTypes \mapsto firstStackInputType \in Lattice \lor firstStackInputType = returnAddress$
grd7 : $\texttt{form} = 40$

then

act1 : $stackSize := stackSize - 1$
act2 : $stackTypes := \{stackSize - 1\} \lhd\!\!\!- stackTypes$
act3 : $localVariablesTypes(prm\_index) := firstStackInputType$

end

Event *astore_0_R08* $\widehat{=}$
extends *astore_0_R07*

any

$firstStackInputType$
prm_index
$\texttt{form}$

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize > 0$
grd3 : $firstStackInputType \in TYPES$
grd4 : $firstStackInputType = stackTypes(stackSize - 1)$
grd5 : $prm\_index \in dom(localVariablesTypes)$

grd6_t : $prm\_index \leq MaxLocalVariablesIndex$
grd6 : $prm\_index = 0$
grd7_t : $referenceTypes \mapsto firstStackInputType \in Lattice \vee firstStackInputType = returnAddress$
grd8 : $\texttt{form} = 43$

then

act1 : $stackSize := stackSize - 1$
act2 : $stackTypes := \{stackSize - 1\} \lhd stackTypes$
act3 : $localVariablesTypes(prm\_index) := firstStackInputType$

end

Event *astore_1_R08* $\widehat{=}$
extends *astore_1_R07*

any

$firstStackInputType$
prm_index
$\texttt{form}$

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize > 0$
grd3 : $firstStackInputType \in TYPES$
grd4 : $firstStackInputType = stackTypes(stackSize - 1)$
grd5 : $prm\_index \in dom(localVariablesTypes)$
grd6_t : $prm\_index \leq MaxLocalVariablesIndex$
grd7 : $prm\_index = 1$
grd8_t : $referenceTypes \mapsto firstStackInputType \in Lattice \vee firstStackInputType = returnAddress$
grd8 : $\texttt{form} = 44$

then

act1 : $stackSize := stackSize - 1$
act2 : $stackTypes := \{stackSize - 1\} \lhd stackTypes$
act3 : $localVariablesTypes(prm\_index) := firstStackInputType$

end

Event *astore_2_R08* $\widehat{=}$
extends *astore_2_R07*

any

$firstStackInputType$
prm_index
$\texttt{form}$

where

grd1 : $programRunning = TRUE$

grd2_t : $stackSize > 0$
grd3 : $firstStackInputType \in TYPES$
grd4 : $firstStackInputType = stackTypes(stackSize - 1)$
grd5 : $prm\_index \in dom(localVariablesTypes)$
grd6_t : $prm\_index \leq MaxLocalVariablesIndex$
grd7 : $prm\_index = 2$
grd8_t : $referenceTypes \mapsto firstStackInputType \in Lattice \vee firstStackInputType = returnAddress$
grd8 : $\texttt{form} = 45$

then

act1 : $stackSize := stackSize - 1$
act2 : $stackTypes := \{stackSize - 1\} \lhd stackTypes$
act3 : $localVariablesTypes(prm\_index) := firstStackInputType$

end

Event *astore_3_R08* $\widehat{=}$
extends *astore_3_R07*

any

$firstStackInputType$
prm_index
$\texttt{form}$

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize > 0$
grd3 : $firstStackInputType \in TYPES$
grd4 : $firstStackInputType = stackTypes(stackSize - 1)$
grd5 : $prm\_index \in dom(localVariablesTypes)$
grd6_t : $prm\_index \leq MaxLocalVariablesIndex$
grd7 : $prm\_index = 3$
grd8_t : $referenceTypes \mapsto firstStackInputType \in Lattice \vee firstStackInputType = returnAddress$
grd8 : $\texttt{form} = 46$

then

act1 : $stackSize := stackSize - 1$
act2 : $stackTypes := \{stackSize - 1\} \lhd stackTypes$
act3 : $localVariablesTypes(prm\_index) := firstStackInputType$

end

Event *baload_R08* $\widehat{=}$
extends *baload_R07*

any

$firstStackInputType$

$secondStackInputType$
$firstStackOutputType$
$\texttt{form}$

where

$\texttt{grd1}$: $programRunning = TRUE$
$\texttt{grd2\_t}$: $stackSize > 0$
$\texttt{grd3\_t}$: $stackSize > 1$
$\texttt{grd4}$: $firstStackInputType \in TYPES$
$\texttt{grd5}$: $secondStackInputType \in TYPES$
$\texttt{grd6}$: $firstStackOutputType \in TYPES$
$\texttt{grd7}$: $firstStackInputType = stackTypes(stackSize - 1)$
$\texttt{grd8}$: $secondStackInputType = stackTypes(stackSize - 2)$
$\texttt{grd9\_t}$: $firstStackInputType = shart$
$\texttt{grd10\_t}$: $secondStackInputType = booleanArray \lor secondStackInputType = byteArray$
$\texttt{grd11}$: $secondStackInputType = booleanArray \Rightarrow firstStackOutputType = boolean$
$\texttt{grd12}$: $secondStackInputType = byteArray \Rightarrow firstStackOutputType = byte$
$\texttt{grd13}$: $\texttt{form} = 37$

then

$\texttt{act1}$: $stackSize := stackSize - 1$
$\texttt{act2}$: $stackTypes := (\{stackSize-1, stackSize-2\} \lhd stackTypes) \cup \{stackSize-2 \mapsto firstStackOutputType\}$

end

Event *bastore_R08* $\widehat{=}$
extends *bastore_R07*

any

$firstStackInputType$
$secondStackInputType$
$thirdStackInputType$
$\texttt{form}$

where

$\texttt{grd1}$: $programRunning = TRUE$
$\texttt{grd2\_t}$: $stackSize - 2 > 0$
$\texttt{grd3\_t}$: $stackSize > 2$
$\texttt{grd4}$: $firstStackInputType \in TYPES$
$\texttt{grd5}$: $secondStackInputType \in TYPES$
$\texttt{grd6}$: $thirdStackInputType \in TYPES$
$\texttt{grd7}$: $firstStackInputType = stackTypes(stackSize - 1)$
$\texttt{grd8}$: $secondStackInputType = stackTypes(stackSize - 2)$
$\texttt{grd9}$: $thirdStackInputType = stackTypes(stackSize - 3)$

grd9_t : $firstStackInputType = shart$
grd10_t : $secondStackInputType = shart$
grd11_t : $thirdStackInputType = booleanArray \lor thirdStackInputType = byteArray$
grd12 : $\texttt{form} = 56$

then

act1 : $stackSize := stackSize - 3$
act2 : $stackTypes := \{stackSize - 1, stackSize - 2, stackSize - 3\} \lhd stackTypes$

end

Event *bspush_R08* $\widehat{=}$
extends *bspush_R07*

any

firstStackOutputType
prm_byte
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize < MaxStackSize$
grd3 : $firstStackOutputType \in TYPES$
grd4 : $firstStackOutputType = byte$
grd5 : $\texttt{prm\_byte} \in \texttt{u1}$
grd6 : $\texttt{form} = 16$

then

act1 : $stackSize := stackSize + 1$
act2 : $stackTypes := stackTypes \cup \{stackSize \mapsto firstStackOutputType\}$

end

Event *dup_R08* $\widehat{=}$
extends *dup_R07*

any

firstStackInputType
firstStackOutputType
secondStackOutputType
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize < MaxStackSize$
grd3_t : $stackSize > 0$
grd4 : $firstStackInputType \in TYPES$
grd5 : $firstStackOutputType \in TYPES$
grd6 : $secondStackOutputType \in TYPES$

grd7 : $firstStackInputType = stackTypes(stackSize - 1)$
grd9 : $firstStackOutputType = firstStackInputType$
grd10 : $secondStackOutputType = firstStackInputType$
grd11 : $\texttt{form} = 61$

then

act1 : $stackSize := stackSize + 1$
act2 : $stackTypes := stackTypes \lhd \{stackSize - 1 \mapsto firstStackOutputType, stackSize \mapsto secondStackOutputType\}$

end

Event *dup_2_R08* $\widehat{=}$
extends *dup_2_R07*

any

$firstStackInputType$
$secondStackInputType$
$firstStackOutputType$
$secondStackOutputType$
$thirdStackOutputType$
$fourthStackOutputType$
$\texttt{form}$

