HAL open science

Modélisation par contraintes de programmes en bytecode Java pour la génération automatique de tests

Florence Charreteur

▶ To cite this version:

Florence Charreteur. Modélisation par contraintes de programmes en bytecode Java pour la génération automatique de tests. Informatique [cs]. Université Européenne de Bretagne, 2010. Français. NNT : . tel-00497785v2

HAL Id: tel-00497785

https://theses.hal.science/tel-00497785v2

Submitted on 29 Sep 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NonCommercial - ShareAlike 4.0 International License

N° d'ordre : 4107 **ANNÉE 2010**

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale Matisse

présentée par

Florence CHARRETEUR SCHADLE

préparée à l'unité de recherche 6074 IRISA
Institut de Recherche en Informatique et Systèmes Aléatoires
IFSIC

Modélisation par contraintes de programmes en bytecode Java pour la génération automatique de tests

Thèse soutenue à Rennes
Le 9 mars 2010

devant le jury composé de :

Mireille DUCASSE
Professeur à l'INSA de Rennes / *président*

Olga KOUCHNARENKO
Professeur à l'Université de Franche-Comté / *rapporteur*

Alexander PRETSCHNER
Professeur à la Technische Universität Kaiserslautern / *rapporteur*

Sébastien BARDIN
Chercheur au CEA Saclay / *examinateur*

Thomas JENSEN
Directeur de recherche CNRS / *directeur de thèse*

Arnaud GOTLIEB
Chargé de recherche INRIA / *co-directeur de thèse*

Table des matières

Remerciements **1**

Introduction **7**

I Contexte et état de l'Art **13**

1 Test des programmes **15**
1.1 Test basé sur le modèle 15
1.1.1 Une taxonomie du test à base de modèle 15
1.1.2 Description de quelques outils 17
1.2 Test basé sur le code 17
1.2.1 Prérequis 17
1.2.2 Exécution symbolique 18
1.2.3 Exécution symbolique et langages de programmation impératifs ou orientés objet 19
1.2.4 Méthodes exploitant l'exécution symbolique 20
1.2.5 Exécution symbolique et model-checking 20
1.2.6 Exécution symbolique dynamique 22
1.2.7 Approche orientée but 28
1.2.8 Bilan 30

2 Mécanismes de la programmation par contraintes **33**
2.1 Problème de satisfaction de contraintes sur des domaines finis 33
2.2 Résolution d'un système de contraintes sur des domaines finis 34
2.2.1 Propagation de contraintes 34
2.2.2 Énumération des variables 37

II Modélisation par contraintes du bytecode Java pour la génération de données de test **39**

3 Intérêt et aperçu du modèle à contraintes d'une JVM **41**
3.1 Rappel des contributions 41
3.2 Intérêt du modèle 42
3.3 Aperçu du modèle 43
3.3.1 Exemple détaillé 44
3.3.2 Aperçu des EMC et des relations qui les lient 47
3.4 Difficultés principales pour la modélisation 50
3.4.1 Détermination de la forme de la mémoire 50

3.4.2 Héritage et polymorphisme 51

4 Modèle mémoire d'une JVM 53

4.1 Description d'une JVM 53

4.1.1 Type des données manipulées par une JVM 53

4.1.2 Description des zones de stockage des données d'une JVM 54

4.1.3 Définition d'un état de la mémoire 55

4.2 Notre modèle mémoire d'une JVM 55

4.2.1 Modélisation des registres et de la pile d'opérandes 55

4.2.2 Modélisation du tas et des objets 55

4.2.3 Modélisation des variables de type primitif ou référence 57

4.2.4 Etat mémoire sous contraintes (EMC) 57

5 Modélisation des instructions 61

5.1 Remarques préliminaires sur la modélisation proposée 61

5.1.1 Une sémantique sans erreurs 61

5.1.2 Instructions modélisées 61

5.1.3 Informations sur la présentation de la modélisation 62

5.2 Contraintes portant sur les VTPR 62

5.2.1 Contraintes portant sur les entiers 62

5.2.2 Contraintes portant sur les références 63

5.2.3 Notation de la contrainte d'égalité 64

5.3 Traduction des instructions en contraintes 65

5.3.1 Principe de l'interprétation en contraintes 65

5.3.2 Instructions d'une JVM et informations de type 65

5.3.3 Instructions d'accès aux registres et de modification des registres . . 66

5.3.4 Instructions de manipulation de la pile 67

5.3.5 Instructions arithmétiques 67

5.3.6 Instructions de branchement conditionnel 67

5.3.7 Instructions de manipulation d'instances de classe 68

5.3.8 Appels de méthodes et de constructeurs 76

6 Génération de données de test 79

6.1 Un parcours de graphe en arrière et incrémental 79

6.1.1 Le choix d'un parcours en arrière 79

6.1.2 Un parcours incrémental 79

6.1.3 Stratégies pour le parcours de graphe 81

6.2 Enumération de la mémoire d'entrée 82

6.3 Correction et complétude de la méthode de génération 83

7 Comparaison de notre approche avec l'état de l'Art 85

7.1 Comparaison avec les approches exploitant le model-checking et l'exécution symbolique 85

7.2 Comparaison avec les méthodes basées sur l'exécution symbolique dynamique 85

7.3 Comparaison avec l'approche orientée but 87

III Validation expérimentale **91**

8 Description de l'outil JAUT **93**
8.1 Principe de fonctionnement 93
8.2 Architecture détaillée 94

9 Validation expérimentale **95**
9.1 Programmes avec arithmétique 97
9.1.1 Programme trityp 98
9.1.2 Programme avec boucle 98
9.2 Programmes avec structures de données 100
9.2.1 Listes doublement chaînées 100
9.2.2 Arbres colorés 101
9.3 Limitation du parcours de graphe 102
9.3.1 Josephus_m 102
9.4 Conclusion sur la validation de l'approche 104

IV Conclusion et perspectives **107**

10 Conclusion **109**

11 Perspectives **111**
11.1 Améliorer le parcours du graphe de flot de contrôle 111
11.2 Étendre le langage traité 112
11.2.1 Gestion des nombres flottants 112
11.2.2 Traitement des tableaux 112
11.2.3 Autres extensions 113
11.3 Génération de séquences d'initialisation 113
11.4 Prendre en compte des spécifications 114
11.5 Améliorer le traitement des appels de méthode 115
11.6 Étendre l'approche pour couvrir d'autres critères 115
11.7 Rendre les opérateurs à contraintes sur le tas plus déductifs en enrichissant le modèle mémoire 116
11.8 Exploiter le modèle de mémoire dans une approche orientée but 116

Annexe A **119**

Annexe B **123**

Bibliographie **137**

Table des figures **139**

Table des listings **141**

Introduction

Depuis de nombreuses années, l'accroissement constant de la complexité des logiciels développés augmente le risque d'erreurs, la vérification des logiciels est donc indispensable pour maintenir un certain niveau de qualité et de fiabilité. Cette étape essentielle du développement des logiciels représente d'ailleurs une part importante de leur coût de production, et en faire son économie expose à un risque d'erreurs plus élevé. Les principales approches pour la vérification de logiciels sont la preuve et le test.

Les *méthodes de vérification basées sur la preuve* s'appuient sur la logique mathématique et permettent d'avoir une très forte assurance de l'absence de certaines classes d'erreurs dans un logiciel. Elles donnent des moyens pour vérifier qu'une exigence est satisfaite par tous les comportements possibles du système, et ont donc l'avantage de l'exhaustivité. Elles peuvent s'intégrer à différentes phases du processus de création du produit. La méthode B s'utilise dès la spécification. Elle assure que toutes les propriétés prouvées sur des modèles abstraits sont conservées sur les modèles concrets, et produit donc par raffinements successifs un programme conforme par construction à la spécification. Utilisables pendant ou après le développement, les analyses statiques par interprétation abstraite permettent d'assurer que des propriétés sont satisfaites en des points de programme pour toutes les exécutions. Les méthodes de vérification basées sur la preuve garantissent un très haut niveau de vérification des logiciels, mais elles sont difficiles à mettre en œuvre dans le cadre industriel. Ainsi, les analyses statiques sont soit précises mais coûteuses en temps et en ressources matérielles, soit moins précises mais génèrent davantage de fausses alertes dont l'étude est coûteuse en terme de ressources humaines. Les méthodes de vérification basées sur la preuve sont de ce fait peu utilisées actuellement dans l'industrie hormis pour les logiciels les plus critiques.

Le *test logiciel* consiste à stimuler uniquement certains comportements du programme. Pour cela, le programme est exécuté avec des données d'entrée particulières, et, pour chacune de ces exécutions, un *oracle*, humain ou automatisé, détermine si le comportement du programme est conforme à sa spécification. Le test ne permet pas de garantir l'absence de fautes mais augmente la confiance portée au programme, et s'applique à tous les stades du cycle de production du logiciel. Le test logiciel est la méthode de vérification la plus diffusée dans l'industrie car même s'il offre moins de garanties que la vérification basée sur la preuve, il est plus facile à mettre en œuvre. Sa pratique reste cependant trop souvent artisanale. Face à ce constat, des modèles et des théories sont proposés pour rendre l'activité plus rigoureuse et permettre d'envisager une automatisation du processus de test, dans le but de réduire les coûts et de garantir sa qualité [Ber07].
Le test exhaustif qui consiste à soumettre à un programme toutes les données d'entrée possibles est généralement impraticable car il y en a un très grand nombre, voire une infinité. Des *critères de test* ont donc été définis pour orienter la sélection des données d'entrée, en

décrivant les propriétés que l'ensemble de données sélectionné doit satisfaire. Les critères de test permettent de caractériser a posteriori la qualité d'un ensemble de données de test, mais aussi de guider la génération de ces données.
Le test peut se faire à partir de modèle ou à partir de code. Le *test basé sur le modèle* vise à couvrir des éléments d'un modèle qui spécifie le comportement attendu du programme sous test. Ce modèle peut être une spécification en UML, un automate d'états finis, etc. Le *test basé sur le code* vise quant à lui à couvrir des éléments du code du programme (instructions, branches...). Le test basé sur le modèle présente l'avantage de permettre la création des données de test dès qu'un modèle du logiciel est disponible sans attendre le code. Pour le mettre en place lors de la création d'un produit, les spécifications doivent être exprimées sous forme de modèle formel. La construction de ce modèle peut être coûteuse, mais la formalisation de la spécification laisse moins de place aux ambiguïtés ce qui est profitable aux programmeurs. Par contre, dans le cas d'un code existant, il n'existe pas toujours de modèle du logiciel, et construire un modèle a posteriori n'est pas forcément intéressant. Quand un modèle est disponible, le test basé sur le modèle et le test basé sur le code sont complémentaires. D'une part, utiliser uniquement le code pour trouver les données de test risque de propager les éventuelles erreurs d'interprétation de la spécification faites par les développeurs. D'autre part, utiliser uniquement la spécification n'est pas toujours suffisant car la couverture de modèle n'implique pas forcément la couverture de code. En effet, le code contient des optimisations, non nécessairement spécifiées dans le modèle, qu'il convient de tester. Par exemple, dans le cadre des logiciels embarqués, les contraintes en ressources ne sont pas forcément spécifiées. Par conséquent, une méthode préconisée quand un modèle est disponible est de compléter l'ensemble de données de test généré pour couvrir le modèle afin de satisfaire également des critères de couverture du code.

Problématique : le test basé sur le code des programmes orientés objet

Les langages orientés objet sont devenus prédominants pour développer des logiciels, car ils favorisent l'écriture de programmes modulaires, réutilisables, extensibles et plus faciles à maintenir. Tester un logiciel écrit dans un langage orienté objet pose différents problèmes [Bin99, MS01]. Pour le test d'intégration, il faut choisir un ordre dans lequel tester les différentes classes en tenant compte des liens très forts qui peuvent exister entre elles en raison de l'agrégation, de la composition et de l'héritage. Une fois déterminé un ordre dans lequel tester les classes, les tester au niveau unitaire pose aussi les problèmes suivants : il faut déterminer et générer la forme de la mémoire en entrée, gérer l'héritage ainsi que le polymorphisme.

Pour pouvoir couvrir un objectif de test (branche, instruction, chemin), il faut inférer à partir du code quelle forme doit avoir la mémoire en entrée de la méthode sous test, c'est-à-dire quels objets doit contenir la mémoire et comment ces objets sont éventuellement liés entre eux. Par exemple, atteindre une instruction d'un programme peut requérir de passer en paramètre une liste cyclique, en ce cas les objets se référencent les uns les autres. Ce problème difficile de **détermination de la forme des structures de données** en entrée de programme pour couvrir un objectif de test, que nous appelons problème *Mem_shape*, n'est pas propre aux langages orientés objet. Il se retrouve pour tous les langages permettant l'allocation dynamique.

De plus, déterminer une donnée de test en entrée, et en particulier la forme de la mémoire, ne suffit pas pour atteindre un objectif : l'**encapsulation** des informations, propre aux langages orientés objet, peut gêner la génération de la donnée. En effet, les attributs des objets sont souvent privés, et placer les objets dans un certain état depuis une classe de test extérieure requiert de déterminer quels appels aux constructeurs et méthodes publics effectuer. L'état peut, de plus, s'avérer inaccessible.

L'**héritage** permet de définir de nouveaux attributs et de nouvelles méthodes pour la classe dérivée, qui viennent s'ajouter à ceux et celles de la classe héritée. Des méthodes de la classe héritée peuvent également être surchargées. Si des données de test ont été générées pour la classe héritée, l'effort de test pour la classe dérivée peut s'en trouver réduit. En effet, des données de test de la classe héritée peuvent être réutilisées, il faut donc déterminer lesquelles et quelles données de test ajouter.
Certaines **classes** sont **abstraites**, leur implémentation est partielle et elles ne sont pas instanciables. Cela pose problème notamment pour tester les méthodes de ces classes. Citons parmi les possibilités la création une classe dérivée de la classe abstraite uniquement pour la tester, incluant des bouchons pour les méthodes abstraites. Ou encore l'utilisation d'une classe du programme qui hérite de la classe abstraite pour le test des méthodes non surchargées.

Dans le cadre du **polymorphisme d'inclusion**, un même code peut être appliqué à des données de types différents, sous réserve que ces types soient liés entre eux par une relation de sous-typage. Ainsi, une méthode prenant en paramètre, d'après sa signature, un objet d'un certain type, pourra aussi être appelée avec tout objet d'un sous-type de ce type. Si cet objet appelle une méthode, le corps de la méthode appelée peut dépendre de son type effectif, on parle de **méthode polymorphe**. Le test de méthodes dont la signature autorise des types différents pour les objets en entrée est donc une difficulté pour la génération de données de test car cela influe sur le code exécuté.

Contributions

Depuis quelques années, la programmation par contraintes est vue comme un moyen efficace pour automatiser la génération de données de test, pour le test basé sur le modèle ou sur le code (et plus généralement pour la vérification [YPA06]). Ainsi [CBG07, CBG09] formalisent un modèle à contraintes utilisé pour générer automatiquement des données de test pour du code C. Notre thèse est que la modélisation de la sémantique d'un programme en bytecode Java exploitant la programmation par contraintes permet de générer automatiquement des données de test pour sensibiliser des instructions sélectionnées. Les principales contributions de cette thèse sont les suivantes.

La première contribution est **la définition d'un modèle à contraintes de la sémantique du bytecode Java**. Ce modèle permet de faire des déductions efficaces y compris en présence d'alias de références ou de polymorphisme. La modélisation avec des contraintes de la sémantique du bytecode Java est plus simple qu'au niveau du code source car chaque instruction bytecode a un comportement élémentaire et une sémantique bien définie. Le modèle permet de traduire la relation entre deux états de la mémoire séparés par une suite d'instructions bytecode et de déterminer les contraintes portant sur l'état de la mémoire en début de séquence, telles que la séquence soit exécutée. Il ne nécessite pas de faire d'hypothèse a priori sur la forme de la mémoire mais infère les contraintes

portant dessus à partir des instructions conditionnelles de la séquence d'instructions. La modélisation par contraintes du bytecode Java a nécessité la définition d'un modèle de la mémoire et de nouvelles contraintes. L'utilisation de variables pour modéliser le type des objets permet de prendre en compte l'héritage et le polymorphisme. Comme cela sera montré dans ce manuscrit, notre modèle est très déductif car il permet d'inférer des formes de mémoire complexes pour satisfaire les contraintes posées. Il n'existe pas d'autre modèle aussi déductif à notre connaissance.

La deuxième contribution est **une méthode de génération automatique de données de test** pour le bytecode Java. Cette méthode vise à couvrir des instructions particulières non couvertes par d'autres méthodes de test. La méthode de génération proposée raisonne dans le sens inverse à celui de l'exécution : en partant d'un objectif (une instruction à couvrir) elle essaie de trouver un chemin menant vers le point d'entrée du programme en explorant progressivement et à l'envers le graphe de flot de contrôle, puis de trouver une donnée d'entrée qui active un tel chemin. Pour ce faire, elle exploite le modèle à contraintes du bytecode. Le sens de parcours "en arrière" du graphe de flot de contrôle est original, et favorise la couverture d'instructions non couvertes par d'autres outils de test qui utilisent une exploration en avant.

La troisième contribution est **un prototype**, JAUT, mettant en application le modèle et la méthode de génération proposés et permettant ainsi une validation expérimentale. Les expériences montrent que JAUT permet d'augmenter la couverture des instructions obtenue avec les autres outils disponibles [CG10], en particulier Pex qui est une référence dans le domaine.

Dans [CG08] nous avions proposé un nouveau modèle de mémoire pour le bytecode Java. Cependant, ce modèle générique souffrait de problèmes d'efficacité. Le modèle de mémoire présenté ici est donc un modèle optimisé, plus spécifique au bytecode Java.

Organisation du document

La partie I de ce document donne le contexte de ces travaux. Le chapitre 1 dresse un état de l'Art de la génération de données de test, en particulier basée sur le code. Le chapitre 2 présente la programmation par contraintes à domaines finis qu'exploite notre approche.

La partie II de cet écrit détaille notre modélisation par contraintes du bytecode et son exploitation pour générer des données de test. Le chapitre 3 donne un aperçu de la modélisation, de son utilité et des difficultés principales. Le chapitre 4 expose la modélisation d'un état de la mémoire d'une JVM et le chapitre 5 montre la traduction des instructions en relations, exprimées par des contraintes, entre des états de la mémoire. Le chapitre 6 s'intéresse à la génération de données de test qui s'appuie sur la modélisation des instructions par des contraintes. Le chapitre 7 compare notre approche avec l'Etat de l'art.

La partie III du document comporte le chapitre 8 qui décrit l'outil de génération de données de test pour le bytecode Java, JAUT, et le chapitre 9 qui analyse les résultats expérimentaux.

La partie IV se compose du chapitre 10 qui conclut ces travaux et du chapitre 11 qui ouvre des perspectives.

Première partie

Contexte et état de l'Art

Chapitre 1

Test des programmes

Le test peut se faire à partir de modèle ou à partir de code. Le test basé sur le modèle vise à couvrir des éléments d'un modèle qui spécifie le comportement attendu du programme sous test. Le test basé sur le code vise quant à lui à couvrir des éléments du code du programme. Une grande partie des méthodes de génération de données de test exploitent la programmation par contraintes.

1.1 Test basé sur le modèle

Le test à base de modèle (MBT pour *model-based testing*) repose sur un modèle du comportement attendu du programme sous test et éventuellement de son environnement. Ce modèle de test utilise un niveau d'abstraction et se concentre sur tout ou partie des comportements en fonction de l'objectif de test. Les processus et les techniques de MBT incluent la dérivation de cas de test abstraits depuis un modèle formel abstrait (un cas de test décrit l'entrée et la sortie attendue), la génération de cas de test concrets à partir des cas de test abstraits, et l'exécution manuelle ou automatique des cas de tests concrets résultants.

1.1.1 Une taxonomie du test à base de modèle

Utting, Pretschner et Legeard [UPL06] proposent des dimensions pour classifier les méthodes de MBT, cette classification permet d'avoir un point d'entrée sur ce qui existe dans ce domaine.

Une première dimension repose sur **les caractéristiques du modèle**. Les caractéristiques possibles du modèle sont l'incorporation d'une *dimension temporelle*, de *non déterminisme*, et la *nature continue, discrète ou hybride* du modèle. Ces caractéristiques sont choisies en fonction du système sous test. Ainsi, la dimension temporelle est pertinente pour les systèmes temps réel, ce type de système est connu pour être difficile à traiter. Le non-déterminisme peut être présent dans le système sous test ou seulement dans son modèle. Si le non-déterminisme vient du système, il se peut que l'entrée à soumettre à un moment donné au système dépende des réactions précédentes du système. Les cas de test doivent alors inclure la gestion de ce non-déterminisme. Enfin les modèles incluant une dynamique continue sont utiles pour certains systèmes embarqués et font l'objet de recherches actives.

Une deuxième dimension est **le paradigme et les notations utilisés pour décrire le modèle**.
Les *notations basées sur les états* permettent de modéliser le système comme un ensemble de variables, qui représentent l'état du système à un instant donné, et un ensemble d'opérations qui modifient l'état de ces variables. Parmi les langages de spécification basés sur les états, citons B, JML ou encore OCL. Les *notations basées sur les transitions* se concentrent davantage sur les transitions entre les différents états du systèmes. Elles incluent les machines à états finis (FSM), dans lesquelles les nœuds sont les principaux états du système et les arcs sont les actions ou les opérations du système, ou encore les machines à états UML, les systèmes de transitions et les automates d'entrées/sorties. Les *notations basées sur l'historique* modélisent un système en décrivant les comportements autorisés du système dans le temps. Différentes logiques temporelles peuvent être utilisées (linéaire ou avec branchement, avec une représentation du temps discrète ou continue, etc.). Les *notations fonctionnelles* décrivent un système comme un ensemble de fonctions mathématiques. Les fonctions peuvent être du premier ordre (spécifications algébriques) ou d'ordre supérieur (logique HOL). Les *notations opérationnelles* décrivent un système comme un ensemble de processus s'exécutant en parallèle. Elles incluent par exemple les algèbres de processus. Les *notations stochastiques* décrivent le système par un modèle probabiliste (par exemple des chaînes de Markov) des valeurs d'entrées et des évènements, elles sont surtout utilisées pour modéliser l'environnement du système sous test. Les *notations basées sur le flot des données* se concentrent plus sur les données que sur le flot de contrôle. Le langage Lustre est un exemple de ce type de notations.

Une troisième dimension est le **critère de sélection des tests**, ces critères étant nombreux, nous n'en citons ici qu'une partie. Certains critères se basent sur la *structure du modèle*, comme les noeuds et les arcs pour un modèle basé sur les transitions (couverture de tous les noeuds, de toutes les transitions...), ou encore les instructions conditionnelles pour un modèle exploitant les notations pre/post (couverture de toutes les propositions d'une disjonction dans la post-condition par exemple). D'autres critères se basent sur les *données* pour choisir quelques valeurs de test dans un espace de données important. Il peut s'agir de test aux limites, ou encore de test par paire (*pairwise testing*) qui consiste à tester, pour chaque paire de paramètres en entrée, toutes les combinaisons possibles de leurs valeurs. Les critères de sélection peuvent également se baser sur des *obligations de test* indiquées directement sur le modèle, par exemple dans les prédicats d'une post-condition d'un modèle pre-post, ou sur des transitions d'une machine à états.

Une quatrième dimension est la **technologie utilisée pour la génération des tests.** La méthode de génération peut être *stochastique*, utiliser des *algorithmes de recherche* dans un graphe, du *model-checking* borné, de l'*exécution symbolique*, de la *preuve de théorèmes* ou encore des *techniques de résolution de contraintes* utiles pour générer des valeurs dans des domaines de données complexes. Souvent les méthodes de génération automatique de données de test à partir de modèle combinent différentes techniques.

Une cinquième dimension concerne **l'exécution des tests**. Pour des systèmes non déterministes, il est intéressant que l'algorithme de génération puisse réagir en fonction des sorties courantes du système sous test, on parle de *génération en ligne*. A l'inverse, dans une méthode de *génération hors ligne*, les cas de tests sont dérivés en intégralité avant leur exécution.

1.1.2 Description de quelques outils

Cette section décrit quelques outils de génération de tests à base de modèle parmi les très nombreux outils existants.
L'outil industriel Test Designer (anciennement LTG, Leirios Test Generator) se base sur un modèle de test en UML, comportant des diagrammes de classes, des machines à états, des diagrammes d'instances, ainsi que des spécifications en OCL (spécifications sous la forme pre-post), pour définir une modélisation comportementale de l'application à tester. L'utilisateur indique dans la spécification des cibles de test à couvrir, et l'outil essaie de générer des données de test pour atteindre l'ensemble de ces cibles. Cet outil supporte également une spécification en B [JL07]. L'outil LTG exploitait la programmation par contraintes pour parcourir symboliquement des chemins de la spécification, Test Designer quant à lui exploite des algorithmes de recherche et des prouveurs de théorèmes.
TGV (Test Generation with Verification technology) [JJ04] est un générateur automatique de tests de conformité à partir de spécifications (en Sdl, en Lotos ou en UML) pour les systèmes réactifs. Il prend en entrée une description du comportement d'un protocole et un objectif de test. Il est basé sur le modèle des IOLTS (Input-Output Labelled Transition Systems).
STG (Symbolic Test Generation) [CJRZ02, RdBJ00] est un générateur de tests exploitant des techniques symboliques. Il prend en entrée des spécifications et des objectifs de test symboliques décrits sous la forme d'IOSTS (Input/Output Symbolic Transition System), et génère des cas de test également sous la forme d'IOSTS. Après être traduits en C++ ou Java, les cas de test peuvent être exécutés par une implémentation dans le langage correspondant. Les contraintes portant sur les entrées sont résolues à la volée durant l'exécution.
Autofocus [SH99, WLPO00] est un outil de spécification graphique pour les systèmes distribués. Il exploite différentes vues du système sous test : la structure du système avec des composants et des canaux, une description du comportement du système incluant ces composants et ces canaux, les données gérées par le système et transmises par les canaux et l'interaction entre les composants et l'environnement via des échanges de messages. Le comportement des composants est spécifié par des automates dont les transitions sont étiquetées par des pré-conditions et des post-conditions. Autofocus traduit la spécification sous forme de logique propositionnelle et génère automatiquement des séquences de test pour couvrir des objectifs de test (par exemple atteindre un état cible ou provoquer l'exécution d'une séquence de transitions).
Le langage Lustre est un langage déclaratif à flots de données synchrones muni d'opérateurs temporels permettant de se référer à des valeurs passées des données. L'outil GATeL [MA00] génère automatiquement des séquences de tests à partir de descriptions Lustre. Il repose sur une interprétation des constructions du langage Lustre sous la forme de contraintes portant sur des variables booléennes ou entières. Pour générer des séquences de test, ces contraintes sont résolues à l'aide de techniques de programmation logique par contraintes.

1.2 Test basé sur le code

1.2.1 Prérequis

Remplir un critère de test pour un ensemble de données de test est considéré comme un gage de qualité, et les critères sont souvent utilisés pour guider la génération manuelle ou automatique de tels ensembles. Les critères de test utilisés pour le test basé sur le code se fondent sur le flot de contrôle ou sur le flot des données.

1.2.1.1 Critères basés sur le flot de contrôle

Une donnée de test décrit quelles valeurs donner aux paramètres de la fonction sous test et en particulier quelle est la forme de la mémoire en entrée. La génération de données de test se fait dans le but de couvrir des éléments spécifiques du graphe de flot de contrôle. Voici quelques uns de ces critères.

- Le critère de **couverture de toutes les instructions**. La couverture de ce critère est considérée comme étant un minimum à atteindre, un prérequis à toute utilisation du programme.
- Le critère de **couverture de toutes les branches**. La couverture de toutes les branches garantit la couverture de toutes les instructions.
- Le critère de **couverture de tous les chemins**. La couverture de tous les chemins garantit la couverture de toutes les branches et de toutes les instructions.
- Le critère de **couverture des k-chemins**. Comme le nombre de chemins exécutables d'un programme peut être infini, le critère de couverture de tous les chemins est souvent relaxé avec le critère de couverture des k-chemins où seuls les chemins pour lesquels chacune des boucles est dépliée moins de k fois sont à couvrir.

