GÉNÉRATION DE TESTS DE VULNÉRABILITÉ POUR LA STRUCTURE DES FICHIERS CAP EN JAVA CARD

par

Mathieu Lassale

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

FACULTÉ DES SCIENCES

UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, 2 mars 2016

Le 2 mars 2016

Le jury a accepté le mémoire de Monsieur Mathieu Lassale dans sa version finale.

Membres du jury

Professeur Marc Frappier
Directeur de recherche
Département d'informatique

Professeur Jean-Louis Lanet
Évaluateur externe au programme
Département d'informatique
Université de Limoges

Professeur Gabriel Girard
Président-rapporteur
Département d'informatique

Sommaire

Les cartes à puce Java comportent plusieurs mécanismes de sécurité, dont le vérifieur de code intermédiaire (≪*Java Card bytecode verifier*≫), qui est composé de deux parties, la vérification de structure et la vérification de type. Ce mémoire porte sur la génération de tests de vulnérabilité pour la vérification de structure. Il s'inspire des travaux sur la vérification de type de l'outil VTG (≪*Vulnerability Tests Generator*≫) développé par Aymerick Savary. Notre approche consiste à modéliser formellement la spécification de la structure des fichiers CAP avec le langage Event-B, en utilisant des contextes. Cette modélisation permet de donner une définition formelle d'un fichier CAP valide. Nous utilisons ensuite la mutation de spécification pour insérer des fautes dans cette définition dans le but de générer des fichiers CAP (≪*Converted APplet*≫) invalides. Nous utilisons ProB, un explorateur de modèles Event-B, pour générer des tests abstraits de fichiers CAP invalides. La spécification formelle étant d'une taille importante qui entraîne une forte explosion combinatoire (plus de 250 constantes, 450 axiomes, 100 contextes), nous guidons ProB dans sa recherche de modèles en utilisant des valeurs prédéterminées pour un sous-ensemble de symboles de la spécification. Ce mémoire propose un ensemble de patrons de spécification pour représenter les structures des fichiers CAP. Ces patrons limitent aussi l'explosion combinatoire, tout en facilitant la tâche de spécification. Notre spécification Event-B comprend toute la définition des structures des fichiers CAP et une partie des contraintes. Des tests abstraits sont générés pour une partie du modèle, à titre illustratif. Ces tests ont permis de mettre en lumière des imprécisions dans la spécification Java Card. Ces travaux ont permis d'étendre la méthode de génération de test de vulnérabilité aux contextes Event-B. De plus, le modèle proposé permet de tester, à l'aide du VTG, une partie plus importante de la vérification de structure

du vérifieur de code intermédiaire.

Mots-clés: Event-B ; vérifieur de byte code ; vérification de structure ; tests de vulnérabilité ; Java Card ; ProB.

Remerciements

Je souhaiterais tout d'abord remercier mes directeurs de recherches, le professeur Marc Frappier et le professeur Jean Louis Lanet, pour m'avoir permis d'effectuer cette maîtrise en cotutelle. Je remercie plus particulièrement Marc Frappier pour le temps qu'il m'a accordé afin de faciliter mon intégration à l'équipe du GRIL

Je voudrais aussi exprimer toute ma gratitude à Aymerick Savary pour tous ses conseils et les encouragements, qu'il m'a prodigués. Je voudrais notamment le remercier de m'avoir associé à son projet de recherche ce qui m'a permis d'effectuer ma maîtrise dans des conditions optimales.

J'ai été heureux et touché de l'accueil que m'ont réservé mes collègues du GRIL, accueil qui m'a assuré une intégration rapide et chaleureuse. Ils ont su m'apporter une aide précieuse et efficace tout au long de ma maîtrise.

Je tiens aussi à remercier Michael Leuschel pour ses réponses rapides et précises à mes questions sur ProB.

Mes remerciements s'adressent enfin à mes parents et à ma famille, qui m'ont toujours encouragé et soutenu dans mes projets. Grâce leur soutien inconditionnel, j'ai profité pleinement de l'opportunité offerte par cet échange.

Grâce à ces nombreux soutiens, cette expérience restera un moment privilégié dans ma vie.

Abréviations

VTG Vulnerability Tests Generator

CAP Converted APplet

JCBCV Java Card ByteCode Verifier

TLV Type/Tag Length Value

JVM Java Virtual Machine

JML Java Modeling Language

Table des matières

Liste des figures

Liste des figures

Introduction

Contexte

Les cartes à puce sont des composants essentiels de notre vie courante. Ces objets permettent de mettre en sécurité un secret placé à l'intérieur. Ils sont donc soumis à une très forte pression offensive. Il existe plusieurs modèles de carte à puce, mais nous ne nous intéressons qu'aux cartes utilisant la technologie Java Card. Cette dernière est un sous-ensemble du langage Java adapté aux cartes à puces. Il met en place un environnement d'exécution dans lequel il est possible d'installer plusieurs applications et d'assurer leur exécution. Le vérifieur de code intermédiaire Java Card (≪*Java Card ByteCode Verifier*≫ (JCBCV)) permet de contrôler une application avant son installation pour vérifier son intégrité. Il est composé de deux processus, la vérification de structure et la vérification de type. Pour tester le JCBCV, Aymerick Savary [12] a proposé un outil, le VTG (≪*Vulnerability Tests Generator*≫) permettant la génération de test de vulnérabilité. Celui-ci se base sur une modélisation formelle qui ne concerne que le processus de vérification de type.

Objectifs

Ces recherches ont pour but d'étendre la modélisation au processus de vérification de structure. Les deux processus de vérification sont très différents. Il faut donc proposer une nouvelle modélisation qui est adaptée au processus de vérification de structure. Celle-ci doit pouvoir s'intégrer au modèle existant et fonctionner avec le VTG. Elle doit cependant garantir une facilité de compréhension et d'utilisation.

Méthodologie

La modélisation se base sur la spécification Java Card [15] pour permettre de définir formellement les fichiers CAP (≪*Converted APplet*≫) valides. À partir de cette modélisation, le VTG va générer des modèles comportant des fautes en utilisant la mutation de spécification. Ceux-ci vont permettre l'extraction de tests de vulnérabilité. Pour assurer une bonne complétude de tests, il faut donc garantir une bonne couverture de la spécification. La modélisation doit donc avoir un niveau d'abstraction très bas pour se rapprocher au maximum de la spécification. La modélisation est exprimée avec le langage Event-B qui est utilisé par l'outil VTG. De plus, il faut rechercher le meilleur compromis pour permettre une génération efficace limitant l'explosion combinatoire.

Résultats

Nous avons mis au point une formalisation de la spécification simple à mettre en œuvre. Celle-ci définit l'ensemble des éléments de la structure contenue dans la spécification. Notre modélisation permet, grâce à son organisation, d'assurer une transition simple pour le lecteur qui doit référer à la spécification Java Card. L'étude de cas sur l'héritage nous a permis de vérifier l'intégration du modèle dans le VTG. Les tests abstraits que nous avons générés sont conformes à nos attentes. De plus, nous avons mis en évidence des incohérences dans la spécification Java Card.

Structure du mémoire

Ce mémoire est structuré en quatre chapitres. Le premier est constitué d'un état de l'art et contient les notions fondamentales. Il présente les attaques existantes sur les cartes à puce Java Card ainsi que le JCBCV. Puis il détaille les différentes méthodes de formalisation appliquées au langage Java Card et au langage C. Les travaux d'Aymerick Savary qui constituent la base de ce mémoire sont aussi détaillés dans ce chapitre.

Le deuxième chapitre contient la méthodologie utilisée pour concevoir le modèle du processus de vérification de structure. Il détaille l'organisation du modèle et la méthode utilisée pour traduire la spécification Java Card en Event-B. Il propose un

algorithme pour effectuer ces opérations.

Le chapitre trois présente les évolutions successives du modèle. Il détaille les choix de modélisation. Ils sont apparus au cours de la modélisation pour assurer une bonne compréhension du modèle, mais aussi pour permettre une bonne extraction des tests.

Le dernier chapitre décrit le cas d'étude. Il s'agit d'une preuve de concept utilisant la modélisation créée pour générer des tests abstraits à l'aide du VTG.

La conclusion présente les résultats obtenus ainsi que les perspectives de nos travaux.

Chapitre 1

Revue de littérature

Les cartes à puce sont des coffres-forts électroniques. Elles subissent une forte pression offensive. La première section détaille les types d'attaques existant contre les cartes à puce. Ces attaques permettent de mettre en contexte le JCBCV. La génération de test de vulnérabilité pour le JCBCV à partir d'une formalisation repose sur trois bases. La première est le vérifieur en lui-même. Son fonctionnement et les recherches effectuées à son sujet sont détaillés dans la deuxième section. Une étude sur les différentes méthodes de formalisation et des tests dérivés de spécification est l'objet de la deuxième section. Le choix de la formalisation constitue en effet la deuxième base permettant la.génération de tests de vulnérabilité. La dernière base est la génération des tests qui est l'objet de la dernière section.

1.1 Les attaques existantes sur les cartes

La sécurité défensive consiste à publier des correctifs pour corriger les failles découvertes. En effet, une fois compromise la carte à puce, comme le coffre-fort, ne peut plus garantir la confiance nécessaire à son usage. Un autre type de sécurité est alors utilisé : la sécurité offensive. Elle consiste à rechercher des failles avant que celles-ci ne soient exploitées. Cette approche est utilisée par les instituts de certification pour garantir la fiabilité d'un système sécurisé.

Il existe deux types principaux d'attaques sur les cartes à puce. Les plus fréquentes sont les attaques physiques. Elles consistent à compromettre une carte en interprétant

les fuites d'information induite par la consommation de courant ou le temps de calcul. Il était, par exemple, possible de découvrir la clé RSA qui cryptait les communications à partir de la consommation de courant [5]. Les communications étant déchiffrables, la carte devenait alors compromise. Ces attaques nécessitent un matériel technique de pointe (oscilloscope, sondes micrométriques, etc.). Cette sophistication limite le nombre d'attaquants. De plus, les contre-mesures à ces attaques sont de plus en plus nombreuses. Une deuxième forme d'attaque est donc apparue, celles-ci sont dites logiques, car elles reposent sur la découverte et l'exploitation d'une faille logicielle. Elles ne nécessitent que peu de matériel et sont donc peu onéreuses à mettre en place, mais elles demandent une bonne connaissance du logiciel. Julien Iguchi-Cartigny et Jean-Louis Lanet [8] ont, par exemple, créé une application malicieuse permettant d'accéder et de modifier les autres applications présentes sur la carte. Pour contrer ces attaques, il existe deux mécanismes en Java Card. Le JCBCV, qui valide statiquement les applications à l'installation, et le pare-feu qui vérifie dynamiquement l'exécution d'une application.

1.2 Le vérifieur de code intermédiaire Java Card

Ludovic Casset [3] a proposé une implémentation prouvée du JCBCV embarqué. Il a modélisé à l'aide du langage B les deux étapes successives du JCBCV, la vérification de structure et la vérification de type. La vérification de structure permet de valider le format du fichier contenant l'application, le fichier CAP. Elle vérifie, entre autres, que les fichiers contiennent tous les composants nécessaires à son fonctionnement, mais aussi que les contraintes entre ses composants sont bien respectées. La vérification de type permet de s'assurer que toutes les conversions de type sont en accord avec la spécification du langage. Le JCBCV effectue d'abord la vérification de structure ; si elle échoue le fichier est rejeté, sinon le fichier est alors donné au processus de vérification de type qui va la valider.

Nous avons choisi de vérifier que le JCBCV rejette bien les fichiers invalides. Des failles de ce type ont déjà été trouvées [4]. Il s'agit donc de vérifier le travail effectué par ce composant dans le but d'identifier automatiquement les failles. Ce travail sera effectué en boîte noire sans avoir connaissance de l'implémentation du JCBCV. Nous

ne nous intéressons qu'au processus de vérification de structure.

1.2.1 Format du fichier CAP

Un fichier CAP est un fichier binaire. Il est le résultat de la compilation de l'application. Il est composé de douze composants inter-connectés. Ceux-ci sont définis dans la spécification Java Card qui sera désignée dans le reste du mémoire comme *la spécification*. Elle permet de définir aussi des composants personnalisés. Cependant, pour des raisons de sécurité, cette possibilité n'est pas utilisée. Nos travaux ne concernent que onze des douze composants définis par la spécification. Chacun d'eux est encodé en suivant le TLV (*type length value ou tag length value*). La première partie, le tag, est un entier compris entre un et douze permettant d'identifier le composant. La deuxième contient la taille de ce composant et la dernière les données. Parmi ces composants, certains sont obligatoirement présents dans un fichier CAP, et d'autres optionnels.

1.2.2 Les douze composants d'un fichier CAP

Le composant Header contient les informations générales concernant le fichier CAP, notamment la version de ce fichier, mais aussi la présence des composants additionnels et l'utilisation du type int. En effet, ce type de donnée n'est pas inclus par défaut dans le langage Java Card. Il contient aussi l'ensemble des paquets Java Card défini par ce fichier.

Le composant Directory comprend la taille de tous les composants présents dans le fichier CAP. Les informations concernant le nombre d'applications, les composants personnalisés et les champs statiques sont aussi stockées dans ce composant.

Le composant Applet contient les informations sur les applets définies dans le fichier CAP. Les applets sont des classes qui héritent de la classe Applet. Ce composant est optionnel, s'il n'y a pas d'applets, il n'est pas présent.

Le composant Import stocke l'ensemble des paquets Java Card qui sont importés par les classes définies dans le fichier CAP.