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize + 1 < MaxStackSize$
grd3_t : $stackSize > 1$
grd4 : $firstStackInputType \in TYPES$
grd5 : $secondStackInputType \in TYPES$
grd6 : $firstStackOutputType \in TYPES$
grd7 : $secondStackOutputType \in TYPES$
grd8 : $thirdStackOutputType \in TYPES$
grd9 : $fourthStackOutputType \in TYPES$
grd10 : $firstStackInputType = stackTypes(stackSize - 1)$
grd11 : $secondStackInputType = stackTypes(stackSize - 2)$
grd15 : $firstStackOutputType = secondStackInputType$
grd16 : $secondStackOutputType = firstStackInputType$
grd17 : $thirdStackOutputType = secondStackInputType$
grd18 : $fourthStackOutputType = firstStackInputType$
grd19 : $\texttt{form} = 62$

then

act1 : $stackSize := stackSize + 2$
act2 : $stackTypes := stackTypes \lhd \{stackSize - 2 \mapsto firstStackOutputType, stackSize - 1 \mapsto secondStackOutputType, stackSize \mapsto thirdStackOutputType, stackSize + 1 \mapsto fourthStackOutputType\}$

end

Event *new_R08* $\widehat{=}$
extends *new_R07*

any

firstStackOutputType
CPool_elem
`form`
`prm_indexbyte1`
`prm_indexbyte2`

where

`grd1` : $programRunning = TRUE$
`grd2_t` : $stackSize < MaxStackSize$
`grd3` : $firstStackOutputType \in TYPES$
`grd4_t` : $classTypes \mapsto CPool\_elem \in Lattice$
`grd5` : $firstStackOutputType = CPool\_elem$
`grd6` : $\texttt{form} = 143$
`grd7` : $\texttt{prm\_indexbyte1} \in \texttt{u1}$
`grd8` : $\texttt{prm\_indexbyte2} \in \texttt{u1}$
`grd9` : $\texttt{prm\_indexbyte1} * 128 + \texttt{prm\_indexbyte2} \in \texttt{dom}(\texttt{ConstantPool})$
`grd10` : $\texttt{CPool\_elem} = \texttt{ConstantPool}(\texttt{prm\_indexbyte1} * 128 + \texttt{prm\_indexbyte2})$

then

`act1` : $stackSize := stackSize + 1$
`act2` : $stackTypes := stackTypes \cup \{stackSize \mapsto firstStackOutputType\}$

end

Event *newarray_R08* $\widehat{=}$
extends *newarray_R07*

any

firstStackInputType
firstStackOutputType
prm_atype
`form`

where

`grd1` : $programRunning = TRUE$
`grd2_t` : $stackSize > 0$
`grd3` : $firstStackInputType \in TYPES$
`grd4` : $firstStackOutputType \in TYPES$
`grd5` : $firstStackInputType = stackTypes(stackSize - 1)$
`grd6_t` : $prm\_atype \in 10 \mathinner{..} 12$
`grd7_t` : $firstStackInputType = shart$

grd8_t : $prm\_atype = 10 \Rightarrow firstStackOutputType = booleanArray$
grd9_t : $prm\_atype = 11 \Rightarrow firstStackOutputType = byteArray$
grd10_t : $prm\_atype = 12 \Rightarrow firstStackOutputType = shartArray$
grd12 : $\texttt{form} = 144$

then

act1 : $stackTypes := stackTypes \lhd\!\!\!- \{stackSize - 1 \mapsto firstStackOutputType\}$

end

Event *nop_R08* $\widehat{=}$
extends *nop_R07*

any

form

where

grd1 : $programRunning = TRUE$
grd2 : $\texttt{form} = 0$

then

skip

end

Event *pop_R08* $\widehat{=}$
extends *pop_R07*

any

firstStackInputType
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize > 0$
grd3 : $firstStackInputType \in TYPES$
grd4 : $firstStackInputType = stackTypes(stackSize - 1)$
grd6 : $\texttt{form} = 59$

then

act1 : $stackSize := stackSize - 1$
act2 : $stackTypes := \{stackSize - 1\} \lhd\!\!\!- stackTypes$

end

Event *pop2_R08* $\widehat{=}$
extends *pop2_R07*

any

firstStackInputType
secondStackInputType
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize - 1 > 0$
grd3 : $firstStackInputType \in TYPES$
grd4 : $secondStackInputType \in TYPES$
grd5 : $firstStackInputType = stackTypes(stackSize - 1)$
grd6 : $secondStackInputType = stackTypes(stackSize - 2)$
grd10 : $\texttt{form} = 60$

then

act1 : $stackSize := stackSize - 2$
act2 : $stackTypes := \{stackSize - 1, stackSize - 2\} \lhd stackTypes$

end

Event *return_R08* $\widehat{=}$
extends *return_R07*

any

$\texttt{form}$

where

grd1 : $programRunning = TRUE$
grd2 : $\texttt{form} = 122$

then

act1 : $programRunning := FALSE$

end

Event *s2b_R08* $\widehat{=}$
extends *s2b_R07*

any

firstStackInputType
firstStackOutputType
$\texttt{form}$

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize > 0$
grd3 : $firstStackInputType \in TYPES$
grd4 : $firstStackOutputType \in TYPES$
grd5 : $firstStackInputType = stackTypes(stackSize - 1)$
grd6_t : $firstStackInputType = shart$
grd7 : $firstStackOutputType = byte$
grd8 : $\texttt{form} = 91$

then

act1 : $stackTypes := stackTypes \lhd\!\!\!- \{stackSize - 1 \mapsto firstStackOutputType\}$

end

Event *sadd_R08* $\widehat{=}$
extends *sadd_R07*

any

firstStackInputType
secondStackInputType
firstStackOutputType
`form`

where

`grd1` : $programRunning = TRUE$
`grd2_t` : $stackSize > 0$
`grd3_t` : $stackSize > 1$
`grd4` : $firstStackInputType \in TYPES$
`grd5` : $secondStackInputType \in TYPES$
`grd6` : $firstStackOutputType \in TYPES$
`grd7` : $firstStackInputType = stackTypes(stackSize - 1)$
`grd8` : $secondStackInputType = stackTypes(stackSize - 2)$
`grd9_t` : $firstStackInputType = shart$
`grd10_t` : $secondStackInputType = shart$
`grd11` : $firstStackOutputType = shart$
`grd12` : `form` $= 65$

then

`act1` : $stackSize := stackSize - 1$
`act2` : $stackTypes := (\{stackSize-1, stackSize-2\} \lhd stackTypes) \cup \{stackSize-2 \mapsto firstStackOutputType\}$

end

Event *saload_R08* $\widehat{=}$
extends *saload_R07*

any

firstStackInputType
secondStackInputType
firstStackOutputType
`form`

where

`grd1` : $programRunning = TRUE$
`grd2_t` : $stackSize > 0$
`grd3_t` : $stackSize > 1$
`grd4` : $firstStackInputType \in TYPES$
`grd5` : $secondStackInputType \in TYPES$
`grd6` : $firstStackOutputType \in TYPES$
`grd7` : $firstStackInputType = stackTypes(stackSize - 1)$

grd8 : $secondStackInputType = stackTypes(stackSize - 2)$
grd9_t : $firstStackInputType = shart$
grd10_t : $secondStackInputType = shartArray$
grd11 : $firstStackOutputType = shart$
grd12 : $form = 38$

then

act1 : $stackSize := stackSize - 1$
act2 : $stackTypes := (\{stackSize-1, stackSize-2\} \lhd stackTypes) \cup \{stackSize-2 \mapsto firstStackOutputType\}$

end

Event *sand_R08* $\widehat{=}$
extends *sand_R07*

any

$firstStackInputType$
$secondStackInputType$
$firstStackOutputType$
$form$

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize > 0$
grd3_t : $stackSize > 1$
grd4 : $firstStackInputType \in TYPES$
grd5 : $secondStackInputType \in TYPES$
grd6 : $firstStackOutputType \in TYPES$
grd7 : $firstStackInputType = stackTypes(stackSize - 1)$
grd8 : $secondStackInputType = stackTypes(stackSize - 2)$
grd9_t : $firstStackInputType = shart$
grd10_t : $secondStackInputType = shart$
grd11 : $firstStackOutputType = shart$
grd12 : $form = 83$

then

act1 : $stackSize := stackSize - 1$
act2 : $stackTypes := (\{stackSize-1, stackSize-2\} \lhd stackTypes) \cup \{stackSize-2 \mapsto firstStackOutputType\}$