1.2.1.2 Critères basés sur le flot des données

Pour les critères basés sur le flot des données, les définitions (*def*) des variables du programme ainsi que leurs différentes utilisations (*p-use* dans un prédicat, *c-use* dans un calcul) sont identifiées. Le test vise alors à couvrir les différentes combinaisons des paires *def-use*.

1.2.1.3 Approches orientées chemins et approches orientées but

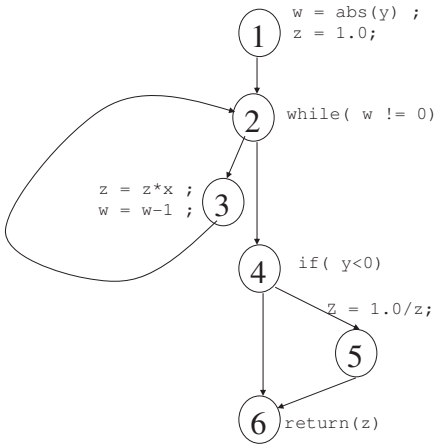
Pour couvrir un objectif donné, la plupart des approches consistent à essayer de trouver un ensemble de chemins exécutables tels que la couverture de chacun de ces chemins par une donnée de test assure la couverture du critère de test considéré. C'est ce que l'on appelle des méthodes orientées chemins. Cet ensemble de chemins peut être sélectionné en amont de la génération des données de test, mais s'il comprend des chemins non exécutables, ceux-ci peuvent faire baisser le taux de couverture. Dans les méthodes de génération de données de test actuelles, les chemins qui mènent à la couverture du critère sont le plus souvent choisis au fur et à mesure de la génération, en prenant en compte les informations acquises sur des sous-chemins non exécutables ou sur la structure du graphe de flot de contrôle.

L'approche orientée but fut proposée par Ferguson et Korel [FK96]. Dans cette approche, pour couvrir un objectif qui peut être une branche ou une instruction, aucun chemin n'est choisi a priori.

1.2.2 Exécution symbolique

L'exécution symbolique est une technique introduite par King [Kin76]. Elle a pour but de déterminer les conditions que doivent respecter des données de test pour suivre un chemin sélectionné. Des valeurs symboliques sont utilisées comme entrées du programme à la place des données concrètes, et des expressions symboliques représentent les valeurs des variables du programme. A chaque chemin exécuté symboliquement est associée une conjonction de contraintes (sans quantifieurs) portant sur les entrées symboliques : cette formule est appelée *condition de chemin*. La condition de chemin accumule les contraintes

```
double P(ush x, ush y){
  int w = abs(y) ;
  int z = 1.0;
  while ( w != 0 ){
    z = z * x ;
    w = w − 1 ;
  }
  if ( y<0 )
  z = 1.0 / z ;
  return(z) ;
}
```



chemin	x	y	w	z	condition de chemin
1	X0	Y0	undef	undef	true
1-2	X0	Y0	abs(Y0)	1.0	true
1-2-3	X0	Y0	abs(Y0)	1.0	abs(Y0)$\neq$0
1-2-3-2	X0	Y0	abs(Y0)-1	X0	abs(Y0)$\neq$0
1-2-3-2-4	X0	Y0	abs(Y0)-1	X0	abs(Y0)$\neq$0 $\wedge$ abs(Y0)-1 = 0
1-2-3-2-4-6	X0	Y0	abs(Y0)-1	X0	abs(Y0)$\neq$ 0 $\wedge$ abs(Y0)-1 = 0 $\wedge$ Y0 $\geq$ 0

FIG. 1.1 – Un programme, son graphe de flot de contrôle et une exécution symbolique de l'un de ses chemins

que les entrées doivent satisfaire pour que l'exécution suive le chemin associé. La figure 1.1 présente un programme et une exécution symbolique associée à l'un des chemins.

La technique de l'exécution symbolique exprime le problème d'activation d'un chemin donné par un problème de résolution de contraintes : trouver une donnée de test pour le chemin revient à déterminer des valeurs pour les variables d'entrée telles qu'elles satisfassent la condition de chemin. Si aucune valeur ne satisfait la condition, le chemin n'est pas exécutable.

1.2.3 Exécution symbolique et langages de programmation impératifs ou orientés objet

L'exécution symbolique telle que présentée ci-dessus a été proposée pour des programmes prenant en entrée des variables numériques. Pendant l'exécution symbolique, des manipulations algébriques sont effectuées pour simplifier les expressions symboliques ce qui peut rendre la méthode coûteuse. Outre ce problème de coût, pour pouvoir appliquer ce type de raisonnement à des langages plus complexes, en particulier les programmes orientés objets qui sont l'objet de cet écrit, il faut pouvoir étendre le traitement symbolique à des structures de données complexes.

Les manipulations de tableaux peuvent poser problème pour l'exécution symbolique. Ainsi, une expression de type t[i] telle que la valeur de i dépende des valeurs d'entrée présente des difficultés, car la valeur symbolique à laquelle on accède n'est pas déterminée. Les méthodes usuelles envisagent toutes les possibilités : i=0, i=1, etc. Cela multiplie le

nombre de chemins à analyser durant l'exécution symbolique.

Le problème de la détermination de la forme de la mémoire quand le langage permet l'allocation dynamique (problème *Mem_shape*) pose également de grandes difficultés pour la mise en œuvre de l'exécution symbolique. Ainsi, pour une méthode qui manipule des listes cycliques comme paramètres, le nombre d'objets, et donc de variables d'instance, peut être différent d'une liste à l'autre. Le nombre de variables d'instance n'étant pas connu, on ne sait pas a priori combien de variables symboliques seront nécessaires pour représenter les états des objets.

1.2.4 Méthodes exploitant l'exécution symbolique

Doyle et al. [DM03b, DM03a] proposent une méthode pour générer des données de test pour des programmes en bytecode Java, implantée dans l'outil IBIS. IBIS effectue des exécutions symboliques pour générer des contraintes de chemins résolues par une librairie de contraintes basée sur le simplexe. La méthode proposée ne couvre qu'une partie très restreinte du bytecode Java puisqu'elle ne traite les objets et les appels de méthode qu'à un "degré limité" selon les auteurs, et les exemples détaillés dans les articles décrivant cette méthode ne manipulent que des entiers.

Müller et al. [MLK04] présentent également une méthode pour tester des programmes en bytecode Java. Son objectif est de couvrir différents critères de couverture : couverture des instructions, des branches, et des couples def-use. Pour cela, les chemins du programme sont exécutés symboliquement par une machine virtuelle Java dite symbolique. La méthode est implantée dans l'outil GlassTT, la résolution des conditions de chemin fait appel, selon le type de contraintes à résoudre, à différents solveurs de contraintes, ou encore au système de calcul algébrique Mathematica.
Dans cette méthode, pour couvrir le critère def-use, chacun des couples def-use rencontrés est mémorisé avec la chaîne def-use ayant permis de le sensibiliser. Si le même couple avec la même chaîne est rencontré, alors l'exploration ne se poursuit pas car cela indique qu'une boucle est rencontrée et que la parcourir plusieurs fois n'augmentera pas la couverture. Ce constat paraît discutable en présence de références, mais la méthode ne semble traiter que les types entiers. Cette limitation n'est pas explicite dans les articles mais la gestion des références n'est pas évoquée et les exemples ne présentent que des entiers.

1.2.5 Exécution symbolique et model-checking

1.2.5.1 Initialisation paresseuse

Kurshid, Pasareanu et Visser proposent de combiner des techniques d'exécution symbolique et de model-checking [KPV03, VPK04, PMB+08] pour générer automatiquement des données de test pour la couverture de code en bytecode Java. L'objectif est de pouvoir tester des programmes qui prennent en entrée des structures complexes avec des données non bornées (nombre d'objets en mémoire initialement inconnu par exemple), ainsi que les programmes concurrents pour l'analyse desquels le model-checking est très utilisé. Pour générer des entrées afin de satisfaire un critère de test basé sur le code, par exemple la couverture de toutes les instructions, la méthode sous test est exécutée symboliquement pour la vérifier par rapport à des propriétés qui encodent le critère de test. Les contre-exemples pour ces propriétés sont des chemins qui satisfont le critère.

La méthode utilise une généralisation de l'exécution symbolique [KPV03] pour supporter les constructions avancées des langages modernes comme Java, en particulier pour traiter le problème de la méconnaissance de la forme de la mémoire en entrée de méthode (problème *Mem_shape*). Le model-checker permet d'explorer les chemins, et de les exécuter symboliquement de la manière suivante. Initialement, les variables d'instance de type référence en entrée de méthode sont non initialisées. Elles le seront au fur et à mesure des besoins lors de l'exécution symbolique, on parle d'*initialisation paresseuse*. Les variables entières sont symboliques comme pour toute exécution symbolique. Quand l'exécution symbolique accède à une variable d'instance référence, l'algorithme initialise de manière aléatoire cette référence qui : (1) soit vaut *null*, (2) soit référence un objet symbolique qu'il crée et dont les attributs sont non initialisés, (3) soit référence un objet symbolique déjà créé ayant un type compatible. Le troisième cas permet de générer des situations d'aliasing. Quand une condition de branchement c portant sur des variables entières est évaluée, l'algorithme choisit une branche de manière non déterministe en ajoutant à la condition de chemin courante soit la condition c, soit la négation de c, et appelle une procédure de décision pour vérifier la satisfiabilité de la condition de chemin. Si la condition de chemin devient contradictoire, l'algorithme revient en arrière (*backtrack*) pour choisir une autre branche et donc un autre chemin. Chaque chemin sensibilisé est caractérisé par une configuration en entrée du tas, qui encode les contraintes sur les attributs de type référence, et une condition portant sur les variables symboliques entières. La valeur des références non utilisées sur le chemin est fixée à *null*. L'utilisateur peut fournir des pré-conditions en Java que doivent vérifier les objets créés, y compris ceux qui sont créés lors de l'exploration des chemins.

Les auteurs mettent en œuvre leur méthode via une extension, appelée JPF-SE, au model-checker Java PathFinder pour du bytecode Java [APV07]. Pour vérifier les conditions de chemins, plusieurs procédures de décisions peuvent être utilisées (Oméga, CVC-lite, YICES et STP).
Dans [VPP06] et [APV09], des abstractions d'états sont utilisées pour comparer l'état courant à des états déjà rencontrés. L'avantage est de réduire l'explosion du nombre d'états, l'inconvénient est d'exclure indûment certains états du parcours à cause de l'abstraction et donc de sous-approximer les comportements du programme.
Des exécutions concrètes peuvent être utilisées pour mettre l'environnement dans un certain état qui sera exploité pour l'exécution symbolique.

Concernant les inconvénients de cette méthode, le nombre de chemins dans un programme peut être non borné, et le model-checker explore ces chemins selon des heuristiques telles que l'exploration en largeur ou en profondeur, bornée par la taille des chemins et des entrées. Cependant, il ne tient pas compte de la structure du programme pour guider l'exploration, ainsi cette méthode ne permet pas de borner le nombre de dépliage d'une boucle. Les références ne sont pas gérées de manière symbolique, un choix est effectué quand la connaissance de leur valeur est nécessaire, multipliant ainsi le nombre de chemins pouvant être explorés. Augmenter ainsi le nombre de chemins augmente aussi potentiellement le nombre de chemins non exécutables dans le cas où le choix effectué n'est pas compatible avec les conditions qui suivent sur le chemin exploré.
Cette méthode a le mérite de proposer une solution pour couvrir des codes de méthodes prenant en entrée des structures complexes. La satisfiabilité de la condition de chemin est testée au fur et à mesure de sa construction ce qui permet de couper dès que possible

l'exploration de l'arbre des exécutions et d'exclure ainsi des chemins non exécutables.

1.2.5.2 *Lazier initialization*

Les travaux sur JPF-SE ont inspiré ceux de Deng, Robby et Hatcliff [DRH07]. Ils utilisent l'exécution symbolique généralisée proposée dans [KPV03] en l'améliorant afin de retarder davantage le choix non déterministe effectué lors de l'utilisation d'une référence et de réduire ainsi le nombre de chemins à explorer. Ils proposent pour cela la *lazier initialization*. L'initialisation paresseuse de [KPV03] initialise une variable d'instance de type référence lorsqu'une instruction y accède. Soit o.f une référence, dans l'initialisation proposée, la *lazier initialization*, son initialisation est retardée jusque ce qu'elle soit utilisée pour un accès à un attribut (o.f.g par exemple) ou un test d'égalité (o.f=o' par exemple). Pour ce faire, un accès à la référence o.f retourne seulement une référence symbolique, de potentiels usages ultérieurs provoqueront l'initialisation de sa valeur en choisissant de manière non déterministe entre un nouvel objet ou un objet déjà présent dans le tas de type compatible. Les références sont donc en partie traitées de manière symbolique, mais des choix non déterministes peuvent encore être nécessaires.
Le fait de retarder l'initialisation limite le nombre de chemins à considérer car le choix non déterministe lié à une référence se fait soit plus tard, soit jamais. Le traitement symbolique des références nécessite de propager en arrière l'information lorsqu'elles sont concrétisées. A l'issue de l'exécution symbolique d'un chemin, un solveur de contraintes est utilisé pour instancier les éléments encore inconnus.

Une borne, k, limite la longueur des chaînes de références, une autre borne limite le dépliage des boucles. L'approche est implantée dans l'outil KUnit, qui exploite l'exécution symbolique de Bogor/Kiasan [DLR06]. Il utilise une procédure de décision (CVC Lite) et un solveur de contraintes. Les éléments manquants du système peuvent être remplacés par des contrats. Comme pour JPF-SE, les pré-conditions portant sur les objets sont exploitées pour la construction des objets.
L'objectif affiché de la méthode n'est pas seulement d'atteindre une couverture élevée du code, mais aussi de tester le programme avec des configurations d'entrée variées. Grâce à la borne k, il est possible de garantir que toutes les formes du tas possibles dans la limite de la borne k seront générées pour chacun des chemins explorés. L'un des inconvénients qui en découle est l'importance du nombre d'entrées générées. Ainsi, pour certains exemples présentés, en fixant k=3 plus de mille données de test sont générées.

1.2.5.3 Représentation symbolique de librairies

Les auteurs de [KS05] partent du constat que les appels aux librairies peuvent être coûteux pour l'exécution symbolique. En effet, les implémentations de librairies représentant, par exemple, des ensembles comportent souvent des optimisations qui multiplient le nombre de chemins. Ils proposent d'utiliser une modélisation symbolique des librairies Java *Set* et *Map* : les opérations sont modélisées sous forme d'opérations symboliques en se défaisant des détails d'implémentation. Leur méthode est implantée dans l'outil Dianju, basé sur (l'ancêtre de) JPF-SE.

1.2.6 Exécution symbolique dynamique

L'exécution symbolique dynamique, ou *exécution concolique*, associe exécution concrète du programme et exécution symbolique afin d'explorer les chemins du programme. L'exé-

cution concrète dans ce cadre sert à guider l'exploration des chemins du graphe de flot de contrôle, et parfois à aider l'exécution symbolique en approximant la valeur d'expressions complexes telles que les expressions non linéaires. Différentes méthodes exploitent l'exécution symbolique dynamique pour la génération de données de test, pour tester des programmes en C [WMMR05, GKS05, God07, SMA05], des programmes en bytecode Java [SA06], des programmes dans le langage intermédiaire du .NET [TdH08, TS06] et des exécutables [BH08]. Certaines méthodes [GKS05, God07, SMA05, SA06, BH08] sont cependant plus spécialisées dans la recherche de bugs, elles essaient de mettre en évidence des erreurs d'exécution ou des failles de sécurité.

1.2.6.1 Exploration du graphe de flot de contrôle

L'exécution symbolique dynamique est une méthode de génération de données de test orientée chemins. L'exploration du graphe de flot de contrôle dans le cadre de l'exécution symbolique dynamique est illustrée sur la figure 1.2. La première donnée de test est générée

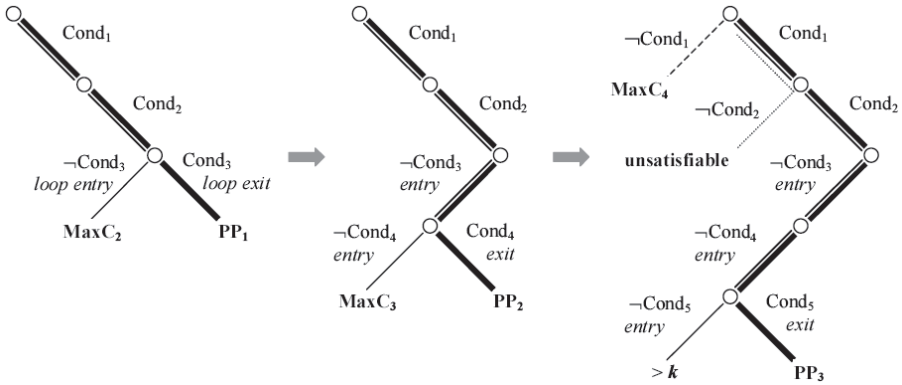


FIG. 1.2 – Méthode de parcours des chemins pour l'exécution symbolique dynamique. Figure extraite de [WN04].

aléatoirement, la condition de chemin du chemin suivi est exprimée sous la forme d'une conjonction de contraintes, sur la figure $cond_1 \wedge cond_2 \wedge cond_3$. La dernière condition est niée, cette nouvelle conjonction de contraintes caractérise un préfixe de chemins qui n'est pas encore couvert. Sur la figure on obtient $cond_1 \wedge cond_2 \wedge \neg cond_3$. Une donnée qui satisfait cette nouvelle conjonction de contraintes est générée si cela est possible (sinon la recherche reprend avec un préfixe non encore exploré). Supposons que cette donnée de test sensibilise le chemin dont la condition est $cond_1 \wedge cond_2 \wedge \neg cond_3 \wedge cond_4$, la dernière condition $cond4$ est niée à son tour et le processus continue de la sorte, menant à une exploration en profondeur du graphe (des bornes sont cependant souvent fixées pour limiter l'exploration), jusqu'à ce que le graphe soit exploré selon le critère fixé.

Le préfixe du chemin dernièrement exploré qui est conservé pour la génération de la donnée de test suivante est forcément exécutable puisqu'une donnée de test qui le sensibilise a été générée. Seule la négation de la dernière condition peut introduire une insatisfiabilité. Si c'est le cas, tous les chemins qui sont issus du préfixe insatisfiable sont écartés de la génération, c'est l'intérêt d'utiliser une méthode incrémentale qui teste au fur et à mesure la satisfiabilité des préfixes de chemins et pas uniquement des chemins complets.

Comme cela sera décrit par la suite, certaines approches symboliques dynamiques [GKS05, TdH08] modifient l'ordre dans lequel sont explorées les branches afin de s'adapter à la couverture de critères autres que tous les chemins et tous les k-chemins, par exemple pour la couverture du critère de toutes les branches. Le principe reste néanmoins de nier la dernière condition d'un préfixe de chemin déjà sensibilisé.

1.2.6.2 Exécution symbolique dynamique pour la couverture des chemins

Williams, Marre, Mouy, et Roger proposent une méthode de génération automatique de données de test pour le langage C [WMMR05, BDHTH+09]. Ils sont parmi les premiers, avec Godefroid, Klarlund, et Sen [GKS05], à proposer une exécution symbolique dynamique. Leur méthode a pour but de générer un ensemble de tests pour couvrir le critère des k-chemins. Comme pour toutes les approches symboliques dynamiques, le graphe de flot de contrôle n'est pas construit a priori mais exploré "à la volée", l'exploration est faite en profondeur d'abord. La méthode est implanté dans un outil, *PathCrawler*. Sous réserve que le temps limite fixé (*timeout*) ne soit pas atteint avant la fin de la résolution, l'approche garantit la complétude pour le critère des k-chemins. Cette complétude repose d'une part sur la méthode d'exploration du graphe de flot de contrôle, qui par construction va explorer tous les k-chemins, et d'autre part sur la capacité du solveur de contraintes utilisé, Colibri, à traiter toutes les expressions des conditions de chemin générées, y compris les expressions non linéaires. En cela la méthode se distingue d'autres approches ayant recours à des approximations, notamment faute de pouvoir gérer la non-linéarité.

La méthode traite les fonctions ayant pour paramètres des pointeurs, des données structurées et des tableaux de dimensions variables. L'hypothèse qu'il n'y a pas d'alias dans les entrées (deux entrées de la méthode sous test ne peuvent pas désigner la même zone de mémoire) est posée, sous cette hypothèse les structures de données d'entrée sont construites de manière paresseuse. L'utilisateur peut néanmoins préciser que des entrées sont des alias à l'aide de pré-conditions peuvant être prises en compte pour générer les entrées. La complétude est donc assurée en excluant par défaut des relations d'alias entre les entrées, sauf indication contraire de l'utilisateur. Une qualité notable de l'approche est la prise en compte sans approximation des expressions manipulant des tableaux, y compris en cas d'accès via un indice variable, ce qui est l'un des avantages de l'utilisation de la programmation par contraintes pour le test.

Les appels de fonctions amplifient le problème de l'explosion combinatoire du nombre de chemins à couvrir lorsqu'ils sont traités par "*inlining*", c'est-à-dire quand le code de la fonction appelée est inclus dans le code de la fonction sous test. En effet, couvrir un chemin de la fonction sous test peut nécessiter de parcourir différents chemins des fonctions appelées. De plus, s'il y a *inlining* des fonctions appelées, aucune distinction n'est faite entre le code des fonctions appelées et celui de la fonction sous test : l'objectif de l'approche est de couvrir le critère des k-chemins y compris pour les fonctions appelées. Pour faire face à ce problème, [MMWG08] propose d'utiliser des spécifications à la place du code des fonctions appelées, sous la forme de couples pre-post. Chaque couple pre-post est vu comme un chemin de la spécification, il y a généralement moins de chemins dans la spécification que dans le code d'une fonction ce qui réduit le nombre de chemins à explorer. De plus, un autre critère est proposé pour lequel couvrir les chemins de la méthode appelée n'est plus nécessaire. Les auteurs précisent que des spécifications formelles sont déjà souvent requises pour les logiciels critiques, utiliser des spécifications à la place des méthodes appelées n'alourdit dans ce cas pas la tâche de l'utilisateur.

1.2.6.3 Exécution concolique

Godefroid, Klarlund, et Sen proposent une méthode basée sur l'exécution symbolique dynamique pour couvrir tous les chemins exécutables d'un programme C [GKS05], ce qui peut être très coûteux. Leur méthode peut aussi s'utiliser pour la couverture de toutes les branches. Elle est implantée dans un outil, *DART* (Directed Automatic Random Testing), qui se sert non seulement de l'exécution symbolique dynamique pour parcourir le graphe de flot de contrôle en profondeur, mais aussi pour approximer la valeur d'expressions complexes qu'il ne peut pas résoudre symboliquement. Ainsi, en cas de multiplication entre deux variables dans une condition de chemin, DART ne peut pas traiter la multiplication symboliquement car il exploite la programmation linéaire entière pour résoudre les conditions de chemins. Pour pallier cette limitation, l'outil utilise le résultat de la multiplication obtenu lors de la dernière exécution concrète à la place de la multiplication symbolique. DART se sert également d'approximations pour les expressions complexes portant sur les pointeurs qui ne sont pas supportées par son modèle mémoire. Par exemple si un pointeur dont la valeur dépend des paramètres d'entrée est déréférencé, alors il est remplacé dans l'expression symbolique par sa valeur concrète lors de la dernière exécution : c'est une concrétisation. Le nombre de comportements du programme pris en considération peut se trouver réduit par ces approximations, par conséquent certains chemins exécutables peuvent ne pas être couverts par des données de test à cause de l'approximation.

Dans [SMA05] et [SA06], Sen, Marinov et Agha reprennent le principe d'exécution symbolique dynamique exploitée dans la méthode de Godefroid, Klarlund et Sen pour couvrir les chemins exécutables de programmes en C ou Java. Ils introduisent le terme d'*exécution concolique* pour désigner la coopération entre exécution concrète et exécution symbolique. Ils proposent un modèle mémoire plus complexe qui réduit le nombre de concrétisations nécessaires sur les pointeurs (ou références pour Java). Leur méthode utilise les instructions conditionnelles du chemin suivi portant sur les pointeurs pour exprimer des contraintes sur les variables modélisant ces pointeurs, y compris en cas de déréférencement. La concrétisation reste cependant parfois nécessaire, ainsi pour l'extrait de code `*p=0 ; *q=1 ; if(*p==1) <instruction cible> ;` leur méthode ne permet pas de générer la contrainte `p==q` qui permet d'atteindre l'instruction cible car le programme ne contient aucune conditionnelle portant sur les pointeurs qui pourrait lui permettre de la déduire. Deux outils exploitent cette méthode : *Cute* pour tester les programmes en langage C, et *JCute* pour tester les programmes en bytecode Java.

Godefroid étend l'approche utilisée dans DART avec un traitement compositionnel des appels de fonction [God07]. Pour chaque chemin de chaque fonction appelée un résumé, sous la forme d'un couple pre-post, est calculé pour être utilisé ensuite à la place du code de la fonction lorsqu'elle est appelée. Une stratégie top-down est recommandée pour calculer les résumés, c'est-à-dire que les résumés sont calculés à la demande, ce qui évite de calculer des résumés qui ne sont jamais utilisés dans le contexte des appels à la fonction par le programme sous test. Initialement, aucun résumé de fonction n'est disponible. Lors de l'exécution symbolique dynamique, quand une méthode est appelée, il faut regarder si un résumé pour cette fonction est disponible pour le contexte d'appel (c'est-à-dire pour les valeurs concrètes courantes) : si oui, le résumé est utilisé à la place de la fonction, si non, le code de la méthode appelée est résumé par exécution symbolique dynamique. Cette méthode est implantée dans l'outil *SMART* (Systematic Modular Automated Random Testing) pour le test de programmes C.

1.2.6.4 Tests paramétrés

Tillmann, de Halleux et Schulte proposent également une méthode de génération de données de test basée sur l'exécution symbolique dynamique [TdH08, TS06]. Leur méthode est implantée dans *Pex*, un outil développé par Microsoft pour générer des données de test couvrant les branches, ou encore mettant en évidence des erreurs d'exécution ou des failles de sécurité. Il supporte de très nombreuses fonctionnalités des programmes en bytecode .NET (hormis le non-déterminisme, la concurrence, le code natif et les flottants), peut être intégré à Visual Studio pour le langage C#, et il génère les scripts de test permettant d'exécuter des séquences d'appels pour créer les objets et les placer dans l'état désiré. Il rapporte un taux de couverture des branches qui est dit "dynamique" : il s'agit du taux de couverture des branches explorées, qui peut être distinct du taux de couverture des branches du graphe de flot de contrôle ; en particulier si l'outil ne parvient pas à générer des entrées pour un préfixe exécutable, tous les chemins débutant par ce préfixe sont exclus de l'exploration.

La méthode de génération utilisée par Pex exploite l'exécution symbolique dynamique pour parcourir le graphe de flot de contrôle en cherchant à couvrir au plus tôt toutes les branches. De multiples paramètres permettent de borner l'exploration : bornes sur le nombre d'exécutions, sur la longueur des chemins, sur le nombre de branches pour un chemin, sur le temps imparti à la génération, sur le temps imparti au solveur, sur le nombre de conditions portant sur les entrées pour un chemin, sur la taille de la pile, sur la taille de la mémoire, etc.
Pour la résolution des conditions de chemin, le solveur SMT Z3 [dMB08] est utilisé. Comme dans les méthodes utilisées dans DART, SMART, Cute et JCute, l'aspect dynamique de l'approche est exploitée pour approximer certaines expressions symboliques complexes par des valeurs concrètes, par exemple pour les accès aux tableaux via des indices variables. Le nombre de comportements du programme pris en compte peut s'en trouver réduit.
Concernant le traitement des références et des alias, le modèle mémoire utilisé par Pex n'étant pas décrit précisément, il est difficile de savoir si la concrétisation est utilisée pour les références, et si oui dans quels cas : cela n'est à notre connaissance pas renseigné, ni dans les articles, ni dans la documentation de l'outil. Les expériences montrent qu'il ne permet pas de résoudre certaines conditions de chemins impliquant des relations entre les références (et par conséquent des structures de données) complexes.
Dans le cas où un paramètre a un type interface (classe abstraite) ou si une classe n'est pas encore développée, Pex permet à l'utilisateur d'utiliser des "*mock objects*" pour simuler le comportement attendu d'un objet. Cette approche peut réduire le nombre de chemins dans les méthodes appelées, de manière similaire à l'approche de PathCrawler.