Le composant Constant Pool comprend l'ensemble des références des éléments du fichier CAP. Il existe six types de référence suivant le type de l'élément concerné. La première référence toutes les classes et les interfaces (créées et importées). La seconde, les méthodes virtuelles qui correspondent à l'ensemble des méthodes qui ne sont ni privées, ni statiques. Les constructeurs ne sont pas des méthodes virtuelles. La troisième, contient les méthodes virtuelles issues d'une super classe. La quatrième contient les champs non statiques instanciés par les classes. Les deux derniers concernent les champs et les méthodes statiques.

Le composant Class décrit l'ensemble des classes et des interfaces définies dans le fichier CAP. Il ne contient que les informations nécessaires pour la création, le transtypage et les appels aux méthodes virtuelles.

Le composant Method contient toutes les méthodes définies dans le fichier CAP. Ce composant stocke le bytecode de chaque méthode non abstraite et les exceptions levées par ses méthodes.

Le composant Static Field contient les informations nécessaires pour créer et initialiser tous les champs statiques du fichier CAP

Le composant Reference Location permet de faire les liens entre les références du Constant Pool et celles du Method.

Le composant Export stocke les champs statiques pouvant être importés par d'autres applications.

Le composant Descriptor Il comprend les informations permettant de vérifier les composants Class et Method. Il stocke le complément d'information concernant les liens entre les classes, les interfaces et les méthodes.

Le composant Debug est un composant optionnel servant à déboguer l'application contenue dans le fichier CAP. Ce composant n'est jamais utilisé en pratique, car il permet l'accès à trop d'informations sur l'application. Ce composant ne sera pas modélisé dans ces recherches.

1.2.3 Les contraintes

Les composants du fichier CAP doivent respecter les contraintes imposées par la spécification Java Card. Comme dans les travaux précédents, celles-ci ont été divisées en deux parties, les contraintes internes et les contraintes externes.

Les contraintes internes sont des contraintes portant sur des champs d'un même composant. Par exemple, chaque composant contient un champ taille qui va contraindre les données en limitant l'espace qui leur est alloué.

Les contraintes externes portent sur des champs de plusieurs composants. Par exemple, toutes les classes déclarées dans le composant Class doivent être référencées dans le composant Constant Pool.

1.3 Modélisation de la spécification

Dans ses travaux, Ludovic Casset a modélisé les composants du fichier CAP et synthétisé en langage C pour pouvoir l'embarquer sur une carte. Dans notre cas, il n'y a pas synthèse, car nous voulons pouvoir générer les tests les plus complets possible. De plus, notre but n'est pas de faire un logiciel embarqué, nous n'avons donc pas de contraintes de stockage et de calcul. Les composants et leurs contraintes sont modélisés en Event-B à l'aide de la plateforme Rodin, car il s'agit du format d'entrée du VTG. Cette plateforme est l'objet de la section 1.3.1. La modélisation de la spécification Java Card nécessite la mise en place d'un processus de formalisation. Il existe des outils de formalisation en Java et Java Card qui sont présentés à la section 1.3.2. Notre approche de formalisation basée sur de la traduction est présentée à la section 1.3.3.

1.3.1 Rodin et le langage **Event-B**

Rodin est un outil qui permet le développement de projets Event-B. Ces projets contiennent des contextes et des machines. L'Event-B est une méthode formelle de modélisation de systèmes fondée sur le raffinement et la preuve de propriétés.

Les contextes définissent la partie statique d'une modélisation. Ils sont donc utilisés dans le cadre de nos travaux pour la modélisation. Ils peuvent contenir des ensembles, des constantes et des axiomes. Un contexte peut étendre d'autres contextes, ce qui lui permet d'utiliser tout le contenu de ces contextes dans ses axiomes. Les constantes déclarées dans le contexte doivent être utilisées dans au moins un axiome. Les axiomes sont composés d'un label et d'un prédicat qui est supposé être vrai dans le reste du modèle.

Les machines décrivent la partie dynamique de la modélisation. Elles sont composées des variables, dont la valeur modifiée par les événements. Elles contiennent aussi des invariants qui doivent être respectés par les variables.

1.3.2 Outils de formalisation existants

En 2001, Luc Moreau et Pieter Hartel avaient déjà mis en lumière la nécessité de formaliser les langages Java et Java Card et leurs JVM ≪*Java Virtual Machine*≫ [6]. Ce travail est important pour la recherche, mais aussi pour l'industrie, car il est nécessaire à un haut niveau de certification. Plusieurs solutions ont été proposées sous la forme d'outils comme TestEra [10], Korat [11] ou KRAKATOA [9].

Korat est un outil qui génère un ensemble d'entrées à tester pour des programmes Java. Il prend en entrée une méthode, appelée *finitization*, qui s'assure que l'espace d'état généré est fini et un prédicat. Il parcourt ensuite l'espace d'état pour vérifier que le prédicat n'est pas violé. Cet outil nécessite donc une version exécutable du programme.

TestEra est basé sur Alloy. Il prend en entrée une méthode, une spécification formelle des pré et post conditions de cette méthode et une limite. Il génère et exécute alors un ensemble de tests à l'aide d'Alloy. Cet outil, contrairement au précédent, peut s'utiliser à toutes les étapes du développement.

KRAKATOA est un outil de développement pour la certification de programme

Java et Java Card. Il repose sur des annotations du code en JML (Java Modelling Language) qui précise les pré et post conditions. Il s'assure que ces conditions sont respectées.

Ces trois logiciels permettent de s'assurer que l'exécution du programme se déroule conformément à la spécification. Cependant, le problème est différent dans le JCBCV. Le but est de s'assurer de la validité du programme lors de son installation à partir d'un fichier compilé. Dans notre cas il faut s'assurer du format de ce fichier. L'effort de modélisation doit donc plus porter sur la traduction de la spécification relative au format du fichier que sur l'exécution des méthodes.

1.3.3 Sémantique formelle

La spécification des composants est écrite dans un langage similaire au C. Il comprend des types primitifs, des tableaux, des structures et des unions. De nombreux travaux ont été faits pour formaliser le langage C dans le but notamment de créer un compilateur prouvé. Ces travaux nous intéressent particulièrement, car le fonctionnement d'un compilateur est assez similaire au processus de vérification de structure. Même s'il n'y a pas de vérification grammaticale dans ce processus, il y a comme dans tout compilateur une vérification sémantique.

Sandrine Blazy et Xavier Leroy ont proposé un langage le Cligth qui est un sous-ensemble du C [2]. Ce sous-ensemble contient une sémantique formelle des types de données qui nous intéresse. Ce langage a l'avantage d'être compilé avec un compilateur vérifié CompCert. La syntaxe abstraite du Clight est formalisée en Coq. Dans notre cas, le but est de formaliser en Event-B l'ensemble de la spécification.

Michael Hohmuth et Hendrik Tews ont décrit une sémantique formelle d'un sous-ensemble du langage C++ [7]. Ce sous-ensemble est utilisé pour vérifier des propriétés du microkernel Fiasco. Comme dans notre cas, le but est d'obtenir une traduction en logique d'ordre supérieur. Cette traduction est ensuite soumise au processus de vérification. Pour ce faire, les auteurs proposent une traduction des principaux types de données. Nous avons choisi une approche similaire. Le langage de destination, PVS dans le cas de ce papier, est différent de celui que nous avons choisi. Il est cependant intéressant de constater que les deux traductions ont des points communs. Ceci

est notamment visible dans les deux cas qui ont nécessité l'ajout d'une information supplémentaire sur la position en mémoire.

1.4 Génération de tests à l'aide de mutations

La modélisation de la spécification obtenue est utilisée en entrée du processus de mutation. La mutation est un changement minimal, mais significatif de la modélisation qui entraîne une divergence minime avec la spécification. Ce processus a notamment été utilisé par Paul Black [1] pour générer des tests via un vérifieur de modèle. Une fois la modélisation mutée, le vérifieur de modèle permet la génération de mutants. Ces mutants ne représentent pas complètement la spécification. Si l'un d'eux peut passer les tests assurant la correspondance avec la spécification alors les tests ne sont pas complets.

Dans le cas du JCBCV, Aymerick Savary a proposé une preuve de concept [12]. Son approche se base sur un outil qu'il a développé, le VTG et sur une modélisation en Event-B du JCBCV. Il a approfondi le concept pour le processus de vérification de type et augmenté le nombre d'instructions concernées [13]. Actuellement, le concept au complet est vérifié et les mutations sont prouvées [14]. La génération des tests a été optimisée, ce qui a permis d'atteindre un temps d'exécution raisonnable 45 minutes (contre 2 ans estimés auparavant). De plus, le nombre d'instructions implémentées a été multiplié par cinq.

Dans le cadre d'une coopération que nous avons mise en place, ce mémoire traite du processus de vérification de structure.

1.4.1 Le VTG

Cet outil est divisé en deux processus comme présenter dans la figure 1.1 extraite de l'article [13]. Le modèle qui est utilisé en entrée du VTG est un modèle Event B. Le premier processus Faulty Model Derivation va induire une mutation du modèle original qui entraîne la création d'un ensemble de modèles mutants. Dans le cas du processus de vérification de structure, la mutation concerne les contraintes imposées par la spécification. La section suivante 1.4.2 va détailler le processus de mutation. Le

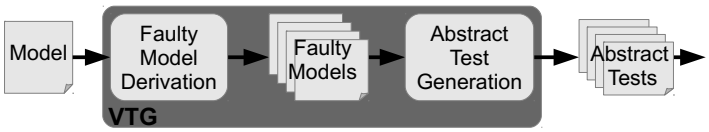

figure 1.1 – Processus du VTG

second processus, Abstract Test Generation, va générer des tests abstraits à partir de ses modèles mutants. Cette étape repose sur l'explorateur de modèle ProB. ProB va résoudre les contraintes mutées et ainsi générer des tests possédant le même niveau d'abstraction que le modèle d'entrée. Il est donc nécessaire de rajouter une étape supplémentaire, le concrétiseur, pour obtenir des tests concrets. Cet outil est pour l'instant en développement.

1.4.2 Le processus de mutation

Notre modèle ne contient que des contextes, nous ne donnons donc ici que des éléments concernant la mutation de contextes. L'ensemble du processus de mutation est donné en détail dans la thèse soutenue par Aymerick Savary. Dans nos contextes, les axiomes contiennent les propriétés de la spécification. Ces axiomes sont divisés en deux ensembles.

Le premier contient les axiomes qui assurent le fonctionnement du modèle. Par exemple, les axiomes qui permettent de typer les constantes. Ces axiomes ne doivent pas être mutés pour pouvoir assurer l'intégrité du modèle. Le deuxième ensemble comprend les axiomes qui décrivent les contraintes. Chaque axiome ne contient qu'une unique contrainte. Le processus de mutation ne va toucher que les axiomes appartenant à cet ensemble

La mutation de chaque axiome va engendrer un ensemble d'axiomes mutants. Pour chaque axiome mutant, le contexte originel va être cloné. Dans ce clone, l'axiome à muter va être remplacé par le mutant pour constituer un contexte mutant. L'application de ce processus à l'ensemble des axiomes mutants permet d'obtenir l'ensemble des contextes mutants. Chaque contexte mutant ne contient alors qu'un unique axiome muté.

1.4.3 Mutation de formule

L'axiome contient un prédicat et c'est celui-ci qui va être muté. Voici quelques exemples de mutations qui sont utilisées dans ce mémoire. Dans ces exemples nous utilisons les notations suivantes :

— $i_1, i_2 \in \mathbb{N}$

— p, p_1, p_2 des prédicats.

La mutation de prédicats simples correspond à la négation du prédicat en question.

— $mut(i_1 > i_2) \rightsquigarrow \{i_1 = i_2\} \cup \{i_1 < i_2\}$

— $mut(i_1 = i_2) \rightsquigarrow \{i_1 < i_2\} \cup \{i_1 > i_2\}$

— $mut(i_1 \neq i_2) \rightsquigarrow \{i_1 = i_2\}$

Dans le cas de prédicat plus complexe, la mutation devient récursive. La récursion s'arrête quand le processus atteint un prédicat simple.

— $mut(p_1 \Rightarrow p_2) \rightsquigarrow (p_1 \wedge mut(p_2))$

— $mut(\forall z \cdot p) \rightsquigarrow (\exists z.mut(p))$

Les exemples ci-dessus ne constituent qu'une partie des règles de mutation. Aymerick Savary propose dans sa thèse un ensemble de règles couvrant toute partie de la grammaire Event-B.

1.5 Conclusion

Nous allons donc effectuer des tests de vulnérabilité sur la vérification de structure. Ces tests se basent sur une modélisation de la spécification Java Card du fichier CAP. Tous les champs de ce fichier sont représentés par des constantes dans des contextes en utilisant le langage Event-B. Ces contextes sont donnés en entrée au VTG qui les mute. ProB extrait ensuite des tests abstraits à partir de ses mutants.

Chapitre 2

Modélisation du vérifieur de structure

Ce chapitre présente la modélisation proposée applicable au vérifieur de structure. Elle est effectuée en Event-B à l'aide de Rodin. Ce langage est composé de machines et de contextes. Les contraintes de structure sont de type statique, notre modélisation ne contient donc que des contextes. Nous avons choisi un niveau d'abstraction très faible, tous les champs de la spécification sont représentés. Seule l'organisation des différents éléments en mémoire est abstraite pour simplifier le modèle. Ce faible niveau d'abstraction est nécessaire pour nos tests. S'il est augmenté alors la concrétisation de nos tests devient beaucoup plus complexe. De plus, la représentation des contraintes est alors moins évidente et donc la couverture des tests est plus complexe à déterminer. Comme nous l'avons vu dans le chapitre précédent, le fichier CAP est composé d'un grand nombre d'éléments. Ces éléments sont très fortement intriqués entre eux. Il a donc été nécessaire d'établir des conventions lors de la création de la modélisation pour que la transition entre la spécification et le modèle soit plus aisée. Elles définissent l'organisation des contextes et des axiomes, mais aussi la dénomination des constantes. Cette organisation a été motivée par le grand nombre de constantes nécessaires (environ 350). Ce nombre de constantes impose un découpage en plusieurs contextes. De plus, ProB instancie un contexte ainsi que tous ceux qu'il étend. Cette décomposition en contextes permet donc aussi une instanciation modulaire.