end

Event *sastore_R08* $\widehat{=}$
extends *sastore_R07*

any

$firstStackInputType$
$secondStackInputType$

thirdStackInputType
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize - 2 > 0$
grd3_t : $stackSize > 2$
grd4 : $firstStackInputType \in TYPES$
grd5 : $secondStackInputType \in TYPES$
grd6 : $thirdStackInputType \in TYPES$
grd7 : $firstStackInputType = stackTypes(stackSize - 1)$
grd8 : $secondStackInputType = stackTypes(stackSize - 2)$
grd9 : $thirdStackInputType = stackTypes(stackSize - 3)$
grd10_t : $firstStackInputType = shart$
grd11_t : $secondStackInputType = shart$
grd12_t : $thirdStackInputType = shartArray$
grd12 : form $= 57$

then

act1 : $stackSize := stackSize - 3$
act2 : $stackTypes := \{stackSize - 1, stackSize - 2, stackSize - 3\} \lhd stackTypes$

end

Event *sconst_m1_R08* $\widehat{=}$
extends *sconst_m1_R07*

any

firstStackOutputType
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize < MaxStackSize$
grd3 : $firstStackOutputType \in TYPES$
grd4 : $firstStackOutputType = shart$
grd5 : form $= 2$

then

act1 : $stackSize := stackSize + 1$
act2 : $stackTypes := stackTypes \cup \{stackSize \mapsto firstStackOutputType\}$

end

Event *sconst_0_R08* $\widehat{=}$
extends *sconst_0_R07*

any

firstStackOutputType
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize < MaxStackSize$
grd3 : $firstStackOutputType \in TYPES$
grd4 : $firstStackOutputType = shart$
grd5 : $\texttt{form} = 3$

then

act1 : $stackSize := stackSize + 1$
act2 : $stackTypes := stackTypes \cup \{stackSize \mapsto firstStackOutputType\}$

end

Event *sconst_1_R08* $\widehat{=}$
extends *sconst_1_R07*

any

firstStackOutputType
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize < MaxStackSize$
grd3 : $firstStackOutputType \in TYPES$
grd4 : $firstStackOutputType = shart$
grd5 : $\texttt{form} = 4$

then

act1 : $stackSize := stackSize + 1$
act2 : $stackTypes := stackTypes \cup \{stackSize \mapsto firstStackOutputType\}$

end

Event *sconst_2_R08* $\widehat{=}$
extends *sconst_2_R07*

any

firstStackOutputType
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize < MaxStackSize$
grd3 : $firstStackOutputType \in TYPES$
grd4 : $firstStackOutputType = shart$
grd5 : $\texttt{form} = 5$

then

act1 : $stackSize := stackSize + 1$
act2 : $stackTypes := stackTypes \cup \{stackSize \mapsto firstStackOutputType\}$

end

Event *sconst_3_R08* $\widehat{=}$
extends *sconst_3_R07*

any

firstStackOutputType

`form`

where

`grd1` : $programRunning = TRUE$
`grd2_t` : $stackSize < MaxStackSize$
`grd3` : $firstStackOutputType \in TYPES$
`grd4` : $firstStackOutputType = shart$
`grd5` : $\texttt{form} = 6$

then

`act1` : $stackSize := stackSize + 1$
`act2` : $stackTypes := stackTypes \cup \{stackSize \mapsto firstStackOutputType\}$

end

Event *sconst_4_R08* $\widehat{=}$
extends *sconst_4_R07*

any

firstStackOutputType

`form`

where

`grd1` : $programRunning = TRUE$
`grd2_t` : $stackSize < MaxStackSize$
`grd3` : $firstStackOutputType \in TYPES$
`grd4` : $firstStackOutputType = shart$
`grd5` : $\texttt{form} = 7$

then

`act1` : $stackSize := stackSize + 1$
`act2` : $stackTypes := stackTypes \cup \{stackSize \mapsto firstStackOutputType\}$

end

Event *sconst_5_R08* $\widehat{=}$
extends *sconst_5_R07*

any

firstStackOutputType

`form`

where

`grd1` : $programRunning = TRUE$
`grd2_t` : $stackSize < MaxStackSize$

grd3 : $firstStackOutputType \in TYPES$
grd4 : $firstStackOutputType = shart$
grd5 : $\texttt{form} = 8$

then

act1 : $stackSize := stackSize + 1$
act2 : $stackTypes := stackTypes \cup \{stackSize \mapsto firstStackOutputType\}$

end

Event *sdiv_R08* $\widehat{=}$
extends *sdiv_R07*

any

firstStackInputType
secondStackInputType
firstStackOutputType
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize > 0$
grd3_t : $stackSize > 1$
grd4 : $firstStackInputType \in TYPES$
grd5 : $secondStackInputType \in TYPES$
grd6 : $firstStackOutputType \in TYPES$
grd7 : $firstStackInputType = stackTypes(stackSize - 1)$
grd8 : $secondStackInputType = stackTypes(stackSize - 2)$
grd9_t : $firstStackInputType = shart$
grd10_t : $secondStackInputType = shart$
grd11 : $firstStackOutputType = shart$
grd12 : $\texttt{form} = 71$

then

act1 : $stackSize := stackSize - 1$
act2 : $stackTypes := (\{stackSize-1, stackSize-2\} \lhd stackTypes) \cup \{stackSize-2 \mapsto firstStackOutputType\}$

end

Event *sinc_R08* $\widehat{=}$
extends *sinc_R07*

any

prm_index
form
prm_const

where

grd1 : $programRunning = TRUE$

grd2 : $prm\_index \in dom(localVariablesTypes)$
grd3_t : $prm\_index \leq MaxLocalVariablesIndex$
grd4_t : $localVariablesTypes(prm\_index) = shart$
grd3 : form = 89
grd4 : prm_const ∈ u1

then

skip

end

Event *sinc_w_R08* $\widehat{=}$
extends *sinc_w_R07*

any

prm_index
form
prm_byte1
prm_byte2

where

grd1 : $programRunning = TRUE$
grd2 : $prm\_index \in dom(localVariablesTypes)$
grd3_t : $prm\_index \leq MaxLocalVariablesIndex$
grd4_t : $localVariablesTypes(prm\_index) = shart$
grd3 : form = 150
grd4 : prm_byte1 ∈ u1
grd5 : prm_byte2 ∈ u1

then

skip

end

Event *sload_R08* $\widehat{=}$
extends *sload_R07*

any

firstStackOutputType
prm_index
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize < MaxStackSize$
grd3 : $firstStackOutputType \in TYPES$
grd4 : $prm\_index \in dom(localVariablesTypes)$
grd5_t : $prm\_index \leq MaxLocalVariablesIndex$
grd6 : $firstStackOutputType = localVariablesTypes(prm\_index)$
grd6_t : $firstStackOutputType = shart$

grd7 : form = 22

then

act1 : $stackSize := stackSize + 1$

act2 : $stackTypes := stackTypes \cup \{stackSize \mapsto firstStackOutputType\}$

end

Event *sload_0_R08* $\widehat{=}$
extends *sload_0_R07*

any

$firstStackOutputType$

prm_index

form

where

grd1 : $programRunning = TRUE$

grd2_t : $stackSize < MaxStackSize$

grd3 : $firstStackOutputType \in TYPES$

grd4 : $prm\_index \in dom(localVariablesTypes)$

grd5_t : $prm\_index \leq MaxLocalVariablesIndex$

grd6 : $firstStackOutputType = localVariablesTypes(prm\_index)$

grd6_t : $firstStackOutputType = shart$

grd7 : $prm\_index = 0$

grd8 : form = 28

then

act1 : $stackSize := stackSize + 1$

act2 : $stackTypes := stackTypes \cup \{stackSize \mapsto firstStackOutputType\}$

end

Event *sload_1_R08* $\widehat{=}$
extends *sload_1_R07*

any

$firstStackOutputType$

prm_index

form

where

grd1 : $programRunning = TRUE$

grd2_t : $stackSize < MaxStackSize$

grd3 : $firstStackOutputType \in TYPES$

grd4 : $prm\_index \in dom(localVariablesTypes)$

grd5_t : $prm\_index \leq MaxLocalVariablesIndex$

grd6 : $firstStackOutputType = localVariablesTypes(prm\_index)$

grd6_t : $firstStackOutputType = shart$

grd7 : $prm\_index = 1$

grd8 : form = 29

then

act1 : $stackSize := stackSize + 1$

act2 : $stackTypes := stackTypes \cup \{stackSize \mapsto firstStackOutputType\}$