1.2.6.5 Test de code exécutable

Bardin et Herrmann proposent une méthode pour générer des données de test pour du code exécutable [BH08], ce qui présente plusieurs avantages par rapport au test de code source. D'une part des programmes dont le code source n'est pas disponible (sous-traitance, appels à des librairies) peuvent être testés. D'autre part, quand la vérification est faite au niveau du code source, il faudrait assurer que la compilation conserve la sémantique ce qui est parfois difficile quand le compilateur utilise des optimisations. Tester directement le code binaire assure que l'on teste ce qui sera réellement exécuté, mais travailler au niveau du binaire est plus difficile (pas de types, pas de flot de contrôle, ...).

La méthode proposée utilise uniquement le code de l'exécutable (pas de spécifications)

pour générer des données de test afin de couvrir les critères de couverture de toutes les branches ou toutes les instructions. L'architecture et l'ensemble d'instructions utilisées dans les exécutables sont très dépendants du système, l'approche présentée se veut générique pour s'adapter à plusieurs processeurs.
La structure du programme peut partiellement être retrouvée statiquement à partir du code binaire. Puis l'exécution symbolique dynamique permet de découvrir davantage de branchements, liés aux sauts dynamiques (des sauts dont l'adresse de l'instruction cible est une variable, par exemple `goto x` où la valeur de `x` est seulement connue à l'exécution). Cependant, il n'est pas garanti que l'intégralité de la structure du graphe de flot de contrôle puisse être retrouvée, la couverture peut donc être incomplète.
Le parcours des chemins se fait en profondeur d'abord et de manière bornée. Les conditions de chemins sont exprimées avec des contraintes sur des vecteurs de bits : les vecteurs de bits facilitent la formalisation des instructions standards des machines au niveau des bits (opérations bit à bit par exemple), et la flexibilité de la programmation par contraintes permet d'encoder toutes les instructions rencontrées.

Concernant les alias, n'ayant pas accès à des informations de haut niveau, les relations d'alias de l'exécution concrète sont extraites et ajoutées aux prédicats de chemin. Cependant, comme dans le cas des méthodes utilisées dans les outils Cute et Pex, cette sous-approximation des comportements peut rendre le prédicat de chemin insatisfiable alors que le chemin est exécutable. Pour atténuer cet inconvénient, différentes alternatives pour les alias sont énumérées en relaxant certaines contraintes.

L'approche est implantée dans l'outil *OSMOSE*.

1.2.6.6 Méthode de Cadar et al.

Cadar et al. [CGP+06, CGP+08] proposent une méthode de génération de données de test pour couvrir les chemins et détecter un maximum d'erreurs d'exécution (par exemple les divisions par zéro) pour les programmes C. Elle exploite elle aussi une complémentarité entre exécution symbolique et exécution concrète du programme, mais pas de la même manière que les autres méthodes détaillées précédemment puisque l'utilisateur indique quelles zones de mémoire traiter symboliquement, les autres zones de mémoire étant traitées avec leur valeur concrète.
Les chemins sont explorés exhaustivement selon une heuristique qui choisit en priorité des chemins couvrant des instructions non encore atteintes. L'exécution symbolique portant sur les zones de mémoire à traiter symboliquement débute par le point d'entrée du programme. Quand une expression conditionnelle est symboliquement exécutée, un *fork* est effectué afin qu'il y ait un processus pour le cas où l'expression est vraie et un autre pour le cas où elle est fausse. De ce mécanisme de création de processus résulte un processus par chemin. La priorité donnée à chacun des processus détermine dans quel ordre sont parcourus les chemins.
L'approche est implantée dans l'outil *EXE*. Quand une erreur d'exécution peut être produite ou quand l'exécution symbolique est achevée, la résolution des contraintes de chemins s'effectue dans cet outil à l'aide d'un solveur de contraintes dédié, STP, qui est une procédure de décision sur les vecteurs de bits et les tableaux. En effet, la mémoire est vue comme un tableau de bits. STP implante toutes les opérations arithmétiques sur les entiers et les booléens, y compris les opérations non linéaires. Les instructions n'impliquant que des opérandes concrets sont exécutées concrètement.
La méthode ne traite les pointeurs que partiellement puisqu'elle ne gère pas le double dé-

référencement symboliquement. En présence d'un double déréférencement **p, la première déréférence *p prend une valeur concrète parmi ses valeurs possibles.

L'utilisation d'une duplication des processus est intéressante dans le cas où une parallélisation est possible (grille de calcul par exemple). Contrairement aux autres méthodes utilisant l'exécution symbolique dynamique, cette approche ne cherche pas à détecter dès que possible les préfixes de chemin non exécutables. En effet, la résolution des contraintes n'est lancée que lorsqu'un chemin complet est parcouru. Cela peut ralentir considérablement le parcours des chemins en présence de chemins partageant le même préfixe de chemin non exécutable.

1.2.7 Approche orientée but

1.2.7.1 Méthode de génération de tests pour le C et C++

Dans [GBR98, GBW06], une méthode de génération de tests pour le C et C++ est proposée. Elle prend comme objectif de test un élément du code source du programme sous test (instruction, branche) et essaie de générer une donnée de test pour couvrir cet élément. L'approche est orientée but, c'est-à-dire que pour couvrir une instruction elle ne fait pas le choix a priori d'un chemin particulier. Ainsi, elle ne borne ni le nombre de dépliages des boucles, ni la longueur des chemins contrairement aux autres approches, elle permet donc potentiellement d'atteindre des instructions que les autres approches ne couvrent pas ou difficilement à cause de ces bornes. Cette méthode gère les programmes avec des pointeurs [GDB05], des allocations dynamiques [CBG09] et des nombres flottants [BGM06].

La méthode est implantée dans un outil, *InKa*, dont le fonctionnement est le suivant. Le programme C ou C++ à tester est d'abord traduit dans un langage intermédiaire. Cette normalisation divise des instructions complexes pour obtenir des instructions plus simples, grâce à l'introduction de variables temporaires. La normalisation permet aussi de rendre explicites des opérations implicites, comme les conversions de type et les surcharges. Puis ces instructions du langage intermédiaire sont interprétées pour générer un système de contraintes. L'obligation d'atteindre l'instruction souhaitée est également traduite sous forme de contraintes ajoutées au système. La résolution du système de contraintes est lancée. Si InKa détermine que le système de contraintes a des solutions, il retourne une donnée de test permettant d'atteindre l'instruction sélectionnée. S'il détermine que le système de contraintes n'a pas de solution, l'instruction à atteindre est inaccessible (code mort). Si la résolution du système de contraintes excède un temps fixé, la résolution du système s'arrête sans que l'outil ne puisse ni fournir une donnée de test, ni affirmer que l'instruction du programme à atteindre est inaccessible. InKa utilise le solveur de contraintes clp(fd) de SICStus prolog pour les entiers et son propre solveur pour les autres contraintes.

Des relations entre états mémoires Dans l'approche proposée, toutes les instructions C et C++ sont traitées comme des relations entre deux états mémoires, un état mémoire étant une représentation abstraite de la mémoire physique en un point de l'exécution. Les contraintes générées par interprétation d'une instruction FINOP traduisent les liens entre la mémoire physique avant instruction et la mémoire physique après instruction, sous forme d'une relation entre ces états mémoires. Si de l'information est acquise sur la mémoire en entrée, les contraintes liant les mémoires en entrée et en sortie permettront de faire des déductions sur la mémoire en sortie, et inversement.

Gestion des structures de contrôle Les structures de contrôle telles que le *if-then-else* et le *while* sont gérées par des opérateurs spécifiques. Seul le cas du *if-then-else* est détaillé ici, les autres opérateurs s'appuyant sur des principes similaires.
Une instruction de contrôle *if-then-else* est traitée par un opérateur nommé `ite/3`. Soit c la contrainte traduisant la conditionnelle sur laquelle porte le *if*, C_{then} la conjonction des contraintes obtenues en interprétant les instructions du corps du *then* et C_{else} la conjonction des contraintes obtenues en interprétant les instructions du corps du *else*. Lorsque la contrainte `ite(`c,C_{then},C_{else}`)` devient active, l'évaluation de la contrainte consiste à déterminer par quelle branche il faut passer. La branche par laquelle il faut passer peut être imposée par l'objectif. Si cela n'est pas le cas, il faut essayer de prouver que $(c \wedge C_{then})$ ou $(\neg c \wedge C_{else})$ est incompatible avec le système de contraintes, prouvant ainsi respectivement qu'il ne faut pas passer par la branche *then* ou par la branche *else* mais par l'autre branche et écartant de la sorte une branche du processus de génération. Si aucune branche n'est écartée, la mémoire en entrée (respectivement en sortie) de la structure de contrôle est une union disjonctive des états mémoires en entrée (respectivement en sortie) de chacune des branches, et l'opérateur `ite` est suspendu, c'est-à-dire qu'il pourra être de nouveau utilisé pour tenter d'éliminer une branche quand davantage d'information sera acquise sur les mémoires en entrée ou en sortie.

1.2.7.2 Vérification de programmes C critiques

Une autre approche à base de contraintes [Got09] a également été proposée pour vérifier des programmes C critiques. Elle permet non seulement générer des données de test pour atteindre un élément d'un code, mais aussi de vérifier des propriétés de sécurité ou de générer des contre-exemples qui les invalident. Cependant comme ces problèmes sont indécidables dans le cas général, la procédure est semi-correcte. La méthode présentée permet de générer une donnée de test pour atteindre un point particulier du programme ou indique qu'il n'est pas atteignable. Dans le cadre de la vérification, l'utilisateur peut ajouter des propriétés de sécurité qui peuvent être données sous la forme de pre et post conditions ou sous la forme d'assertions directement dans le code.

Cette méthode est orientée but et reprend des principes proposés dans [GBR98, GBW06], en particulier les opérateurs à contraintes pour les structures de contrôle. Les pointeurs vers des zones nommées sont traités grâce à une analyse de *points-to*.
La détection de la non satisfiabilité d'un système de contraintes, nécessaire pour réfuter des branches non exécutables, est améliorée par l'utilisation de relaxations linéaires dynamiques [DGD07b]. En effet, les techniques de programmation linéaire telles que le simplexe peuvent résoudre des problèmes linéaires très efficacement. La relaxation linéaire pour cette approche peut être vue comme une manière de sur-approximer par des contraintes linéaires les relations maintenues par le système de contraintes généré. En particulier le traitement des boucles est rendue plus efficace par des relaxations linéaires spécifiques basées sur un raisonnement par cas et des techniques d'interprétation abstraite [DGD07a].

Cette méthode est implantée dans un outil, *Euclide*, qui a été utilisé pour vérifier une version préliminaire d'un composant critique du TCAS (un système d'aide à la décision pour les pilotes d'avions) [Got09].

1.2.7.3 Lien avec notre approche

Notre approche exploite les travaux menés pour la génération automatique de données tests orientée but [GBR98, GBW06]. [CBG07] formalise la représentation du tas utilisée dans l'outil InKa, cette représentation est extrêmement complexe ce qui ralentit la génération de données de test. Euclide [Got09] possède un modèle mémoire plus simple, mais en contrepartie il ne gère pas l'allocation dynamique. Notre modélisation se distingue de ces approches par la modélisation de la pile d'opérandes, qui n'existe pas en C ou C++, et la gestion de l'héritage et des appels de méthodes polymorphes. De plus, notre méthode de génération de test est orientée chemins et non but. Cependant, le modèle mémoire que nous proposons a été pensé avec l'objectif d'étendre notre méthode à l'approche orientée but. Les opérateurs utilisés dans ces approches orientées but pour modéliser les structures de contrôle, tels que le `ite`, ne sont pas directement applicables pour le bytecode Java car le bytecode est déstructuré et cela nécessite de retrouver les structures de contrôle.

1.2.8 Bilan

Les tableaux suivants récapitulent les caractéristiques des principales approches présentées dans cette section. Le positionnement de notre approche face à ces travaux est détaillé dans le chapitre 7, après les chapitres expliquant notre approche.

	critère de couverture visé	type d'approche	spécificités des algorithmes	complétude
Méthode de Kurshid, Pasareanu et Visser	toutes les instructions	statique, orientée chemins	utilisation du model-checking et de l'exécution symbolique, initialisation paresseuse	sous-approximation des comportements (non déterminisme)
Méthode de Deng, Robby et Hatcliff	toutes les instructions	statique, orientée chemins	utilisation du model-checking et de l'exécution symbolique, *lazier initialization*	sous-approximation des comportements (non déterminisme)
Méthode de Williams, Marre, Mouy, et Roger	tous les k-chemins	dynamique, orientée chemins	exécution symbolique dynamique	pas de sous-approximation, sous les hypothèses fournies par l'utilisateur sur les relations d'alias en entrée de programme
Méthode de Godefroid, Klarlund, et Sen	tous les chemins, toutes les branches	dynamique, orientée chemins	exécution concolique, concrétisation pour les expressions non linéaires et pour certaines expressions portant sur des pointeurs	sous-approximation des comportements liée à la concrétisation
Méthode de Sen, Marinov et Agha	tous les chemins, toutes les branches, failles de sécurité	dynamique, orientée chemins	exécution concolique, concrétisation pour les expressions non linéaires et pour certaines expressions portant sur des pointeurs, traitement de certaines relations d'alias portant sur les entrées du programmes	sous-approximation des comportements liée à la concrétisation
Méthode de Tillmann, de Halleux et Schulte	toutes les branches, failles de sécurité	dynamique, orientée chemins	tests paramétrés, traitement de certaines relations d'alias portant sur les entrées du programmes, concrétisation	sous-approximation des comportements liée à la concrétisation
Méthode de Bardin et Herrmann	toutes les branches, toutes les instructions	dynamique, orientée chemins	exécution concolique pour le test d'exécutables, approximation de certaines relations d'alias due à la perte d'informations de haut niveau	sous-approximation des comportements (perte d'informations de haut niveau)
Méthode de Cadar et al.	tous les chemins, erreurs d'exécution	dynamique, orientée chemins	exécution concolique, un processus par chemin, concrétisation pour les doubles déréférencements	sous-approximation des comportements liée à la concrétisation
Méthode de Gotlieb, Botella et Rueher	toutes les instructions	statique, orientée but	pas de choix de chemin a priori, gestion des relations d'alias entre références	pas de sous-approximation
Méthode de Gotlieb	toutes les instructions	statique, orientée but	pas de choix de chemin a priori, gestion des relations d'alias entre références, utilisation de relaxations linéaires	pas de sous-approximation

FIG. 1.3 – Tableau comparatif des différentes approches de génération de test basées sur le code (1/2)

	sens de parcours du graphe	détection de sous-chemins non exécutables	techniques de résolution utilisées	outil	langage cible	références bibliographiques
Méthode de Kurshid, Pasareanu et Visser	en avant	oui	procédures de décision	JPF-SE	Java	[KPV03, VPK04, PMB+08]
Méthode de Deng, Robby et Hatcliff	en avant	oui	procédures de décision et programmation par contraintes	Kiasan/KUnit	Java	[DRH07]
Méthode de Williams, Marre, Mouy, et Roger	en avant	oui	programmation par contraintes	PathCrawler	C	[WMMR05, BDHTH+09]
Méthode de Godefroid, Klarlund, et Sen	en avant	oui	programmation linéaire entière	DART, SMART	C	[GKS05, God07]
Méthode de Sen, Marinov et Agha	en avant	oui	programmation par contraintes	Cute JCute	C Java	[SMA05, SA06]
Méthode de Tillmann, de Halleux et Schulte	en avant	oui	solveur SMT (Z3)	Pex	bytecode .NET	[TdH08, TS06]
Méthode de Bardin et Herrmann	en avant	oui	contraintes sur des vecteurs de bits	Osmose	code exécutable	[BH08]
Méthode de Cadar et al.	en avant	non	solveur de contraintes, procédures de décision	EXE	C	[CGP+06, CGP+08]
Méthode de Gotlieb, Botella et Rueher	en avant	oui	programmation par contraintes	InKa	C, C++	[GBR98, GBW06]
Méthode de Gotlieb	en avant	oui	programmation par contraintes	Euclide	C	[Got09]

FIG. 1.4 – Tableau comparatif des différentes approches de génération de test basées sur le code (2/2)

Chapitre 2

Mécanismes de la programmation par contraintes

De nombreuses approches actuelles pour la génération automatique de données de test se basent sur la programmation par contraintes. Notre approche exploite également la programmation par contraintes afin de modéliser la sémantique du bytecode Java pour générer des tests. Ce chapitre contient une brève introduction à la programmation par contraintes, le lecteur désireux d'en savoir plus peut consulter la référence [KS98].

2.1 Problème de satisfaction de contraintes sur des domaines finis

Le problème de satisfaction de contraintes est un problème indécidable dans le cas général. Cependant le problème est décidable dans le cas où les domaines des variables sont finis. De plus, pour certaines classes de problèmes NP-complets de satisfaction de contraintes sur des domaines finis, il existe des algorithme de résolution très efficaces en pratique, d'où l'attractivité de ce type de contraintes.

Definition 2.1 (CSP) [1] *Un problème de satisfaction de contraintes (CSP pour* Constraint Satisfaction Problem*) est un triplet* $\langle V, C, D\rangle$ *où*

- V *est un ensemble fini de variables,*
- $D = \{dom(x) | x \in V\}$ *est un ensemble de domaines finis associés aux variables,*
- C *est un ensemble de contraintes*
  Soit $W \subseteq V$*, on note* D^W *l'ensemble de n-uplets sur* W*. Chaque contrainte* c *est un couple* (W, R) *où*
  - $W \subseteq V$ *est l'ensemble des variables sur lesquelles elle porte, il est noté* $vars(c)$*,*
  - $R \subseteq D^W$ *est l'ensemble des solutions de* c *et est noté* $sol(c)$

Le domaine des variables est en général numérique (nombres entiers) ou symbolique (termes, structures de données).

Chaque contrainte de C exprime une relation entre des variables de V et spécifie les combinaisons de valeurs autorisées pour ce sous-ensemble de variables.

1. Nous utilisons les notations courantes en intelligence artificielle. Une autre manière de présenter un CSP, en termes cette fois de logique, est de le voir comme une formule du premier ordre : $\phi(x_1, ..., x_n) = R_1(\overline{s_1}) \wedge ... \wedge R_q(\overline{s_q})$. La question est alors de déterminer si ϕ est satisfiable.

Exemple 1. Soit la contrainte $X < Y$ qui lie les variables entières X et Y de domaines respectifs $[1..5]$ *et* $[3..5]$*, X ne peut pas prendre la valeur* 4 *quand Y vaut* 3*.*

On distingue les **contraintes primitives** et les **contraintes non primitives**. Les contraintes primitives utilisent l'opérateur ensembliste $\in$, les opérateurs arithmétiques $\{+, -, \times, \div, modulo\}$ et des relations $\{>, \geq, =, \neq, \leq, <\}$. La négation d'une contrainte primitive c, notée $\neg\, c$ est aussi une contrainte primitive. Par opposition, une contrainte notée $element(X, LISTE, Y)$, qui contraint le $X^{\text{ième}}$ élément de la liste $LISTE$ à valoir Y, est une contrainte non primitive ou opérateur à contraintes.

Définition 2.2 (instanciation d'une variable) *Une variable x est instanciée lorsqu'une valeur lui est affectée. Son domaine $dom(x)$ est alors réduit à une unique valeur (singleton).*

Définition 2.3 (satisfaction d'une contrainte) *Une instanciation des variables de $vars(c)$ satisfait la contrainte c ssi elle appartient à $sol(c)$.*

Définition 2.4 (solution d'un CSP) *Une solution d'un CSP est une instanciation de ses variables telle que toutes les contraintes de C soient satisfaites.*

Résoudre un CSP consiste à donner une ou plusieurs solutions de ce CSP. Le problème de satisfaction de contraintes général est NP-complet. Cependant, il existe des classes de CSP pour lesquels des algorithmes ont été développés pour rendre la recherche de solution efficace en pratique. Ainsi le mécanisme de propagation de contraintes vise à obtenir une solution plus rapidement qu'en énumérant toutes les instanciations possibles de l'ensemble des variables.

2.2 Résolution d'un système de contraintes sur des domaines finis

Pour trouver une solution à un CSP, ou système de contraintes à domaines finis, il est possible d'énumérer tous les n-uplets possibles et de vérifier s'ils satisfont ou non toutes les contraintes du système. Cependant, cela s'avère impraticable pour des problèmes de taille moyenne en raison du grand nombre de combinaisons possibles. Utiliser un processus de propagation de contraintes avant l'énumération permet de réduire l'espace de recherche. La propagation de contraintes consiste à regarder les contraintes séparément les unes des autres, et à supprimer des domaines des variables les valeurs impossibles. Ce faisant,chaque réduction de domaine définit un CSP équivalent au CSP initial (c'est-à-dire ayant les mêmes solutions) mais pour lequel la recherche d'une solution par énumération est plus rapide car les domaines sont plus restreints.
Les processus de propagation et d'énumération sont en général entrelacés.

2.2.1 Propagation de contraintes

Le processus de propagation de contraintes exploite une *file* qui contient les contraintes susceptibles de permettre une réduction des domaines des variables du problème. Initialement, toutes les contraintes du problème s'y trouvent. Chacune des contraintes de la file est évaluée à son tour par le processus de propagation dans le but de supprimer certaines

valeurs des domaines des variables incompatibles avec la satisfaction de la contrainte considérée. De telles valeurs sont dites inconsistantes pour le système de contraintes car elles ne font pas partie des éventuelles solutions du système. Pour déterminer des valeurs inconsistantes, un *algorithme de filtrage* est associé à chaque contrainte. Les algorithmes de filtrage se basent sur la notion de *consistance*. Deux types de consistances sont présentées ici : la consistance de domaine et la consistance de bornes.

Définition 2.5 (consistance de domaine) *Soit un CSP $\langle V, C, D\rangle$, une contrainte c de C est consistante de domaine ssi pour chaque variable v de $vars(c)$ et pour chaque valeur de $dom(v)$ il existe une instanciation des variables de $vars(c) - \{v\}$ telle que c soit satisfaite.*
Les algorithmes de filtrage ne peuvent en général pas garantir cette consistance lorsque les domaines des variables sont grands. D'une part il faut tester beaucoup d'instanciations, et d'autre part stocker l'appartenance ou non de chaque valeur au domaine de la variable peut se révéler trop lourd à traiter.

En présence de domaines larges, la consistance de bornes est une alternative à la consistance de domaine pour les contraintes portant sur des variables entières. La consistance de bornes raisonne sur les valeurs minimum et maximum que les variables peuvent prendre.
Définition 2.6 (consistance de bornes) *Soit un CSP $\langle V, C, D\rangle$, une contrainte c de C est consistante de bornes ssi pour chaque variable v de $vars(c)$, en instanciant v à $min(dom(v))$ (resp. à $max(dom(v))$), il existe une instanciation des variables de $vars(c) - \{v\}$ telle que c soit satisfaite.*

Quand le domaine d'une variable est réduit par l'évaluation d'une contrainte, chacune des autres contraintes qui impliquent cette variable est *réveillée*, c'est-à-dire qu'elle retourne dans la file de contraintes si elle n'y est pas déjà. En effet, une telle contrainte peut exploiter la réduction du domaine de la variable pour réduire les domaines des autres variables sur lesquelles elle porte, et doit donc être réévaluée par le processus de propagation de contraintes.

Si le domaine d'une variable devient vide, la résolution échoue : le système de contraintes est *insatisfiable*. Si la contrainte qui est en cours d'évaluation *réussit*, c'est-à-dire si chacun des n-uplets des domaines courants sont compatibles avec la contrainte, alors la contrainte est retirée du système de contraintes et passe dans l'état *impliquée*. Dans ce cas, la contrainte ne sera plus considérée dans le processus de résolution du système car elle n'est plus utile.

Quand plus aucune déduction n'est possible (un point fixe est atteint), la file devient vide. En effet, comme les domaines des variables n'évoluent plus, plus aucune contrainte n'est réveillée et donc n'est remise en file. Le processus de propagation de contraintes s'arrête.

Soit la contrainte de supériorité stricte dont les caractéristiques sont données sur la figure 2.2. Sur cette figure apparaissent :
- l'*algorithme de filtrage* qui permet de supprimer des valeurs du domaine des variables liées par la contrainte lorsque celle-ci est évaluée,
- la *condition d'échec* qui indique sous quelles conditions l'évaluation de la contrainte provoque l'échec de la résolution du système de contraintes,

Etape	File	Contraintes suspendues	Contraintes impliquées	Domaines des variables
0	$X > Y, Y > Z$			$X \in \{1,2,3,4\}, Y \in \{1,2,3\}, Z \in \{1,3\}$
1	$\mathbf{X}>\mathbf{Y}, Y > Z$			$X \in \{2,3,4\}, Y \in \{1,2,3\}, Z \in \{1,3\}$
2	$\mathbf{Y}>\mathbf{Z}$	$X > Y$		$X \in \{2,3,4\}, Y \in \{2,3\}, Z \in \{1\}$
3	$\mathbf{X}>\mathbf{Y}$		$Y > Z$	$X \in \{3,4\}, Y \in \{2,3\}, Z \in \{1\}$
4		$X > Y$	$Y > Z$	$X \in \{3,4\}, Y \in \{2,3\}, Z \in \{1\}$

FIG. 2.1 – Illustration de la propagation de contraintes

- la *condition d'implication* qui indique la condition à satisfaire pour que la contrainte passe dans l'état impliquée indiquant que plus aucune réduction de domaine ne sera possible grâce à la contrainte,
- la *condition de réveil* qui indique sous quelle condition la contrainte est remise en file si elle n'est pas impliquée.

Exemple 2. *Le tableau de la figure 2.1 donne une illustration du mécanisme de propagation, pour un système comportant de telles contraintes de supériorité stricte. Le système de contraintes comporte deux contraintes, $X > Y$ et $Y > Z$, qui sont initialement mises dans la file, et trois variables, X, Y et Z ; dont les domaines initiaux sont donnés à l'étape 0. A l'étape 1, la contrainte $X > Y$ passe en phase d'évaluation (contrainte en gras dans le tableau). Les règles de déduction associées à cette contrainte d'inégalité, via un algorithme de filtrage basé sur la consistance de domaine, retirent la valeur 1 du domaine de X car aucune valuation telle que X vaille* 1 *ne satisfait la contrainte courante. Puis la contrainte $X > Y$ est suspendue et c'est au tour de la seconde contrainte, $Y > Z$ d'être évaluée (étape 2). Cette évaluation retire 1 du domaine de Y et 3 du domaine de Z. La modification du domaine de Y réveille la contrainte suspendue $X > Y$ car celle-ci implique la variable Y : elle est réintroduite dans la file. La contrainte $Y > Z$ est impliquée car toutes les combinaisons de valeurs des domaines de Y et Z après ces déductions satisfont la contrainte. La contrainte $X > Y$ est à nouveau évaluée (étape 3), une autre valeur, 2, est retirée du domaine de X. Puis la contrainte est suspendue. La file est vide (étape 4) et la phase de propagation de contraintes s'achève.*

CONTRAINTE TRAITÉE : $X > Y$

ALGORITHME DE FILTRAGE
pour chaque $a \in dom(X)$ **faire**
  si $\nexists b \in dom(Y)$ *tel que* $a > b$ **alors**
    $dom(X) \longleftarrow dom(X) - \{a\}$

pour chaque $b \in dom(Y)$ **faire**
  si $\nexists a \in dom(X)$ *tel que* $a > b$ **alors**
    $dom(Y) \longleftarrow dom(Y) - \{b\}$

CONDITION D'ÉCHEC : $dom(X) = \emptyset \vee dom(Y) = \emptyset$

CONDITION D'IMPLICATION : $\forall (a,b) \in dom(X) \times dom(Y).\ a > b$

CONDITION DE RÉVEIL : $change(X) \vee change(Y)$

FIG. 2.2 – Un exemple de traitement possible de la contrainte $X > Y$ avec un filtrage basé sur la consistance de domaine

Les différents états dans lesquels se trouve une contrainte lors du processus de résolution peuvent s'illustrer par la machine à états de la figure 2.3.