2.1 Organisation des contextes

Cette section présente les conventions concernant l'architecture des contextes de la modélisation. Elle comprend leur organisation globale, qui concerne l'ensemble des composants et leurs interactions, et leur organisation au sein d'un composant.

2.1.1 Organisation globale

La figure 2.1 décrit les différents contextes utilisés pour la modélisation. Elle ne comprend que deux composants génériques afin de permettre une meilleure lecture. Ce schéma donne l'ensemble des contextes principaux avec leurs noms. Les contextes sont représentés par des bulles. Les carrés en pointillés représentent des ensembles de contextes dont la source, qui est un contexte principal, est au sommet . Pour ne pas alourdir le schéma, le détail de cette organisation n'est pas précisé, il sera l'objet de la section 2.1.4. Les noms de contextes en italique sont génériques. Ils sont particuliers à chaque composant. Cette dénomination des contextes est l'objet de la section 2.5.1. Il suffit de généraliser ce schéma pour obtenir l'architecture des douze composants. Les extensions entre les contextes sont représentées par des flèches. Chaque flèche signifiant que le contexte de départ étend le contexte pointé.

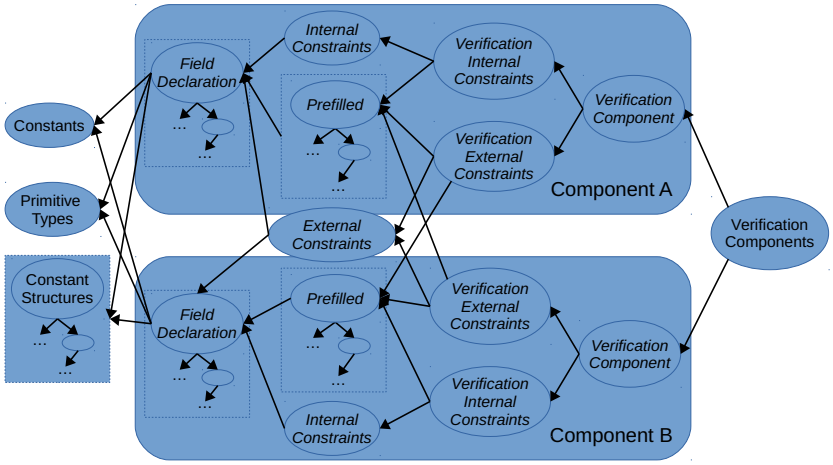

figure 2.1 – patron pour un composant

Les contextes sont soit inclus dans un composant, soit communs à tous les composants. Ils constituent le cœur de la modélisation. Les extensions représentées forment les relations entre ces contextes.

2.1.2 Contextes principaux communs

Ces contextes composent le socle sur lequel sont bâtis tous les contextes utilisés dans les composants. Ils permettent d'éviter la redondance d'informations et servent à regrouper l'ensemble des éléments utilisés par plusieurs contextes. Ils se différencient par le type d'informations qu'ils contiennent.

Le contexte Common Structures décrit les structures utilisées dans plusieurs composants du fichier CAP. Elles sont communes afin de pouvoir les maintenir facilement et de ne pas avoir à les créer dans chaque composant. Chacune de ses structures est contenue dans son propre contexte. Le contexte Common Structures étend chacun de ses contextes. Cette organisation forme un graphe dont Common Structures est la source. Les structures concernées sont Package_Info, Type_Descriptor, Class_ref et Static_ref.

Le contexte Primitive Types contient l'ensemble des types primitifs utilisés par Java Card. Ce sont les types définis dans le fichier CAP. Il y a trois types soit u1 , u2 et u4 qui correspondent respectivement à un, deux et quatre octets. Ces octets sont non signés et sont représentés dans Rodin par le sous-ensemble d'entiers naturels qui leur correspond. Par exemple u1, soit un octet, est représenté par l'intervalle $[0, 255]$, qui correspond à l'ensemble de ses valeurs possibles.

En plus de ces trois types explicitement définis dans la spécification, il est nécessaire d'en rajouter trois. En effet, certains champs du fichier CAP nécessitent de descendre jusqu'au niveau de l'organisation des bits au sein d'un octet. Le bit_4 constitue la moitié d'un octet, il est donc composé de 4 bits consécutifs. Le bit et le bit_15 correspondent au découpage de deux octets en deux parties. Le bit représentant le bit de poids fort et le bit_15 représentant les 15 bits restant des deux octets. Ces types sont spécifiques d'un champ appelé bitfield, qui est détaillé dans la section 2.2.1.

Le contexte Constants contient l'ensemble des constantes utilisées dans un fichier CAP. Il stocke aussi les tags des différents composants et des différentes références du Constant Pool. Il contient les tailles limites des données. Certaines, comme la taille maximale des tableaux, sont fournies explicitement, mais d'autres sont à extrapoler, comme le nombre maximal de classes.

Le contexte Constants étend un contexte flags qui contient l'ensemble des marqueurs utilisés dans le fichier CAP. Ces marqueurs servent à identifier le type d'un champ notamment dans le cas des unions.

Le contexte Verification components permet d'unir l'ensemble des composants pour l'instanciation avec ProB. Chacun des contextes de vérification est vide, ces contextes ne servant que pour l'union de contextes qu'ils étendent. Ils permettent d'apporter de la modularité dans l'instanciation.

2.1.3 Contextes principaux d'un composant

Ces contextes sont présents dans tous les composants. Seuls leur contenu et les contextes qu'ils étendent varient. Ils feront l'objet de la prochaine section.

Le contexte Field Declaration définit l'ensemble des champs qui forment le composant. Ces définitions se basent sur les contextes précédents notamment pour le typage des champs.

Le contexte Prefilled contient l'initialisation des champs définis dans *Field Declaration*. Cette initialisation est effectuée par une application CAP2Rodin en se basant sur une application Java Card valide. Ce contexte permet d'encadrer ProB pour éviter une explosion combinatoire. En effet, l'ensemble des solutions pour tous les champs d'un fichier CAP est trop grand pour être analysé par ProB.

Le contexte Internal Constraints contient les contraintes qui s'appliquent entre les champs appartenant au composant. Elle est donc dépendante de *Field Declaration*

Le contexte* *Verification Internal Constraints permet d'unir *Prefilled* et *Internal Constraints*.

Le contexte* *External Constraints contient les contraintes qui s'appliquent entre les champs appartenant à deux composants. Elle est donc dépendante des contextes *Field Declaration* des deux composants. Chacun de ces contextes représente une paire de composants qui sont liés par des contraintes.

Le contexte* *Verification External Constraints permet d'unir *Prefilled* et *External Constraints*.

Le contexte* *Verification Component unit les deux vérifications précédentes ce qui entraîne la vérification complète du composant.

2.1.4 Organisation du contexte principal *Field Declaration*

Cette organisation varie suivant les structures contenues dans le composant. En effet, la définition d'un composant dans la spécification peut contenir des champs qui ne sont pas de type primitif. Ces champs sont des structures et correspondent aux structures utilisées dans le C. Ces champs peuvent être des types primitifs, des tableaux ou des structures. Chacune des structures d'un composant est détaillée dans un contexte qui lui est propre. Leur architecture forme alors un graphe qui illustre les imbrications des structures dans la spécification. Elle permet ainsi de faciliter la relation entre le modèle et la spécification. Le contexte parent étend la structure qu'il contient. Un graphe est ainsi créé, chaque nœud étant un contexte, et la source est le contexte principal *Field Declaration*. Ce contexte étend donc l'ensemble des contextes de structures qui appartiennent au composant qu'il définit.

Le contexte principal Prefilled suit l'organisation du contexte *Field Declaration* qu'il étend. Ses extensions forment un graphe identique à celui de *Field Declaration*. On peut ainsi conserver la même facilité de transition avec la spécification. External Constraints et Internal Constraints ne suivent pas cette architecture, car ils mettent en relation des champs de différentes structures pour leurs contraintes. Il devient alors impossible de conserver cette organisation. Les contextes de vérification n'ont pas à suivre cette architecture. En effet, la vérification s'applique à tous les champs concernés par les

contraintes à appliquer, il n'est donc plus nécessaire de connaître l'appartenance d'un champ à une structure. De plus, ces contextes ne contiennent aucun axiome, ils n'ont donc pas besoin d'une architecture particulière.

2.2 Patron de représentation des types de données

Cette section présente la traduction des différents types de données en Event-B. Dans le fichier CAP, il existe trois types de données soient les types primitifs, les tableaux et les structures. Dans la suite de la section, nous allons nous servir des types abstraits pour présenter le concept de la traduction. Les types primitifs sont notés ti, les tableaux vi et les structures si où i représente un indice. Les sections suivantes définissent les traductions pour chacun de ses types ainsi que pour la combinaison de ces types rencontrés dans le fichier CAP.

2.2.1 Définition des types primitifs

Il y a trois types primitifs dans le fichier CAP soient u1,u2 et u4. Ils sont définis comme des ensembles d'entiers représentant leur intervalle de valeurs possibles.

Voici leur spécification Event-B

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

Soit c un champ et t un type primitif. La déclaration suivante : $t\ c$ est traduite en Event-B de la façon suivante : $c \in t$. Dans le contexte issu de la traduction, c et t sont des constantes.

En plus de ces trois types, nous avons dû en ajouter trois autres pour pouvoir modéliser de champs particuliers, les bitfields. Ces champs fixent un cadre de lecture particulier aux types primitifs. Ils sont utilisés comme des tableaux de bits. Il en existe

deux formes, une basée sur u1 et une sur u2. Dans la forme basée sur u1, l'octet de données est découpé en deux champs contenant chacun 4 bits de données consécutifs. Dans la forme basée sur u2, les deux octets de données sont découpés en deux champs, un contenant le bit de poids fort et l'autre contenant les quinze bits restants.

2.2.2 Définition des tableaux de types primitifs

Il existe deux déclarations possibles pour un tableau. Dans la première, la taille est spécifiée par une constante. Soit k un entier naturel, $t\ v[k]$ est la déclaration d'un tableau de taille k contenant des éléments de type t. Ce champ est traduit à l'aide d'une fonction totale de la façon suivante : $v \in 1 \mathinner{..} k \rightarrow t$. Dans la traduction v, t et k sont des constantes du contexte. La fonction associe un indice à un élément du tableau. Elle est totale, car par définition tous les indices d'un tableau doivent correspondre à un unique élément du tableau.

Dans la seconde déclaration, la taille est fixée par un autre champ qui est de type primitif. Le champ c est traduit comme dans la section précédente et est utilisé pour définir le domaine de la fonction représentant le tableau. La définition de la structure est à gauche et la traduction correspondante à droite.

Voici leur spécification JavaCard	Voici leur spécification Event-B
$t1 \quad c$	$c \in t1$
$t2 \quad v[c]$	$v \in 1 \mathinner{..} c \rightarrow t2$

2.2.3 Définition des structures

Une structure comprend plusieurs champs. Une valeur de la structure est donc constituée des valeurs de chaque champ de la structure. On peut donc représenter une structure par un ensemble porteur reliant les champs qui la constitue. Soit s une structure contenant un champ c et un tableau v, tous deux de type primitif. $t1$ et $t2$ sont des types primitifs qui peuvent être identiques.

Voici leur spécification JavaCard	Voici leur spécification Event-B
s {	$finite(s)$
...	....
$t1$ c	$c \in s \rightarrow t1$
$t2$ $v[c]$	$v \in s \rightarrow (1..t1 \nrightarrow t2)$
...	$\forall x \cdot x \in s \Rightarrow dom(v(x)) = 1 \,..\, c(x)$
}	...

Dans le contexte, s est un ensemble fini, c et v sont des constantes. L'ensemble s est lié aux champs qui le constitue par une fonction totale, car chaque élément de s doit contenir l'ensemble des champs. La définition du tableau change, car la taille est particulière à chaque élément de l'ensemble. Les indices du tableau sont donc liés aux éléments par une fonction partielle. Le domaine va ensuite être contraint pour les indices du tableau de chaque élément correspondant bien à la taille du tableau de cet élément.

2.2.4 Définition des champs de types structures

La structure peut être utilisée pour typer un champ ou un tableau. Soit $s1$ et $s2$ deux structures.

Voici leur spécification JavaCard	Voici leur spécification Event-B
$s1$ c	$c = s1$
$s2$ $v[k]$	$v \in 1 \,..\, k \rightarrow s2$

Comme $s1$ type un champ, sa cardinalité est forcément de 1. La constante c est créée, car nous avons décidé de représenter tous les champs du fichier CAP par une constante, mais elle ne sera jamais utilisée. Le tableau possède une taille constante, mais on peut transposer la traduction à l'autre définition de tableau de la même manière que celle

utilisée dans le cas de type primitif.

Les structures peuvent aussi être imbriquées les unes dans les autres.

Voici leur spécification JavaCard

	Voici leur spécification Event-B
$s1$ {	
...	$finite(s1)$
$s2$ $c1$	....
...	$c1 \in s1 \rightarrow s2$
}	...
$s2$ {	$finite(s2)$
...	....
t $c2$	$c1 \in s2 \rightarrow t$
...	...
}	

2.2.5 Définition des unions

Les unions sont un type de structures particulières. Elles sont composées de deux éléments de même taille mémoire, par exemple l'union ref définie dans la structure suivante. Soit t un type primitif de n bits. L'union représente un choix, un élément de type ref, dans notre exemple, peut être soit internal soit external. Ce choix est traduit par une partition de ref. Dans le fichier CAP, la distinction entre les deux parties d'une union se fait à l'aide du bit de poids fort. Comme t est composé de n bits, savoir si le bit de poids fort de t est à 1 revient donc à savoir si le champ est un multiple de 2^{n-1}.