end

Event *sload_2_R08* $\widehat{=}$
extends *sload_2_R07*

any

firstStackOutputType

prm_index

form

where

grd1 : $programRunning = TRUE$

grd2_t : $stackSize < MaxStackSize$

grd3 : $firstStackOutputType \in TYPES$

grd4 : $prm\_index \in dom(localVariablesTypes)$

grd5_t : $prm\_index \leq MaxLocalVariablesIndex$

grd6 : $firstStackOutputType = localVariablesTypes(prm\_index)$

grd6_t : $firstStackOutputType = shart$

grd7 : $prm\_index = 2$

grd8 : form = 30

then

act1 : $stackSize := stackSize + 1$

act2 : $stackTypes := stackTypes \cup \{stackSize \mapsto firstStackOutputType\}$

end

Event *sload_3_R08* $\widehat{=}$
extends *sload_3_R07*

any

firstStackOutputType

prm_index

form

where

grd1 : $programRunning = TRUE$

grd2_t : $stackSize < MaxStackSize$

grd3 : $firstStackOutputType \in TYPES$

grd4 : $prm\_index \in dom(localVariablesTypes)$

grd5_t : $prm\_index \leq MaxLocalVariablesIndex$

grd6 : $firstStackOutputType = localVariablesTypes(prm\_index)$

grd6_t : $firstStackOutputType = shart$

grd7 : $prm\_index = 3$

grd8 : form = 31

then

act1 : $stackSize := stackSize + 1$

act2 : $stackTypes := stackTypes \cup \{stackSize \mapsto firstStackOutputType\}$

end

Event *smul_R08* $\widehat{=}$
extends *smul_R07*

any

firstStackInputType
secondStackInputType
firstStackOutputType
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize > 0$
grd3_t : $stackSize > 1$
grd4 : $firstStackInputType \in TYPES$
grd5 : $secondStackInputType \in TYPES$
grd6 : $firstStackOutputType \in TYPES$
grd7 : $firstStackInputType = stackTypes(stackSize - 1)$
grd8 : $secondStackInputType = stackTypes(stackSize - 2)$
grd9_t : $firstStackInputType = shart$
grd10_t : $secondStackInputType = shart$
grd11 : $firstStackOutputType = shart$
grd12 : form = 69

then

act1 : $stackSize := stackSize - 1$
act2 : $stackTypes := (\{stackSize-1, stackSize-2\} \lhd stackTypes) \cup \{stackSize-2 \mapsto firstStackOutputType\}$

end

Event *sneg_R08* $\widehat{=}$
extends *sneg_R07*

any

firstStackInputType
firstStackOutputType
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize > 0$
grd3 : $firstStackInputType \in TYPES$

grd4 : $firstStackOutputType \in TYPES$
grd5 : $firstStackInputType = stackTypes(stackSize - 1)$
grd6_t : $firstStackInputType = shart$
grd7 : $firstStackOutputType = shart$
grd8 : $\texttt{form} = 75$

then

act1 : $stackTypes := stackTypes \lhd\!\!\!- \{stackSize - 1 \mapsto firstStackOutputType\}$

end

Event *sor_R08* $\widehat{=}$
extends *sor_R07*

any

firstStackInputType
secondStackInputType
firstStackOutputType
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize > 0$
grd3_t : $stackSize > 1$
grd4 : $firstStackInputType \in TYPES$
grd5 : $secondStackInputType \in TYPES$
grd6 : $firstStackOutputType \in TYPES$
grd7 : $firstStackInputType = stackTypes(stackSize - 1)$
grd8 : $secondStackInputType = stackTypes(stackSize - 2)$
grd9_t : $firstStackInputType = shart$
grd10_t : $secondStackInputType = shart$
grd11 : $firstStackOutputType = shart$
grd12 : $\texttt{form} = 85$

then

act1 : $stackSize := stackSize - 1$
act2 : $stackTypes := (\{stackSize-1, stackSize-2\} \lhd\!\!\!- stackTypes) \cup \{stackSize-2 \mapsto firstStackOutputType\}$

end

Event *srem_R08* $\widehat{=}$
extends *srem_R07*

any

firstStackInputType
secondStackInputType
firstStackOutputType
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize > 0$
grd3_t : $stackSize > 1$
grd4 : $firstStackInputType \in TYPES$
grd5 : $secondStackInputType \in TYPES$
grd6 : $firstStackOutputType \in TYPES$
grd7 : $firstStackInputType = stackTypes(stackSize - 1)$
grd8 : $secondStackInputType = stackTypes(stackSize - 2)$
grd9_t : $firstStackInputType = shart$
grd10_t : $secondStackInputType = shart$
grd11 : $firstStackOutputType = shart$
grd12 : $\texttt{form} = 73$

then

act1 : $stackSize := stackSize - 1$
act2 : $stackTypes := (\{stackSize-1, stackSize-2\} \lhd stackTypes) \cup \{stackSize-2 \mapsto firstStackOutputType\}$

end

Event *sreturn_R08* $\widehat{=}$
extends *sreturn_R07*

any

firstStackInputType
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize > 0$
grd3 : $firstStackInputType \in TYPES$
grd4 : $firstStackInputType = stackTypes(stackSize - 1)$
grd5_t : $firstStackInputType = shart$
grd6 : $\texttt{form} = 120$

then

act1 : $programRunning := FALSE$

end

Event *sshl_R08* $\widehat{=}$
extends *sshl_R07*

any

firstStackInputType
secondStackInputType
firstStackOutputType
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize > 0$
grd3_t : $stackSize > 1$
grd4 : $firstStackInputType \in TYPES$
grd5 : $secondStackInputType \in TYPES$
grd6 : $firstStackOutputType \in TYPES$
grd7 : $firstStackInputType = stackTypes(stackSize - 1)$
grd8 : $secondStackInputType = stackTypes(stackSize - 2)$
grd9_t : $firstStackInputType = shart$
grd10_t : $secondStackInputType = shart$
grd11 : $firstStackOutputType = shart$
grd12 : $\texttt{form} = 77$

then

act1 : $stackSize := stackSize - 1$
act2 : $stackTypes := (\{stackSize-1, stackSize-2\} \lhd stackTypes) \cup \{stackSize-2 \mapsto firstStackOutputType\}$

end

Event *sshr_R08* $\widehat{=}$
extends *sshr_R07*

any

firstStackInputType
secondStackInputType
firstStackOutputType
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize > 0$
grd3_t : $stackSize > 1$
grd4 : $firstStackInputType \in TYPES$
grd5 : $secondStackInputType \in TYPES$
grd6 : $firstStackOutputType \in TYPES$
grd7 : $firstStackInputType = stackTypes(stackSize - 1)$
grd8 : $secondStackInputType = stackTypes(stackSize - 2)$
grd9_t : $firstStackInputType = shart$
grd10_t : $secondStackInputType = shart$
grd11 : $firstStackOutputType = shart$
grd12 : $\texttt{form} = 79$

then

act1 : $stackSize := stackSize - 1$

act2 : $stackTypes := (\{stackSize-1, stackSize-2\} \lhd stackTypes) \cup \{stackSize-2 \mapsto firstStackOutputType\}$

end

Event *sspush_R08* $\widehat{=}$
extends *sspush_R07*

any

firstStackOutputType
prm_byte1
prm_byte2
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize < MaxStackSize$
grd3 : $firstStackOutputType \in TYPES$
grd4 : $firstStackOutputType = shart$
grd5 : $prm\_byte1 \in u1$
grd6 : $prm\_byte2 \in u1$
grd7 : $form = 17$

then

act1 : $stackSize := stackSize + 1$
act2 : $stackTypes := stackTypes \cup \{stackSize \mapsto firstStackOutputType\}$

end

Event *sstore_R08* $\widehat{=}$
extends *sstore_R07*

any

firstStackInputType
prm_index
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize > 0$
grd3 : $firstStackInputType \in TYPES$
grd4 : $firstStackInputType = stackTypes(stackSize - 1)$
grd5 : $prm\_index \in dom(localVariablesTypes)$
grd6_t : $prm\_index \leq MaxLocalVariablesIndex$
grd7_t : $firstStackInputType = shart$
grd7 : $form = 41$