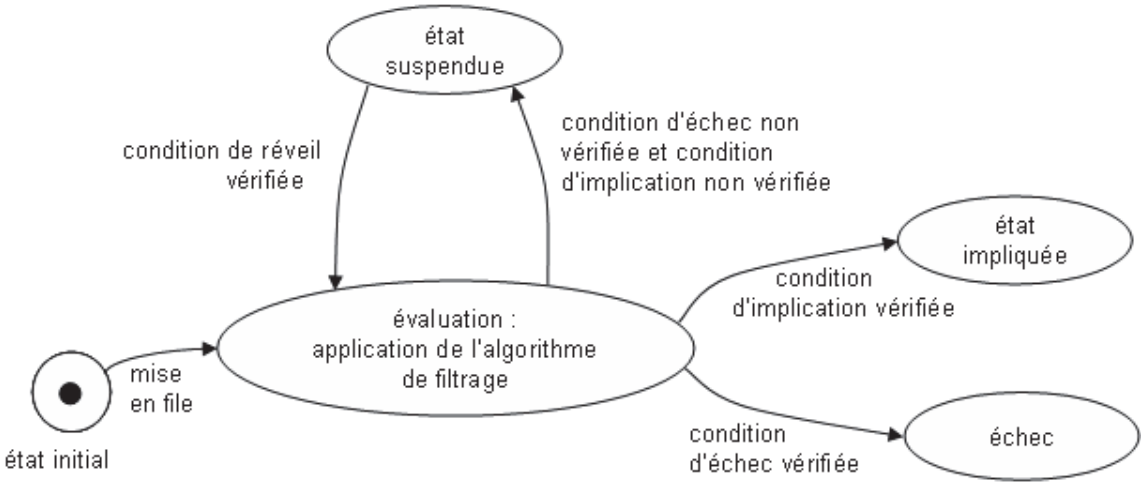


FIG. 2.3 – Les différents états possibles d'une contrainte au cours de la résolution

2.2.2 Énumération des variables

Quand la propagation de contraintes s'achève, c'est-à-dire quand tous les domaines des variables sont stables, l'énumération des valeurs possibles pour les variables est généralement requise pour obtenir une solution. La procédure d'énumération essaie d'affecter une valeur à chacune des variables, l'une après l'autre. Quand une valeur est choisie dans son domaine pour une variable, la propagation de contraintes est relancée pour réduire le domaine des autres variables sous l'hypothèse courante. Si une contradiction apparaît au cours du processus de résolution (si le système devient insatisfiable), la procédure revient en arrière (*backtrack*) pour tester d'autres valeurs. Le processus s'arrête quand la file de contraintes est vide et qu'une valeur est assignée à chaque variable ce qui forme une solution. Si aucune solution n'est trouvée après combinaison de toutes les valeurs des domaines des variables, le système de contraintes n'a pas de solution : il est *insatisfiable*.

Différentes stratégies d'énumération sont possibles dans le but d'obtenir une solution rapidement. En effet, l'ordre dans lequel sont considérées les variables pour leur affecter une valeur, ainsi que l'ordre dans lequel sont affectées les valeurs de son domaine à une variable, peuvent influencer le temps de résolution. Certaines stratégies d'énumération sont indépendantes du système de contraintes à résoudre, comme la stratégie qui consiste à considérer d'abord les variables ayant le plus petit domaine. Des stratégies d'énumération propres au problème à résoudre peuvent aussi s'avérer bénéfiques.

Exemple 3. *Soit le système de contraintes utilisé dans l'exemple 2. A l'issue de la propagation,* $dom(X) = \{3, 4\}$, $dom(Y) = \{2, 3\}$ *et* $dom(Z) = \{1\}$. *L'énumération a pour but de fixer des valeurs à* X *et* Y *(*Z *étant déjà instancié) telles que les contraintes* $X > Y$ *et* $Y > Z$ *soient satisfaites. Supposons que l'énumération considère d'abord* X *puis* Y*, et que la première valeur affectée à* X *soit* 3. *En ce cas, la propagation de contraintes est relancée comme illustré sur la figure 2.4. La contrainte* $X > Y$ *est la seule restant en file donc la seule à considérer, elle permet de supprimer la valeur 3 du domaine de* Y *(étape*

Etape	File	Contraintes suspendues	Contraintes impliquées	Domaines des variables
0	$X > Y$		$Y > Z$	$X \in \{3\}, Y \in \{2,3\}, Z \in \{1\}$
1	$\mathbf{X}>\mathbf{Y}$		$Y > Z$	$X \in \{3\}, Y \in \{2\}, Z \in \{1\}$
2			$Y > Z, X > Y$	$X \in \{3\}, Y \in \{2\}, Z \in \{1\}$

FIG. 2.4 – Illustration de l'énumération

1) puis passe dans l'état impliquée (étape 2). Les domaines des trois variables sont réduits à des singletons et plus aucune contrainte ne reste à considérer, $X = 3$, $Y = 2$, $Z = 1$ *est donc une solution du système.*

Deuxième partie

Modélisation par contraintes du bytecode Java pour la génération de données de test

Chapitre 3

Intérêt et aperçu du modèle à contraintes d'une JVM

Dans le but de contribuer à l'automatisation de la génération de données de test pour couvrir un programme en bytecode Java, cette thèse propose un modèle à contraintes d'une JVM (Java Virtual Machine, permettant d'exécuter le bytecode). Après un rappel des contributions de cette thèse, l'intérêt d'un tel modèle à contraintes pour attaquer cette problématique de test est présenté. Un aperçu des déductions que le modèle à contraintes proposé permet d'obtenir sur les données de test à soumettre à un programme est ensuite montré, suivi d'un aperçu plus détaillé des contraintes menant à ces déductions. Enfin les difficultés principales pour la conception du modèle à contraintes sont exposées.

3.1 Rappel des contributions

Voici les principales contributions de cette thèse.

La première contribution est **la définition d'un modèle à contraintes de la sémantique du bytecode Java**. Étant donnée une séquence d'instructions, le modèle a pour but de permettre la détermination d'un état de la mémoire en début de séquence, s'il en existe, tel que celle-ci soit exécutée. Il ne nécessite pas de faire d'hypothèse a priori sur la forme de la mémoire et les valeurs numériques, mais les infère à partir des instructions conditionnelles de la séquence d'instructions. La modélisation par contraintes du bytecode Java a nécessité d'une part la définition d'un modèle de la mémoire, décrit à la section 4.2, et de nouvelles contraintes, décrites au chapitre 5, traduisant le lien entre les états de la mémoire avant et après l'exécution de chacune des instructions. L'utilisation de variables pour modéliser le type des objets permet de prendre en compte l'héritage et le polymorphisme. Comme cela sera montré dans ce manuscrit, notre modèle est très déductif car il permet d'inférer des formes de mémoire complexes pour satisfaire les contraintes posées. Il ne fait pas de sous-approximations, y compris en présence d'expressions non linéaires et de déréférencements multiples (par exemple `p.left.parent`). Les états de la mémoire obtenus à l'issue de la propagation de contraintes sont une sur-approximation des états de la mémoire possibles. Ainsi, aucun état de la mémoire en entrée qui permettrait l'exécution de la séquence d'instructions choisie n'est indûment écarté.

La deuxième contribution est **une méthode de génération automatique de données de test** pour le bytecode Java,, détaillée au chapitre 6. Cette méthode vise à couvrir des instructions particulières non couvertes par d'autres méthodes de test. La méthode

de génération proposée raisonne dans le sens inverse à celui de l'exécution : en partant d'un objectif (une instruction à couvrir) elle essaie de trouver un chemin menant vers le point d'entrée du programme en explorant progressivement et à l'envers le graphe de flot de contrôle, puis de déterminer une donnée d'entrée qui active un tel chemin. Pour ce faire, elle exploite le modèle à contraintes du bytecode. Le sens de parcours "en arrière" du graphe de flot de contrôle est original, et favorise la couverture d'instructions non couvertes par d'autres outils de test qui utilisent une exploration en avant.

La troisième contribution est **un prototype**, JAUT, mettant en application le modèle et la méthode de génération proposés et permettant ainsi une validation expérimentale. L'outil est décrit au chapitre 8. Les expériences, décrites au chapitre 9, montrent que JAUT permet d'augmenter la couverture des instructions obtenue avec les autres outils disponibles [CG10], en particulier Pex qui est une référence dans le domaine.

3.2 Intérêt du modèle

Nous proposons une méthode de génération automatique de données de test orientée chemins. Pour un chemin allant de l'entrée du programme à l'objectif de test, les conditions à remplir par les données d'entrée pour que le chemin soit exécuté (conditions de chemins) sont exprimées sous la forme d'un système de contraintes. Les méthodes existantes de génération de données de test orientées chemins pour les programmes orientés objet ou impératifs ont chacune leurs limitations. Comme détaillé dans l'état de l'Art, certaines méthodes ([GKS05, God07, SMA05, SA06, TdH08]) utilisent des conditions de chemins incluant des approximations, ceci en raison des limites du modèle mémoire, des solveurs de contraintes ou des prouveurs qu'elles utilisent qui ne permettent pas de traiter des conditions de chemins exactes. Les méthodes qui expriment avec exactitude les conditions de chemins ([DM03b, MLK04, WMMR05]) sont quant à elles limitées dans le traitement des alias de pointeurs ou références.

Nous proposons une méthode de génération automatique de données de test sans approximation des conditions de chemins, qui exploite un modèle mémoire et des contraintes capables d'exprimer les relations d'alias de pointeurs, l'héritage et le polymorphisme. Son but est d'obtenir une couverture de 100% des instructions exécutables. La contrepartie de l'absence d'approximation étant la complexité du modèle mémoire et des contraintes utilisées, la méthode proposée se veut être une méthode complémentaire aux méthodes déjà existantes, qui permet de couvrir des instructions que ces méthodes ne couvrent pas. Notre méthode contribue à la résolution du problème d'atteignabilité d'une instruction d'un programme en bytecode Java. Rappelons que ce problème est un problème indécidable dans le cas général [Wey79].

Le principe général de notre méthode est le suivant. Étant donnée une instruction à couvrir d'un programme en bytecode Java, le graphe de flot de contrôle de la méthode sous test est parcouru d'une manière guidée par l'objectif afin d'énumérer tous les chemins du graphe reliant l'entrée de la méthode à cette instruction cible (le parcours est détaillé au chapitre 6). Une fois un chemin sélectionné, le problème à résoudre est alors le suivant : étant donnée une suite d'instructions successives dans le graphe de flot de contrôle, séparant l'entrée de la méthode de l'instruction cible, peut-on déterminer un état de la mémoire en début de cette suite d'instructions tel que celle-ci soit exécutée ? L'état de la mémoire

en entrée de méthode donne en effet la valeur des paramètres avec lesquels elle est appelée. *Le modèle à contraintes proposé vise donc à résoudre le problème de trouver un état de la mémoire en début d'une séquence d'instructions en bytecode Java tel que cette séquence soit exécutée.* Selon la séquence d'instructions, les conditions de chacune des instructions conditionnelles de la séquence doivent être vérifiées ou non, ce qui limite les valeurs possibles pour l'état de la mémoire en entrée de séquence. Ainsi la satisfaction de la condition `p.next==p` nécessite que `p` désigne une structure de données cyclique avant le test de la condition, comme illustré sur la figure 3.1, la valeur de `p` étant potentiellement liée à l'état de la mémoire en début de séquence.

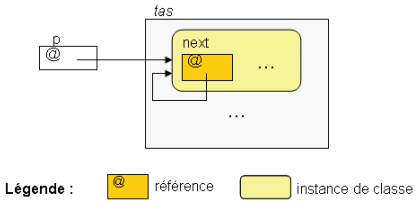


FIG. 3.1 – Extrait d'un état de la mémoire avant le test de la condition `(p.next==p)` tel que la condition soit satisfaite. `p` doit référencer un objet `o` dont l'attribut `next` référence également `o`

Le modèle à contraintes que nous proposons modélise la sémantique relationnelle du bytecode Java et permet un raisonnement "dans les deux sens". Il donne la possibilité de déterminer un état de la mémoire en début d'une séquence d'instructions tel que celle-ci soit exécutée, ou de détecter que la séquence est non exécutable. Il permet également de poser des contraintes sur l'état de la mémoire en sortie de séquence, par exemple une référence est contrainte à être nulle ou un entier est contraint à être supérieur à une constante, et de réduire ainsi les possibilités pour les états de la mémoire en entrée de séquence grâce au processus de propagation de contraintes.

3.3 Aperçu du modèle

Afin d'exprimer le lien existant entre deux états de la mémoire séparés par une séquence d'instructions, chacune des instructions de la séquence est elle-même traduite sous la forme d'une relation entre deux états de la mémoire : l'état avant et l'état après l'exécution de l'instruction. Les états de la mémoire mis en relation sont en réalité des modèles des états de la mémoire, ce sont des *états de la mémoire sous contraintes*, abrégé en *EMC*. Un EMC contient l'information connue sur l'état physique de la mémoire à un instant de la résolution du problème, par exemple les objets obligatoirement présents dans le tas ou encore une sur-approximation des valeurs possibles pour les variables locales ou les attributs des objets. Un EMC est formé de variables sous contraintes, quand toutes ces variables sont instanciées l'EMC modélise un unique état de la mémoire physique. A l'inverse, quand toutes ces variables ne sont pas instanciées l'EMC englobe plusieurs états physiques de la mémoire.
Cette section a pour but de donner un aperçu des états mémoires sous contraintes (EMC) et des relations qui expriment les effets de chacune des instructions bytecode sur la mémoire. La modélisation des EMC sera détaillée en section 4.2, la traduction des instructions sous forme de relations entre deux EMC, notamment maintenues par des contraintes, sera

```
public class Coord {

       private int x,y;

       public Coord(int cx,int cy){ x = cx; y = cy;}

       public Coord deplaceY(Chrono chrono, int vitesse){
              if(chrono.temps <= 0 || vitesse <= 0)
                     return this;
              int ytemp = y + chrono.temps * vitesse;
              chrono.temps = 0;
              if( ytemp > 65536 )
                     return new Coord(x,65536);//INSTRUCTION CIBLE
              else
                     return new Coord(x,ytemp);
       }
}//classe Coord


public class Chrono {

       public int temps;
       ...

}//classe Chrono
```

Listing 3.1 – Code Java de la classe `Coord` qui modélise des coordonnées

détaillée en chapitre 5.

3.3.1 Exemple détaillé

Le code du listing 3.1 est celui d'une classe `Coord` représentant les coordonnées cartésiennes d'un point : `x` et `y`. La méthode `deplaceY` permet de créer de nouvelles coordonnées pour exprimer un déplacement caractérisé par un temps écoulé, donné par un objet de type `Chrono`, et une vitesse entière. Le résultat de la compilation de cette méthode en bytecode Java est donné [1] sur le listing 3.2. Si l'on veut atteindre l'instruction cible 49, il n'y a ici pas d'autre possibilité que d'exécuter la séquence d'instructions bytecode $0, 1, 4, 7, 8, 13, 14, 17, 18, 21, 22, 23, 24, 25, 26, 27, 30, 31, 33, 36, 39, 40, 41, 44, 46, 49$. Trouver des valeurs de paramètres telles que l'instruction soit exécutée revient à trouver un état de la mémoire en entrée tel que cette séquence d'instructions soit exécutée. Cet état de la mémoire en entrée est un EMC dans notre modèle. Un tel état est trouvé quand toutes les variables que l'EMC comporte sont instanciées et satisfont les contraintes à respecter pour que la séquence soit exécutée.

Les instructions de la séquence sont traduites en relations entre EMC, ce qui génère un système de contraintes à satisfaire pour obtenir un état de la mémoire en entrée. Nous allons montrer l'évolution de l'EMC en entrée de la méthode *deplaceY* au fur et à mesure

1. sous une forme lisible grâce à la commande javap du JDK de Sun

```
public Coord deplaceY(Chrono, int);
  Code:
   Stack=4, Locals=4, Args_size=3
   0:  aload_1          //empile le contenu du registre 1
   1:  getfield #4;     //remplace le sommet de pile par la valeur de l'attribut
       temps
   4:  ifle 11          //si le sommet de pile est inférieur ou égal à 0, aller
       en 11
   7:  iload_2          //empile le contenu du registre 2
   8:  ifgt 13          //si le sommet de pile est plus grand que 0 aller en 13
   11: aload_0
   12: areturn
   13: aload_0          //empile le contenu du registre 0
   14: getfield #3;     //remplace le sommet de pile par la valeur de l'attribut
                        //y
   17: aload_1          //empile le contenu du registre 1
   18: getfield #4;     //remplace le sommet de pile par l'attribut temps
   21: iload_2          //empile le contenu du registre 2
   22: imul             //multiplie les deux valeurs entières en sommet de pile
                        //et les remplace par le résultat
   23: iadd             //additionne les deux valeurs entières en sommet de pile
                        //et les remplace par le résultat
   24: istore_3         //stocke le sommet de pile dans le registre 3
   25: aload_1          //empile le contenu du registre 1
   26: iconst_0         //empile la constante 0
   27: putfield #4;     //modifie la valeur de l'attribut temps
   30: iload_3          //empile le contenu du registre 3
   31: ldc #5;          //charge la constante 65536
   33: if_icmple 50     //si le deuxième élément de la pile est inférieur ou égal
                        //au premier élément, aller en 50
   36: new #6;          //alloue dynamiquement de la mémoire pour un objet de
                        //type Coord
   39: dup              //duplique le sommet de pile
   40: aload_0          //empile le contenu du registre 0
   41: getfield #2;     //remplace le sommet de pile par la valeur de l'attribut
                        //x
   44: ldc #5;          //empile la constante 65536
   46: invokespecial #7;//invoque le constructeur de la classe Coord
   49: areturn          //retourne l'élément en sommet de pile
                        //INSTRUCTION CIBLE
   50: new #6;
   53: dup
   54: aload_0
   55: getfield #2;
   58: iload_3
   59: invokespecial #7;
   62: areturn
```

Listing 3.2 – Code machine de la méthode `deplaceY` de la classe `Coord`

de la résolution du système de contraintes, telle qu'elle se fait dans le prototype JAUT qui implante le modèle à contraintes proposé.
Voici un extrait de l'EMC en entrée de méthode à un instant de la propagation de contraintes initiale :

```
lin: this  -<[Coord],  dom=[0] ndom=[]>,
     chrono-<[Chrono], dom=all ndom=[]>,
     vitesse -< int,   [-268435456..268435455]>,
hin:
   0::([Coord],[x-<int,[-268435456..268435455]>,
                y-<int,[-268435456..268435455]>])
```

`lin` désigne ici les **registres** [2] qui stockent la référence de l'objet sur laquelle s'applique la méthode (*this*) si la méthode n'est pas statique, les paramètres de la méthode, et les variables locales à la méthode (ces dernières ne sont pas montrées ici). `hin` décrit le **tas** qui est une zone de mémoire dans laquelle sont stockés les objets alloués dynamiquement (instruction Java `new`). Une autre zone de stockage des données est la **pile d'opérandes**, omise ici car elle est forcément vide en entrée de méthode. Les types de données manipulées par une JVM sont les **entiers**, les **flottants**, et les **références** qui sont des adresses du tas permettant d'accéder aux objets. `lin` contient deux références (*this* et *chrono*) et un entier (*vitesse*). L'objet désigné par *this* est déjà créé dans le tas `hin` à l'adresse 0, car invoquer la méthode *deplaceY* nécessite qu'un objet de type *Coord* soit créé. *this* le désigne effectivement car son domaine (dom) ne comporte que l'adresse *0*. *chrono* au contraire ne désigne pour l'instant aucun objet du tas, il peut encore valoir *null* ou référencer n'importe quel objet du tas (dom=all). Les variables entières vitesse, *this.x* et *this.y* sont des entiers 32-bits dans le programme en bytecode Java, les variables les modélisant ont le domaine par défaut des entiers pour le solveur utilisé [3].

La résolution du système de contraintes est classique et fait intervenir deux processus entrecroisés : la propagation de contraintes qui exploite les contraintes pour élaguer le domaine des EMC en supprimant des valeurs inconsistantes, et l'énumération qui énumère progressivement les valeurs possibles pour les variables de l'EMC et lance à chaque hypothèse faite le processus de propagation. A un moment donné de la propagation de contraintes, l'EMC d'entrée suivant est obtenu :

```
lin: this  -<[Coord],  dom=[0] ndom=[]>,
     chrono-<[Chrono], dom=all ndom=[null,0]>,
     vitesse -<int,[1..268435455]>,
hin:
  0::([Coord],[x-<int,[-268435456..268435455]>,
               y-<int,[-268435456..268435455]>])
```

L'ensemble *ndom* de chrono, qui indique des adresses que chrono ne peut pas référencer, vaut maintenant *[null,0]*, ce qui signifie que chrono ne peut pas être nul, ni référencer l'objet stocké à l'adresse *0*. En effet, atteindre l'instruction 49 du bytecode nécessite que chrono soit non nul car plusieurs instructions du programme le déréférencent (`chrono.temps`). De plus *chrono* doit référencer un objet de type *Chrono* ce qui n'est pas le cas avec l'objet stocké à l'adresse *0*. Le domaine de la vitesse a été élagué en raison de la contrainte *vitesse*

2. variables locales dans la spécification Sun. 3. les entiers sont actuellement limités à 28 bits dans notre outil en raison de l'utilisation du solveur clpfd de SICStus prolog, mais il ne s'agit pas d'une limitation de notre modèle

> 0.
Ensuite, l'énumération génère un objet de type *Chrono* référencé par *chrono*, et essaie d'instancier l'une après l'autre les variables de type entier. Supposons que ce processus ait instancié *vitesse*, *this.x* et *this.y* à *1*, nous obtenons alors l'EMC suivant :

```
lin: this  -<[Coord],  dom=[0] ndom=[]>,
     chrono-<[Chrono], dom=[1] ndom=[]>,
     vitesse -< int,[1]>,
hin:
  0::([Coord],[x-<int,[1]>,
               y-<int,[1]> ]),
  1::([Chrono],[temps-<int,[65536..268435454]> ])
```

Un objet de type *Chrono* est créé à l'adresse *1* et est référencé par *chrono* dont le domaine est maintenant réduit à l'adresse *1*. Son attribut *chrono.temps* a pour domaine [65536..268435454] à cette phase de la résolution car les contraintes $ytemp = this.y + chrono.temps * vitesse$ et $ytemp > 65536$ ont été prises en compte par le solveur de contraintes à domaines finis. Enfin la variable *chrono.temps* est instanciée, et l'EMC suivant est obtenu :

```
lin: this  -<[Coord],  dom=[0] ndom=[]>,
     chrono-<[Chrono], dom=[1] ndom=[]>,
     vitesse -< int,[1]>,
hin:
  0::([Coord],[x-<int,[1]>,
               y-<int,[1]> ]),
  1::([Chrono],[temps-<int,[65536]> ])
```

Toutes les variables de l'EMC sont ainsi instanciées ; l'EMC caractérise un état de la mémoire d'une JVM en entrée de méthode qui permet d'atteindre l'instruction cible 49.

3.3.2 Aperçu des EMC et des relations qui les lient

Nous appelons état de la mémoire l'état des registres pour la méthode courante (les valeurs stockées), de la pile d'opérandes pour la méthode courante (la taille de la pile et les valeurs empilées) et du tas (la forme du tas et les valeurs des instances stockées). Nous proposons une modélisation des données manipulées par une JVM et des zones de stockage de ces données pour créer des états de la mémoire sous contraintes (EMC).

Les variables entières manipulées par le programme sont des variables entières à domaines finis dans notre modèle [4]. Le domaine de ces variables à domaines finis est une sur-approximation des valeurs possibles des variables du programme qu'elles modélisent. Leur domaine initial dépend du type de l'entier (`int`, `long`,...) dans le programme. Les références manipulées par le programme sont représentées dans notre modèle par des variables dont le domaine peut être considéré comme un ensemble d'adresses qui désignent des zones mémoires du tas vers lesquelles la référence peut pointer.

Les zones de stockage des données sont les registres, la pile d'opérandes et le tas. Les registres sont modélisés par une fonction qui à un numéro de registre associe la variable

4. Pour une variable entière v, $dom(v) \in \mathcal{P}(\mathbb{N})$.

modélisant la valeur qui y est stockée. La pile est représentée par une séquence de variables dont le premier élément est le sommet de pile. Le tas est modélisé comme une fonction dont le domaine de définition est un ensemble d'entiers. Cette fonction associe à chaque adresse (représentée par un entier) l'instance de classe qui lui est associée dans le tas. Une fonction qui modélise le tas est une variable de notre problème, des contraintes peuvent porter sur ce type de variables. Ces contraintes ainsi que la représentation des instances de classe seront détaillées dans les sections 4.2 et 5.3.7.

Soit la méthode `deplaceY` du listing 3.2. Pour trouver une mémoire en entrée pour la séquence d'instructions qui mène à l'instruction 49, une possibilité est de suivre l'ordre d'exécution des instructions pour générer des états mémoires et poser les contraintes qui les lient [5]. Dans ce cas un premier EMC est généré, qui est la mémoire en entrée. Puis en considérant cet EMC et la première instruction de la séquence, l'EMC qui suit l'instruction est généré, et des contraintes qui lient des variables de ces deux EMC sont éventuellement posées. On continue de même pour les autres instructions de la séquence. Ce faisant, les EMC et le système de contraintes donnés sur la figure 3.2 sont générés. Nous notons $m_j = (f_j, s_j, H_j)$ l'EMC après l'instruction j, avec f_j les registres, s_j la pile d'opérandes et H_j le tas. Cette notation est possible pour l'exemple détaillé car la séquence ne contient pas deux fois la même instruction [6]. Par souci de simplicité, l'appel au constructeur via l'instruction `invokespecial` n'est pas détaillé, son effet est directement pris en considération.
ϵ est la séquence vide, $v.s$ désigne la pile s sur laquelle v est empilé.

On distingue quatre étapes dans l'exécution : la vérification de la positivité du temps et de la vitesse, le calcul de la valeur affectée à `ytemp`, la mise à 0 de l'attribut `temps` de l'instance de classe référencée par `chrono` et la création de la nouvelle instance de type `Coord`.

Etape 1. L'instruction 0 charge sur la pile la référence $Chrono_r$ stockée dans le premier registre f0(1). L'instruction 1 dépile la référence $Chrono_r$ pour la remplacer par la valeur de l'attribut temps de l'instance qu'elle référence dans le tas $H0$. La contrainte $getfield(H0, 4, Chrono_r, Temps_i)$ exprime cette relation. La référence $Chrono_r$ doit être non nulle pour pouvoir être déréférencée (accès à un attribut) sans provoquer d'erreur. L'instruction 4 est l'instruction conditionnelle `ifle 11`. Si le sommet de pile est inférieur ou égal à 0, l'exécution doit se poursuivre avec l'instruction 11, sinon elle se poursuit avec l'instruction suivante 7. Comme dans la séquence d'instructions qui nous intéresse l'instruction 7 suit l'instruction 4, le sommet de pile $Temps_i$ doit être strictement supérieur à 0 ce qui est traduit par une contrainte arithmétique. L'instruction 7 charge le contenu du registre 2, la vitesse Vit_i, sur la pile. L'instruction 8 est une instruction conditionnelle. Comme l'instruction 13 la suit dans la séquence, l'instruction 8 contraint la vitesse Vit_i à être strictement positive.

5. une autre possibilité, qu'exploite notre outil pour générer des données de test, est de partir de l'instruction à atteindre pour remonter vers le point d'entrée de la méthode. Cependant, le raisonnement avant étant plus habituel que le raisonnement arrière, il est privilégié dans cette section 6. Pour une séquence dans laquelle la même instruction figure plusieurs fois (si elle fait partie d'une boucle), une mémoire après exécution différente est générée pour chaque occurrence de l'instruction : une notation indexant M, f, s et H par le numéro d'instruction n'est pas possible, car elle ne permettrait pas la distinction des différentes exécutions de la même instruction. Une telle notation, qui n'est donc pas générique, est choisie ici pour faciliter la lecture en permettant de relier facilement un état de la mémoire à l'instruction correspondante.