Voici leur spécification JavaCard

ref {
  t $internal$
  t $external$
  }

Voici leur spécification Event-B

$finite(ref)$
$partition(ref, internal, external)$
$internal\_ref \in t$
$external\_ref \in t$
$\forall ref \cdot (ref \in internal$
$\Rightarrow internal\_ref(ref)/2^{n-1} = 1)$
$\forall ref \cdot (ref \in external$
$\Rightarrow internal\_ref(ref)/2^{n-1} = 0)$

2.2.6 Définition des structures partagées

Une structure partagée est une structure qui est utilisée par plusieurs composants. Comme les autres structures, elle est traduite par un ensemble. Cependant, cet ensemble est partitionné en autant de sous-ensembles qu'il y a de composant qui utilise la structure. Chaque composant utilise le sous-ensemble qui lui est attribué ce qui permet de trouver facilement les éléments qui lui appartiennent. Soit s une structure partagée par les composants A et B. Les constantes s_A et s_B représentent les sous-ensembles appartenant respectivement au composant A et au composant B.

Voici leur spécification Event-B

$$finite(s)$$
$$partition(s, s\_A, s\_B)$$

Tous les contextes contenant les structures partagées sont étendus par le contexte Common_Structures.

2.2.7 Cas particulier des décalages

Dans le fichier CAP , certains tableaux de structures ont des tailles non spécifiées. Les éléments contenus dans ses tableaux sont identifiés par leur position en mémoire. Ils sont situés à l'aide d'un décalage qui correspond à la différence de leur position avec la position de début du composant qui les contient. Ce décalage est nécessaire pour identifier les éléments et donc pour créer les contraintes entre les composants. Or le niveau d'abstraction choisi entraîne la perte de cette information. Il est donc nécessaire de rajouter un champ contenant le décalage de l'élément. Ce champ est une référence interne il est donc de type u2. Les structures concernées sont le method_info, le class_info et le static_field_ref. Un exemple est présenté dans le chapitre 4.

2.3 Le contexte Prefilled

Les contextes principaux Prefilled permettent de remplir les relations définies dans la section précédente. Ces contextes servent de base à ProB pour générer des tests abstraits lors de la mutation. Les contextes Prefilled sont générés à l'aide du logiciel Cap2Rodin à partir d'une application Java Card valide. Les données contenues dans cette application sont traduites en Event-B, puis sont affectées aux constantes créées dans la section précédente. Le contexte principal Prefilled étend le contexte principal *Field Declaration*. Ces deux contextes principaux se basent sur la même organisation de contextes afin de permettre une meilleure transition entre les deux. Cette section se base aussi sur le composant Directory. Le but de cette section est de montrer le format des axiomes. Leur contenu est donc secondaire. Aussi, pour simplifier cet exemple, nous n'utilisons pas de données réelles, car elles nécessiteraient des explications qui ne sont pas l'objet de ce chapitre.

2.3.1 Patron des axiomes

Cette section reprend les champs définis dans la section 2.2.

Les types primitifs

La figure 2.3.1 montre le format d'attribution de valeur pour les types primitifs contenus dans le contexte principal Prefilled.

axm : $tag = 2$

figure 2.2 – Exemple avec un type primitif

Les tableaux de types primitifs

Les tableaux sont définis comme une fonction totale entre l'indice et la valeur. Chaque élément de cette relation est ajouté dans un axiome distinct. La figure 2.3.1 illustre le cas du tableau défini dans la section 2.2.2. Cette règle permet une meilleure lisibilité des éléments contenus dans la relation. L'opérateur utilisé est l'inclusion ce qui permet à ProB de créer des éléments dans cette relation. La fonction totale oblige ProB à garder une structure de tableau cohérente.

axm : $\{1 \mapsto 4\} \subseteq component\_sizes$
axm : ...
axm : $\{12 \mapsto 2\} \subseteq component\_sizes$

figure 2.3 – relation avec un tableau de types primitifs

Les structures

Pour le cas des structures, l'ensemble qui les modélise est partitionné en autant d'éléments que nécessaire. ProB peut comme dans le cas précédent rajouter des éléments à l'ensemble si nécessaire. Dans le cas défini à la section 2.2.3, la structure est unitaire. Sa définition suit donc la forme de la figure 2.3.1. La partition a lieu dans le contexte père et non dans le contexte dédié à la structure.

Les structures tabulaires

Dans ce cas, plusieurs structures peuvent coexister. Pour l'exemple, nous en avons défini deux à l'aide d'une partition. Comme pour les tableaux de types simples, l'ajout

axm : partition($static\_field\_size\_info$, $static\_field\_size\_info1$)

axm : $static\_field\_size = static\_field\_size\_info1$

figure 2.4 – relation avec un tableau de types primitifs

de chaque valeur dans une structure tabulaire se fait dans un axiome distinct via l'opérateur d'inclusion.

axm : $custom\_count = 2$

axm: partition($custom\_component\_info, custom\_component\_info1$, $custom\_component\_info2$)

axm : $\{1 \mapsto custom\_component\_info1\} \subseteq custom\_components$

axm : $\{2 \mapsto custom\_component\_info2\} \subseteq custom\_components$

figure 2.5 – relation avec un tableau de types primitifs

2.3.2 Exemple

Dans cette section les deux éléments créés précédemment dans l'ensemble custom_component_info sont réutilisés. L'utilisation de fonction totale oblige ProB à créer toutes les parties d'une structure pour chaque élément qui est contenu dans son ensemble. Comme pour les tableaux, chaque axiome ne contient que l'ajout d'un élément et l'opérateur d'inclusion est utilisé.

Les types primitifs

axm : $\{custom\_component\_info1 \mapsto 2\} \subseteq component\_tag$

axm : $\{custom\_component\_info2 \mapsto 4\} \subseteq component\_tag$

figure 2.6 – Axiome de base d'une structure

Les tableaux

Pour l'exemple, nous attribuons à la première structure un tableau de deux éléments et un tableau vide à la deuxième. Comme son tableau est vide, l'élément custom_component_info2 n'apparaît pas dans les relations indices et AID.

$$\texttt{axm} : \{custom\_component\_info1 \mapsto 2\} \subseteq AID\_length$$
$$\texttt{axm} : \{custom\_component\_info2 \mapsto 0\} \subseteq AID\_length$$
$$\texttt{axm} : \{custom\_component\_info1 \mapsto 1\} \subseteq indices$$
$$\texttt{axm} : \{custom\_component\_info1 \mapsto 2\} \subseteq indices$$
$$\texttt{axm} : \{(custom\_component\_info1 \mapsto 1) \mapsto 1\} \subseteq AID$$
$$\texttt{axm} : \{(custom\_component\_info1 \mapsto 2) \mapsto 1\} \subseteq AID$$

figure 2.7 – Axiome de base d'une structure

2.3.3 Les cas particuliers

Il n'y a pas de cas particuliers dans les contextes de Prefilled. En effet, les unions et les bitfields ne sont que des cas particuliers au niveau de la définition de leur modélisation. Une fois que celle-ci est définie, les ensembles contenus dans les unions peuvent être assimilées à des structures et les champs des bitfields à des types primitifs.

2.4 Contraintes

Dans les contextes de contraintes, nous avons pris pour convention de définir chaque contrainte par un axiome. Le motif de cette règle est de limiter la présence de «et» logique dans les axiomes. En effet, une concaténation de plusieurs contraintes devient rapidement illisible et incompréhensible même si les outils développés permettent de les muter.

Pour respecter cette règle, nous avons aussi adopté une forme particulière pour le quantificateur $\forall$. L'utilisation de cet opérateur suit le modèle suivant :

$$\forall(x_1 \cdots x_n) \cdot (P_1[x_1 \cdots x_n] \Rightarrow p_2)$$

P_1 est un ensemble de prédicats et p_2 est un prédicat unique contenant la contrainte à définir. Cette notation permet d'identifier rapidement le prédicat définissant la contrainte et facilite sa mutation.

2.5 Convention de dénomination

Les parties précédentes entraînent la création d'un grand nombre de contextes et de constantes. Il est donc nécessaire de pouvoir les identifier précisément. Le nom de chacun permet de déterminer leurs places et leurs rôles au sein de l'architecture de la modélisation.

2.5.1 Les contextes

Le nom des contextes suit le schéma suivant :
$\langle pr\acute{e}fixe\rangle\_\langle tag\rangle\_[\langle nomdu(des)contexte(s)p\grave{e}re(s)\rangle\_]\langle nomdel'\acute{e}l\acute{e}ment\rangle$

Le prefix est déterminé par le tableau de la figure 2.8. Il permet d'identifier rapidement le contexte principal dont dépend le contexte nommé. Le chiffre placé devant permet d'ordonner plus significativement le classement effectué par Rodin.

Contexte principal	préfixe
communs	0_Com
Field Declaration	1_FD
Prefilled	2_Pre
Internal Constraints	3_IC
External Constraints	4_EC
Verification Internal Constraints	5_VIC
Verification External Constraints	6_VEC
Verification Component	7_VC

figure 2.8 – préfixe associé aux contextes principaux

Le tag correspond au tag du composant que décrit le contexte nommé. Les contextes principaux communs ne possèdent pas de tag, car ils concernent l'ensemble des composants.

Le nom du(des) contexte(s) père(s) permet de situer rapidement le contexte courant. Comme nous l'avons vu précédemment les contextes forment un graphe dont la source est un contexte principal. Le nom des contextes pères correspond aux noms des contextes constituant le chemin entre le contexte courant et le contexte principal. Pour éviter que le nom final ne soit trop long, l'acronyme du nom de ses contextes est utilisé. Cette partie est absente pour la dénomination des contextes principaux.

Le nom de l'élément est constitué du nom de la structure ou du nom du composant dans le cas où le contexte décrit le composant.

Les contraintes externes sont définies entre deux composants. Elles ont donc un nom qui suit le format :

$\langle préfixe \rangle\_\langle tagpremiercomposant \rangle\_\langle tagdeuxièmecomposant \rangle$

Les relations entre les deux composants ne sont pas ordonnées, mais par simplicité les tags sont classés dans l'ordre croissant.

Les constantes utilisée pour représenter les champs suit ce format de dénomination. Le nom du contexte d'origine ne comporte pas de préfixe dans ce cas.

$\langle nomducontexted'origine \rangle\_\langle nomduchamps \rangle$

Les champs ajoutés à la modélisation vont avoir une dénomination particulière. Les champs de décalage définis dans la section 2.2.7 sont nommés en suivant ce format :

$\langle nomducontexted'origine \rangle\_offset\_\langle nomdestructure \rangle$

Les ensembles issus de la partition d'une structure du contexte principal Common_Structures (section 2.2.6) suivent ce format pour leur nom.

$\langle nomducomposant \rangle\_\langle tag \rangle\_\langle nomdelastructure \rangle$

2.6 Processus de modélisation

Dans cette section, nous présentons le processus utilisé pour la construction du contexte principal *Field Declaration* associé au composant Directory. Ce composant contient l'ensemble des types que nous avons présenté précédemment. La construction de ce contexte peut être rationalisée en une suite d'actions simples. C'est cet algorithme qui est utilisé lors de la création des contextes principaux *Field Declaration* des autres composants. Les contextes principaux communs ont déjà été créés. L'ensemble de ces contextes constitue la base de laquelle dérivent tous les autres.

La création des contextes Prefilled est entièrement automatisée. Ils sont créés à l'aide de l'application Cap2Rodin. Les autres composants principaux ne peuvent être construits qu'après les contextes précédents. La modélisation des contraintes ne peut pas être assimilée à une suite d'opération simple, car chaque contrainte est particulière et possède alors sa propre modélisation. Les contextes de vérification ne contiennent aucun axiome et leur création a été détaillée au début du chapitre.

La construction du contexte principal *Field Declaration* et de son organisation de contextes dépend de deux processus. Le premier processus permet de créer le contexte principal *Field Declaration* pour chaque composant. Il construit aussi les axiomes à partir des champs du composant 2.2.

```
function MODÉLISATION
    for all composant do
        création du contexte principal Field Declaration
        for all champ do
            traduction du champ
            if champ = structure then
                CRÉATION STRUCTURE(type du champ)
            end if
        end for
    end for
end function
```

figure 2.9 – Processus de modélisation des contextes principaux *Field Declaration*

L'application de ce processus à la spécification du composant Directory nous permet d'obtenir la traduction suivante.

Voici leur spécification JavaCard

$u1$ tag

$u2$ $size$

$u2$ $component\_sizes[12]$

$static\_field\_size\_info$ $static\_field\_size$

$u1$ $import\_count$

$u1$ $applet\_count$

$u1$ $custom\_count$

$custom\_component\_info$ $custom\_components[custom\_count]$

Voici leur spécification Event-B

$$dic\_tag \in u1$$
$$dic\_size \in u2$$
$$dic\_component\_sizes \in 1 \mathinner{..} 12 \rightarrow u2$$
$$dic\_static\_field\_size = dic\_sfsi\_static\_field\_size\_info$$
$$dic\_import\_count \in u1$$
$$dic\_applet\_count \in u1$$
$$dic\_custom\_count \in u1$$
$$dic\_custom\_component \in$$
$$1 \mathinner{..} dic\_custom\_count \rightarrow dic\_cci\_custom\_component\_info$$

Le composant Directory contient deux structures. Ces structures suivent un processus particulier détaillé dans la figure 2.10. Ce processus est récursif car les structures peuvent être imbriquées. Il prend en charge la création du contexte propre à la structure et l'extension du contexte père vers ce contexte. Dans le cas du composant Directory, deux contextes de structure sont créés et sont étendus pour le contexte *Field Declaration* de ce composant. L'ensemble de l'organisation du contexte principal *Field Declaration* est ainsi engendré.

```
function création Structure(name)
    création du contexte
    création du set name
    for all champ do
        traduction du champ
        if champ = structure then
            création Structure(type du champ)
        end if
    end for
end function
```

figure 2.10 – Processus de modélisation pour les structures

Nous ne détaillons ici que la structure custom_component_info. Les contextes com-

plets correspondant à la modélisation du composant Directory sont contenus dans l'annexe A.