then

act1 : $stackSize := stackSize - 1$
act2 : $stackTypes := \{stackSize - 1\} \lhd stackTypes$

act3 : $localVariablesTypes(prm\_index) := firstStackInputType$

end

Event $sstore\_0\_R08 \widehat{=}$
extends $sstore\_0\_R07$

any

$firstStackInputType$
prm_index
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize > 0$
grd3 : $firstStackInputType \in TYPES$
grd4 : $firstStackInputType = stackTypes(stackSize - 1)$
grd5 : $prm\_index \in dom(localVariablesTypes)$
grd6_t : $prm\_index \leq MaxLocalVariablesIndex$
grd7_t : $firstStackInputType = shart$
grd8 : $prm\_index = 0$
grd9 : form = 47

then

act1 : $stackSize := stackSize - 1$
act2 : $stackTypes := \{stackSize - 1\} \lhd stackTypes$
act3 : $localVariablesTypes(prm\_index) := firstStackInputType$

end

Event $sstore\_1\_R08 \widehat{=}$
extends $sstore\_1\_R07$

any

$firstStackInputType$
prm_index
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize > 0$
grd3 : $firstStackInputType \in TYPES$
grd4 : $firstStackInputType = stackTypes(stackSize - 1)$
grd5 : $prm\_index \in dom(localVariablesTypes)$
grd6_t : $prm\_index \leq MaxLocalVariablesIndex$
grd7_t : $firstStackInputType = shart$
grd8 : $prm\_index = 1$
grd9 : form = 48

then

act1 : $stackSize := stackSize - 1$
act2 : $stackTypes := \{stackSize - 1\} \lhd stackTypes$
act3 : $localVariablesTypes(prm\_index) := firstStackInputType$

end

Event *sstore_2_R08* $\widehat{=}$
extends *sstore_2_R07*

any

firstStackInputType
prm_index
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize > 0$
grd3 : $firstStackInputType \in TYPES$
grd4 : $firstStackInputType = stackTypes(stackSize - 1)$
grd5 : $prm\_index \in dom(localVariablesTypes)$
grd6_t : $prm\_index \leq MaxLocalVariablesIndex$
grd7_t : $firstStackInputType = shart$
grd8 : $prm\_index = 2$
grd9 : $form = 49$

then

act1 : $stackSize := stackSize - 1$
act2 : $stackTypes := \{stackSize - 1\} \lhd stackTypes$
act3 : $localVariablesTypes(prm\_index) := firstStackInputType$

end

Event *sstore_3_R08* $\widehat{=}$
extends *sstore_3_R07*

any

firstStackInputType
prm_index
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize > 0$
grd3 : $firstStackInputType \in TYPES$
grd4 : $firstStackInputType = stackTypes(stackSize - 1)$
grd5 : $prm\_index \in dom(localVariablesTypes)$
grd6_t : $prm\_index \leq MaxLocalVariablesIndex$
grd7_t : $firstStackInputType = shart$
grd8 : $prm\_index = 3$

grd9 : $\texttt{form} = 50$

then

act1 : $stackSize := stackSize - 1$

act2 : $stackTypes := \{stackSize - 1\} \lhd stackTypes$

act3 : $localVariablesTypes(prm\_index) := firstStackInputType$

end

Event *ssub_R08* $\widehat{=}$
extends *ssub_R07*

any

firstStackInputType

secondStackInputType

firstStackOutputType

form

where

grd1 : $programRunning = TRUE$

grd2_t : $stackSize > 0$

grd3_t : $stackSize > 1$

grd4 : $firstStackInputType \in TYPES$

grd5 : $secondStackInputType \in TYPES$

grd6 : $firstStackOutputType \in TYPES$

grd7 : $firstStackInputType = stackTypes(stackSize - 1)$

grd8 : $secondStackInputType = stackTypes(stackSize - 2)$

grd9_t : $firstStackInputType = shart$

grd10_t : $secondStackInputType = shart$

grd11 : $firstStackOutputType = shart$

grd12 : $\texttt{form} = 67$

then

act1 : $stackSize := stackSize - 1$

act2 : $stackTypes := (\{stackSize-1, stackSize-2\} \lhd stackTypes) \cup \{stackSize-2 \mapsto firstStackOutputType\}$

end

Event *sushr_R08* $\widehat{=}$
extends *sushr_R07*

any

firstStackInputType

secondStackInputType

firstStackOutputType

form

where

grd1 : $programRunning = TRUE$

grd2_t : $stackSize > 0$
grd3_t : $stackSize > 1$
grd4 : $firstStackInputType \in TYPES$
grd5 : $secondStackInputType \in TYPES$
grd6 : $firstStackOutputType \in TYPES$
grd7 : $firstStackInputType = stackTypes(stackSize - 1)$
grd8 : $secondStackInputType = stackTypes(stackSize - 2)$
grd9_t : $firstStackInputType = shart$
grd10_t : $secondStackInputType = shart$
grd11 : $firstStackOutputType = shart$
grd12 : $form = 81$

then

act1 : $stackSize := stackSize - 1$
act2 : $stackTypes := (\{stackSize-1, stackSize-2\} \lhd stackTypes) \cup \{stackSize-2 \mapsto firstStackOutputType\}$

end

Event *swap_11_R08* $\widehat{=}$
extends *swap_11_R07*

any

firstStackInputType
secondStackInputType
firstStackOutputType
secondStackOutputType
prm_m
prm_n
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize > 1$
grd3 : $firstStackInputType \in TYPES$
grd4 : $secondStackInputType \in TYPES$
grd5 : $firstStackOutputType \in TYPES$
grd6 : $secondStackOutputType \in TYPES$
grd7 : $firstStackInputType = stackTypes(stackSize - 1)$
grd8 : $secondStackInputType = stackTypes(stackSize - 2)$
grd12 : $prm\_m \in 1 \mathinner{..} 2$
grd13 : $prm\_n \in 1 \mathinner{..} 2$
grd14 : $prm\_m = 1 \land prm\_n = 1$
grd15 : $firstStackOutputType = firstStackInputType$
grd16 : $secondStackOutputType = secondStackInputType$
grd19 : $form = 64$

then

act1 : $stackTypes := stackTypes \lhd \{stackSize - 2 \mapsto firstStackOutputType, stackSize - 1 \mapsto secondStackOutputType\}$

end

Event *swap_12_R08* $\widehat{=}$
extends *swap_12_R07*

any

$firstStackInputType$
$secondStackInputType$
$thirdStackInputType$
$firstStackOutputType$
$secondStackOutputType$
$thirdStackOutputType$
prm_m
prm_n
`form`

where

grd1 : $programRunning = TRUE$
grd3_t : $stackSize > 2$
grd4 : $firstStackInputType \in TYPES$
grd5 : $secondStackInputType \in TYPES$
grd6 : $thirdStackInputType \in TYPES$
grd7 : $firstStackOutputType \in TYPES$
grd8 : $secondStackOutputType \in TYPES$
grd9 : $thirdStackOutputType \in TYPES$
grd10 : $firstStackInputType = stackTypes(stackSize - 1)$
grd11 : $secondStackInputType = stackTypes(stackSize - 2)$
grd12 : $thirdStackInputType = stackTypes(stackSize - 3)$
grd13 : $prm\_m \in 1 \mathinner{..} 2$
grd14 : $prm\_n \in 1 \mathinner{..} 2$
grd15 : $prm\_m = 1 \land prm\_n = 2$
grd16 : $firstStackOutputType = secondStackInputType$
grd17 : $secondStackOutputType = firstStackInputType$
grd18 : $thirdStackOutputType = thirdStackInputType$
grd21 : `form` $= 64$

then

act2 : $stackTypes := (\{stackSize - 1, stackSize - 2, stackSize - 3\} \lhd stackTypes) \cup \{stackSize - 3 \mapsto firstStackOutputType, stackSize - 2 \mapsto secondStackOutputType, stackSize - 1 \mapsto thirdStackOutputType\}$

end

Event *swap_21_R08* $\widehat{=}$

extends *swap_21_R07*

any

firstStackInputType
secondStackInputType
thirdStackInputType
firstStackOutputType
secondStackOutputType
thirdStackOutputType
prm_m
prm_n
`form`