Instruction interprétée	Génération des mémoires	Contraintes créées
initialisation	$f0 = \{0 \mapsto This_r, 1 \mapsto Chrono_r, 2 \mapsto Vit_i, 3 \mapsto YTemp_i\}$ $m_{in} = (f0, \epsilon, H0)$	
Etape 1		
0 : `aload_1`	$m_0 = (f0, Chrono_r, H0)$	
1 : `getfield #4`	$m_1 = (f0, Temps_i, H0)$	$Chrono_r \neq null$, $getfield(H0, 4, Chrono_r, Temps_i)$
4 : `ifle 11`	$m_4 = (f0, \epsilon, H0)$	$Temps_i > 0$ car l'instruction 7 suit dans la séquence
7 : `iload_2`	$m_7 = (f0, Vit_i, H0)$	
8 : `ifgt 13`	$m_8 = (f0, \epsilon, H0)$	$Vit_i > 0$ car l'instruction 13 suit dans la séquence
Etape 2		
13 : `aload_0`	$m_{13} = (f0, This_r, H0)$	
14 : `getfield #3`	$m_{14} = (f0, Y_i, H0)$	$This_r \neq null$, $getfield(H0, 3, This_r, Y_i)$
17 : `aload_1`	$m_{17} = (f0, Chrono_r.Y_i, H0)$	
18 : `getfield #4`	$m_{18} = (f0, Temps_i.Y_i, H0)$	$Chrono_r \neq null$, $getfield(H0, 4, Chrono_r, Temps_i)$
21 : `iload_2`	$m_{21} = (f0, Vit_i.Temps_i.Y_i, H0)$	
22 : `imul`	$m_{22} = (f0, Mul_i.Y_i, H0)$	$Mul_i = Vit_i * Temps_i$
23 : `iadd`	$m_{23} = (f0, Exp_i, H0)$,	$Exp_i = Mul_i + Y_i$
24 : `istore_3`	$f1 = \{0 \mapsto This_r, 1 \mapsto Chrono_r, 2 \mapsto Vit_i, 3 \mapsto Exp_i\}$ $m_{24} = (f1, \epsilon, H0)$	
Etape 3		
25 : `aload_1`	$m_{25} = (f1, Chrono_r, H0)$	
26 : `iconst_0`	$m_{26} = (f1, 0.Chrono_r, H0)$	
27 : `putfield #4`	$m_{27} = (f1, \epsilon, H1)$	$Chrono_r \neq null$, $putfield(H0, H1, 4, Chrono_r, 0)$
Etape 4		
30 : `iload_3 ;`	$m_{30} = (f1, Exp_i, H1)$	
31 : `ldc #5 ;`	$m_{31} = (f1, 65536.Exp_i, H1)$	
33 : `if_icmple 50`	$m_{31} = (f1, \epsilon, H1)$	$Exp_i > 65536$ car l'instruction 36 suit dans la séquence
36 : `new #6 ;`	$m_{36} = (f1, Ref_r, H2)$, $Ref_r = fresh(reference, ad, Coord)$	$new(H1, H2, Ref_r)$
39 : `dup`	$m_{39} = (f1, Ref_r.Ref_r, H2)$	
40 : `aload_0`	$m_{40} = (f1, This_r.Ref_r.Ref_r, H2)$	
41 : `getfield #2`	$m_{41} = (f1, X_i.Ref_r.Ref_r, H2)$	$This_r \neq null$, $getfield(H2, 2, This_r, X_i)$
44 : `ldc #5 ;`	$m_{44} = (f1, 65636.X_i.Ref_r.Ref_r, H2)$	
46 : `invokespecial #7 ;`	$m_{46} = (f1, Ref_r, H3)$ $H2$ et $H3$ sont identiques excepté pour l'objet référencé par Ref_r. Dans $H3$ cet objet a pour valeurs d'attributs X_i pour x et 65636 pour y.	
49 : `areturn`	$m_{49} = (f1, \epsilon, H3)$	

FIG. 3.2 – Mémoires générées et contraintes créées au cours de l'interprétation de la méthode `deplaceY`

Etape 2. L'instruction 13 charge sur la pile la référence $This_r$, stockée dans le registre 0, l'instruction 14 la remplace par la valeur de l'attribut `y` de l'instance de classe qu'elle référence. Les instructions 17 et 18 procèdent de manière similaire pour l'attribut `temps` de l'objet référencé par $Chrono_r$. L'instruction 21 charge depuis le registre 2 le paramètre `vitesse` sur la pile. L'instruction 22 remplace les deux éléments en sommet de pile par le résultat de leur multiplication, d'où la contrainte arithmétique $Mul_i = Vit_i * Temps_r$. L'instruction 23 procède de manière similaire avec l'addition. A cette étape, en sommet de pile se trouve la variable qui modélise le résultat de `y`+`chrono.temps`*`vitesse`, notée Exp_i. L'instruction 24 stocke cette valeur dans le registre 3 qui correspond à la variable locale `ytemp`. Le contenu d'un registre est donc modifié, d'où la création d'une nouvelle fonction f1 pour décrire les registres.

Etape 3. L'instruction 25 charge sur la pile la référence $Chrono_r$ contenue dans le registre 1, et l'instruction 26 empile la constante 0. L'instruction 27 stocke la valeur du sommet de pile, ici 0, dans l'attribut `temps` de l'instance de classe référencée par $Chrono_r$. Cette instruction modifie le tas puisqu'une variable d'instance est mise à jour, la relation $putfield(H0, H1, 4, Chrono_r, 0)$ exprime le lien qui existe entre le tas avant l'instruction de mise à jour, $H0$, et le tas après l'instruction, $H1$.

Etape 4. L'instruction 30 charge sur la pile le contenu Exp_i du registre 3, l'instruction 31 empile la constante 65536. L'instruction 33 est une instruction conditionnelle : l'instruction 36 la suit dans la séquence qui nous intéresse, ce qui impose que Exp_1 soit strictement supérieur à 65536. L'instruction 36 réserve de la place dans le tas pour une instance de classe de type `Coord`, et la référence vers ce nouvel objet est empilée. Le tas est modifié par cette instruction (dans notre modèle une adresse supplémentaire y est ajoutée) ce qui s'exprime par la relation $new(H1, H2, Ref_r)$, où Ref_r référence une adresse ad n'étant pas encore utilisée. L'instruction 39 duplique le sommet de pile. L'instruction 40 empile $This_r$ et l'instruction 41 le remplace par `this.x`. L'instruction 44 empile la constante 65536. L'instruction 46 appelle le constructeur qui initialise les valeurs des attributs de l'objet `Coord` nouvellement créé, ce mécanisme n'est pas détaillé. Enfin l'instruction 46 est l'instruction cible, elle retourne comme résultat l'élément en sommet de pile qui est une référence vers l'objet nouvellement créé.

3.4 Difficultés principales pour la modélisation

3.4.1 Détermination de la forme de la mémoire

La forme de la mémoire en entrée d'une méthode n'est pas toujours connue : la quantité de mémoire allouée pour stocker les paramètres peut ne pas être déterminée (problème *Mem_shape*). Prenons le cas d'une classe *Maillon* permettant de décrire une liste chaînée grâce à deux attributs : un attribut *valeur* de type entier et un attribut *suivant* de type *Maillon*. Si l'on veut modéliser par contraintes le comportement d'une méthode appelée par un *Maillon* m, on ne veut pas faire initialement d'hypothèses sur la valeur de m. Ainsi, on ne sait pas si le champ *suivant* de m vaut `null`, ou s'il référence un autre maillon dont la valeur est elle aussi indéterminée. La figure 3.3 représente quatre cas de figure possibles (mais il en existe une infinité). Dans notre modélisation, l'objet de type *Maillon* appelant la méthode est stocké dans le tas, et l'attribut *suivant* de cet objet est représenté par une variable référence N_r, variable sous contraintes. Initialement (tant que l'on n'a considéré

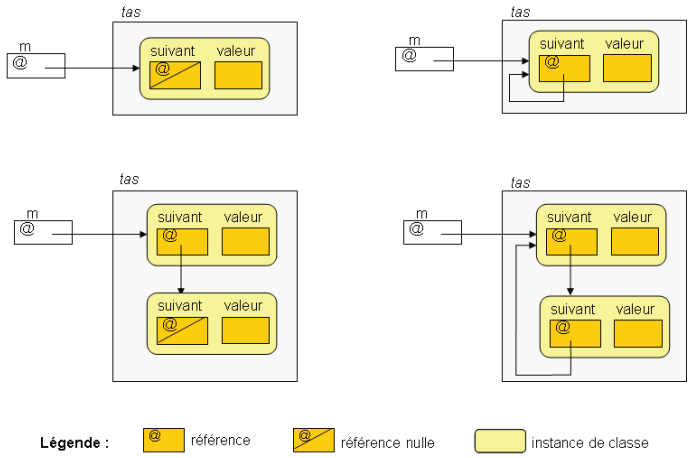


FIG. 3.3 – Exemple d'indétermination de la forme de la mémoire, `m` est une référence vers un objet de type *Maillon*

aucune contrainte) aucune information sur sa valeur n'est disponible. N_r peut référencer n'importe quelle zone de la mémoire ou valoir `null` : on notera ***all*** son domaine initial pour signifier que toutes ces possibilités sont ouvertes. Les possibilités rétrécissent au cours de la résolution, le domaine de N_r devient un ensemble énuméré d'adresses du tas.
Initialement le nombre d'objets en mémoire est inconnu : le tas en entrée de méthode est dit **non clos**. Des objets sont en effet susceptibles d'être ajoutés en mémoire, par exemple selon l'hypothèse qui sera faite sur la valeur de *m.suivant* lors de la phase d'énumération. La notion de tas non clos traduit donc le fait qu'il est partiellement connu : il n'est pas exclu qu'il contienne d'autres objets en plus des objets dont on est certain de l'existence. A l'issue du processus de résolution, le tas est clos et les instances de classe sont connues.

3.4.2 Héritage et polymorphisme

Le polymorphisme en Java permet à une variable déclarée de type C de référencer tout objet appartenant à la famille de types de C. Une **famille de types** est un ensemble de classes qui partagent un comportement commun via la classe de base C, par exemple par héritage. Chaque descendant de C est membre de la famille de C. Si D fait partie de la famille de C, le polymorphisme signifie que n'importe quelle instance de D peut être utilisée librement quand une instance de C est attendue. Toute classe C définit une famille de types dont la classe C fait elle-même partie.
En Java, les appels de méthode peuvent être polymorphes. En effet, soient deux classes C et D telles que D hérite de C, et qui définissent toutes deux la méthode $m()$. Faisons l'hypothèse que la famille de types de C ne contient que C et D. Si une référence p est déclarée de type C et n'est pas nulle, alors p référence soit une instance de type C, soit une instance de type D. Pour un appel de méthode p.m(), le corps de la méthode appelée dépend du type effectif de p lors de l'appel : on parle de liaison dynamique. Si p est de type C, la fonction $C :: m()$ sera appelée, si p est de type D ce sera la fonction $D :: m()$.
Si la méthode modélisée contient un appel de méthode par une instance référencée par un paramètre, alors le corps de la méthode appelée peut dépendre du type effectif de la référence paramètre. Dans notre modèle, le type d'une référence est une variable. De même les instances du tas possèdent une variable de type pour le cas où leur type précis n'est pas encore déterminé. En effet, le modèle est actuellement utilisé pour exprimer l'effet

d'une séquence déterminée d'instructions. Cependant, il a été conçu de manière à ce que son utilisation puisse être élargie, par exemple en imaginant la fusion en un seul EMC (état mémoire sous contraintes) d'informations provenant d'EMC issus de deux chemins différents.

Chapitre 4

Modèle mémoire d'une JVM

4.1 Description d'une JVM

La description donnée ici d'une JVM est tirée de la spécification de Sun [LY99].

L'exécution d'un programme en bytecode par une JVM se fait dans un environnement qui comprend notamment des zones de mémoire pour stocker les données lors de l'exécution du programme. Les types des données manipulées par une JVM sont les **entiers**, les **flottants** [1], et les **références**.
Une JVM est une machine à base de pile, chaque thread a une pile pour stocker les *frames*. Une frame est créée à chaque fois qu'une méthode est invoquée, elle contient notamment deux zones de mémoire propres à la méthode : des **registres** et une **pile d'opérandes**. Le **tas** est une zone de mémoire partagée entre tous les appels de méthodes.
Un programme est une suite d'instructions bytecode exécutées par une JVM. Une instruction consiste en un **mnémonique** spécifiant l'opération à effectuer, suivi par zéro ou plusieurs opérandes servant d'arguments ou de données qui seront utilisés par l'opération (par exemple `aload_0`, `iadd`, `ifle n`). Certaines de ces opérandes font référence à des entrées d'une table de constantes, le ***constant pool*** (par exemple `getfield #3`, `ldc #5`). Il y a un *constant pool* par classe ou interface publique. Il est utilisé notamment pour connaître la valeur de constantes numériques définies dans le programme Java, la signature des méthodes et le type des attributs.

4.1.1 Type des données manipulées par une JVM

4.1.1.1 Les entiers et les booléens

Différents types d'entiers sont définis dans le langage Java : les types entiers sont *byte*, *short*, *int* et *long* dont les valeurs sont les entiers signés respectivement sur 8 bits, 16 bits, 32 bits et 64 bits, et *char* dont les valeurs sont les entiers non signés sur 16 bits. Java définit également le type booléen dont les valeurs sont *true* et *false*.
Tous ces types se retrouvent au niveau de la machine virtuelle Java, hormis le type booléen, le type int étant utilisé à la compilation pour représenter les variables booléennes.

1. Les flottants sont laissés de côté pour le moment, l'arithmétique des flottants étant très délicate à traiter. Leur prise en compte est discutée dans les perspectives (section 11.2.1).

4.1.1.2 Les références

Les références permettent de désigner des objets (instances de classe ou tableaux) créés dynamiquement et stockés dans le tas. Dans cet écrit, nous les considérons comme des adresses du tas. Elles peuvent être de type classe, interface ou tableaux, seules les références de type classe sont décrites ici. Une référence de type `class c` a pour valeur l'adresse dans le tas d'une instance de la famille de *c*. Elle peut également valoir `null`. Le tas, où sont stockées les instances de classe, sera décrit dans la partie 4.1.2.4

4.1.2 Description des zones de stockage des données d'une JVM

4.1.2.1 Notion de *word*

La machine virtuelle définit la notion abstraite de *word*, qui est une quantité de mémoire qui dépend de l'implémentation. [LY99] donne en effet la spécification d'une machine abstraite et non d'une implémentation particulière d'une JVM. Un *word* est suffisant pour stocker des valeurs entières de type *byte*, *char*, *short*, *int*, ainsi que les références. Deux *words* sont utilisés pour stocker des valeurs entières de type *long*.

4.1.2.2 Les registres

A chaque invocation de méthode, la machine virtuelle Java alloue une *frame* qui contient un tableau de *words* pour stocker les valeurs des variables propres à la méthode : paramètres, référence de l'objet qui invoque la méthode si la méthode n'est pas statique, variables locales de la méthode. Les valeurs qui y sont stockées sont donc soit de type primitif (entier, flottant) soit des références. Les cases de ce tableau sont les registres (ou variables locales). Le nombre de registres est fixe et connu à la compilation du programme Java. On accède à un registre grâce à un index qui donne le décalage en *word* par rapport à la base du tableau. Les données stockées sur deux *words* occupent deux cases consécutives, on y accède par l'index de la première case. La taille du tableau est connue à la compilation.

4.1.2.3 La pile d'opérandes

A chaque invocation de méthode, la JVM alloue une *frame* qui contient la pile d'opérandes dont la taille maximale est connue à la compilation. Une JVM fournit des instructions pour charger des constantes, des valeurs depuis les registres ou encore la valeur de variables d'instance (attributs d'objet) sur la pile d'opérandes. Certaines instructions bytecode prennent des valeurs de cette pile, opèrent sur ces valeurs, et mettent le résultat de l'opération sur la pile. Ainsi une instruction d'addition dépile les deux éléments en sommet de pile, les additionne et empile le résultat. La pile d'opérandes sert également lors des appels de méthodes pour passer les arguments et recevoir le résultat.

4.1.2.4 Le tas

Le tas, contrairement aux registres et à la pile d'opérandes, est partagé entre toutes les invocations de méthodes. Chaque objet est créé dans le tas par appel à un constructeur via l'instruction Java `new`. La suppression d'un objet est gérée automatiquement, elle ne peut se faire que s'il n'existe plus de références vers lui. Il n'existe pas de directives Java pour détruire explicitement un objet.

4.1.3 Définition d'un état de la mémoire

On appelle état de la mémoire l'état des registres (les valeurs stockées), de la pile d'opérandes (la taille de la pile et les valeurs empilées) et du tas (la forme du tas et les valeurs des instances stockées).

Plusieurs formalisations d'un état de la mémoire d'une JVM sont disponibles, par exemple dans [FM99, Qia99, Siv04, GPR07]. Voici la formalisation d'un état mémoire *State*, utilisée dans [GPR07] [2].

$\mathcal{N}$: valeurs numériques $\mathcal{L}$: adresses de mémoire $\mathcal{X}$: noms de variables
$\mathcal{C}$: noms de classes $\mathcal{F}$: noms des attributs

$$
\begin{array}{rcll}
\mathcal{V} & = & \mathcal{N} \cup \mathcal{L} \cup null & \text{valeurs} \\
\text{LocalVar} & = & \mathcal{X} \rightarrow \mathcal{V} & \text{registres} \\
\text{OpStack} & = & \mathcal{V}^* & \text{pile d'opérandes} \\
\mathcal{O} & = & \mathcal{C} \times \mathcal{F} \rightarrow \mathcal{V} & \text{objets (instances de classe)} \\
\mathcal{A} & = & \mathcal{N} \times (\mathcal{N} \rightarrow \mathcal{V}) & \text{tableaux} \\
\text{Heap} & = & \mathcal{L} \rightarrow \mathcal{O} & \text{tas} \\
\text{State} & = & \text{LocalVar} \times \text{OpStack} \times \text{Heap} & \text{état mémoire}
\end{array}
$$

4.2 Notre modèle mémoire d'une JVM

Les entiers et les références sont modélisés par des variables sous contraintes dites VTPR (pour variables de type primitif ou référence), dont le domaine sera décrit en 4.2.3 et les contraintes les impliquant dans le chapitre suivant.

4.2.1 Modélisation des registres et de la pile d'opérandes

Les registres sont modélisés par une fonction qui à un index de registre associe la VTPR modélisant la valeur contenue dans le registre. Si une variable locale à une méthode occupe deux *words*, la JVM la stocke dans deux registres consécutifs. Soient i et $i+1$les index de ces registres, dans notre modèle seul l'index i apparaît, auquel on associe la VTPR modélisant la variable locale [3].
La pile d'opérandes est modélisée par une séquence de VTPR dont la première est le sommet de pile.

4.2.2 Modélisation du tas et des objets

4.2.2.1 Modélisation du tas

Un état du tas associe un objet à une adresse. Les adresses physiques d'une JVM sont modélisées par des entiers positifs. Les objets sont modélisés comme décrit dans la partie suivante (partie 4.2.2.2). La fonction modélisant le tas, qui associe à un entier un objet,

2. Nous omettons le point de programme que le formalisme utilisé dans cet article fait figurer dans l'état mémoire, ainsi que le niveau de sécurité pour les tableaux qui est propre aux besoins des auteurs. 3. Par souci de lisibilité, les EMC montrés dans la partie 3.3.1 ne correspondent pas exactement à la modélisation présentée ici : ainsi, dans le chapitre précédent, les registres sont indexés par le nom de la variable du programme qu'ils contiennent alors que dans notre modèle la notion de nom disparaît au profit d'un numéro de registre.

est une variable sous contraintes. En effet, tous les objets présents en mémoire ne sont pas toujours connus à un instant donné de la résolution du système de contraintes, le domaine de définition de la fonction est donc amené à s'enrichir au fur et à mesure que l'information se propage et que des choix sont faits lors de l'énumération. On appellera ces variables sous contraintes *variables fonctions*, abrégé en VF. Le domaine d'une VF notée H, modélisant le tas, est défini par deux éléments : un ensemble de couples et un statut.

1. L'ensemble de couples, noté E_H, contient les couples de la forme *(adresse,objet)* obligatoirement présents dans la fonction.
2. Le statut, noté $statut_H$, est un symbole qui vaut soit *non clos*, soit *clos*. Si le statut vaut *non clos*, aux couples de E_H sont susceptibles de s'ajouter d'autres couples. Si le statut vaut *clos*, aucun couple ne peut s'ajouter à ceux de E_H.

Les couples *(adresse,objet)* contenus dans E_H ont des adresses toutes distinctes. On note $domaine(H)$ l'ensemble de ces antécédents. Au cours du processus de résolution du système de contraintes, l'ensemble de couples E_H ne peut qu'être enrichi, réduisant ainsi le nombre de valeurs possibles pour la fonction puisque les fonctions ne possédant pas les couples ajoutés s'en retrouvent exclues.
Lorsque le statut associé à une VF vaut *clos*, le domaine de définition de la fonction est intégralement connu et E_H ne peut plus être enrichi. Au cours du processus de résolution du système de contraintes, le statut ne peut que passer de *non clos* à *clos*.
Une VF est instanciée quand le statut associé à la variable vaut *clos*. Les objets peuvent encore être inconnus ou partiellement connus (si les valeurs des variables d'instances ne sont pas toutes connues).

En raison de la complexité du modèle, des raccourcis de langage seront utilisés dans ce document afin de simplifier la compréhension. Ainsi *le domaine de la VF H* désigne le domaine de E_H, c'est-à-dire la partie du domaine déjà connu pour la fonction. *L'image d'une adresse par H* désigne en fait son image par E_H. *H est dite close* si son statut vaut *clos*.

4.2.2.2 Modélisation des objets

Nous restreignons ici les objets aux instances de classe [4]. Une instance de classe stockée dans le tas est modélisée par un couple.

1. Le premier élément est une variable qui représente le type de l'instance de classe : son domaine est un ensemble de classes qui sont les types encore possibles pour l'instance à une étape de la résolution. Une telle variable est appelée *variable de type* et est utile pour gérer l'héritage et le polymorphisme.
2. Le deuxième élément est généralement une fonction qui associe à chaque identifiant d'attribut non statique une VTPR qui représente la valeur de l'attribut pour l'instance. A un instant donné de la résolution du système de contraintes, si plusieurs types sont possibles pour l'instance, tous les identifiants des attributs possibles doivent être dans le domaine de la fonction. Cependant, dans le cas particulier où de la place est réservée en mémoire pour un objet mais que cet objet n'est pas encore initialisé par l'appel à un constructeur, la fonction vaut *inconnu* : tout accès ou mise à jour de la valeur d'un attribut est interdit.

Exemple : Soit une classe A qui définit les attributs a1 et a2, et une classe B qui hérite de A et définit l'attribut b. $([A, B], [a1 - Vtpr1, a2 - Vtpr2, b - Vtpr3])$ *représente soit un objet de classe A dont les variables d'instance pour a1 et a2 sont modélisées respectivement*

4. Notre modèle peut être étendu pour prendre en compte des tableaux et interfaces, cela est discuté en perspectives (sections 11.2.2 et 11.2.3)

par $Vtpr1$ et $Vtpr2$, soit un objet de classe B dont les variables d'instance a1, a2, et b sont modélisées respectivement par $Vtpr1$, $Vtpr2$ et $Vtpr3$.

Les raccourcis de langage suivants seront utilisés. Une instance $(Type, inconnu)$ sera dite inconnue (le type pouvant quant à lui être connu). *L'attribut associé à id dans une instance* $(Type, id_to_attr)$ désigne la VTPR image de l'identifiant d'attribut id par la fonction id_to_attr.

4.2.3 Modélisation des variables de type primitif ou référence

Les variables de type primitif ou référence sont modélisées par des variables sous contraintes, dites VTPR.

4.2.3.1 Modélisation des entiers et les booléens

Les entiers sont modélisés par des variables entières à domaines finis. Leur domaine par défaut dépend du type de l'entier, ainsi le domaine $-2^{31}..2^{31}-1$ est associé aux variables de type *int*.

4.2.3.2 Modélisation des références

Dans notre modèle, les adresses des objets dans le tas sont des entiers positifs. Les références sont modélisées par des variables dont le domaine est soit 1) un ensemble énuméré pouvant contenir des entiers positifs et le symbole *null*, soit 2) *all*. Dans le premier cas, les entiers de l'ensemble représentent les adresses du tas vers lesquelles la référence peut pointer. Dans le deuxième cas, la référence peut pointer vers n'importe quelle adresse du tas ou valoir *null*, sachant qu'à un instant donné de la résolution du système de contraintes toutes les adresses du tas ne sont pas forcément connues (le domaine de la VF le modélisant peut encore s'enrichir) ce qui empêche de définir l'ensemble en extension.
De plus, des contraintes du type $N_r \neq a$ sont ajoutées au système, indiquant que la variable référence N_r ne peut pas référencer l'adresse a. Deux cas sont à distinguer. Soit le domaine de N_r vaut *all*, en ce cas N_r peut valoir n'importe quelle adresse mais pas les adresses a figurant dans de telles contraintes [5]. Soit le domaine de N_r est énuméré et les adresses a figurant dans ce type de contraintes sont enlevées du domaine. De même, on peut exprimer qu'une référence doit être non nulle par la contrainte $N_r \neq null$.
Le domaine d'une référence ne peut que rétrécir au cours de la propagation, *all* étant le plus grand domaine.

A chaque variable référence on associe également une variable de type dont le domaine est un ensemble de symboles désignant les classes des instances vers lesquelles la référence peut pointer. Soit t le type de la référence r en Java (le type qui a été déclaré), initialement la variable de type associée à r aura pour domaine l'ensemble des sous-types de t (famille de types de t). On notera $Type_{Ref}$ la variable de type de la variable référence Ref.

4.2.4 Etat mémoire sous contraintes (EMC)

Dans notre modèle, un état de la mémoire est un triplet (f, s, H) où f est la fonction pour les registres, s la séquence de VTPR qui modélise la pile, et H une variable fonction

5. En pratique, quand une référence vaut *all*, un ensemble *ndom* retient les adresses vers lesquelles elle ne peut pas pointer ce qui permet d'accélérer les déductions.

(VF) représentant le tas.

On formalise de la manière suivante le modèle mémoire que nous proposons :

VarInt	:	ensemble des variables sous contraintes modélisant les variables entières du programme	
VarRef	:	ensemble des variables sous contraintes modélisant les variables références du programme	
VarType	:	ensemble des variables sous contraintes modélisant le type d'une instance ou d'une référence	
VarHeap	:	ensemble des variables sous contraintes modélisant le tas, dites variables fonctions (VF)	
$\mathcal{C}$	:	ensemble des classes	
IdAttr	:	ensemble des identifiants des attributs d'instance	
Loc	$\subset$	$\mathbb{N}$	ensemble des adresses de la mémoire
VTPR	$=$	VarInt $\cup$ VarRef	variables de type primitif ou référence
Inst	$=$	(VarType $\times$ Id_to_attr)	instance de classe
Id_to_attr	$=$	$\{inconnu\} \cup$ (SubIdAttr $\rightarrow$ VTPR) avec SubIdAttr $\subseteq$ IdAttr	valeur des attributs d'une instance
type	:	VarRef $\rightarrow$ VarType	types possibles de l'instance désignée par une référence
$\mathcal{F}$	$=$	$\mathbb{N} \rightarrow$ VTPR	registres
$\mathcal{S}$	$=$	VTPR*	pile d'opérandes
$\mathcal{EMC}$	$=$	$\mathcal{F} \times \mathcal{S} \times$ VarHeap	état mémoire sous contraintes

Pour $Ref \in$ VarRef, type(Ref) est noté $type_{Ref}$

Les domaines des variables sous contraintes sont donnés par la fonction *dom* :

$$\begin{array}{lll} \text{dom} & \in & \text{VarInt} \rightarrow \mathcal{P}(\mathbb{N}) \\ & \cup & \text{VarRef} \rightarrow \{all\} \cup \mathcal{P}(\text{Loc} \cup \{null\}) \\ & \cup & \text{VarType} \rightarrow \mathcal{P}(\mathcal{C}) \\ & \cup & \text{VarHeap} \rightarrow (\text{Loc} \rightarrow \text{Inst}) \times \text{Statut} \end{array}$$

avec Statut $= \{clos, non\ clos\}$: statut du tas

Pour $H \in$ VarHeap, dom(H) est noté (E_H,$statut_H$).

Un état mémoire *(f,s,H)* est complètement spécifié quand les conditions suivantes sont remplies :

- les VTPR des registres f sont instanciées,
- les VTPR de la pile s sont instanciées,
- H est instanciée c'est à dire que son statut est clos,
- toutes les instances de classe stockées dans le tas sont instanciées : leur type et la valeur des attributs pour ce type sont connus.