Voici leur spécification JavaCard

$$
\begin{array}{ll}
custom\_component\_info\{ & \\
\quad u1 & component\_tag \\
\quad u2 & size \\
\quad u1 & AID\_length \\
\quad u1 & AID[AID\_length] \\
\quad \} &
\end{array}
$$

Voici leur spécification Event-B

$$
\begin{array}{l}
finite(dic\_cci\_custom\_component\_info) \\
dic\_cci\_component\_tag \in dic\_cci\_custom\_component\_info \rightarrow u1 \\
dic\_cci\_size \in dic\_cci\_custom\_component\_info \rightarrow u2 \\
dic\_cci\_AID\_length \in dic\_cci\_custom\_component\_info \rightarrow u1 \\
dic\_cci\_AID \in dic\_cci\_custom\_component\_info \rightarrow (u1 \nrightarrow u1) \\
\quad \forall d \cdot (d \in dic\_cci\_custom\_component\_info \\
\qquad \Rightarrow dom(dic\_cci\_AID(d)) = 1 \mathinner{..} dic\_cci\_AID\_length(d))
\end{array}
$$

2.7 Conclusion

Les conventions décrites dans ce chapitre assurent que la modélisation reste homogène. De plus, elles minimisent les preuves à effectuer, l'auto-prouveur de Rodin arrive à en résoudre la grande majorité. Elles assurent aussi une transition rapide entre la spécification et le modèle. Elles facilitent donc l'écriture des contraintes. Cette modé-

lisation constitue le socle des différents outils utilisés pour la négation des contextes. Il est donc important de bien choisir les règles pour pouvoir permettre la meilleure articulation possible entre ces outils et la modélisation. Ces règles ont été créées de manière empirique suite aux différentes versions de la modélisation. Elles sont suffisamment génériques pour s'appliquer à d'autres cas dont la forme est similaire à la spécification.

Chapitre 3

Évolution de la modélisation et du VTG

Le chapitre précédent détaille les conventions choisies pour modéliser le vérifieur de structure. Celles-ci sont le résultat d'une évolution à travers plusieurs étapes. Ce chapitre présente ces étapes, il détaille aussi les modifications apportées au VTG pour permettre la mutation de contextes. Ces changements de modélisation ont été induits par plusieurs facteurs, les principaux étant de proposer un modèle permettant de rester au plus près de la spécification tout en gardant un modèle lisible avec le minimum de preuve. À ces facteurs s'en rajoute un autre, soit permettre à ProB d'animer le modèle efficacement. L'organisation des contextes constitue la première partie de chapitre. Les deux sections suivantes traitent des modélisations des structures et des structures tabulaires. La troisième section s'intéresse aux contextes Prefilled. La dernière porte sur l'évolution du VTG.

3.1 Évolution des contextes

La figure 3.1 constitue le point de départ de la modélisation. Dans cette première approche, chaque ellipse représente un unique contexte. Cette approche avec un nombre limité de contextes est entrée en conflit avec notre choix de modélisation, qui est de représenter l'ensemble des champs des composants. Cet ensemble est modélisé sous forme de constantes. Dans le contexte Structure et suivant le composant concerné,

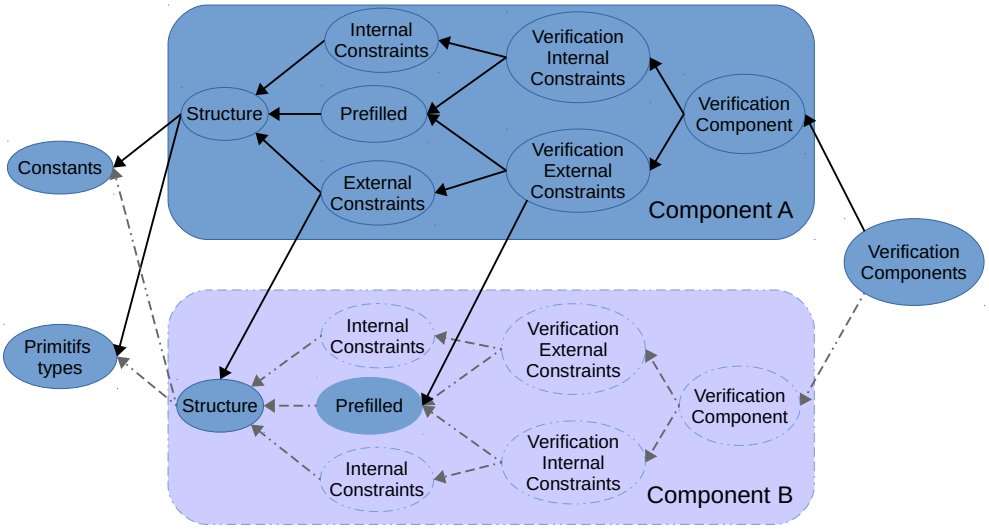

figure 3.1 – ancien modèle pour un composant

le nombre de constantes peut devenir grand ce qui rend le contexte moins lisible et donc moins exploitable. Pour résoudre ce problème, la notion de contextes principaux a été introduite. En créant un contexte par structure, le nombre de constantes par contexte diminue et donc la lisibilité augmente. Le lien entre les contextes est créé à l'aide d'extension ce qui forme alors l'arbre de contexte dont Structure est la racine.

Une des conséquences de ce système est de créer des contextes redondants, quand une structure est présente dans plusieurs composants. Pour l'éviter, un nouveau contexte global Common Strucure a été créé. Ce contexte étend les deux structures redondantes présentes dans le modèle Class_Ref et type_descriptor.

Le contexte External Constraints a évolué. D'abord présent dans chaque composant, il est ensuite devenu un contexte partagé par une paire de composants. En effet, il contient les contraintes entre deux composants, il est donc impossible d'associer ces contraintes à un des deux composants.

3.2 Modélisation des structures

Nous sommes partis d'une modélisation naïve à l'aide de fonctions partielles. Celles-ci permettent de voir les structures comme des tableaux. Les fonctions devaient être restreintes via des contraintes. Les tableaux et les structures étaient donc modélisés de la même manière. Ceci permet une grande simplicité d'utilisation. Les structures sont les éléments les plus complexes à modéliser. En effet, elles peuvent contenir les quatre différents formats de données. Dans le fichier CAP, elles contiennent fréquemment des structures plus internes.

Le passage entre les différents niveaux d'imbrication se fait à l'aide d'une fonction partielle. Cette modélisation a rapidement montré ses limites qui sont facilement visibles dans l'exemple suivant. Pour trois structures A, B, C imbriquées successivement, la représentation naïve donne une fonction d'ordre 2, une liant A à B et une autre B à C. Il y a donc un effet d'accumulation qui se crée. Il est aggravé par la présence de tableaux, qui sont aussi représentés par une fonction partielle liant leurs indexes à leurs contenus. Leur modélisation augmente encore l'ordre. Certains champs de la modélisation composée de l'intrication de ces deux formats se retrouvaient alors représentés par une fonction d'ordre cinq : ce qui rend ces champs inutilisables et illisibles.

Il est donc devenu évident qu'il fallait séparer la modélisation de ces deux formats. Celle du tableau fait l'objet de la section suivante. L'Event-B permet de représenter des types de données particuliers via des ensembles. Nous avons donc décidé de représenter chaque structure par un ensemble. Comme présenté au chapitre précédant, chaque élément de la structure est lié aux ensembles qui définissent cette structure. L'ensemble sert alors d'interface entre les éléments qu'il contient et le reste du modèle. L'imbrication est alors représentée par une fonction entre deux ensembles. L'imbrication successive de trois structures A, B, C est donc modélisée par deux fonctions une liant A à B et une autre liant B à C. L'effet d'accumulation n'est plus présent et l'imbrication n'entraîne alors plus d'explosion de l'ordre des fonctions dans la modélisation.

En plus de cet effet sur l'imbrication, l'ensemble a alors permis une simplification d'un type de champs particulier les unions. En effet, les contraintes de ces champs

peuvent alors être représentées très simplement par une partition (voir section 2.2.5). La partition assure l'ensemble des propriétés de l'union.

Les fonctions partielles qui faisaient le lien entre les éléments ont été remplacées par des fonctions totales. Les fonctions totales permettent de simplifier les contraintes. Elles obligent par définition chaque structure à implémenter l'ensemble de ses éléments. Cette représentation possède un deuxième avantage : elle permet à ProB de travailler de façon plus efficace.

3.3 Modélisation des structures tabulaires

Ces éléments ont demandé un effort particulier dans la modélisation. En effet, il s'agit des éléments les plus complexes, car ils sont composés de relations d'ordre deux. Elles mettent en relation les ensembles détaillés dans la figure 2.2.4 du chapitre précédent. Ces relations concernent l'ensemble des structures d'origine E_1, l'ensemble des indexes d'un tableau E_2 et l'ensemble des valeurs du tableau E_3. Une première relation , $r1$, lie une structure à l'ensemble de ses indexes. Chaque association, créée dans celle-ci, est reliée à sa valeur dans une deuxième relation, $r2$. Nous allons détailler trois manières de les modéliser. Chacune d'elles a un impact sur l'animation avec ProB et sur les preuves à fournir.

3.3.1 Essai basé sur une relation simple

Dans cette version une constante est ajoutée, elle n'est pas contenue dans la spécification. Elle définit $r1$ sous la forme d'une relation à laquelle on ajoute une contrainte pour s'assurer que les indices possèdent des valeurs correctes pour des indexes de tableau. Chacun des éléments de cette relation est associé à l'élément de E_3 qui lui correspond via une fonction totale. Par rapport à la version définitive proposée dans

axm: $AID\_length \in custom\_component\_info \rightarrow u1$
axm: $indices \in custom\_component\_info \leftrightarrow\!\!\!\!\rightarrow u1$
axm: $\forall t \cdot (t \in custom\_component\_info \Rightarrow indices[\{t\}] = 1 \mathinner{..} AID\_length(t))$
axm: $AID \in indices \rightarrow u1$

le chapitre 2, celle-ci nécessite la création d'une constante. Il y a un nombre important de ce type de tableau dans la spécification. L'ajout de cette constante a donc un coût important pour ProB. De plus, cette version se base sur une relation, très peu restrictive, qui entraîne rapidement une explosion combinatoire dans ProB. Le coût en preuve de cette version et la complexité des contraintes sont légèrement plus élevés que dans la version définitive. Le surcoût entraîné dans ProB a motivé l'abandon de cette version.

3.3.2 Essai basé sur une séquence

La modélisation d'un tableau est généralement associée aux séquences en langage B. Ce concept n'existe pas en Event-B, cependant il est possible de l'ajouter via le plug-in Theory. Les séquences ont l'avantage d'avoir nativement un domaine qui correspond aux indexes désirés du tableau. La figure 3.3.2 donne la modélisation de l'exemple avec une séquence. Il est nécessaire de rajouter une contrainte pour définir la taille de la séquence en fonction de la taille du tableau. L'utilisation d'une séquence n'entraîne

axm: $AID\_length \in custom\_component\_info \rightarrow u1$
axm: $AID \in custom\_component\_info \rightarrow seq(u1)$
axm: $\forall t \cdot (t \in custom\_component\_info \quad \Rightarrow seqSize(AID(t)) = AID\_length(t))$

pas de modification significative du temps de l'animation dans ProB. Par contre, elle nécessite plus de preuves, qui, par ailleurs, sont plus complexes. La complexité des contraintes reste similaire à celle de la version définitive.

La version définitive constitue le meilleur compromis entre facilité d'animation dans ProB et quantité de preuve à effectuer. Ceci est permis par l'emploi de fonctions totales et de fonctions partielles dont le domaine est défini à l'aide d'une contrainte. Cette simplicité de définition permet à l'auto prouveur de Rodin d'effectuer la grande majorité des preuves.

3.4 Les contextes Prefilled

Ces contextes étendent les contextes Structure dans chaque composant. Les contextes Prefilled ont donc dû suivre les évolutions des contextes Structure. Les contextes Prefilled jouent aussi un rôle central dans la génération des tests. Ils contiennent la traduction d'une application JavaCard valide en Event-B. Chaque constante définie dans les contextes Structure est donc associée à sa valeur ou à l'ensemble de valeurs, qui lui correspondent dans le fichier CAP de l'application. En effet, ces contextes vont être complétés par ProB lors de la génération des tests abstraits. Ils doivent donc contenir suffisamment d'informations pour éviter une explosion combinatoire dans ProB. Cependant, ces informations ne doivent pas non plus influer sur la couverture des tests. Il est donc nécessaire de mettre en place une stratégie pour définir un moyen automatique résolvant ce compromis.