where

grd1 : $programRunning = TRUE$
grd3_t : $stackSize > 2$
grd4 : $firstStackInputType \in TYPES$
grd5 : $secondStackInputType \in TYPES$
grd6 : $thirdStackInputType \in TYPES$
grd7 : $firstStackOutputType \in TYPES$
grd8 : $secondStackOutputType \in TYPES$
grd9 : $thirdStackOutputType \in TYPES$
grd10 : $firstStackInputType = stackTypes(stackSize - 1)$
grd11 : $secondStackInputType = stackTypes(stackSize - 2)$
grd12 : $thirdStackInputType = stackTypes(stackSize - 3)$
grd13 : $prm\_m \in 1 \mathinner{..} 2$
grd14 : $prm\_n \in 1 \mathinner{..} 2$
grd15 : $prm\_m = 2 \land prm\_n = 1$
grd16 : $firstStackOutputType = firstStackInputType$
grd17 : $secondStackOutputType = thirdStackInputType$
grd18 : $thirdStackOutputType = secondStackInputType$
grd21 : `form` $= 64$

then

act2 : $stackTypes := (\{stackSize - 1, stackSize - 2, stackSize - 3\} \lhd stackTypes) \cup \{stackSize - 3 \mapsto firstStackOutputType, stackSize - 2 \mapsto secondStackOutputType, stackSize - 1 \mapsto thirdStackOutputType\}$

end

Event *swap_22_R08* $\widehat{=}$
extends *swap_22_R07*

any

firstStackInputType
secondStackInputType
thirdStackInputType

$fourthStackInputType$
$firstStackOutputType$
$secondStackOutputType$
$thirdStackOutputType$
$fourthStackOutputType$
prm_m
prm_n
$\texttt{form}$

where

grd1 : $programRunning = TRUE$
grd3_t : $stackSize > 3$
grd4 : $firstStackInputType \in TYPES$
grd5 : $secondStackInputType \in TYPES$
grd6 : $thirdStackInputType \in TYPES$
grd7 : $fourthStackInputType \in TYPES$
grd8 : $firstStackOutputType \in TYPES$
grd9 : $secondStackOutputType \in TYPES$
grd10 : $thirdStackOutputType \in TYPES$
grd11 : $fourthStackOutputType \in TYPES$
grd12 : $firstStackInputType = stackTypes(stackSize - 1)$
grd13 : $secondStackInputType = stackTypes(stackSize - 2)$
grd14 : $thirdStackInputType = stackTypes(stackSize - 3)$
grd15 : $fourthStackInputType = stackTypes(stackSize - 4)$
grd16 : $prm\_m \in 1 \mathinner{..} 2$
grd17 : $prm\_n \in 1 \mathinner{..} 2$
grd18 : $prm\_m = 2 \land prm\_n = 2$
grd19 : $firstStackOutputType = secondStackInputType$
grd20 : $secondStackOutputType = firstStackInputType$
grd21 : $thirdStackOutputType = fourthStackInputType$
grd22 : $fourthStackOutputType = thirdStackInputType$
grd26 : $\texttt{form} = 64$

then

act2 : $stackTypes := (\{stackSize - 1, stackSize - 2, stackSize - 3, stackSize - 4\} \lhd stackTypes) \cup \{stackSize - 4 \mapsto firstStackOutputType, stackSize - 3 \mapsto secondStackOutputType, stackSize - 2 \mapsto thirdStackOutputType, stackSize - 1 \mapsto fourthStackOutputType\}$

end

Event $sxor\_R08 \mathrel{\widehat{=}}$
extends $sxor\_R07$

any

$firstStackInputType$

secondStackInputType
firstStackOutputType
form

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize > 0$
grd3_t : $stackSize > 1$
grd4 : $firstStackInputType \in TYPES$
grd5 : $secondStackInputType \in TYPES$
grd6 : $firstStackOutputType \in TYPES$
grd7 : $firstStackInputType = stackTypes(stackSize - 1)$
grd8 : $secondStackInputType = stackTypes(stackSize - 2)$
grd9_t : $firstStackInputType = shart$
grd10_t : $secondStackInputType = shart$
grd11 : $firstStackOutputType = shart$
grd12 : form = 87

then

act1 : $stackSize := stackSize - 1$
act2 : $stackTypes := (\{stackSize-1, stackSize-2\} \lhd stackTypes) \cup \{stackSize-2 \mapsto firstStackOutputType\}$

end

Event *createInitObject_R08* $\widehat{=}$
extends *createInitObject_R07*

any

firstStackOutputType
prm_localVariables_index
prm_constantPool_index

where

grd1 : $programRunning = TRUE$
grd2_t : $stackSize < MaxStackSize$
grd3 : $firstStackOutputType \in TYPES$
grd4 : $prm\_localVariables\_index = 0$
grd5 : $prm\_constantPool\_index \in dom(ConstantPool)$
grd6 : $firstStackOutputType = ConstantPool(prm\_constantPool\_index)$

then

act1 : $stackSize := stackSize + 1$
act2 : $stackTypes := stackTypes \cup \{stackSize \mapsto firstStackOutputType\}$

end

END

Bibliographie

[1] M. Leuschel and M. Butler, "ProB : A model checker for B," *FME 2003 : Formal Methods*, pp. 855–874, 2003. [Online]. Available : http://link.springer.com/chapter/10.1007/978-3-540-45236-2_46

[2] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert, "Fault attacks on RSA with CRT : Concrete results and practical countermeasures," in *Cryptographic hardware and embedded systems - CHES 2002*, vol. 2523, 2003, pp. 260–275.

[3] G. Piret, "A differential fault attack technique against SPN structures, with application to the AES and KHAZAD," *Hardware and Embedded Systems-CHES 2003*, pp. 77–88, 2003. [Online]. Available : http://www.springerlink.com/index/WQ7JX5HB6XGBU3X7.pdf

[4] G. Barbu, H. Thiebeauld, and V. Guerin, "Attacks on Java Card 3.0 combining fault and logical attacks," in *Smart Card Research and Advanced Application*, D. Gollmann, J.-L. Lanet, and J. Iguchi-Cartigny, Eds., vol. 6035. Springer Berlin Heidelberg, 2010, pp. 148–163. [Online]. Available : http://www.springerlink.com/index/W041831734511301.pdfhttp://link.springer.com/chapter/10.1007/978-3-642-12510-2_11

[5] M. Barreaud, G. Bouffard, N. Kamel, and J.-L. Lanet, "Fuzzing on the HTTP protocol implementation in mobile embedded web server," *Proceeding of C&ESAR*, p. 125, 2011.

[6] B. Mathieu, J. Iguchi-Cartigny, and J.-L. Lanet, "Analysis vulnerabilities in smart card web server," *SAR-SSI 2011 Network and Information Systems Security*, 2011.

[7] W. Mostowski and E. Poll, "Malicious Code on Java Card Smartcards : Attacks and Countermeasures," in *Smart Card Research and Advanced Applications*. Springer Berlin Heidelberg, 2008, vol. 5189, pp. 1–16. [Online]. Available : http://dx.doi.org/10.1007/978-3-540-85893-5_1

[8] E. Hubbers and E. Poll, "Transactions and non-atomic API methods in Java Card : specification ambiguity and strange implementation behaviours," *cs.ru.nl*, pp. 1–22. [Online]. Available : http://cs.ru.nl/E.Poll/papers/acna_new.pdf

[9] W. Mostowski and E. Poll, "Testing the Java Card Applet Firewall," Technical Report ICIS–R07029, Radboud University Nijmegen, Tech. Rep., 2007. [Online]. Available : https://vsm.cs.utwente.nl/~mostowskiwi/papers/firewall2007.pdf

[10] T. Razafindralambo, G. Bouffard, B. N. Thampi, and J.-L. Lanet, "A dynamic syntax interpretation for java based smart card to mitigate logical attacks," in *Communications in Computer and Information Science*, vol. 335 CCIS, 2012, pp. 185–194. [Online]. Available : http://link.springer.com/chapter/10.1007/978-3-642-34135-9_19

[11] J. Iguchi-Cartigny and J.-L. Lanet, "Developing a Trojan applets in a smart card," *Journal in Computer Virology*, vol. 6, no. 4, pp. 343–351, Sep. 2010. [Online]. Available : http://link.springer.com/10.1007/s11416-009-0135-3

[12] G. Bouffard, J. Iguchi-Cartigny, and J.-L. Lanet, "Combined software and hardware attacks on the Java Card control flow," in *Lecture Notes in Computer Science*, vol. 7079 LNCS, 2011, pp. 283–296. [Online]. Available : http://link.springer.com/chapter/10.1007/978-3-642-27257-8_18

[13] "Defensive Java Virtual Machine website : http://computationallogic.com/software/djvm/index.html."