Soit $M = (f, s, H)$ avec :

$f = \{1 \rightarrow V1, 2 \rightarrow V2\}$

$s = V2.V3$

$E_H = \{\ (1, ([a], [t1 - V1, t2 - V2])),$

$$(2, ([b], [t3 - V3]))\},$$
$$statut_H = \text{clos}$$

où $V1$ vaut 3, $V2$ vaut -5 et $V3$ est une référence de type b qui vaut 2. M est un état de la mémoire entièrement connu. Le tas contient exactement deux objets, l'un de type a dont les attributs $t1$ et $t2$ valent respectivement 3 et -5, et l'autre de type b dont l'attribut $t3$ est une référence vers l'objet 2.

Ce chapitre décrit ce qu'est un état mémoire sous contraintes (EMC), qui représente un ensemble d'états mémoires concrets. Soit une séquence d'instructions, on associe un EMC au début de la séquence et après chaque instruction de cette séquence. Le chapitre suivant décrit les contraintes qui lient les EMC avant et après chacune des instructions de la séquence. Les contraintes traduisent la transition d'un état mémoire à l'autre par l'exécution d'une instruction.

Chapitre 5

Modélisation des instructions

Lors de son exécution, chaque instruction modifie l'état de la mémoire d'une manière définie par sa sémantique. L'effet de l'exécution de chaque instruction bytecode est exprimé sous la forme d'une relation entre deux états de la mémoire : l'état de la mémoire avant l'exécution de l'instruction et l'état de la mémoire après l'exécution de l'instruction. La sémantique modélisée est donc une sémantique opérationnelle petits pas, qui décrit les relations entre les états mémoires. Dans notre modèle les états mémoires sont des états mémoires sous contraintes (EMC), décrits au chapitre précédent. La relation entre les états de la mémoire avant et après une instruction se traduit notamment par des contraintes qui portent sur des éléments composant les EMC : les variables de type primitif ou référence (VTPR), les variables de type, les instances de classe, et les variables fonctions (VF) qui modélisent le tas.

Dans ce chapitre, après quelques remarques préliminaires sur la modélisation, les contraintes portant sur les VTPR, et se retrouvant dans la traduction de plusieurs instructions, sont décrites. Puis la traduction de certaines instructions sous la forme de relations est détaillée.

5.1 Remarques préliminaires sur la modélisation proposée

La sémantique modélisée actuellement est une sémantique sans erreurs. Nous traitons une partie des instructions bytecode, l'extension du modèle est discutée en perspectives. Des informations sont données pour comprendre la présentation de la modélisation des instructions.

5.1.1 Une sémantique sans erreurs

L'objectif de notre approche est la recherche de fautes fonctionnelles. La sémantique modélisée est par conséquent une sémantique sans erreurs. Pour modéliser une division, 0 est supprimé du domaine de la variable entière en dénominateur, les chemins du graphe de flot de contrôle dans lesquels sont lancées des exceptions ne sont pas considérés [1] et les opérations arithmétiques sont modélisées sans *overflows*.

5.1.2 Instructions modélisées

Les instructions actuellement modélisées sont :
– les instructions qui accèdent aux registres ou les modifient,

1. La prise en compte des exceptions est discutée en section 11.2.3.

– la plupart des instructions de manipulation de la pile,
– les opérations arithmétiques sur les entiers,
– les instructions de branchement conditionnel,
– les instructions qui manipulent les instances de classe : leur création dynamique (instruction `new`), l'accès à la valeur d'une variable d'instance et la modification d'une telle variable (instructions `getfield` et `putfield`),
– les appels aux méthodes statiques et virtuelles.

Notre modèle ne supporte pas actuellement :
– les nombres flottants,
– les exceptions,
– le multithreading.

L'extension du langage modélisé est discutée en section 11.2. Concernant la destruction des données, le ramasse miettes (*garbage collector*) désalloue les zones de la mémoire auxquelles le programme n'accède plus. Il n'est pas modélisé car son activation n'affecte pas le résultat de l'analyse sauf en présence de méthodes `finalize`.

5.1.3 Informations sur la présentation de la modélisation

Description des contraintes. L'interprétation des instructions sous forme de relations génère des contraintes, chacune de ces contraintes est décrite dans ce chapitre en suivant le format utilisé pour la contrainte $X > Y$ sur la figure 2.2 au chapitre 2.
Dans les algorithmes de filtrage, *change(V)* dénote un changement de domaine de la variable V (pour les variables fonctions il s'agit de l'enrichissement du domaine de la fonction ou du passage du statut de non clos à clos). *is_instanciated(V)* indique que la valeur de la variable est connue. $a \leftarrow b$ indique que a est mis à jour par b.

Optimisations. Des optimisations permettent de rendre plus efficace l'implémentation de la modélisation présentée dans ce chapitre. Les principales optimisations sont les suivantes. Premièrement, quand une référence vaut *all*, un ensemble *ndom* retient les adresses vers lesquelles elle ne peut pas pointer ce qui permet d'accélérer les déductions. Ensuite, les algorithmes de filtrage des contraintes modélisant les instructions portant sur les instances de classe (`new`, `getfield`, `putfield`) sont optimisés pour être plus rapides et pour réduire le nombre de réveils. Cela se fait notamment en mémorisant des informations d'un réveil à l'autre, ou encore en créant des contraintes secondaires, qui réalisent des parties ciblées du mécanisme de déduction, et qui peuvent éviter d'avoir à reconsidérer toutes les déductions lorsque le domaine de certaines des variables sont réduits.

5.2 Contraintes portant sur les VTPR

Cette section décrit les contraintes qui portent sur les VTPR. Ces contraintes sont introduites dans le système de contraintes lors de l'interprétation des instructions en contraintes.

5.2.1 Contraintes portant sur les entiers

Les librairies de contraintes sur les entiers étant nombreuses, nous n'avons pas redéfini les contraintes de ce type. La modélisation des instructions bytecode que nous traitons actuellement nécessite les contraintes $<,\leq,=,\geq,>,\neq$ ainsi que les opérateurs arithmétiques de l'addition $+$, de la soustraction $-$, de la multiplication $*$, de la division euclidienne $/$, et du modulo $\%$.

5.2.2 Contraintes portant sur les références

5.2.2.1 Egalité d'une référence et d'une adresse ou *null*

La contrainte $Ref = a$, avec a une adresse, force la référence Ref à valoir a. En pratique, cette contrainte ne reste pas dans le système de contraintes, elle est évaluée dès qu'elle est postée. La figure 5.1 détaille le traitement de la contrainte.

CONTRAINTE TRAITÉE : $Ref = a$

ALGORITHME DE FILTRAGE
si $a \in dom(Ref)$ **alors**
$\llcorner$ $dom(Ref) \longleftarrow \{a\}$

CONDITION D'ÉCHEC : $a \notin dom(Ref)$

CONDITION D'IMPLICATION : toujours vérifiée

CONDITION DE RÉVEIL : sans objet

FIG. 5.1 – Traitement de la contrainte d'égalité d'une référence et d'une adresse

a appartient au domaine de la référence si le domaine de la référence vaut *all* ou si il fait partie de l'ensemble énuméré d'adresses qui décrit le domaine. Si a n'appartient pas au domaine de Ref la contrainte échoue.
La contrainte $Ref = null$ force la référence à valoir *null* et est traitée de manière similaire.

5.2.2.2 Inégalité d'une référence et d'une adresse ou *null*

La contrainte $Ref \neq a$, avec a une adresse, force la référence Ref à être distincte de a. La figure 5.2 détaille le traitement de la contrainte. Elle est évaluée puis impliquée dès qu'elle est postée si le domaine de la référence ne vaut pas *all*. Sinon la contrainte est suspendue jusqu'à ce que le domaine de la référence devienne un ensemble énuméré.
La contrainte $Ref \neq null$ force Ref à être distinct de *null*.

CONTRAINTE TRAITÉE : $Ref \neq a$

ALGORITHME DE FILTRAGE
si $a \in dom(Ref) \wedge dom(Ref) \neq all$ **alors**
$\llcorner$ $dom(Ref) \leftarrow dom(Ref) - \{a\}$

CONDITION D'ÉCHEC : $dom(Ref) = \emptyset$

CONDITION D'IMPLICATION : $dom(Ref) \neq all$

CONDITION DE RÉVEIL : $change(Ref)$

FIG. 5.2 – Traitement de la contrainte d'inégalité d'une référence et d'une adresse

5.2.2.3 Egalité de références

La contrainte $Ref1 = Ref2$, décrite sur la figure 5.3, force deux références à être égales. Elle est réveillée dès que le domaine de l'une ou l'autre des références est modifié (donc

réduit) afin de maintenir égaux les domaines des deux références. Elle est impliquée quand les deux références sont connues. Quand cette contrainte est postée, une autre contrainte est aussi postée pour imposer l'égalité des variables de type associées aux références. Il s'agit de la contrainte $Type_{Ref1} =_{type} Type_{Ref2}$, décrite sur la figure 5.4.

CONTRAINTE TRAITÉE : $Ref1 = Ref2$

ALGORITHME DE FILTRAGE
si $dom(Ref1) = all$ **alors**
  $dom(Ref1) \leftarrow dom(Ref2)$
sinon
  si $dom(Ref2) = all$ **alors**
    $dom(Ref2) \leftarrow dom(Ref1)$
  sinon
    $dom(Ref1), dom(Ref2) \leftarrow dom(Ref1) \cap dom(Ref2)$

CONDITION D'ÉCHEC : $dom(Ref1) = \emptyset$

CONDITION D'IMPLICATION : $is_instanciated(Ref1)$

CONDITION DE RÉVEIL : $change(Ref1) \vee change(Ref2)$

FIG. 5.3 – Traitement de la contrainte d'égalité de références

CONTRAINTE TRAITÉE : $T1 =_{type} T2$

ALGORITHME DE FILTRAGE
$dom(T1), dom(T2) \leftarrow dom(T1) \cap dom(T2)$;

CONDITION D'ÉCHEC : $dom(T1) = \emptyset$

CONDITION D'IMPLICATION : $is_instanciated(T1)$

CONDITION DE RÉVEIL : $change(T1) \vee change(T2)$

FIG. 5.4 – Traitement de la contrainte d'égalité de types

5.2.2.4 Inégalité de références

La contrainte $Ref1 \neq Ref2$, décrite sur la figure 5.5, force les deux références à être distinctes. Ce type de contraintes est généré par interprétation d'instructions bytecode traduisant les conditions Java du type `p !=q` où p et q sont des références. Dès que l'une des références est connue, avec a sa valeur, cette contrainte génère une autre contrainte qui force la deuxième référence à être distincte de a. La contrainte $Ref1 \neq Ref2$ est alors impliquée. La contrainte est réveillée dès que le domaine de l'une des deux références évolue.

5.2.3 Notation de la contrainte d'égalité

Dans la suite, on notera $V1 =_{vtpr} V2$ la contrainte d'égalité de deux variables de type primitif. Selon le type de $V1$ et $V2$ la contrainte d'égalité posée sera soit l'égalité d'entiers décrite en 5.2.1, soit l'égalité de références décrite en 5.2.2.

CONTRAINTE TRAITÉE : $Ref1 \neq Ref2$

ALGORITHME DE FILTRAGE
soient :
$Ref1 \neq Ref2$ la contrainte à traiter
C le reste du système de contraintes
si $dom(Ref1) = \{a\}$ **alors**
  $C \longleftarrow C \cup \{Ref2 \neq a\}$
sinon
  si $dom(Ref2) = \{a\}$ **alors**
    $C \longleftarrow C \cup \{Ref1 \neq a\}$

CONDITION D'ÉCHEC : $dom(Ref1) = \emptyset \vee dom(Ref2) = \emptyset$

CONDITION D'IMPLICATION : $dom(Ref1) = \{a\} \vee dom(Ref2) = \{a\}$

CONDITION DE RÉVEIL : $change(Ref1) \vee change(Ref2)$

FIG. 5.5 – Traitement de la contrainte d'inégalité de références

5.3 Traduction des instructions en contraintes

5.3.1 Principe de l'interprétation en contraintes

Cette section décrit la modélisation des instructions bytecode sous la forme de relations entre des états de la mémoire sous contraintes (EMC). Un EMC M est de la forme (f, s, H) avec f les registres, s la pile et H la variable fonction (VF) modélisant le tas. L'effet de chaque instruction est exprimé comme une relation entre l'état de la mémoire avant l'instruction M_{in} et l'état de la mémoire après l'instruction M_{out}.
Dans cette section, nous considérons que la génération des mémoires se fait en sens inversé de l'exécution : lorsque l'instruction est interprétée, la mémoire après l'instruction a déjà été générée (mémoire créée initialement ou issue de l'interprétation d'une autre instruction), la mémoire avant instruction est générée à partir de celle-ci. C'est en effet dans ce sens que sont générées les mémoires pour le problème de la génération de données de test (cf. section 6.1).

Le tableau de la figure 5.6 décrit, pour chaque instruction, la mémoire générée et les éventuelles contraintes créées, des explications pour chaque instruction sont données au fil des paragraphes. $fresh(VTPR)$ dénote la création d'une variable fraîche, dont le domaine est le domaine par défaut pour son type. $fresh_ref(d, c)$ désigne la création d'une variable sous contrainte Ref modélisant une référence, avec d son domaine initial, i.e. $dom(Ref) = d$, et c l'ensemble des types possibles de l'instance qu'elle désigne, i.e. $dom(type_{Ref}) = c$.

5.3.2 Instructions d'une JVM et informations de type

La plupart des instructions d'une machine virtuelle Java encodent des informations de type pour les opérations qu'elles effectuent. Par exemple l'instruction `iload` charge le contenu d'un registre, qui doit être un *int*, sur la pile d'opérandes. L'instruction `lload` fait de même avec une valeur de type *long*. Dans notre modèle les types des variables stockées dans les registres sont initialement inconnus, ils sont inférés grâce à ces informations (ce mécanisme n'est pas décrit ici). De même certaines variables de la pile sont de type inconnu.

Instruction interprétée	Génération de mémoire	Contraintes créées
`aload_`$\langle n \rangle$, `iload_`$\langle n \rangle$	$M_{out} = (f, V.s, H)$, $M_{in} = (f, s, H)$	$V =_{vtpr} f(n)$
`astore_`$\langle n \rangle$ `istore_`$\langle n \rangle$	$M_{out} = (f, s, H)$, $M_{in} = (f\,[n \mapsto V'], V.s, H)$ avec V'=fresh(VTPR)	$V =_{vtpr} f(n)$
`dup`	$M_{out} = (f, V2.V1.s, H)$, $M_{in} = (f, V1.s, H)$	$V1 =_{vtpr} V2$
`pop`	$M_{out} = (f, s, H)$, $M_{in} = (f, V.s, H)$ avec V=fresh(VTPR)	
`iadd`, `ladd`	$M_{out} = (f, Int1.s1, H)$, $M_{in} = (f, Int3.Int2.s1, H)$ avec Int2,Int3=fresh(VTPR)	$Int1 = Int2 + Int3$
`irem`, `lrem`	$M_{out} = (f, Int1.s1, H)$, $M_{in} = (f, Int3.Int2.s1, H)$ avec Int2,Int3=fresh(VTPR)	$Int1 = Int2\ mod\ Int3$ avec *mod* l'opération modulo
`iinc n const`	$M_{out} = (f, s, H)$, $M_{in} = (f\,[n \mapsto Int1], s, H)$ avec Int1=fresh(VTPR)	$f(n) = Int1 + const$
`i : if_acmpeq n` arc(i,i+1) suivi arc(i,n) suivi	 $M_{out} = (f, s, H)$, $M_{in} = (f, Ref1.Ref2.s, H)$ avec Ref1,Ref2 = fresh(VTPR) $M_{out} = (f, s, H)$, $M_{in} = (f, Ref1.Ref2.s, H)$ avec Ref1,Ref2 = fresh(VTPR)	 $Ref1 \neq_{vtpr} Ref2$ $Ref1 =_{vtpr} Ref2$
`i : ifle n` arc(i,i+1) suivi arc(i,n) suivi	 $M_{out} = (f, s, H)$, $M_{in} = (f, Int1.s, H)$ avec Int1 = fresh(VTPR) $M_{out} = (f, s, H)$, $M_{in} = (f, Int1.s, H)$ avec Int1 = fresh(VTPR)	 $Int1 > 0$ $Int1 \leq 0$
`checkcast #c` `c` désigne le type t	$M_{out} = (f, Ref.s, H), M_{in} = (f, Ref.s, H)$	$Ref \neq null$, $checkcast(Ref, t)$
`new #n` `n` désigne le type c	$M_{out} = (f, Ref.s, H2), M_{in} = (f, s, H1)$	$new(H1, H2, Ref)$, $Ref = a$, $Type_{Ref} = c$ avec a l'adresse de l'instance à créer
`getfield #f` `f` désigne l'attribut id	$M_{out} = (f, Val.s, H), M_{in} = (f, Ref.s, H)$ $Ref = fresh(VTPR)$	$Ref \neq null$, $getfield(H, id, Ref, Val)$
`putfield #f` `f` désigne l'attribut id	$M_{out} = (f, s, H2), M_{in} = (f, Val.Ref.s, H1)$ $Ref = fresh(VTPR)$, $Val = fresh(VTPR)$	$Ref \neq null$, $putfield(H1, H2, id, Ref, Val)$
`invokevirtual #m` `m` désigne la méthode appelée	$M_{out} = (f, ReturnValue.s, H2)$, $M_{in} = (f, Prm_n ... Prm_1.This.s, H1)$ $Prm_1, Prm_n = fresh(VTPR)$, $This = fresh_ref(all, family(Class))$	$invokevirtual(Class, Method, This,$ $PrmTypes, [Prm_1 ... Prm_n],$ $ReturnValue, H1, H2)$, $This \neq null$

FIG. 5.6 – Génération des mémoires et création des contraintes selon l'instruction interprétée

5.3.3 Instructions d'accès aux registres et de modification des registres

Les instructions `aload_`$\langle n \rangle$, `iload`$\langle n \rangle$, `lload`$\langle n \rangle$ chargent le contenu du nième registre sur la pile, qui doit avoir pour type respectivement référence, *int* ou *long*. Dans notre modélisation, cela se traduit par une contrainte d'égalité entre la valeur stockée dans le nième registre avant l'instruction et la valeur en sommet de pile après l'instruction.

Les instructions `astore_`$\langle n \rangle$, `istore`$\langle n \rangle$ stockent la valeur en sommet de pile avant l'instruction dans le nième registre. Dans notre modélisation cela se traduit par une contrainte d'égalité entre la valeur en sommet de pile avant l'instruction et la valeur stockée dans le nième registre après l'instruction.

5.3.4 Instructions de manipulation de la pile

L'instruction `dup` duplique le sommet de pile. La modélisation de cette instruction, lorsque la génération des mémoires se fait en sens inversé par rapport à l'exécution, consiste à générer une mémoire en entrée qui est la même que la mémoire en sortie avec un élément en moins sur la pile d'opérandes, et à contraindre les deux éléments en sommet de pile dans la mémoire en sortie à être égaux.
L'instruction `pop` dépile l'élément en sommet de pile. La mémoire en entrée est la même que la mémoire en sortie hormis qu'elle comporte une variable supplémentaire en sommet de pile. Cette variable est une VTPR (variable de type primitif ou référence) nouvelle, dont le type est inconnu.
D'autres instructions de manipulation de la pile sont traitées de manière similaire (par exemple `dup_x1` et `swap`).

Enfin les instructions `iconst_m1`, `i_const_<i>`, `aconst_null`, `ldc` permettent d'empiler une constante entière ou la référence *null* sur la pile, leur modélisation est immédiate.

5.3.5 Instructions arithmétiques

Les instructions arithmétiques `iadd`, `isub`, `imul`, `idiv`, `irem` dépilent les deux éléments en sommet de pile qui doivent être des *int* et les remplacent respectivement par le résultat de l'addition, la soustraction, la multiplication, la division et le reste de la division euclidienne. `ladd`, `lsub`, `lmul`, `ldiv` et `lrem` procèdent de même avec les *long*. La modélisation de ces opérations pose une contrainte arithmétique qui lie l'élément en sommet de pile après l'instruction aux deux éléments en sommet de pile avant l'instruction.
Dans le tableau de la figure 5.6, les instructions `iadd`, `ladd`, `irem` et `lrem` sont détaillées.

L'instruction `iinc n const` permet d'incrémenter de la quantité *const* la valeur contenue dans le nième registre sans avoir à passer par la pile d'opérandes. Les contenus du nième registre avant et après instruction sont donc liés par une contrainte arithmétique d'addition.

5.3.6 Instructions de branchement conditionnel

Une instruction de branchement conditionnel fixe une condition sous laquelle l'exécution ne se poursuivra pas par l'instruction qui la suit mais par une autre instruction désignée par son numéro. Ainsi l'instruction `if_acmpeq n` compare les deux références qui sont au sommet de la pile d'opérandes : si elles sont égales, l'exécution se poursuit avec la nième instruction, sinon l'exécution continue avec l'instruction suivante. Les contraintes générées pour modéliser les instructions de branchement conditionnel dépendent de la branche choisie, deux cas sont à distinguer à chaque fois. Toutes les instructions de branchement conditionnel impliquant des entiers et des références sont modélisées. Étant nombreuses, seules deux d'entre elles seront détaillées : `if_acmpeq n` et `ifle n`.

Le rôle de l'instruction `if_acmpeq n` est donné au paragraphe précédent. La mémoire avant l'instruction est la même qu'après avec deux références supplémentaires sur la pile d'opérandes. Selon la branche suivie, les deux références sont contraintes à être égales ou différentes.

L'instruction `if_le n` compare la valeur de type *int* en sommet de pile avec 0 : si elle est inférieure ou égale à 0, l'exécution se poursuit avec l'instruction n, dans le cas contraire l'exécution continue avec l'instruction suivante. La mémoire avant l'instruction possède un entier supplémentaire sur la pile d'opérandes par rapport à la mémoire après instruction. Selon la branche suivie, l'entier est contraint soit à être strictement supérieur à 0, soit à être inférieur ou égal à 0.

5.3.7 Instructions de manipulation d'instances de classe

Les instructions modélisées qui mettent en jeu des instances de classe sont : `checkcast` qui contraint le type d'une référence, `new` pour créer une instance, `getfield` pour accéder à la valeur d'un attribut (ou variable d'instance) et `putfield` pour modifier la valeur d'un attribut. Des contraintes spéciales ont dû être créées, par exemple les instructions `new` et `putfield` nécessitent de maintenir des relations complexes entre les mémoires. En effet, à un instant de la résolution du système de contraintes le contenu du tas (les instances de classe stockées, leur type ou la valeur de leurs attributs) n'est pas toujours connu intégralement. Lorsque de l'information supplémentaire est acquise sur un état du tas, il faut la propager aux autres états du tas : cela nécessite de prendre en compte l'effet des instructions qui modifient le tas (`new` et `putfield`) pour connaître par exemple les éléments du tas qui restent inchangés.
Les contraintes modélisant les instructions `new`, `getfield` et `putfield` portent sur les variables fonctions modélisant le tas (VF) mais aussi sur des variables de type primitif ou référence (VTPR) contenues dans la pile d'opérandes. Ces contraintes ont trois rôles :

- la propagation des zones allouées dans la mémoire (si le domaine d'une VF contient une adresse a, que peut-on déduire sur la présence de a dans le domaine d'une autre VF qui lui est liée par une telle contrainte ?),
- le filtrage des domaines des variables de type primitif ou référence (VTPR), ainsi que des variables de type,
- la propagation de l'information sur la clôture du tas (si une VF est close, que peut-on déduire sur la clôture d'une autre VF qui lui est liée ?).

5.3.7.1 Instruction de vérification de type

L'instruction `checkcast #c` vérifie que la référence en sommet de pile n'est pas *null* et qu'elle désigne une instance de la famille de types du type t donné à l'index `c` du *constant pool*. Si c'est le cas, l'exécution de l'instruction ne modifie pas l'état de la mémoire, sinon elle lance une exception. La modélisation de cette instruction est immédiate dans le cadre d'une sémantique sans erreurs : la mémoire en entrée est la même que la mémoire en sortie, la référence en sommet de pile dans ces mémoires est contrainte à ne pas valoir *null* et son type est contraint à être inclus dans la famille de types de t par la contrainte, $checkcast(Ref, t)$, décrite sur la figure 5.7.

L'instruction de vérification de type `instanceof` empile un entier dont la valeur dépend du type de l'instance désignée par la référence en sommet de pile avant l'instruction. Elle est modélisée de manière similaire.

5.3.7.2 Instruction pour la création d'un objet : new

L'instruction `new #n` réserve de la place pour une nouvelle instance de classe et empile une référence vers cette instance sur la pile. Le type c de l'instance à créer est donné par le *constant pool* à l'index n.

CONTRAINTE TRAITÉE : $checkcast(Ref, t)$

ALGORITHME DE FILTRAGE
$dom(Type_{Ref}) \leftarrow dom(Type_{Ref}) \cap family(t)$

CONDITION D'ÉCHEC : $dom(Type_{Ref}) = \emptyset$

CONDITION D'IMPLICATION : toujours vérifiée

CONDITION DE RÉVEIL : sans objet

FIG. 5.7 – Traitement de la contrainte de vérification de type

Pour modéliser cette instruction, on utilise une nouvelle adresse *aref*, c'est à dire un entier positif qui n'est pas encore utilisé dans les domaines des VF modélisant le tas. L'élément en sommet de pile d'opérandes après l'instruction est une référence *Ref* qui doit être de type c et valoir *aref*. La pile d'opérandes avant l'instruction est la même qu'après l'instruction avec cet élément en moins. La contrainte $new(H1, H2, Ref)$, décrite sur la figure 5.8, lie les VF $H1$ et $H2$ modélisant le tas respectivement avant et après instruction. Son rôle est de garantir que les couples (adresse,instance) de la VF après instruction sont les mêmes que pour la VF avant instruction avec un couple supplémentaire d'adresse *aref*. L'instance associée à *aref* est de type c, et est inconnue car l'initialisation par appel à un constructeur n'a pas encore eu lieu. La contrainte $instances_egales$ figurant dans l'algorithme de filtrage de la contrainte *new* est décrite au paragraphe suivant.

L'algorithme de filtrage, donné sur la figure 5.8, procède de la manière suivante (les chiffres indiqués sur l'algorithme font référence à ces règles de déduction) :

(1) Soit $aref$ la valeur de Ref. Si $aref$ n'est pas dans le domaine de définition de $H2$, il doit y être ajouté. Son image par $H2$ est une instance de classe inconnue de même type que Ref.
(2) Tous les éléments du domaine de définition de $H1$ doivent être dans le domaine de définition de $H2$.
(3) Tous les éléments du domaine de définition de $H2$ qui sont distincts de l'adresse $aref$ référencée par Ref doivent être dans le domaine de définition de $H1$.
(4) Tout élément qui est à la fois dans le domaine de définition de $H1$ et dans celui de $H2$ a la même image (instance de classe) par $H1$ et $H2$.
(5) Si $H1$ ou $H2$ est clos, alors $H1$ et $H2$ sont tous les deux clos. En effet, si $H1$ ou $H2$ est clos, l'un des deux domaines de définition est connu. Ref étant instancié, l'élément à ajouter au domaine de $H1$ pour obtenir le domaine de $H2$ est également connu. Donc les deux domaines de définition sont connus.

La contrainte *new* est réveillée si l'ensemble de couples (adresse,instance) associé à $H1$ ou à $H2$ est enrichi ou si le statut de $H1$ ou de $H2$ est modifié.

Remarque. L'algorithme de filtrage proposé a une complexité plus élevée que l'algorithme réellement implanté. Ainsi, l'algorithme implanté ne considère lors d'un réveil que les adresses des domaines des VF qui n'ont pas été traitées lors des réveils précédents.