3.4.1 Masquage

La première stratégie mise en place consistait à masquer les champs intervenant dans la contrainte mutée. Le masquage consiste à enlever la valeur ou l'ensemble de valeurs associés à la constante représentant le champ. ProB doit alors compléter l'information manquante, afin qu'elle réponde aux contraintes du modèle. Cette méthode permet d'insérer une erreur dans l'application valide, qui est contenue dans les contextes Prefilled. Ainsi les informations de chaque champ impliqué sont supprimées des contextes Prefilled. Pour éviter des incohérences, ce masquage est récursif. Il remonte en masquant chaque champ impliqué dans une contrainte avec un champ précédemment masqué. Cette récursivité constitue la faiblesse de la stratégie, car son effet cumulatif peut entraîner le masquage de tous les champs. En effet les contraintes sont très intriquées, notamment celles reliant les tailles des éléments du fichier CAP. Les tailles de tous les éléments sont sommées, donc le masquage d'une seule de celles-ci entraînent le masquage de toutes. De plus, le masquage s'étend alors aux éléments que décrivent ces tailles, ce qui aboutit alors au masquage complet des contextes Prefilled. Cette stratégie a donc dû être abandonnée.

3.4.2 Choix des fonctions partielles

Pour répondre au problème précédent, nous avons décidé d'utiliser des fonctions partielles. Par définition, leur domaine n'est pas défini strictement, il est constitué d'un sous-ensemble. Cette liberté permet à ProB de travailler. Il peut ainsi rajouter des éléments aux relations précédemment définies. Cependant, la fonction partielle implique un coût de calcul plus important dans ProB. Comme le modèle contient un grand nombre de relations de ce type, le coût final devient trop important. Cette approche a donc dû être aussi abandonnée.

3.4.3 Choix des fonctions totales

Les fonctions totales ont un coût moindre dans ProB, car leur domaine est clairement défini. L'utilisation de ces fonctions est donc recommandée si l'on veut diminuer ce coût. Par contre, l'utilisation de telles fonctions nous fait perdre le degré de liberté permis par les fonctions partielles. Nous avons donc décidé de réutiliser la notion de masquage en lui fixant des limites nettes pour éviter les effets de bords. La stratégie qui est alors utilisée consiste à masquer seulement les structures impliquées dans une contrainte. Cependant pour que ProB arrive à résoudre les contraintes il ne faut pas masquer les informations relatives au contenu des ensembles représentant ces structures. Chaque ensemble introduit dans les contextes Structure voit ses éléments détaillés dans les contextes de Prefilled. Nous utilisons pour cela une partition, car elle permet de garantir la cohérence. Cette partition ne doit pas être masquée, car ProB n'est pas capable de générer des éléments de ce format sans entrer en explosion combinatoire.

3.5 Évolution du VTG

Le VTG a été conçu, dans sa première version [12], pour muter des machines. Il a donc fallu étendre le processus de mutation aux contextes. De plus, l'ajout de nouvelles règles de mutation dans ce processus nécessitait la modification du code source et s'avérait donc peu pratique. Nous avons donc développé une nouvelle version de l'application permettant de prendre en compte la mutation de contextes et d'ajouter

simplement de nouvelles règles. Cette dernière contrainte a constitué la principale difficulté. La nouvelle version du processus de la mutation de formule (cf section 1.4.3) est détaillée dans cette section. Elle repose sur un analyseur de formule utilisant TOM.

3.5.1 Utilisation de TOM

Au départ, les règles de mutation devaient être contenues dans un fichier XML qui aurait constitué la bibliothèque sur laquelle se baserait l'analyseur. Mais l'analyseur devenait alors très lourd et très complexe. TOM apporte une solution efficace à ces problèmes. TOM permet de créer et surtout de parcourir un arbre de données orientées objet. La grammaire Rodin est définie dans ce langage.

L'analyseur en TOM permet donc de naviguer dans l'ensemble de la grammaire Rodin. Les programmes en TOM peuvent être compilés pour générer du code dans plusieurs langages (C, C++, Java). Dans notre cas, le programme sera compilé en JAVA. Une des principales contraintes est donc respectée, le programme en TOM est contenu dans un fichier externe à l'application. Ce fichier est facilement modifiable et sa compilation en Java assure son intégration à l'application.

3.5.2 Analyseur de formule

L'analyseur utilise une structure conditionnelle particulière match. Celle-ci fonctionne comme un switch et reconnaît les éléments de la grammaire qui composent une formule. Ceux-ci sont définis dans le fichier TOM qui contient la grammaire Rodin comme le prédicat *equal* ci-dessous.

```
%op Predicate Equal (left: Expression, right: Expression) {
      is_fsym(t) { t != null && t.getTag() == Formula.EQUAL }
      get_slot(left,t) { ((RelationalPredicate) t). getLeft() }
      get_slot(right,t) { ((RelationalPredicate) t).getRight() }
}
```

Pour détecter les prédicats d'égalité, il suffit de les ajouter au match comme ci-dessous. TOM permet la détection du prédicat *equal*, mais il permet aussi d'accéder facilement aux expressions composant ce prédicat via les paramètres left et right.

Cette propriété permet de mettre facilement en place la récursion de certaines règles de mutation.

```
%match(formula) {
       ...
       Equal(left,right)->
       {
       ...
       }
       ...
}
```

L'ajout de nouvelles règles est donc très simple. Le seul inconvénient est que cet ajout se fait dans un fichier TOM donc à chaque modification il faut le recompiler pour obtenir le fichier Java correspondant.

3.6 Conclusion

Les évolutions successives ont permis au modèle de gagner en cohérence et en lisibilité. L'utilisation de ProB pour animer le modèle nous semblait dès le départ de la modélisation comme un facteur important. Il a, en fait, peu à peu, pris une grande importance en conditionnant les choix. ProB ne peut pas animer le modèle si les relations sont moins restrictives. Ces évolutions ont aussi permis de répondre au problème d'accumulation. En représentant les structures avec des ensembles, nous avons abaissé la complexité du modèle. La modélisation ne dépend plus alors du niveau d'imbrication des structures, mais seulement de son élément le plus complexe : le tableau. Celui-ci est représenté avec une fonction d'ordre deux, il constitue l'ordre maximum dans ce modèle. L'évolution de l'analyseur de formule a permis l'ajout simplifié de règles de mutation. De plus, celui-ci repose sur la grammaire Event-B.

Chapitre 4

Cas d'étude

Nous allons développer ici deux cas d'étude. Ceux-ci portent sur une notion fondamentale en programmation orientée objet (POO), l'héritage. Nous avons choisi d'étudier cette notion, car elle est bien connue. Dans le fichier CAP, cette notion est représentée dans deux composants, le Class et le Constant Pool. Elle est traduite par des contraintes simples. Ces cas nous permettent d'apporter la preuve de concept de la bonne intégration de la modélisation dans le VTG.

4.1 Présentation du cas d'étude

4.1.1 Notions sur l'héritage

La POO manipule des objets qui sont décrits dans des classes. L'héritage est une manière de créer des liens entre les classes. Il permet à partir d'une classe mère de créer une classe fille qui possédera, en plus de ses comportements propres, des comportements hérités de sa mère. Cette relation permet d'ordonner les classes en créant une hiérarchie. Suivant le nombre de parents permis, l'héritage peut être simple avec un unique parent ou multiple avec plusieurs parents. La forme d'héritage choisi dépend du langage de programmation. Cependant, cette relation possède, indépendamment du langage, deux propriétés fondamentales : la transitivité et l'absence de cycle

4.1.2 L'héritage en Java

Le Java Card est un sous-ensemble du Java. Ces deux langages ont notamment en commun toutes les caractéristiques qui concernent les types de référence et l'héritage. Ils possèdent deux types de référence : les classes et les interfaces. Ces deux objets ont les mêmes propriétés en Java et en Java Card. La seule différence se situe dans le nombre maximal d'objets instanciables. En Java Card la mémoire est limitée : seul un nombre réduit de ces objets peut être créé. Il ne peut exister au maximum que 255 classes et interfaces définies dans un fichier Cap.

En Java, l'héritage est simple pour les classes et multiple pour les interfaces. Chaque classe ne peut donc avoir qu'une unique classe mère. Il existe une classe originelle appelée Object dont héritent toutes les classes créées. Object contient les caractéristiques minimales d'un type de référence en Java. Les interfaces et les classes héritent par défaut de la classe Object. L'ensemble des relations d'héritage forme alors un arbre dont la racine est la classe Object. Cet arbre est en fait une partie d'un treillis contenant l'ensemble des éléments du langage, dont les classes et interfaces. Les bornes de ce treillis ne seront pas définies par souci de simplification. Son ordre est détaillé par la suite. Ce treillis permet d'assurer les propriétés de la relation d'héritage.

4.1.3 Représentation dans le fichier CAP

Deux composants sont concernés, Constant Pool et Class. Le composant Constant Pool contient, entre autres, l'ensemble des références et notamment les références des classes et interfaces. Le composant Class contient les informations sur l'héritage des classes et des interfaces. Afin de ne pas surcharger le cas, nous ne nous intéresserons qu'aux champs nécessaires à la modélisation des contraintes. Ces champs sont détaillés par composant dans les parties suivantes. Ils constituent une sous-partie de la modélisation effectuée. Pour les modéliser, nous utilisons les conventions définies dans le chapitre 2.

4.2 Les champs du composant Constant Pool

Le composant Constant Pool contient l'ensemble des références du fichier CAP. Il en existe six différentes sortes, chacune représentée par une structure particulière. Dans la spécification de ce composant, il n'y pas de distinction entre les classes et les interfaces. Les classes et interfaces sont désignées par le terme générique de classe.

4.2.1 Description dans la spécification

Dans le composant Constant Pool, les références des classes sont contenues dans la structure CONSTANT_Classref. Cette structure est définie comme suit :

```
CONSTANT_Classref {
 u1 tag
 union {
  u2 internal_class_ref
  {
   u1 package_token
   u1 class_token
  } external_class_ref
 } class_ref
 u1 padding
}
```

Cette structure est composée d'un tag, d'une union et d'un padding. L'union est utilisée pour décrire la Class_ref, car ce type de référence existe sous deux formes. Une interne qui désigne le décalage, par rapport au début du composant Class, d'une classe décrite dans le composant. Ce décalage est calculé à l'aide de la position en mémoire de la classe dans le composant Class. L'autre forme est une référence externe qui pointe vers un élément du composant Import. Elle désigne une classe d'une bibliothèque externe qui a été importée par l'application.

4.2.2 Modélisation

L'union Class_ref est d'abord modélisée, comme défini dans la section 2.2.5, par la figure 4.1.

$partition(class\_ref, internal\_class\_ref, external\_class\_ref)$
$ec\_package\_token \in external\_class\_ref \rightarrow u1$
$ec\_class\_token \in external\_class\_ref \rightarrow u1$
$ic\_class\_ref \in internal\_class\_ref \rightarrow u2$

figure 4.1 – modélisation de l'union

Ce contexte fait normalement partie du Constant Pool, mais il est plus tard utilisé dans le composant Class. Il doit donc être, par définition, une extension du contexte racine Common Structures. Le même type de réflexion a conduit l'inclusion Static_ref dans ce contexte racine.

La structure CONSTANT_Classref est ensuite modélisée dans la figure 4.2. Le champ représentant l'union n'est pas explicitement défini dans la spécification. Nous lui avons donc attribué un nom dans notre modèle pour assurer sa cohérence.

$tag \in constant\_classref \rightarrow u1$
$union\_class\_ref \in constant\_classref \rightarrow class\_ref$
$padding \in constant\_classref \rightarrow u1$

figure 4.2 – modélisation du CONSTANT_Classref

Les contraintes de ce composant ne sont pas détaillées, car elles ne sont pas pertinentes dans ce cas d'étude.

4.3 L'héritage entre les classes

4.3.1 Les champs requis

Dans le composant Class, les informations sur les classes sont stockées dans la structure class_info. Dans cette structure, seuls deux champs nous intéressent. Le

champ offset qui permet d'identifier chaque élément de class_info via sa position en mémoire. Il est ajouté lors de la modélisation (cf section 2.2.7) et correspond à une référence interne. Le champ super_class qui contient la référence vers la classe mère. Il est de type Class_ref, car la classe mère peut être interne ou externe. La figure 4.3 détaille la modélisation de ces champs.

$$cc\_ci\_offset \in class\_info \rightarrow internal\_class\_ref$$
$$cc\_ci\_super\_class\_ref \in class\_info \rightarrow class\_ref$$

figure 4.3 – Champs impliqués dans l'héritage entre les classe

4.3.2 Les contraintes

La contrainte définit l'ordre du treillis pour les classes. Elle s'assure que chaque classe mère est définie avant sa fille. Cette vérification est inutile pour les classes mères externes. Par contre, pour les classes définies dans l'application, il est nécessaire que le décalage de la classe mère soit inférieur à celui de sa fille.

$$\begin{aligned}&\forall class \cdot (class \in class\_info \\ &\quad \wedge cc\_ci\_super\_class\_ref(class) \in ran(cc\_ci\_offset) \\ &\quad \Rightarrow ic\_class\_ref(cc\_ci\_offset(class)) > \\ &\qquad ic\_class\_ref(cc\_ci\_super\_class\_ref(class)))\end{aligned}$$

L'ordre du treillis est donc assuré par l'agencement des classes en mémoire. Il permet d'assurer l'absence de cycle dans la relation d'héritage. La transitivité ne peut pas être testée au niveau de la vérification de structure, car elle appartient à la vérification de type.