[14] G. Bouffard, B. N. Thampi, and J. L. Lanet, "Detecting Laser Fault Injection for Smart Cards Using Security Automata," in *Security in Computing and Communications*, ser. Communications in Computer and Information Science, S. Thampi, P. Atrey, C.-I. Fan, and G. Perez, Eds. Springer Berlin Heidelberg, 2013, vol. 377 CCIS, pp. 18–29.

[15] J.-C. Laprie, *Guide de la sûreté de fonctionnement*. Cépaduès, 1995. [Online]. Available : http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Guide+de+la+S\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{u\global\mathchardef\accent@spacefactor\spacefactor}\accent94u\egroup\spacefactor\accent@spacefactorret\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{e\global\mathchardef\accent@spacefactor\spacefactor}\accent19e\egroup\spacefactor\accent@spacefactor+de+fonctionnement#0

[16] T. Lindholm and F. Yellin, *Java Virtual Machine Specification*, 2nd ed. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1999.

[17] S. Doyon, "On the security of Java : The Java bytecode verifier," Master's thesis, Université Laval, Québec City, Canada, 1999.

[18] P. H. Hartel and L. Moreau, "Formalizing the safety of Java, the Java Virtual Machine and Java Card," *ACM Computing Surveys*, vol. 33, no. 4, pp. 517–558, 2001. [Online]. Available : http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Formalising+the+Safety+of+Java+,+the+Java+Virtual+Machine+and+Java+Card#1

[19] S. N. Freund and J. C. Mitchell, "A formal framework for the java bytecode language and verifier," in *In OOPSLA Proceeedings*. ACM press, 1999, pp. 147–166.

[20] A. Goldberg, "A specification of java loading and bytecode verification," in *Proceedings of the 5th ACM conference on Computer and communications security*, ser. CCS '98. New York, NY, USA : ACM, 1998, pp. 49–58. [Online]. Available : http://doi.acm.org/10.1145/288090.288104

[21] T. Nipkow and D. von Oheimb, "Javalight is type-safe—definitely," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '98. New York, NY, USA : ACM, 1998, pp. 161–170. [Online]. Available : http://doi.acm.org/10.1145/268946.268960

[22] J.-L. Lanet, "Cartes à puce et méthodes formelles, une lente intégration..." in *Technique et Sciences Informatique*, 2001, pp. 959–964.

[23] L. Casset, "Construction correcte de logiciels pour carte à puce," *These de doctorat, Université d'Aix-Marseille II*, 2002. [Online]. Available : http://atelierb.eu/pdf/theseLudovic.pdf

[24] ——, "Development of an embedded verifier for Java Card byte code using formal methods," in *FME 2002 : Formal Methods—Getting IT Right*. Springer Berlin Heidelberg, 2002, pp. 290–309. [Online]. Available : http://link.springer.com/chapter/10.1007/3-540-45614-7_17http://www.springerlink.com/index/4u93fl7kajwct265.pdf

[25] L. Casset and J.-L. Lanet, "How to formally specify the java bytecode semantics using the b method," in *Proceedings of the Workshop on Object-Oriented Technology*. London, UK : Springer-Verlag, 1999, pp. 104–105.

[26] M. C. Reynolds, "Modeling the Java Bytecode Verifier," *Science of Computer Programming*, vol. 78, no. 3, pp. 327–342, 2013. [Online]. Available : http://www.sciencedirect.com/science/article/pii/S0167642311000943

[27] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011. [Online]. Available : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5487526$\delimiter"026E30F$nhttp://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5487526

[28] K. N. King and A. J. Offutt, "A fortran language system for mutation-based software testing," *Software : Practice and Experience*, vol. 21, no. 7, pp. 685–718, 1991. [Online]. Available : http://dx.doi.org/10.1002/spe.4380210704

[29] H. Agrawal, R. A. Demillo, B. Hathaway, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. H. Spafford, "Design of Mutant Operators for the C Programming Language," Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University, Tech. Rep., 1989. [Online]. Available : http://web.soccerlab.polymtl.ca/log6305/protected/papers/CMutation.pdf

[30] A. J. Offutt, J. M. Voas, and J. Payne, "Mutation Operators for Ada," Reliable Software Technologies Corps., Tech. Rep., 1996. [Online]. Available : http://www.cs.gmu.edu/~offutt/rsrch/papers/ada-ops.pdf

[31] S. Kim, J. Clark, and J. McDermid, "The Rigorous Generation of Java Mutation Operators Using HAZOP," University of York, Tech. Rep., 1999. [Online]. Available : http://citeseerx.ist.psu.edu/viewdoc/download?rep=rep1&type=pdf&doi=10.1.1.145.7736

[32] A. Gal, C. W. Probst, and M. Franz, "A Denial of Service Attack on the Java Bytecode Verifier," University of California, Tech. Rep., 2003.

[33] K. Sohr, "Die Sicherheitsaspekte von mobilem Code," Ph.D. dissertation, University of Marburg, 2001.

[34] M. Shafique and Y. Labiche, "A systematic review of state-based test tools," *International Journal on Software Tools for Technology Transfer*, vol. 17, pp. 59 –76, 2015. [Online]. Available : http://dx.doi.org/10.1007/s10009-013-0291-0

[35] M. Utting and B. Legeard, *Practical Model Based Testing : A Tools Approach*. Kaufmann, Morgan, 2007. [Online]. Available : http://store.elsevier.com/product.jsp?isbn=9780123725011

[36] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting, "A subset of precise UML for model-based testing," in *Proceedings of the 3rd international workshop on Advances in model-based testing*, ser. A-MOST '07. London, United Kingdom : ACM, 2007, pp. 95–104. [Online]. Available : http://portal.acm.org/citation.cfm?doid=1291535.1291545http://doi.acm.org/10.1145/1291535.1291545

[37] S. Khurshid and D. Marinov, "TestEra : Specification-based testing of Java programs using SAT," *Automated Software Engineering*, vol. 11, no. 4, pp. 403–434, 2004.

[38] B. K. Aichernig and F. Lorber, "Model-based Mutation Testing with Timed Automata," *Technical Report IST-MBT-2013-02, TU Graz*, pp. 1–21, 2013.

[39] M. Mikucionis, K. G. Larsen, and B. Nielsen, "T-UPPAAL : Online model-based testing of real-time systems," in *Proceedings - 19th International Conference on Automated Software Engineering, ASE 2004*, 2004, pp. 396–397.

[40] Q. A. Malik, J. Lilius, and L. Laibinis, "Model-based testing using scenarios and event-B refinements," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, E. Butler, Michael and Jones, Cliff and Romanovsky, Alexander and Troubitsyna, Ed., vol. 5454 LNCS. Springer Berlin Heidelberg, 2009, pp. 177–195. [Online]. Available : http://dx.doi.org/10.1007/978-3-642-00867-2_9

[41] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software Testing Verification and Reliability*, vol. 22, no. 5, pp. 297–312, 2012. [Online]. Available : http://onlinelibrary.wiley.com/doi/10.1002/stvr.450/pdfhttp://dx.doi.org/10.1002/stvr.456

[42] M. Felderer, P. Zech, R. Breu, M. Büchler, and A. Pretschner, "Model-Based Security Testing : A Taxonomy and Systematic Classification," *Software Testing, Verification and Reliability*, 2015. [Online]. Available : http://onlinelibrary.wiley.com/doi/10.1002/stvr.450/pdf

[43] H. Martin, "Une méthodologie de génération automatique de suite de tests pour applets Java Card," Ph.D. dissertation, Université de Lille 1, 2001. [Online]. Available : http://cat.inist.fr/?aModele=afficheN&cpsidt=14195659

[44] P.-A. Masson, M.-L. Potet, J. Julliand, R. Tissot, G. Debois, B. Legeard, B. Chetali, F. Bouquet, E. Jaffuel, L. Van Aertrick, J. Andronick, and A. Haddad, "An Access Control Model Based Testing Approach for Smart Card Applications : Results of the {POSÉ} Project," *JIAS, Journal of Information Assurance and Security*, vol. 5, pp. 335–351, 2010. [Online]. Available : http://hal.archives-ouvertes.fr/hal-00943158/