CONTRAINTE TRAITÉE : $new(H1, H2, Ref)$

ALGORITHME DE FILTRAGE (simplifié)
soient :
$new(H1, H2, Ref)$ la contrainte à traiter
C le reste du système de contraintes
$aref$ la valeur de Ref
début
  si $aref \notin domaine(E_{H2})$ **alors**
    $E_{H2} \longleftarrow E_{H2} \cup \{(aref, (Type_{Ref}, inconnu))\}$ ***(1)***
  pour chaque $(a, (Type2, id_to_attr2)) \in E_{H2}$ **faire**
    si $(a, (Type1, id_to_attr1)) \in E_{H1}$ **alors**
      $C \longleftarrow C \cup \{Type1 =_{Type} Type2\}$; ***(4)***
      $C \longleftarrow C \cup \{instances_egales(id_to_attr1, id_to_attr2, Type1)\}$ ***(4)***
    sinon
      si $a \neq aref$ **alors**
        $E_{H1} \longleftarrow E_{H1} \cup \{(a, (Type2, id_to_attr2))\}$ ***(3,4)***
  pour chaque $(a, (Type1, id_to_attr1)) \in E_{H1}$ **faire**
    si $a \notin domaine(E_{H2})$ **alors**
      $E_{H2} \longleftarrow E_{H2} \cup \{(a, (Type1, id_to_attr1))\}$ ***(2,4)***
  si $statut_{H1} = clos$ *ou* $statut_{H2} = clos$ **alors**
    $statut_{H1} \longleftarrow clos$;
    $statut_{H2} \longleftarrow clos$ ***(5)***

CONDITION D'ÉCHEC : ajout d'un élément dans le domaine d'une VF close

CONDITION D'IMPLICATION : $statut_{H1} = clos \wedge status_{H2} = clos$

CONDITION DE RÉVEIL : $change(H1) \vee change(H2)$

FIG. 5.8 – Traitement de la contrainte *new*

5.3.7.3 Contrainte d'égalité d'instances.

La contrainte *instances_egales* force deux instances à être égales. Tous les attributs des deux instances ne sont pas à égaler car les *attributs significatifs* pour une instance dépendent de son type. En effet, reprenons l'exemple du 4.2.2.2 avec une classe A qui définit les attributs a1 et a2, et une classe B qui hérite de A et définit l'attribut b. Si deux instances $([A, B], [a1 - V1, a2 - V2, b - V3])$ et $([A, B], [a1 - V'1, a2 - V'2, b - V'3])$, doivent être égales, elles ont le même type (cette obligation n'est pas gérée dans la contrainte *instance_egales*, c'est un prérequis pour ajouter la contrainte). Ces instances sont soit de type B, soit de type A. Si elles s'avèrent être de type B, il faut que $V1 = V'1$, $V2 = V'2$ et $V3 = V'3$. Par contre, si elles s'avèrent être de type A, la dernière égalité n'est plus nécessaire, b n'étant en ce cas pas un attribut significatif. Les contraintes à poser pour égaler les attributs dépendent donc du type des instances.

La contrainte $instances_egales(id_to_attr1, id_to_attr2, Type)$, où $Type$ donne le type des instances et id_to_attr1 et id_to_attr2 sont les fonctions qui décrivent les attributs pour chacune des deux instances, fonctionne de la manière suivante.

(1) Pour les attributs forcément significatifs pour les instances, c'est-à-dire requis pour tous les types encore possibles pour ces instances, des contraintes d'égalité entre les valeurs des attributs sont posées. C'est le cas, dans l'exemple précédent, pour les attributs a1 et a2, les contraintes $V1 =_{vtpr} V'1$ et $V2 =_{vtpr} V'2$ sont posées.

(2) Pour chacun des autres attributs, si les domaines des variables les modélisant dans les deux instances sont disjoints alors les variables ne peuvent pas être égalées. Un tel attribut doit donc être non significatif pour ces instances, et les types qui requièrent cet attribut sont supprimés du domaine des variables de type des instances. Ainsi, dans l'exemple précédent, si $V3$ et $V'3$ ne peuvent pas être égalés alors B doit être enlevé de l'ensemble des types possibles pour les instances.

Cette contrainte est réveillée notamment quand le domaine d'une variable de type associée aux instances est modifié. En effet, si un type est supprimé du domaine, des attributs non significatifs pour ce type peuvent être significatifs pour tous les autres types restant possibles, et des égalités pour ces attributs peuvent alors être posées.

5.3.7.4 Instruction pour l'accès à la valeur d'un attribut : getfield

L'instruction `getfield #f` accède à la valeur d'un attribut de l'instance désignée par la référence en sommet de la pile d'opérandes. L'identifiant de l'attribut accédé est donné par le *constant pool* à l'index `f`. La référence en sommet de pile y est remplacée par la valeur de l'attribut. Pour modéliser cette instruction, l'élément en sommet de pile après l'instruction est la valeur à laquelle on accède, la pile avant instruction a pour sommet une référence dont les types possibles dépendent de l'attribut concerné, le tas et les registres ne sont pas modifiés par l'instruction. La contrainte $getfield(H, idAttr, Ref, Val)$, décrite sur la figure 5.10, garantit que Val est l'attribut d'identifiant *idAttr* de l'instance du tas H désignée par Ref. Les types possibles de Ref se déduisent de l'attribut auquel on accède.

L'algorithme de filtrage de la contrainte $getfield$, donné sur la figure 5.10, procède de la manière suivante.

(1) Si Ref est instancié et vaut $aref$, $aref$ doit être ajouté au domaine de H s'il n'y est pas. Si l'instance associée à $aref$ par H a pour variable de type $Type$ alors $Type$ et la variable de type de Ref doivent être égales. De plus, soit Val' la valeur de l'attribut d'identifiant *idAttr* dans l'instance, Val' et Val doivent également être égales.

CONTRAINTE TRAITÉE : $instances_egales(id_to_attr1, id_to_attr2, Type)$

ALGORITHME DE FILTRAGE (simplifié)
soient :
$instances_egales(id_to_attr1, id_to_attr2, Type)$ la contrainte à traiter
C le reste du système de contraintes
pour chaque $attr \in \bigcup_{t \in dom(Type)} attributs(t)$ **faire**
  soient : $V1 = id_to_attr1(attr)$ et $V2 = id_to_attr2(attr)$
  si $V1$ *et* $V2$ *ne peuvent pas être égalées* **alors**
    retirer du domaine de $Type$ tous les types qui exigent la définition de $attr$ ***(2)***
  sinon
    si $attr \in \bigcap_{t \in dom(Type)} attributs(t)$ **alors**
      $C \longleftarrow C \cup \{V1 =_{ptrv} V2\}$ ***(1)***

CONDITION D'ÉCHEC : $dom(Type) = \emptyset$

FIG. 5.9 – Traitement de la contrainte d'égalité d'instances

(2) Pour toute adresse a du domaine de H, si l'instance associée est inconnue, si l'instance a un type incompatible avec le type de Ref, ou si l'attribut $idAttr$ de l'instance ne peut pas être égalé avec Val, alors Ref est différent de a.
(3) Si le statut de H est clos, le domaine de Ref doit être inclus dans celui de H.

5.3.7.5 Instruction pour modifier la valeur d'un attribut : putfield

L'instruction `putfield #f` modifie la valeur de l'attribut dont l'identifiant est donné par le *constant pool* à l'index `f`. Les deux éléments au sommet de la pile d'opérandes avant l'exécution de l'instruction sont la référence vers l'instance modifiée et la nouvelle valeur que l'on associe à l'attribut. Ils sont dépilés lors de l'exécution de l'instruction. Pour modéliser cette instruction, une référence est créée, dont les types possibles dépendent de l'attribut concerné, ainsi qu'une VTPR qui modélise la nouvelle valeur de l'attribut. Ces variables sont placées en sommet de la pile d'opérandes avant l'instruction. Les registres ne sont pas modifiés par l'instruction, par contre le tas l'est. On ne connaît pas initialement la valeur de la référence, on ne sait donc pas quelle instance du tas est modifiée. Les VF $H1$ et $H2$ modélisent le tas respectivement avant et après l'instruction, la contrainte $putfield(H1, H2, idAttr, Ref, Val)$, décrite sur la figure 5.11, garantit que Val modélise l'attribut $idAttr$ de l'objet référencé par Ref dans $H2$. Il garantit également que toutes les adresses présentes dans l'une des VF ($H1$ ou $H2$) le sont également dans l'autre et que les instances associées à une adresse sont identiques, hormis pour l'attribut $idAttr$ de l'instance référencée par Ref.

Deux nouvelles contraintes figurent dans l'algorithme de filtrage. La contrainte *instances_egales_except_index(ita1,ita2,Type1,idAttr)* contraint deux instances de type $Type1$ à être égales hormis pour l'attribut d'identifiant $idAttr$, son fonctionnement est très proche de celui de la contrainte *instances_egales* et n'est pas détaillé. La contrainte $attributs_egaux(Val1, Val2, idAttr, Type)$ contraint les variables $Val1$ et $Val2$ à être égales si l'attribut $idAttr$ (auquel elles sont associées dans des instances) est significatif pour les types donnés par $Type$, son fonctionnement n'est pas non plus détaillé.

CONTRAINTE TRAITÉE : $getfield(H, idAttr, Ref, Val)$

ALGORITHME DE FILTRAGE (simplifié)
soient :
$getfield(H, idAttr, Ref, Val)$ la contrainte à traiter
C le reste du système de contraintes
début
- **si** $dom(Ref) = \{aref\}$ **alors**
  - **si** $aref \notin domaine(E_H)$ **alors**
    - $E_H \longleftarrow E_H \cup \{(aref, (Type_{Ref}, id_to_attr))\}$ ***(1)***
    - avec id_to_attr une fonction donnant la valeur des attributs telle que tous les types du domaine de $Type_{Ref}$ soient pris en compte. A chaque attribut est associée une variable créée par défaut en fonction du type de l'attribut hormis pour $idAttr$ auquel est associé Val.
  - **sinon**
    - **soient** $(aref, (Type, id_to_attr)) \in E_H$ et $Val' = id_to_attr(idAttr)$
    - $C \longleftarrow C \cup \{Type =_{type} TypeRef, Val' =_{vtpr} Val\}$ ***(1)***
- **sinon**
  - **pour chaque** $(a, (Type, id_to_attr)) \in E_H$ *tel que* $a \in dom(Ref)$ **faire**
    - **si** $id_to_attr = inconnu$ **alors**
      - $C \longleftarrow C \cup \{Ref \neq a\}$ ***(2)***
    - **sinon**
      - **si** $dom(Type) \cap dom(TypeRef) = \emptyset$ **alors**
        - $C \longleftarrow C \cup \{Ref \neq a\}$ ***(2)***
      - **sinon**
        - Soit $Val' = id_to_attr(idAttr)$
        - **si** $dom(Val) \cap dom(Val') = \emptyset$ **alors**
          - $C \longleftarrow C \cup \{Ref \neq a\}$ ***(2)***
  - **si** $statut(H) = clos$ **alors**
    - $dom(Ref) \longleftarrow dom(Ref) \cap domaine(E_H)$
  - ***(3)***

CONDITION D'ÉCHEC : le domaine d'une variable devient vide ou un élément est ajouté dans le domaine d'une VF close

CONDITION D'IMPLICATION : $is_instanciated(Ref)$

FIG. 5.10 – Traitement de la contrainte $getfield$

L'algorithme de filtrage de la contrainte $putfield$, donné sur la figure 5.11, procède de la manière suivante.

(1) Toutes les adresses du domaine de $H1$ doivent être dans le domaine de $H2$ et inversement. Les instances associées à une adresse dans $H1$ et $H2$ doivent avoir le même type.

(2) Toute adresse du domaine de définition de $H1$ et de $H2$ qui n'est pas dans le domaine de Ref a la même image (types possibles et valeur des attributs pour les types possibles) par $H1$ et par $H2$.

(3) Pour toute instance de $H1$ ou $H2$ associée à une adresse a, si l'instance est inconnue ou si ses types possibles sont incompatibles avec ceux de Ref, alors Ref ne vaut pas a.

(4). Si Ref vaut a, soit $Val2$ l'attribut $idAttr$ de l'instance associée à a dans $H2$, $Val2$ et Val sont égaux. Les instances associées à a dans $H1$ et $H2$ sont identiques hormis pour cet attribut.

(5) Pour toute instance de $H2$, associée à une adresse a, dont les types possibles sont compatibles avec ceux de Ref, si l'attribut d'identifiant $idAttr$ pour cette instance, et la variable Val qui donne la nouvelle valeur de l'attribut, ne peuvent pas être égaux, alors Ref ne vaut pas a.

(6) Pour chaque adresse a du domaine de $H1$ ou $H2$, dans le cas où les types possibles des instances associées à a sont compatibles avec ceux de Ref, si les domaines des deux variables modélisant l'attribut $idAttr$ de l'instance stockée à l'adresse a respectivement avant et après l'instruction sont incompatibles alors l'attribut est modifié par l'instruction `putfield`, d'où Ref vaut a.

(7) Pour chaque adresse a du domaine de $H1$ ou $H2$, dans le cas où les types possibles des instances associées sont compatibles avec ceux de Ref, soient $Val1$ et $Val2$ les variables modélisant l'attribut $idAttr$ de l'instance image de a respectivement dans $H1$ et $H2$, $Val2$ a un domaine inclus dans l'union des domaines de Val et $Val1$. En effet, soit Ref désigne l'adresse a et $Val2$ est égal à Val, soit Ref ne désigne pas a et $Val2$ est égal à $Val1$ si $idAtt$ est un attribut significatif pour les instances images de a.

(8) Si le statut de $H1$ ou de $H2$ est clos, alors ce statut se propage à l'autre tas. En effet, $H1$ et $H2$ ont le même domaine de définition (ils contiennent les mêmes adresses).

Voici une illustration des capacités de déduction de la contrainte $putfield$ sur un exemple simple. Soient les tas $H1$ et $H2$ suivants :

$$\begin{aligned} E_{H1} = \{ & (1, ([a], [t1 - V1, t2 - V2])), \\ & (2, ([a], [t1 - V3, t2 - V4])), \\ & (3, ([b], [t3 - V5]))\}, \\ statut_{H1} = & \text{ non clos} \end{aligned}$$

$$\begin{aligned} E_{H2} = \{ & (1, ((a], [t1 - V6, t2 - V7])), \\ & (2, ([a], [t1 - V8, t2 - V9])), \\ & (3, ([b], [t3 - V10]))\} \\ statut_{H2} = & \text{ clos} \end{aligned}$$

Ces deux tas contiennent chacun deux objets de la classe a et un objet de la classe b. La classe a comporte deux attributs entiers, $t1$ et $t2$. La classe b n'hérite pas de a et

CONTRAINTE TRAITÉE : $putfield(H1, H2, idAttr, Ref, Val)$
ALGORITHME DE FILTRAGE (simplifié)
soient :
$putfield(H1, H2, idAttr, Ref, Val)$ la contrainte à traiter
C le reste du système de contraintes
début
- **pour chaque** $a \in dom(H1) \cup dom(H2)$ **faire**
  - **si** $a \notin dom(Ref)$ **alors**
    - **si** $\exists((a, (Type1, ita1))), (a, (Type2, ita2)) \in E_{H1} \times E_{H2}$ **alors**
      - $C \longleftarrow C \cup \{Type1 =_{type} Type2,\ instances_egales(ita1, ita2, Type1)\}$ ***(1,2)***
    - **sinon**
      - $\exists(a, (Typei, itai)) \in E_{Hi}$ avec $i \in \{1, 2\}$
      - $E_{Hj} \longleftarrow E_{Hj} \cup \{(a, (Typei, itai))\}$ avec $j \in \{1, 2\}, j \neq i$ ***(1,2)***
  - **sinon**
    - **si** $\exists((a, (Type1, ita1))), (a, (Type2, ita2)) \in E_{H1} \times E_{H2}$ **alors**
      - $C \longleftarrow C \cup \{Type1 =_{type} Type2\}$, ***(1)***
      - **si** $dom(Type_{Ref}) \cap dom(Type1) = \emptyset$ *ou* $ita1 = inconnu$ *ou* $ita2 = inconnu$ **alors**
        - $C \longleftarrow C \cup \{Ref \neq a,\ instances_egales(ita1, ita2, Type1)\}$ ***(2,3)***
      - **sinon**
        - $C \longleftarrow C \cup \{instances_egales_except_id(ita1, ita2, Type1, idAttr)\}$ ***(4)***
    - **sinon**
      - $\exists(a, (Typei, itai)) \in E_{Hi}$ avec $i \in \{1, 2\}$
      - **si** $dom(Type_{Ref}) \cap dom(Typei) = \emptyset$ *ou* $itai = inconnu$ **alors**
        - $C \longleftarrow C \cup \{Ref \neq a\}$;
        - $E_{Hj} \longleftarrow E_{Hj} \cup \{(a, (Typei, itai))\}$ avec $j \in \{1, 2\}, j \neq i$ ***(1,2,3)***
      - **sinon**
        - $itaj$ est une fonction identique à $itai$ mis à part qu'à $idAttr$ est associée une variable fraîche
        - $E_{Hj} \longleftarrow E_{Hj} \cup \{(a, (Typei, itaj))\}$ avec $j \in \{1, 2\}, j \neq i$; ***(1)***
- **pour chaque** $a \in dom(Ref)$ **faire**
  - $\exists((a, (Type, ita1))), (a, (Type, ita2)) \in E_{H1} \times E_{H2}$
  - Soient $Val1 = ita1(idAttr)$ et $Val2 = ita2(idAttr)$
  - **si** $dom(Val2) \cap dom(Val) = \emptyset$ **alors**
    - $C \longleftarrow C \cup \{Ref \neq a, attributs_egaux(Val1, Val2, idAttr, Type)\}$ ***(5,2)***
  - **sinon**
    - **si** $dom(Val2) \cap dom(Val1) = \emptyset$ **alors**
      - $C \longleftarrow C \cup \{Ref = a\}$ ***(6)***
    - **sinon**
      - $dom(Val2) \longleftarrow dom(Val1) \cap dom(Val)$ ***(7)***
- **si** $is_instanciated(Ref)$ **alors**
  - $\exists(a, (Type2, ita2)) \in E_{H2}$
  - Soit $Val2 = ita2(idAttr)$
  - $C \longleftarrow C \cup \{Val2 =_{ptrv} Val, Type2 = Type_{Ref}\}$ ***(4)***
- **si** $statut_{H1} = clos$ *ou* $statut_{H2} = clos$ **alors**
  - $status_{H1} \longleftarrow clos$; $status_{H2} \longleftarrow clos$ ***(8)***

CONDITION D'ÉCHEC : le domaine d'une variable devient vide ou un élément est ajouté dans le domaine d'une VF close

FIG. 5.11 – Traitement de la contrainte $putfield$

possède un attribut entier, $t3$. Soit la contrainte $putfield(H1, H2, t1, A, Val)$. Supposons que $dom(V1) = 0..10$, $dom(V3) = 1..3$, $dom(V8) = 15..2^{31} - 1$, que les domaines des autres variables entières ne sont pas contraints, que $dom(A) = \{all\}$, et que $dom(Val) = 10..40$. En utilisant la contrainte $putfield(H1, H2, t1, A, Val)$, différentes déductions peuvent être faites :

- Comme la contrainte opère sur l'attribut $t1$, elle traduit la modification de la valeur d'un attribut d'un objet de type a. On peut déduire que $dom(A) = all - \{3\}$ car l'objet 3 est de type b. De plus, la contrainte ne modifie pas les variables associées aux attributs $t2$ et $t3$ dans le tas, par conséquent les égalités suivantes peuvent être ajoutées : $V2 = V7, V4 = V9$ et $V5 = V10$.
- On considérant les variables $V3$ et $V8$, on constate que $dom(V3) \cap dom(V8) = \emptyset$, et on peut déduire que l'attribut $t1$ de l'objet 2 est modifié par l'instruction `putfield`. Donc A désigne nécessairement l'objet 2, d'où $A = 2$. Par conséquent l'objet stocké en 1 n'est pas modifié, d'où $V1 = V6$. Val modélise la valeur stockée par l'instruction `putfield`, donc $V8 = Val$, d'où $dom(V8) = dom(Val) = 15..40$.
- Finalement, le statut clos de $H2$ se propage à $H1$ car $putfield$ n'ajoute pas de nouvel objet dans le tas. Donc toutes les adresses des deux tas sont connues.

5.3.8 Appels de méthodes et de constructeurs

5.3.8.1 Appels de méthodes

On distingue les appels, par l'instruction `invokestatic #m`, de méthodes statiques, des appels, par l'instruction `invokevirtual #m`, de méthodes non statiques, dites virtuelles. Le traitement des appels de méthodes statiques n'est pas détaillé car il est proche de celui des appels de méthodes virtuelles sans présenter de problèmes de polymorphisme. Lorsqu'une méthode virtuelle est appelée, le code appelé dépend du type effectif de l'instance de classe qui l'appelle, comme cela est détaillé en 3.4.2.

La contrainte $invokevirtual(Class, Method, This, PrmTypes, [Prm1...Prmn], ReturnValue, H1, H2)$ décrite sur la figure 5.12, traduit l'appel de méthode virtuelle. La classe $Class$, le nom de la méthode $Method$ et le type des paramètres $PrmTypes$ permettent de déterminer quelles méthodes peuvent être appelées en fonction du type de la référence $This$ qui se trouve sur la pile avant l'appel de méthode et qui désigne l'instance de classe appelante. Les valeurs des paramètres, $[Prm1...Prmn]$, se trouvent également sur la pile avant l'exécution de l'instruction `invokevirtual`. Comme $This$, ils sont dépilés après l'exécution, et remplacés en sommet de pile par la valeur de retour $ReturnValue$ de la méthode. Le tas est partagé entre la méthode appelée et la méthode appelante, $H1$ et $H2$ le représentent respectivement avant et après l'exécution de l'instruction. Les registres ne sont pas modifiés par l'appel de méthode.

La contrainte $invokevirtual$ procède de la manière suivante. Si le type de l'objet appelant est inconnu, c'est-à-dire si la valeur de la variable de type associée à $This$ n'est pas connue, alors la contrainte est suspendue en attendant d'avoir davantage d'information sur le type de l'instance de classe appelante. Davantage d'information sur ce type pourra être acquise grâce à la phase d'énumération sur la mémoire en entrée, ou encore grâce à la traduction en contraintes d'instructions telles que `checkcast` qui contraignent le type des références.
Si le type c de l'instance de classe appelante est connu, la méthode appelée est également connue, et l'un des chemins de cette méthode est interprété sous forme de contraintes de

la manière suivante. Des registres sont créés, le nombre de registres à créer est fixé par la méthode appelée. Le registre 0 contient la VTPR $This$ qui est la référence vers l'instance de classe qui appelle la méthode. La mémoire en sortie de la méthode appelante, notée $MCallee_{out}$, comporte ces registres, une pile qui contient la valeur de retour de la méthode et le tas représenté par $H2$. Un chemin de la méthode appelée est parcouru en sens inverse de celui de l'exécution, il est interprété sous forme de contraintes en partant de la mémoire de sortie $MCallee_{out}$. Signalons que si plusieurs chemins existent dans la méthode appelée, un point de choix est créé, auquel il sera possible de revenir en cas d'échec de la résolution du système de contraintes. L'interprétation génère la mémoire $MCallee_{in}$ en entrée de méthode, qui comporte le tas $H1$, une pile vide, et des registres qui contiennent notamment les paramètres d'appel de la méthode, ceux-ci vont être contraints à être égaux aux valeurs données par la liste $[Prm1...Prmn]$.

CONTRAINTE TRAITÉE :
$invokevirtual(Class, Method, This, PrmTypes, [Prm_1...Prm_n], ReturnValue, H1, H2)$

ALGORITHME DE FILTRAGE
Soient :
$invokevirtual(Class, Method, This, PrmTypes, [Prm_1...Prm_n], ReturnValue, H1, H2)$ la contrainte à traiter
C le reste du système de contraintes **début**
  si $domaine(Type_{This}) = \{c\}$ **alors**
    $initialisation_registres(f_{out}, This, method(c, Method, PrmTypes))$;
    $MCallee_{out} = (f_{out}, ReturnValue, H2)$;
    $interpret(method(c, Method, PrmTypes), MCallee_{in}, MCallee_{out})$;
    soit $MCallee_{in} = (f_{in}, \epsilon, H1)$
    pour chaque Prm_i *avec* $1 \leq i \leq n$ **faire**
      $C \longleftarrow C \cup \{Prm_i =_{vtpr} f_{in}(i);\}$

CONDITION D'ÉCHEC : échec lors de l'interprétation de la méthode

CONDITION D'IMPLICATION : $is_instanciated(Type_{This})$

CONDITION DE RÉVEIL : $change(Type_{This})$

FIG. 5.12 – Traitement de la contrainte d'appel de méthode virtuelle

5.3.8.2 Appels de constructeurs

Un appel de constructeur se fait en bytecode par l'instruction `invokespecial #m` où `m` est l'index du *constant pool* auquel se trouve la description du constructeur à appeler. Un appel de constructeur se fait soit par une instance de classe non initialisée, dite inconnue dans notre modèle, soit par une instance de classe en cours d'initialisation quand un constructeur appelle un autre constructeur. Le traitement des appels aux constructeurs est très proche du traitement des appels de méthodes et n'est pas détaillé ici.

Un constructeur est cependant particulier, c'est le constructeur de la classe *Object* qui est toujours appelé en début d'initialisation d'une instance de classe et qui fait passer une instance de l'état non initialisée à l'état en cours d'initialisation. Dans notre modèle, l'appel à ce constructeur a pour effet d'initialiser la fonction id_to_attr qui permet d'accéder à la valeur des variables d'instance. Le domaine initialement donné aux VTPR qui modélisent

les variables d'instance est le domaine par défaut pour leur type (référence, *int*, *long*, ...).

Chapitre 6

Génération de données de test

6.1 Un parcours de graphe en arrière et incrémental

6.1.1 Le choix d'un parcours en arrière

L'objectif de notre approche est de générer automatiquement des données de test pour compléter la couverture des instructions d'une méthode sous test obtenue avec d'autres outils. Pour atteindre une instruction particulière de la méthode, les autres outils de génération de données de test utilisent tous, à notre connaissance, un parcours du graphe de flot de contrôle débutant par le point d'entrée de la méthode. En effet, comme détaillé en section 1.2.6, beaucoup d'outils utilisent une approche concolique, approche qui fait appel à des exécutions du programme : ces outils raisonnent donc principalement dans le sens de l'exécution. Or, notre modèle étant relationnel, il nous permet de raisonner aussi bien dans le sens de l'exécution que dans le sens inverse. Notre méthode de génération automatique de données de test se base sur un parcours du graphe partant de l'instruction à atteindre vers le point d'entrée de la méthode. On peut montrer, et cela sera fait dans la partie 6.1.2, que certains objectifs à atteindre le sont plus rapidement grâce à un raisonnement arrière car la détection de chemins non exécutables est plus rapide, d'autres objectifs à l'inverse favorisent le raisonnement avant. Comme notre objectif est de proposer un outil complémentaire aux autres, le raisonnement arrière est plus intéressant pour pouvoir couvrir des instructions atteignables moins facilement par raisonnement avant et donc non couvertes par les autres outils.

6.1.2 Un parcours incrémental

Pour générer une donnée d'entrée qui sensibilise l'instruction cible, les chemins qui relient le point d'entrée de la méthode à cette instruction sont énumérés afin d'en trouver un qui soit exécutable et pour lequel est générée la donnée de test. La construction d'un chemin lors de l'énumération se fait en partant de l'instruction cible et en remontant vers le point d'entrée de la méthode. Au cours de cette construction du chemin, les instructions parcourues sont modélisées par des contraintes : à un moment donné du parcours, si le système de contraintes est insatisfiable, le sous-chemin dont sont issues les contraintes du système n'est pas exécutable, ce qui exclut de l'énumération tous les chemins ayant pour suffixe ce sous-chemin. Le processus de propagation de contraintes est donc lancé régulièrement au cours de la construction d'un chemin en vue de détecter une insatisfiabilité au plus tôt.
Initialement, l'instruction courante est l'instruction cible, son interprétation génère un état mémoire sous contraintes (EMC) qui représente l'état de la mémoire avant l'instruction,

et éventuellement des contraintes. Si l'instruction courante a un seul prédécesseur dans le graphe de flot de contrôle, celui-ci devient à son tour instruction courante. Sinon, un point de choix est créé, et l'un des prédécesseurs est choisi pour devenir instruction courante. Le parcours du graphe se fait en profondeur. Si un (sous-)chemin est prouvé non exécutable, alors le parcours reprend à partir du dernier point de choix laissé en suspend.
En guise d'illustration, considérons l'extrait d'un graphe de flot de contrôle d'une méthode donné à la figure 6.1.