4.3.3 Les mutations obtenues

La mutation de cette contrainte donne les deux cas suivants. La première mutation, représentée dans la figure 4.4, crée une classe héritant d'elle-même. Elle rend donc la relation d'héritage réflexive. Les mutations utilisées sont celles décrites à la section 1.4.3. La mutation du $\forall$ a été modifiée, car elle entraîne la création d'un $\exists$. Or ce

quantificateur ne concerne qu'un élément de l'ensemble concerné, laissant tous les autres éléments libres de toutes contraintes. Ce comportement entraînerait des fautes multiples que nous ne souhaitons pas. Pour éviter ce problème, le quantificateur ∃ 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.

$$
\begin{aligned}
&\exists class\_mut \cdot (cc\_ci\_super\_class\_ref(class\_mut) \in ran(cc\_ci\_offset) \\
&\quad \wedge ic\_class\_ref(cc\_ci\_offset(class\_mut)) = \\
&\quad ic\_class\_ref(cc\_ci\_super\_class\_ref(class\_mut)) \\
&\quad \wedge \forall class \cdot (class \in class\_info \\
&\qquad \wedge class\_mut \neq class \\
&\qquad \wedge cc\_ci\_super\_class\_ref(class) \in ran(cc\_ci\_offset) \\
&\qquad \Rightarrow ic\_class\_ref(cc\_ci\_offset(class)) > \\
&\qquad\quad ic\_class\_ref(cc\_ci\_super\_class\_ref(class))) \\
&\quad)
\end{aligned}
$$

figure 4.4 – Classe s'héritant d'elle-même

La seconde mutation, représentée dans la figure 4.5, crée une classe héritant d'une classe ayant un décalage supérieur au sien. L'ordre est donc rompu ce qui rend la relation d'héritage cyclique.

$$
\begin{aligned}
&\exists class\_mut \cdot (cc\_ci\_super\_class\_ref(class\_mut) \in ran(cc\_ci\_offset) \\
&\quad \wedge ic\_class\_ref(cc\_ci\_offset(class\_mut)) < \\
&\quad ic\_class\_ref(cc\_ci\_super\_class\_ref(class\_mut)) \\
&\quad \wedge \forall class \cdot (class \in class\_info \\
&\qquad \wedge class\_mut \neq class \\
&\qquad \wedge cc\_ci\_super\_class\_ref(class) \in ran(cc\_ci\_offset) \\
&\qquad \Rightarrow ic\_class\_ref(cc\_ci\_offset(class)) > \\
&\qquad\quad ic\_class\_ref(cc\_ci\_super\_class\_ref(class))) \\
&\quad)
\end{aligned}
$$

figure 4.5 – Inversion de l'ordre des décalages

4.4 L'héritage entre les interfaces

4.4.1 Les champs requis

Dans le composant Class, les informations sur les interfaces sont stockées dans la structure interface_info. Dans cette structure, trois champs sont impliqués dans l'héritage. Comme pour la structure Class_info, nous avons ajouté à interface_info un champ offset. L'héritage des interfaces est multiple, il est donc stocké sous la forme d'un tableau. Celui-ci contient l'ensemble des parents directs et indirects d'une interface ,sauf la classe Object. Sa taille est contenue dans le champ cc_ii_bitfield_interface_count. Comme dans le cas des classes, les éléments de ce tableau sont des Class_ref. Le tableau est défini dans le champ cc_ii_superinterfaces. Il peut être vide si l'interface n'hérite que de Object. La figure suivante décrit la modélisation de ses champs.

$$
\begin{array}{l}
cc\_ii\_offset \in interface\_info \rightarrow internal\_class\_ref \\
cc\_ii\_bitfield\_interface\_count \in interface\_info \rightarrow bit\_4 \\
cc\_ii\_superinterfaces \in interface\_info \rightarrow (bit\_4 \nrightarrow class\_ref) \\
\forall interface \cdot (interface \in interface\_info \\
\quad \Rightarrow dom(cc\_ii\_superinterfaces(interface)) = \\
\qquad 1 \,..\, cc\_ii\_bitfield\_interface\_count(interface))
\end{array}
$$

4.4.2 Les contraintes

Les interfaces possèdent aussi une contrainte appliquant un ordre dans le treillis. Comme pour les classes, cet ordre s'applique sur les ensembles des interfaces définies dans l'application. Chaque interface doit avoir un décalage plus grand que celui de ses parents.

$$
\begin{array}{l}
\forall interface \cdot (\\
\quad interface \in interface\_info \\
\quad \Rightarrow (\forall sup \cdot (\\
\qquad sup \in ran(cc\_ii\_superinterfaces(interface)) \\
\qquad \wedge\, sup \in ran(cc\_ii\_offset) \\
\qquad \Rightarrow ic\_class\_ref(cc\_ii\_offset(interface)) > ic\_class\_ref(sup) \\
\qquad)))
\end{array}
$$

La notion d'ordre est explicitée dans la spécification ; par contre, il n'y a aucune information sur l'unicité des éléments du tableau des super-interfaces. Or, cette unicité est capitale, car sans elle on ne peut pas garantir la relation d'ordre. En effet, si une interface apparaît deux fois dans la fermeture transitive, il peut s'agir d'une boucle dans la relation d'héritage. Nous avons donc choisi de rajouter cette contrainte même si elle n'est pas présente dans la spécification pour pouvoir assurer la relation d'ordre. Elle nous permet ainsi d'avoir des tests plus complets.

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

4.4.3 Les mutations obtenues

Comme pour les classes, les contraintes appliquées aux interfaces sont mutées à l'aide des formules contenues à la section 1.4.3.

$$
\begin{array}{l}
\exists interface, sup \cdot (interface \in interface\_info \\
\quad \wedge cc\_ii\_bitfield\_interface\_count(interface) > 0 \\
\quad \wedge sup \in ran(cc\_ii\_superinterfaces(interface)) \\
\quad \wedge sup \in ran(cc\_ii\_offset) \\
\quad \wedge ic\_class\_ref(cc\_ii\_offset(interface)) = ic\_class\_ref(sup) \\
\quad \wedge \forall int \cdot (int \in interface\_info \\
\qquad \wedge cc\_ii\_bitfield\_interface\_count(int) > 0 \\
\qquad \wedge interface \neq int \\
\qquad \Rightarrow \forall sup1 \cdot (sup1 \in ran(cc\_ii\_superinterfaces(int)) \\
\qquad\quad \wedge sup1 \in ran(cc\_ii\_offset) \\
\qquad\quad \Rightarrow ic\_class\_ref(cc\_ii\_offset(int)) = ic\_class\_ref(sup1) \\
\quad)))
\end{array}
$$

figure 4.6 – Interface héritant d'elle-même

$$
\begin{array}{l}
\exists interface, sup \cdot (interface \in interface\_info \\
\quad \wedge cc\_ii\_bitfield\_interface\_count(interface) > 0 \\
\quad \wedge sup \in ran(cc\_ii\_superinterfaces(interface)) \\
\quad \wedge sup \in ran(cc\_ii\_offset) \\
\quad \wedge ic\_class\_ref(cc\_ii\_offset(interface)) < ic\_class\_ref(sup) \\
\quad \wedge \forall int \cdot (int \in interface\_info \\
\qquad \wedge cc\_ii\_bitfield\_interface\_count(int) > 0 \\
\qquad \wedge interface \neq int \\
\qquad \Rightarrow \forall sup1 \cdot (sup1 \in ran(cc\_ii\_superinterfaces(int)) \\
\qquad\quad \wedge sup1 \in ran(cc\_ii\_offset) \\
\qquad\quad \Rightarrow ic\_class\_ref(cc\_ii\_offset(int)) < ic\_class\_ref(sup1) \\
\quad)))
\end{array}
$$

figure 4.7 – Inversion de l'ordre de décalages

$$
\begin{aligned}
&\exists interface, sup1, sup2 \cdot (interface \in interface\_info \\
&\quad \wedge cc\_ii\_bitfield\_interface\_count(interface) > 0 \\
&\quad \wedge sup1 \in dom(cc\_ii\_superinterfaces(interface)) \\
&\quad \wedge sup2 \in dom(cc\_ii\_superinterfaces(interface)) \\
&\quad \wedge sup1 \neq sup2 \\
&\quad \wedge cc\_ii\_superinterfaces(interface)(sup1) = \\
&\qquad cc\_ii\_superinterfaces(interface)(sup2) \\
&\quad \wedge \forall int \cdot (int \in interface\_info \\
&\qquad \wedge cc\_ii\_bitfield\_interface\_count(int) > 0 \\
&\qquad \wedge interface \neq int \\
&\qquad\quad \Rightarrow (\forall supi1, supi2 \cdot (sup1 \in dom(cc\_ii\_superinterfaces(interface)) \\
&\qquad\qquad \wedge sup2 \in dom(cc\_ii\_superinterfaces(interface)) \\
&\qquad\qquad \wedge sup1 \neq sup2 \\
&\qquad\qquad \Rightarrow cc\_ii\_superinterfaces(interface)(supi1) \neq \\
&\qquad\qquad\quad cc\_ii\_superinterfaces(interface)(supi2) \\
&\qquad\qquad\quad))))
\end{aligned}
$$

figure 4.8 – Duplication de l'héritage

4.5 Résultats

Pour obtenir les tests abstraits, nous soumettons au VTG le contexte contenant la description des champs requis et les contraintes. Une de ses contraintes est mutée ce qui va amener à la création du contexte mutant. Ce problème entraîne une explosion combinatoire dans ProB. Il est donc nécessaire d'utiliser un contexte Prefilled. Celui-ci contient une énumération pour chacun des trois ensembles du cas d'étude. ProB va compléter ces informations pour générer les tests abstraits. Dans le cas d'étude, nous allons utiliser un contexte de Prefilled pour le cas des classes et un pour celui des interfaces.

4.5.1 Les classes

Voici le contexte Prefilled utilisé pour générer les tests abstraits des classes.

$partition(class\_info, \{class\_info1\}, \{class\_info2\})$
$partition(internal\_ref, \{internal\_ref1\}, \{internal\_ref2\})$
$partition(external\_ref, \{external\_ref1\})$

Voici la partie générée par ProB avec le modèle non muté

$\{external\_class\_ref\_1 \mapsto 0\} \subseteq ec\_package\_token$
$\{external\_class\_ref\_1 \mapsto 0\} \subseteq ec\_class\_token$
$\{internal\_ref1 \mapsto 1, internal\_ref2 \mapsto 2\} \subseteq ic\_class\_ref$
$\{class\_info1 \mapsto internal\_class\_ref1, class\_info2 \mapsto internal\_class\_ref2\}$
$\subseteq cc\_ci\_offset$
$\{class\_info1 \mapsto external\_class\_ref1, class\_info2 \mapsto internal\_class\_ref1\}$
$\subseteq cc\_ci\_super\_class\_ref$

Ce contexte correspond à la figure 4.9. Les bulles représentent des classes. La bulle en pointillé décrit la classe importée Object. Comme elle est importée, c'est une référence externe. Dans cette figure, les classes sont organisées de droite à gauche suivant l'ordre croissant de leur référence. Cette figure sert de référence pour constater les modifications issues des mutations.

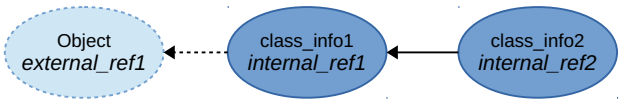

figure 4.9 – Situation normale d'héritage pour une classe

Les tests abstraits suivants ne contiennent que la partie générée par ProB. L'effet de la mutation est mis en rouge dans le contexte et illustré par une figure. Les deux mutations concernent l'ordre associé à l'héritage. Voici le test, généré à partir de la

mutation, contenu dans la figure **??**.

$$\{external\_class\_ref\_1 \mapsto 0\} \subseteq ec\_package\_token$$
$$\{external\_class\_ref\_1 \mapsto 0\} \subseteq ec\_class\_token$$
$$\{internal\_ref1 \mapsto 1, internal\_ref2 \mapsto 2\} \subseteq ic\_class\_ref$$
$$\{class\_info1 \mapsto internal\_ref1, class\_info2 \mapsto internal\_ref2\} \subseteq cc\_ci\_offset$$
$$\{class\_info1 \mapsto external\_class\_ref1, class\_info2 \mapsto internal\_ref2\} \subseteq cc\_ci\_super\_class\_ref$$

La figure 4.10 montre l'effet de la mutation. La classe représentée par class_info2 hérite d'elle-même. Dans le contexte, on peut le constater dans la partie en rouge, l'offset et la référence de la classe mère de class_info2sont identiques.

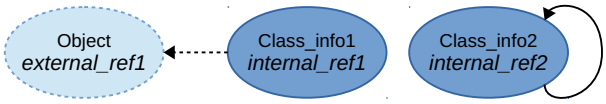

figure 4.10 – Classe héritant d'elle-même

Voici le test, généré à partir de la mutation, contenu dans la figure 4.5.

$$\{external\_ref\_1 \mapsto 0\} \subseteq ec\_package\_token$$
$$\{external\_ref\_1 \mapsto 0\} \subseteq ec\_class\_token$$
$$\{internal\_ref1 \mapsto 1, internal\_ref2 \mapsto 2\} \subseteq ic\_class\_ref$$
$$\{class\_info1 \mapsto internal\_ref1, class\_info2 \mapsto internal\_ref1\} \subseteq cc\_ci\_offset$$
$$\{class\_info1 \mapsto external\_ref1, class\_info2 \mapsto internal\_ref2\} \subseteq cc\_ci\_super\_class\_ref$$

La figure 4.11 montre l'effet de la mutation. Ici, c'est la référence et donc la position en mémoire de class_info1 et de class_info2 qui ont été inversées. Ce qui a pour résultat

d'inverser la relation.

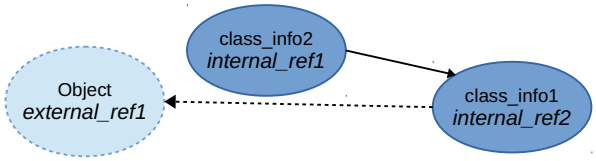

figure 4.11 – Inversion de l'ordre des décalages

4.5.2 Les interfaces

Nous avons utilisé la même méthodologie que pour les classes.