[45] E. Jaffuel and B. Legeard, "LEIRIOS Test Generator : Automated Test Generation from B Models," in *B 2007 : Formal Specification and Development in B*. Springer Berlin Heidelberg, 2006, vol. 4355, pp. 277–280. [Online]. Available : http://link.springer.com/chapter/10.1007/11955757_29http://www.springerlink.com/content/026252518x28211h/

[46] F. Bouquet, B. Legeard, F. Peureux, and E. Torreborre, "Mastering Test Generation from Smart Card Software Formal Models," *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pp. 70–85, 2005. [Online]. Available : http://link.springer.com/chapter/10.1007/978-3-540-30569-9_4

[47] M. Satpathy, M. Butler, M. Leuschel, and S. Ramesh, "Automatic testing from formal specifications," in *Tests and Proofs*, 2007, pp. 95–113. [Online]. Available : http://link.springer.com/chapter/10.1007/978-3-540-73770-4_6

[48] M. Satpathy, M. Leuschel, and M. Butler, "ProTest : An Automatic Test Environment for B Specifications," *Electronic Notes in Theoretical Computer Science*, vol. 111, no. SPEC. ISS., pp. 113–136, Jan. 2005. [Online]. Available : http://linkinghub.elsevier.com/retrieve/pii/S1571066104052351http://www.sciencedirect.com/science/article/pii/S1571066104052351

[49] S. Wieczorek, V. Kozyura, A. Roth, M. Leuschel, J. Bendisposto, D. Plagge, and I. Schieferdecker, "Applying model checking to generate model-based integration tests

from choreography models," *Lecture Notes in Computer Science*, vol. 5826 LNCS, no. Mcm, pp. 179–194, 2009.

[50] T. a. Budd and A. S. Gopal, "Program testing by specification mutation," pp. 63–73, Jan. 1985. [Online]. Available : http://linkinghub.elsevier.com/retrieve/pii/0096055185900116

[51] B. K. Aichernig and P. A. P. Salas, "Test Case Generation by OCL Mutation and Constraint Solving," *Fifth International Conference on Quality Software (QSIC'05)*, vol. 2005, pp. 64–71, 2005. [Online]. Available : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1579121

[52] T. Srivatanakul, J. A. Clark, S. Stepney, and F. Polack, "Challenging formal specifications by mutation : a CSP security example," in *Asia-Pacific Software Engineering Conference*, 2003. [Online]. Available : http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.6.7991

[53] P. Black, V. Okun, and Y. Yesha, "Mutation of Model Checker Specifications for Test Generation and Evaluation," in *Mutation Testing for the New Century*, vol. 24. Springer Berlin Heidelberg, 2001, pp. 14–20. [Online]. Available : http://link.springer.com/chapter/10.1007/978-1-4757-5939-6_5http://dx.doi.org/10.1007/978-1-4757-5939-6_5

[54] A. Calvagna, A. Fornaia, and E. Tramontana, "Combinatorial interaction testing of a Java card static verifier," in *Proceedings - IEEE 7th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2014*, 2014, pp. 84–87.

[55] Sun Microsystems, "Virtual machine specification Java Card platform, may 2009, http://www.oracle.com."

[56] J.-R. Abrial, *Modeling in Event-B : system and software engineering*. Cambridge University Press, 2010.

[57] R. K. Ahuja, "Network flows," Ph.D. dissertation, Technische Hochschule Darmstadt, 1993.

[58] N. Ouerdi, M. Azizi, M. H. Ziane, A. Azizi, J.-L. Lanet, and A. Savary, "Security Vulnerabilities Tests Generation from SysML and Event-B Models for EMV Cards," *International Journal of Security and Its Applications*, vol. 8, no. 1, pp. 373–388, 2013. [Online]. Available : http://www.sersc.org/journals/IJSIA/vol8_no1_2014/35.pdf

[59] M. Lassale, "Génération de tests de vulnérabilité pour la structure des fichiers cap en java card," Master's thesis, Université de Sherbrooke, Département d'informatique, Sherbrooke, Québec, Canada, 2015.

[60] A. Savary, M. Frappier, M. Leuschel, and J. L. Lanet, "Model-based robustness testing in EVENT-B using mutation," in *Software Engineering and Formal Methods : 13th International Conference, SEFM 2015*, vol. 9276, 2015, pp. 132–147.

[61] A. Savary, M. Frappier, and J.-L. Lanet, "Detecting vulnerabilities in java-card bytecode verifiers using model-based testing," in *Integrated Formal Methods (IFM)*, vol. 7940 LNCS, 2013, pp. 223–237. [Online]. Available : http://link.springer.com/chapter/10.1007/978-3-642-38613-8{_}16

[62] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification Java SE 7 Edition*. Pearson Education, 2014.

[63] "TOM website : http://tom.loria.fr/wiki/index.php5/Main_Page."

[64] A. C. Noubissi, A. A.-k. Séré, J. Iguchi-cartigny, and J.-L. Lanet, "Cartes à puce : Attaques et contremesures." *MajecSTIC*, vol. 16, 2009.

[65] A. Savary, M. Frappier, and J.-L. Lanet, "Vtg – vulnerability test generator, a plug-in for rodin," 2012, workshop Deploy.

[66] A. Savary, "Piratons formellement des cartes à puce," 2012, colloqueTI.

[67] A. Savary, M. Frappier, and J.-L. Lanet, "Toolbox for penetration testing based on rodin and prob," 2014, workshop Rodin.

Termine la biblio du Chapitre 2

Détection de vulnérabilités appliquée à la vérification de code intermédiaire de Java Card

Résumé : La vérification de la résistance aux attaques des implémentations embarquées des vérifieurs de code intermédiaire Java Card est une tâche complexe. Les méthodes actuelles n'étant pas suffisamment efficaces, seule la génération de tests manuelle est possible. Pour automatiser ce processus, nous proposons une méthode appelée VTG (≪*Vulnerability Test Generation*≫, génération de tests de vulnérabilité). En se basant sur une représentation formelle des comportements fonctionnels du système sous test, un ensemble de tests d'intrusions est généré. Cette méthode s'inspire des techniques de mutation et de test à base de modèle. Dans un premier temps, le modèle est muté selon des règles que nous avons définies afin de représenter les potentielles attaques. Les tests sont ensuite extraits à partir des modèles mutants. Deux modèles Event-B ont été proposés. Le premier représente les contraintes structurelles des fichiers d'application Java Card. Le VTG permet en quelques secondes de générer des centaines de tests abstraits. Le second modèle est composé de 66 événements permettant de représenter 61 instructions Java Card. La mutation est effectuée en quelques secondes. L'extraction des tests permet de générer 223 tests en 45 min. Chaque test permet de vérifier une précondition ou une combinaison de préconditions d'une instruction. Cette méthode nous a permis de tester différents mécanismes d'implémentations de vérifieur de code intermédiaire Java Card. Bien que développée pour notre cas d'étude, la méthode proposée est générique et a été appliquée à d'autres cas d'études..

Mots clés : Sécurité, Java Card, Vérification de code intermédiaire, Test d'intrusion, Mutation de spécification, Test à base de modèle, Event-B, ProB.

Vulnerability detection for Java Card bytecode verifier

Abstract : Verification of the resistance of attacks against embedded implementations of the Java Card bytecode verifiers is a complex task. Current methods are not sufficient, only the generation of manual testing is possible. To automate this process, we propose a method called VTG (Vulnerability Test Generation). Based on a formal representation of the functional behavior of the system under test, a set of intrusion test is generated. This method is based on techniques of mutation and model-based testing. Initially, the model is transferred according to rules that we have defined to represent potential attacks. The tests are then extracted from the mutant models. Two Event-B models have been proposed. The first represents the structural constraints of the Java Card application files. The VTG allows in seconds to generate hundreds of abstract tests. The second model is composed of 66 events to represent 61 Java Card instructions. The mutation is effected in a few seconds. Extraction tests to generate 223 test 45 min. Each test checks a precondition or a combination of preconditions of a statement. This method allowed us to test different implementations of mechanisms through Java Card bytecode verifier. Although developed for our case study, the proposed method is generic and has been applied to other case studies.

Keywords : Security, Java Card, Bytecode Verifier, Penetration Testing, Specification Mutation, Model-Based Testing, Event-B, ProB.