$partition(interface\_info, \{interface\_info1\}, \{interface\_info2\})$
$partition(internal\_ref, \{internal\_ref1\}, \{internal\_ref2\})$
$partition(external\_ref, \{external\_ref1\})$

Voici la partie générée par ProB avec le modèle non muté

$\{internal\_ref1 \mapsto 1, internal\_ref2 \mapsto 2\} \subseteq ic\_class\_ref$
$\{interface\_info1 \mapsto internal\_ref1, interface\_info2 \mapsto internal\_ref2\}$
$\subseteq cc\_ii\_offset$
$\{interface\_info1 \mapsto 0, interface\_info2 \mapsto 1\} \subseteq cc\_ii\_bitfield\_interface\_count$
$\{interface\_info1 \mapsto \{\}, interface\_info2 \mapsto \{(1 \mapsto interface\_info1)\}\}$
$\subseteq cc\_ii\_superinterfaces$

Nous partons de la figure 4.12 représentant une situation normale pour voir les variations dues aux mutations.

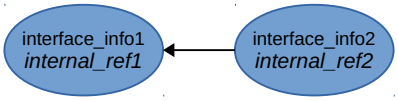

figure 4.12 – Situation normale d'héritage pour une interface

Le test suivant est généré à partir de la mutation issue de la figure 4.8. Cette mutation permet à une interface d'hériter plusieurs fois de la même interface. Les effets de la mutation sont en rouge.

$$\{internal\_ref1 \mapsto 1, internal\_ref2 \mapsto 2\} \subseteq ic\_class\_ref$$
$$\{interface\_info1 \mapsto internal\_ref1, interface\_info2 \mapsto internal\_ref2\} \subseteq cc\_ii\_offset$$
$$\{interface\_info1 \mapsto 0, interface\_info2 \mapsto 2\} \subseteq cc\_ii\_bitfield\_interface\_count$$
$$\{interface\_info1 \mapsto \{\}, interface\_info2 \mapsto \{(1 \mapsto interface\_info1), (2 \mapsto interface\_info1)\}\} \subseteq cc\_ii\_superinterfaces$$

Dans ce cas, on peut voir que interface_info2 hérite deux fois de la même interface.

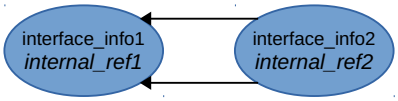

figure 4.13 – Duplication de l'héritage

Les deux mutations suivantes modifient l'ordre associé à l'héritage. Il s'agit d'une transposition aux interfaces des cas vus dans la section précédente. La mutation de

la figure 4.6.

$\{internal\_ref1 \mapsto 1, internal\_ref2 \mapsto 2\} \subseteq ic\_class\_ref$
$\{interface\_info1 \mapsto internal\_ref1, interface\_info2 \mapsto internal\_ref2\}$
$\subseteq cc\_ii\_offset$
$\{interface\_info1 \mapsto 0, interface\_info2 \mapsto 2\} \subseteq cc\_ii\_bitfield\_interface\_count$
$\{interface\_info1 \mapsto \{\}, interface\_info2 \mapsto \{(1 \mapsto interface\_info2)\}\}$
$\subseteq cc\_ii\_superinterfaces$

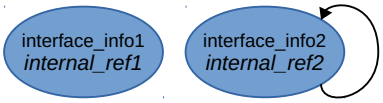

figure 4.14 – Interface héritant d'elle-même

La mutation de la figure 4.7 permet d'obtenir le prefilled suivant.

$\{internal\_ref1 \mapsto 1, internal\_ref2 \mapsto 2\} \subseteq ic\_class\_ref$
$\{interface\_info1 \mapsto internal\_class\_ref2,$
$interface\_info2 \mapsto internal\_class\_ref1\}$
$\subseteq cc\_ii\_offset$
$\{interface\_info1 \mapsto 0, interface\_info2 \mapsto 1\} \subseteq cc\_ii\_bitfield\_interface\_count$
$\{interface\_info2 \mapsto \{(1 \mapsto interface\_info1)\},$
$interface\_info1 \mapsto \{\}\} \quad \subseteq cc\_ii\_superinterfaces$

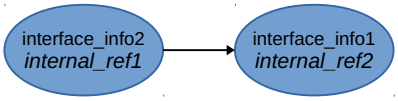

figure 4.15 – Inversion de l'ordre des décalages

4.6 Conclusion

La preuve de concept est donc apportée. Le VTG permet bien d'extraire l'ensemble des mutations désirées à partir de notre modélisation. De plus, ProB génère l'ensemble des tests attendus en une dizaine de millisecondes. Il manque cependant une partie de la relation d'héritage que nous n'avons pas pu tester : l'implémentation des interfaces par les classes. En effet, une interface ne peut être implémentée par une classe que si elle n'est pas implémentée par l'ensemble des parents de cette classe. Cette contrainte repose donc sur une fermeture transitive de la relation d'héritage qui permet de déterminer l'ensemble des parents d'une classe. Cependant, la mutation d'une contrainte contenant une fermeture transitive entraîne une explosion combinatoire dans ProB. Cette limitation n'a pour l'instant pas pu être dépassée.

Conclusion

Nous avons mis en place une modélisation du vérifieur de structure. Celle-ci donne une formalisation de l'ensemble des champs du fichier CAP. Nous avons aussi créé une organisation et une dénomination des champs qui permet une transition facile entre la spécification et la modélisation. La formalisation des champs tient compte des problèmes d'explosion combinatoire dans ProB. En effet, une recherche a été faite dans la déclaration des champs pour que ceux-ci aient le moins d'impact possible. Cette formalisation a été utilisée sur un cas d'étude pour montrer sa validité. Le modèle a montré la capacité du VTG à générer les mutants attendus. De plus, ProB a pu extraire l'ensemble des tests abstraits attendu.

La modélisation a permis de mettre en évidence le manque de clarté de la spécification. Nous avons pu constater que plusieurs notations coexistent ce qui rend la compréhension difficile. De plus, le cas d'étude nous a permis de relever une contradiction entre la spécification et la grammaire Java. La spécification déclare que les classes pouvaient ne pas avoir de classe mère, ce qui va à l'encontre d'une des règles principales de la grammaire Java dans laquelle chaque classe possède une classe mère (par défaut il s'agit d'Object).

Le cas d'étude nous a permis de mettre en évidence des imprécisions de la spécification. Premièrement, la définition de l'héritage des interfaces ne permet pas de garantir l'absence de cycle. Deuxièmement, le cas de l'héritage entre classes et interface n'est pas entièrement défini. L'exemple de la spécification montre bien l'interdiction d'avoir une interface commune ente une classe et sa classe mère. En revanche, elle ne généralise pas à l'ensemble des classes parentes d'une classe.

Le cas d'étude a mis en lumière une limitation due à la fermeture dans ProB. En effet, ProB cherche à générer l'ensemble des états possibles à partir des contraintes

ce qui engendre à une explosion combinatoire. Une approche différente permettrait de contourner le problème. Plutôt que de générer l'ensemble des cas, il faudrait pouvoir limiter la recherche de ProB sur certains axiomes. Par exemple, limiter la recherche sur les axiomes qui définissent la position en mémoire, car la combinaison des positions en mémoire possibles n'est que peu pertinente dans notre cas et consomme beaucoup de temps de calcul. Cette amélioration permettrait de concentrer la résolution sur les points les plus intéressants.

Notre modélisation est, pour l'instant, incomplète. L'ensemble des champs du fichier CAP a bien été défini. Cependant, l'ensemble des contraintes décrites dans la spécification n'a pu être modélisé durant la maîtrise. De plus, notre modélisation ne prend pas en compte la position des éléments en mémoire, ce qui diminue la couverture de nos tests. La modélisation devra donc être complétée pour pouvoir atteindre ces deux objectifs dans de futurs travaux. L'application, qui permet de concrétiser les tests abstraits générés par ProB en applications Java Card, n'est pas encore développée.

Annexe A

Modélisation du composant

Directory

CONTEXT 1_Str_02_dc_Static_Field_Size
EXTENDS 0_Com_PrimitifTypes
SETS

$dic\_sfsi\_static\_field\_size\_info$

CONSTANTS

$dic\_sfsi\_image\_size$

$dic\_sfsi\_array\_init\_count$

$dic\_sfsi\_array\_init\_size$

AXIOMS

axm1 : $finite(dic\_sfsi\_static\_field\_size\_info)$

axm2 : $dic\_sfsi\_image\_size \in dic\_sfsi\_static\_field\_size\_info \rightarrow u2$

axm3 : $dic\_sfsi\_array\_init\_count \in$
$dic\_sfsi\_static\_field\_size\_info \rightarrow u2$

axm4 : $dic\_sfsi\_array\_init\_size \in$
$dic\_sfsi\_static\_field\_size\_info \rightarrow u2$

END

CONTEXT 1_Str_02_dc_Custom_Component_Info
EXTENDS 0_Com_PrimitifTypes
SETS

$dic\_cci\_custom\_component\_info$

CONSTANTS

$dic\_cci\_component\_tag$

dic_cci_size

$dic\_cci\_AID\_length$

dic_cci_AID

AXIOMS

axm1 : $finite(dic\_cci\_custom\_component\_info)$

axm2 : $dic\_cci\_component\_tag \in dic\_cci\_custom\_component\_info \rightarrow u1$

axm3 : $dic\_cci\_size \in dic\_cci\_custom\_component\_info \rightarrow u2$

axm4 : $dic\_cci\_AID\_length \in dic\_cci\_custom\_component\_info \rightarrow u1$

axm5 : $dic\_cci\_AID \in dic\_cci\_custom\_component\_info \rightarrow (u1 \nrightarrow u1)$

axm6 : $\forall d \cdot (d \in dic\_cci\_custom\_component\_info$
$\Rightarrow dom(dic\_cci\_AID(d)) = 1 \mathinner{..} dic\_cci\_AID\_length(d))$

END

CONTEXT 1_Str_02_Directory_Component
EXTENDS 1_Str_02_dc_Static_Field_Size
CONSTANTS

dic_tag

dic_size

$dic\_component\_sizes$

$dic\_static\_field\_size$

dic_import_count

dic_applet_count

dic_custom_count

$dic\_custom\_component$

AXIOMS

axm1: $dic\_tag \in u1$

axm2: $dic\_size \in u2$

axm3: $dic\_component\_sizes \in 0 \mathinner{..} 11 \rightarrow u2$

axm4: $dic\_static\_field\_size = dic\_sfsi\_static\_field\_size\_info$

axm5: $dic\_import\_count \in u1$

axm6: $dic\_applet\_count \in u1$

axm7: $dic\_custom\_count \in u1$

axm8: $dic\_custom\_component \in$
$1 \mathinner{..} dic\_custom\_count \rightarrow dic\_cci\_custom\_component\_info$

END

Bibliographie

[1] Paul E. Black.
« Test generation using model checking and specification mutation ».
IT Professional, 16(2):17–21, 2014.

[2] Sandrine Blazy et Xavier Leroy.
« Mechanized Semantics for the Clight Subset of the C Language ».
Journal of Automated Reasoning, 43(3):263–288, 2009.

[3] Ludovic Casset, Lilian Burdy et Antoine Requet.
« Formal development of an embedded verifier for Java Card byte code ».
Dans *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 51–56, 2002.

[4] Andreas Gal, Christian W. Probst et Michael Franz.
« A Denial of Service Attack on the Java Bytecode Verifier ».
Rapport Technique, University of California, 2003.

[5] Karine Gandolfi, Christophe Mourtel et Francis Olivier.
« Electromagnetic analysis : Concrete results ».
Dans *Cryptographic Hardware and Embedded Systems—CHES 2001*, pages 251–261. Springer, 2001.

[6] Pieter H. Hartel et Luc Moreau.
« Formalizing the safety of Java, the Java virtual machine, and Java card », 2001.

[7] Michael Hohmuth et Hendrik Tews.
« The semantics of C++ data types : Towards verifying low-level system components ».
Dans *TPHOLs*, pages 127–144, 2003.

[8] Julien Iguchi-Cartigny et Jean-Louis Lanet.
« Developing a Trojan applets in a smart card ».
Journal in Computer Virology, 6(4):343–351, 2010.

[9] C. Marché, C. Paulin-Mohring et X. Urbain.
« The KRAKATOA tool for certificationof JAVA/JAVACARD programs annotated in JML ».
Dans *Journal of Logic and Algebraic Programming*, volume 58, pages 89–106, 2004.

[10] D. Marinov et S. Khurshid.
« TestEra : a novel framework for automated testing of Java programs ».
Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001), pages 22–31, 2001.

[11] Aleksandar Milićević, Saša Misailović, Darko Marinov et Sarfraz Khurshid.
« Korat : A tool for generating structurally complex test inputs ».
Dans *Proceedings - International Conference on Software Engineering*, pages 771–774, 2007.

[12] Aymerick Savary.
« Génération de tests de vulnérabilité pour le vérifieur de bytecode Java Card ».
Mémoire de maîtrise, Université de Sherbrooke, Département d'informatique, Sherbrooke, Québec, Canada, 2013.

[13] Aymerick Savary, Marc Frappier et Jean-Louis Lanet.
« Detecting Vulnerabilities in Java-Card Bytecode Verifiers Using Model-Based Testing ».
Dans *Integrated Formal Methods*, pages 223–237. Springer, 2013.

[14] Aymerick Savary, Marc Frappier, Michael Leuschel et Jean-Louis Lanet.
« Model-Based Robustness Testing in Event-B Using Mutation ».
Dans *Software Engineering and Formal Methods*, pages 132–147. Springer, 2015.

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