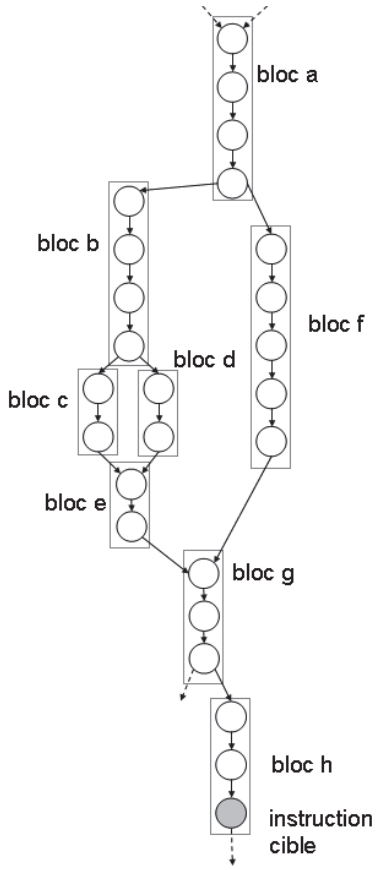


Fig. 6.1 – Regroupement par blocs des instructions du graphe de flot de contrôle pour le parcours incrémental des chemins

Les instructions sont regroupées par blocs, à la fin de l'interprétation des instructions d'un bloc, la propagation de contraintes est lancée. L'instruction cible se trouve sur cet exemple dans le bloc h. La propagation est lancée avant tout point de choix et avant l'interprétation d'une instruction conditionnelle. Supposons que la satisfiabilité de la condition dans le bloc g, nécessaire pour passer au bloc h dans une exécution, est contradictoire avec des instructions du bloc e. Par exemple le bloc e contient l'assignation `i=1` et l'instruction conditionnelle formée par les instructions du bloc g est `if(i>2)`. Les contraintes correspondant au bloc h puis au bloc g sont générées. Le bloc g ayant deux prédécesseurs, les blocs e et f, un point de choix est créé. Faisons l'hypothèse que le bloc e est choisi, les contraintes correspondantes sont générées. La propagation de contraintes est alors lancée, comme à chaque fois qu'un bloc a fini d'être interprété : le système de contraintes est incohérent. Tous les chemins entre le point d'entrée du programme et l'instruction cible ayant pour suffixe le chemin formé par les blocs e,g,h (soit deux chemins) sont donc écartés des

candidats potentiels pour la génération d'une donnée de test, sans qu'il soit nécessaire de les parcourir un par un.
Cet exemple est favorable à notre méthode de parcours du graphe en sens inverse de celui de l'exécution car le sous-chemin non exécutable est "proche" de l'instruction cible. D'autres cas, ceux où les sous-chemins non exécutables sont "proches" du point d'entrée de la méthode, sont plus favorables aux stratégies de parcours "en avant". Comme expliqué précédemment, l'objectif étant de compléter la couverture des autres outils de test qui parcourent le graphe "en avant", parcourir le graphe "en arrière" est plus intéressant.

6.1.3 Stratégies pour le parcours de graphe

Pour trouver un chemin exécutable menant à une instruction, les chemins du graphe sont énumérés et le chemin courant est construit progressivement en partant de l'instruction à atteindre vers le point d'entrée. Cette section décrit certaines stratégies qui sont mises en place pour trouver au plus vite un chemin exécutable. En particulier, l'ordre de parcours des prédécesseurs lors d'un point de choix, et le choix des chemins explorés selon le nombre de passages par certains arcs et leur longueur, peuvent accélérer la génération de donnée de test.

6.1.3.1 Ordre de choix des prédécesseurs

Afin de parvenir plus rapidement à trouver un chemin exécutable de l'entrée à l'instruction cible, si une instruction a plusieurs prédécesseurs alors le prédécesseur permettant d'atteindre le point d'entrée avec le chemin intraprocédural le plus court en termes de nombre d'instructions est choisi prioritairement. Le chemin le plus court est déterminé grâce à l'algorithme de Dijkstra. Les limites de cette stratégie sont les suivantes. Premièrement, les longueurs des chemins suivis dans les méthodes appelées ne sont pas prises en compte car cela peut être coûteux en présence de méthodes polymorphes (cela deviendrait coûteux de considérer tous les types possibles de l'instance de classe appelante) et nécessite d'avoir un graphe de flot de contrôle interprocédural. La deuxième est que le chemin le plus court n'est pas forcément exécutable, ce qui n'est pas détectable sans avoir entamé son parcours pour poser les contraintes.

6.1.3.2 Limitation du nombre de passages par le même arc

Considérons le graphe de la figure 6.2. L'instruction 12 est l'instruction cible. Nous faisons l'hypothèse que le chemin 0-1-2-3-4-5-6 est exécutable et que le chemin 0-1-2-3-7-8 ne l'est pas.

Pour atteindre l'instruction cible, les instructions 12, 10 et 9 sont tour à tour interprétées. L'instruction 9 ayant deux prédécesseurs, la priorité est donnée au prédécesseur 8 pour lequel il existe un chemin plus court en termes de nombre d'instructions vers le point d'entrée. Les instructions 8, 7, 3 et 2 sont interprétées l'une après l'autre. 2 ayant plusieurs prédécesseurs (point de choix), l'instruction 1 est choisie et les instructions 1 et 0 sont interprétées. Le chemin interprété est donc 0-1-2-3-7-8-9-10-12 qui est non exécutable. Le processus revient sur le dernier point de choix, c'est-à-dire sur le choix du prédécesseur de 2. Cette fois, l'instruction 10 est choisie puis interprétée, les instructions 9, 8, 7, 3 et 2 le sont à leur tour. Là encore le premier prédécesseur choisi est 1, là encore le chemin interprété 0-1-2-3-7-8-9-10-2-3-7-8-9-10-12 est non exécutable. Le processus peut se poursuivre ainsi infiniment sans revenir sur le choix du prédécesseur de 9. Pour éviter ce type de comportement lors du parcours de graphe en présence de boucles, un paramètre est ajouté,

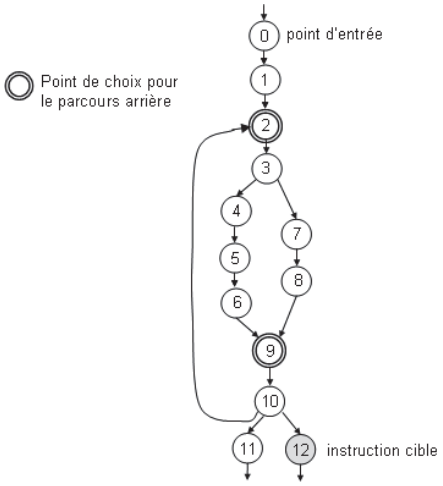


Fig. 6.2 – Exemple de graphe de flot de contrôle avec boucle

dit *borne des arcs* : il limite le nombre de fois qu'un arc issu d'un point de choix peut être emprunté. Par exemple 2-1 et 2-10 sont des arcs issus d'un point de choix.
Ce paramètre peut être assimilé au paramètre qui borne le nombre de tours de boucle dans le cadre de stratégies orientées chemin pour la génération de données de test pour des langages sources (comme le fait PathCrawler). Cependant, au niveau du bytecode, la structure du code n'est pas visible syntaxiquement comme pour les codes source (structure de contrôle *while* par exemple). Pour borner le nombre de passages dans une boucle au niveau du bytecode, il faudrait partir du graphe de flot de contrôle pour reconstruire les boucles. Utiliser un paramètre pour borner le nombre de passages par un arc issu d'un point de choix ne nécessite pas une telle étude car le nombre de passages par un tel arc est mémorisé, qu'il fasse partie d'une boucle ou pas. Notons qu'il n'y a pas d'équivalence entre borner k fois le dépliage des boucles et borner k fois le passage par les arcs issus de points de choix car un arc peut potentiellement être emprunté par plusieurs chemins ne faisant pas forcément partie de la même boucle.

6.1.3.3 Limitation de la longueur des chemins explorés

Lors du parcours des chemins, certains d'entre eux peuvent s'avérer très longs, alors que d'autres chemins plus courts n'ont pas encore été explorés. Les chemins les plus longs, outre le temps nécessaire pour interpréter les instructions qui le composent, génèrent généralement plus de contraintes et la résolution peut s'en retrouver plus longue. C'est pourquoi, une borne sur la longueur des chemins parcourus, dite *borne de longueur*, est passée en paramètre. Elle porte sur les chemins interprocéduraux, c'est-à-dire qu'elle prend en compte la longueur des chemins dans les fonctions appelées. En pratique, la borne sur la longueur des chemins n'est pas testée dès qu'une instruction supplémentaire est interprétée, mais seulement en certains points : après avoir interprété un chemin d'une fonction appelée et avant de choisir un prédécesseur à un point de choix.

6.2 Enumération de la mémoire d'entrée

Si un chemin est suivi entre le point d'entrée de la méthode et l'instruction cible et qu'aucune inconsistance n'est détectée, la phase d'énumération peut débuter. Voici la mé-

thode que nous proposons, et qui est implantée dans l'outil. L'énumération débute avec les paramètres de la méthode. Pour les références, la première valeur testée est *null*. Les valeurs testées ensuite sont les adresses des objets déjà présents dans le tas ayant un type compatible. Enfin, en cas d'échec pour toutes les hypothèses précédentes, un nouvel objet de type compatible est créé. Quand toutes les valeurs des paramètres sont fixées, les valeurs des attributs des instances de classe doivent être fixées à leur tour. Les attributs des objets atteignables depuis les paramètres avec un seul déréférencement sont énumérés les premiers. Ces objets sont dits être au niveau 1 pour l'énumération (cf. figure 6.3). Puis l'énumération se poursuit avec les objets situés au niveau 2 et ainsi de suite.

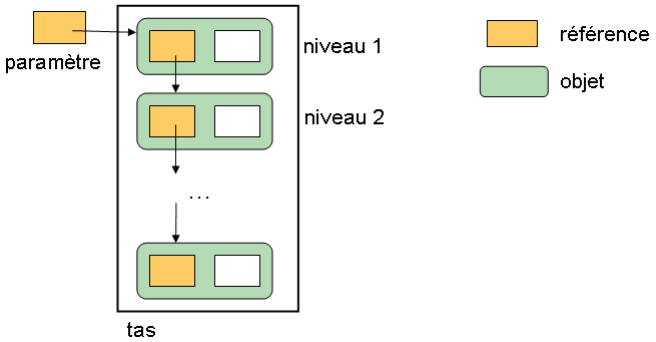


FIG. 6.3 – Illustration de la notion de niveau pour les objets lors de l'énumération

Un paramètre pour la génération de données de test est donné par l'utilisateur, nous l'appelons *borne mémoire*. Si aucune solution n'est trouvée sans que le niveau d'énumération pour les objets ne dépasse cette borne alors le processus d'énumération cesse et on revient en arrière pour suivre un autre chemin. En effet, le processus d'énumération peut être coûteux en temps et il peut être préférable de tester un autre chemin plutôt que de faire une énumération coûteuse. Si tous les chemins ont été testés et que la borne mémoire a été atteinte au moins une fois, celle-ci est augmentée et le processus de génération est relancé. Un *timeout* borne le temps de génération, la procédure est semi-correcte.

6.3 Correction et complétude de la méthode de génération

L'objectif de notre outil, JAUT, est de générer une donnée de test pour couvrir une instruction cible de la méthode sous test si celle-ci est exécutable, et d'indiquer qu'elle n'est pas atteignable sinon. Il s'agit d'une procédure semi-correcte, le problème de sensibilisation d'une instruction donnée est un problème indécidable dans le cas général : si aucune réponse n'est fournie par l'outil dans le délai qui lui est imparti, aucune conclusion ne peut être tirée sur la possibilité d'atteindre ou non l'instruction.

La correction d'une méthode de génération automatique de données de test n'est pas cruciale. En effet, en exécutant le programme avec la donnée de test générée, on peut vérifier que l'objectif de test est bien atteint. Cependant, pour éviter de générer des données de test inutiles qui n'atteignent pas l'objectif, il est préférable que la méthode soit correcte. Dans ce but, nous nous sommes basés sur la spécification SUN d'une JVM pour notre modélisation.

La complétude est par contre cruciale lorsque l'on souhaite atteindre 100% de couverture des instructions exécutables et pouvoir détecter les éléments cibles non exécutables.

L'utilisation de bornes en paramètres de génération porte atteinte à la complétude, cependant des indicateurs signalent si l'une des bornes (sur la mémoire, sur la longueur des chemins, sur le nombre de passages par les arcs issus des points de choix) est atteinte. Si la procédure de résolution de contraintes utilisée est complète, alors notre méthode est complète dans les bornes car nous ne faisons pas d'approximation des conditions de chemins comme peuvent le faire d'autres méthodes de génération de tests. Ainsi, si aucune donnée d'entrée n'est trouvée et qu'aucune des bornes n'est atteinte, alors on pourra affirmer que l'instruction est non atteignable (ou du moins pour un chemin sans exceptions comme cela est discuté en perspectives).

Chapitre 7

Comparaison de notre approche avec l'état de l'Art

7.1 Comparaison avec les approches exploitant le model-checking et l'exécution symbolique

Notre méthode et les approches de génération automatique de tests basées sur l'exécution symbolique avec model-checking, qui ont été présentées dans la section 1.2.5, sont orientées chemins. Elles utilisent un parcours incrémental et essaient de détecter au fur et à mesure les chemins non exécutables.
Dans la méthode basée sur l'initialisation paresseuse et utilisée dans l'outil JPF-SE [KPV03, VPK04, PMB+08], les références ne sont pas symboliques. Dans l'approche exploitant la *lazier initialization* et implantée dans l'outil Kiasan/Kunit [DRH07], elles le sont tant qu'il n'y pas de test d'égalité ou d'accès à un attribut. Dans notre méthode, les variables du programme, y compris les références, sont représentées par des variables sous contraintes, un système à contraintes traduit les conditions à remplir pour suivre le chemin. Outre l'utilisation de la programmation par contraintes et non pas de procédures de décision, notre approche se distingue parce que les choix non déterministes sur la valeur des références sont retardés au plus tard, quand toutes les contraintes d'un chemin sont posées et jamais au cours du parcours. En outre, leurs travaux n'abordent pas le problème du polymorphisme.

7.2 Comparaison avec les méthodes basées sur l'exécution symbolique dynamique

Parcours du graphe de flot de contrôle. Les méthodes exploitant l'exécution symbolique dynamique, présentées dans la section 1.2.6, parcourent le graphe de flot de contrôle "à la volée", ce qui permet de garantir que le préfixe de chemin sensibilisé par la donnée de test précédemment trouvée, et réutilisé pour explorer le chemin courant, est exécutable. Le test de la faisabilité des conditions de chemins est de la sorte incrémentale. Dans notre approche nous cherchons également à détecter au fur et à mesure de leur parcours la non exécutabilité des chemins, notre approche est donc également incrémentale.
Les approches basées sur l'exécution symbolique dynamique mènent une exploration des chemins dans le sens de l'exécution. Notre approche, au contraire, raisonne dans le sens inverse à celui de l'exécution : en partant d'un objectif (une instruction à couvrir) elle essaie de trouver un chemin menant vers le point d'entrée du programme en explorant progressivement le graphe de flot de contrôle. Selon les programmes traités, le parcours

des chemins dans le sens de l'exécution, ou leur parcours dans le sens inverse, peut être plus favorable pour trouver rapidement une donnée de test (cf. partie 6.1.2), en cela notre approche et celles basées sur l'exécution symbolique dynamique sont complémentaires.
Le méthode exploitée dans l'outil OSMOSE [BH08] est confrontée au problème de reconstituer le graphe de flot de contrôle. En effet, au niveau des exécutables, les sauts dans le code peuvent être dynamiques, c'est-à-dire dépendre de la valeur d'une variable. Ce problème ne se pose pas au niveau du bytecode car les adresses des instructions cibles sont désignées par des constantes.

Utilisation de la programmation par contraintes. Comme dans les approches utilisées dans les outils PathCrawler [WMMR05, BDHTH+09], OSMOSE [BH08] et EXE [CGP+06, CGP+08], la résolution des conditions de chemins dans notre approche se fait à l'aide d'un solveur de contraintes, les expressions non linéaires ne posent donc pas problème comme c'est le cas pour d'autres approches qui utilisent une concrétisation qui approxime les comportements du programme. La programmation par contraintes permet également d'exprimer des relations complexes, utiles pour raisonner sur les alias potentiels entre les références. Les méthodes utilisées dans OSMOSE et EXE utilisent la théorie des vecteurs de bits qui est bien adaptée pour modéliser les opérations de bas niveau. Notre approche ne traite pas pour l'instant les opérations bit à bit.

Traitement des pointeurs et des références. Les autres méthodes ont chacunes leurs limitations dans le traitement des pointeurs et références. Comme discuté précédemment, des méthodes basées sur l'exécution concolique [GKS05, God07, SMA05, SA06] utilisent parfois la concrétisation de ce type de variable pour résoudre les conditions de chemin : une telle approximation peut réduire le nombre de comportements pris en compte, et empêcher la couverture d'objectifs exécutables. OSMOSE utilise également les exécutions concrètes pour trouver les relations d'alias entre variables puisqu'il ne dispose que d'un code binaire, sans information de ce type. Dans notre approche et dans celle de Williams, Marre, Mouy, et Roger (implantée dans PathCrawler), aucune approximation n'est utilisée, ne portant ainsi pas atteinte à la complétude. Cependant, la méthode de Williams, Marre, Mouy, et Roger demande à l'utilisateur d'expliciter les relations d'alias entre les pointeurs en entrée sous la forme de pré-conditions ; notre méthode, elle, infère les relations d'alias nécessaires grâce aux conditions et aux instructions du chemin. Il est difficile de cerner les capacités de Pex [TdH08, TS06], les exemples montrent qu'il ne permet pas de résoudre certaines conditions de chemins impliquant des relations complexes entre les références.
En résumé, notre approche se base sur un modèle mémoire bien adapté pour prendre en compte des relations entre les références (inégalités, alias) dans le but d'obtenir une couverture complète des instructions exécutables du programme. Mais la complexité de notre modèle a aussi un coût en termes de temps d'exécution.

Des approches complémentaires. En conclusion de cette comparaison, notre approche s'avère très utile pour compléter la couverture obtenue avec les méthodes basées sur l'exécution symbolique dynamique. Les outils basés sur l'exécution symbolique dynamique sont souvent rapides, ils sont donc plus adaptés que notre outil pour les chemins ne présentant pas des conditions de chemin faisant appel à des alias de pointeurs ou de références trop complexes. Notre approche permet ensuite d'accroître la couverture obtenue avec ces outils, grâce d'une part à son modèle mémoire prenant en compte statiquement des relations d'alias, et d'autre part au sens de parcours des chemins différent de celui des autres outils, qui peut être plus favorable pour atteindre certaines instructions.

7.3 Comparaison avec l'approche orientée but

Notre approche exploite les travaux menés pour pour la génération de tests orientée but [GBR98, GBW06], présentés dans la section 1.2.7, en particulier la représentation par des variables sous contraintes des pointeurs (références en Java).

Des modèles mémoires différents. La représentation du tas utilisée dans l'outil InKa [GBR98, GBW06], formalisée dans [CBG07], est extrêmement complexe ce qui ralentit la génération de données de test. Euclide [Got09] possède un modèle mémoire plus simple, mais en contrepartie il ne gère pas l'allocation dynamique. Le modèle du tas que nous proposons est suffisamment complexe pour traiter les alias de références et l'allocation dynamique, mais il est moins complexe que celui d'InKa et s'avère efficace comme le montrent les expériences. L'approche que nous proposons pour le bytecode Java modélise la pile d'opérandes qui n'existe pas en C ou C++. De plus, InKa ne gère pas l'héritage et les appels de méthodes polymorphes alors que notre modèle le prend en compte grâce à des variables sous contraintes modélisant le type des objets et des références.

Approche orientée but et approche orientée chemins. Les approches utilisées dans les outils InKa et Euclide sont orientées but alors que notre approche est orientée chemins. Cependant, le modèle mémoire que nous proposons a été pensé avec l'objectif d'étendre notre méthode à l'approche orientée but. Les opérateurs utilisés dans ces approches orientées but pour modéliser les structures de contrôle, tels que le `ite`, ne sont pas directement applicables pour le bytecode Java car le bytecode est déstructuré et cela nécessite de retrouver les structures de contrôle. Enfin, InKa et Euclide explorent le programme dans le sens de l'exécution alors que notre approche explore les chemins dans le sens inverse. Cependant, InKa, Euclide et notre outil pourraient tous les trois implanter les deux sens de parcours.

Les tableaux récapitulatifs utilisés en section 1.2 sont repris ici et complétés avec notre approche sur les figures 7.1 et 7.2 .

	critère de couverture visé	type d'approche	spécificités des algorithmes	complétude
Méthode de Kurshid, Pasareanu et Visser	toutes les instructions	statique, orientée chemins	utilisation du model-checking et de l'exécution symbolique, initialisation paresseuse	sous-approximation des comportements (non déterminisme)
Méthode de Deng, Robby et Hatcliff	toutes les instructions	statique, orientée chemins	utilisation du model-checking et de l'exécution symbolique, *lazier initialization*	sous-approximation des comportements (non déterminisme)
Méthode de Williams, Marre, Mouy, et Roger	tous les k-chemins	dynamique, orientée chemins	exécution symbolique dynamique	pas de sous-approximation, sous les hypothèses fournies par l'utilisateur sur les relations d'alias en entrée de programme
Méthode de Godefroid, Klarlund, et Sen	tous les chemins, toutes les branches	dynamique, orientée chemins	exécution concolique, concrétisation pour les expressions non linéaires et pour certaines expressions portant sur des pointeurs	sous-approximation des comportements liée à la concrétisation
Méthode de Sen, Marinov et Agha	tous les chemins, toutes les branches, failles de sécurité	dynamique, orientée chemins	exécution concolique, concrétisation pour les expressions non linéaires et pour certaines expressions portant sur des pointeurs, traitement de certaines relations d'alias portant sur les entrées du programmes	sous-approximation des comportements liée à la concrétisation
Méthode de Tillmann, de Halleux et Schulte	toutes les branches, failles de sécurité	dynamique,orientée chemins	tests paramétrés, traitement de certaines relations d'alias portant sur les entrées du programmes, concrétisation	sous-approximation des comportements liée à la concrétisation
Méthode de Bardin et Herrmann	toutes les branches, toutes les instructions	dynamique, orientée chemins	exécution concolique pour le test d'exécutables, approximation de certaines relations d'alias due à la perte d'informations de haut niveau	sous-approximation des comportements (perte d'informations de haut niveau)
Méthode de Cadar et al.	tous les chemins, erreurs d'exécution	dynamique, orientée chemins	exécution concolique, un processus par chemin, concrétisation pour les doubles déréférencements	sous-approximation des comportements liée à la concrétisation
Méthode de Gotlieb, Botella et Rueher	toutes les instructions	statique, orientée but	pas de choix de chemin a priori, gestion des relations d'alias entre références	pas de sous-approximation
Méthode de Gotlieb	toutes les instructions	statique, orientée but	pas de choix de chemin a priori, gestion des relations d'alias entre références, utilisation de relaxations linéaires	pas de sous-approximation
Méthode présentée dans cette thèse	**toutes les instructions**	**statique, orientée chemins**	**gestion des relations d'alias entre références, modèle mémoire lié au bytecode (pile), analyse de type (héritage, polymorphisme)**	**pas de sous-approximation**

FIG. 7.1 – Tableau comparatif des différentes approches de génération de test basées sur le code, avec notre approche (1/2)

	sens de parcours du graphe	détection de sous-chemins non exécutables	techniques de résolution utilisées	outil	langage cible	références bibliographiques
Méthode de Kurshid, Pasareanu et Visser	en avant	oui	procédures de décision	JPF-SE	Java	[KPV03, VPK04, PMB+08]
Méthode de Deng, Robby et Hatcliff	en avant	oui	procédures de décision et programmation par contraintes	Kiasan/KUnit	Java	[DRH07]
Méthode de Williams, Marre, Mouy, et Roger	en avant	oui	programmation par contraintes	PathCrawler	C	[WMMR05, BDHTH+09]
Méthode de Godefroid, Klarlund, et Sen	en avant	oui	programmation linéaire entière	DART, SMART	C	[GKS05, God07]
Méthode de Sen, Marinov et Agha	en avant	oui	programmation par contraintes	Cute JCute	C Java	[SMA05, SA06]
Méthode de Tillmann, de Halleux et Schulte	en avant	oui	solveur SMT (Z3)	Pex	bytecode .NET	[TdH08, TS06]
Méthode de Bardin et Herrmann	en avant	oui	contraintes sur des vecteurs de bits	Osmose	code exécutable	[BH08]
Méthode de Cadar et al.	en avant	non	solveur de contraintes, procédures de décision	EXE	C	[CGP+06, CGP+08]
Méthode de Gotlieb, Botella et Rueher	en avant	oui	programmation par contraintes	InKa	C, C++	[GBR98, GBW06]
Méthode de Gotlieb	en avant	oui	programmation par contraintes	Euclide	C	[Got09]
Méthode présentée dans cette thèse	**en arrière**	**oui**	**programmation par contraintes**	**JAUT**	**bytecode Java**	[CG10]

FIG. 7.2 – Tableau comparatif des différentes approches de génération de test basées sur le code, avec notre approche (2/2)

Troisième partie

Validation expérimentale

Chapitre 8

Description de l'outil JAUT

8.1 Principe de fonctionnement

La méthode proposée est implantée dans un outil, JAUT (Java Automatic Unit Testing), développé en SICStus prolog (version 4.0.2). L'architecture de l'outil est donnée sur la figure 8.1.

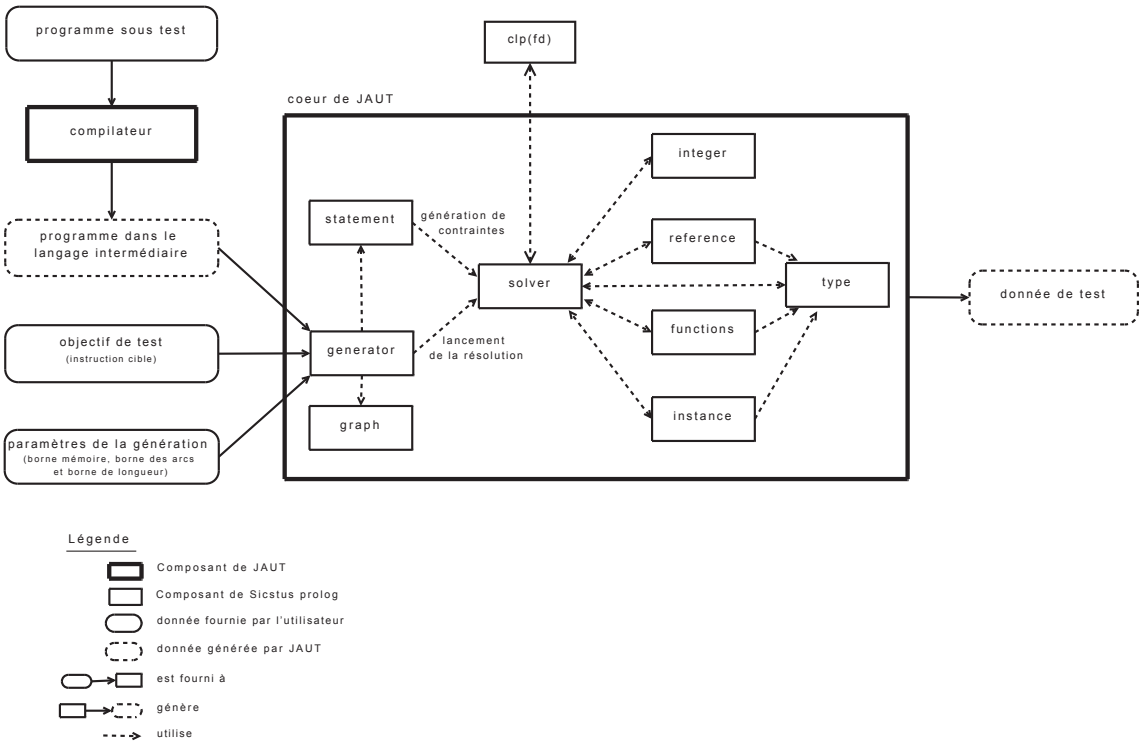


Fig. 8.1 – Architecture de l'outil JAUT

JAUT prend en entrée les fichiers des programmes Java compilés et les traduit en un langage qui lui est propre. L'utilisateur indique à l'outil une liste d'instructions à couvrir ainsi que les paramètres de la génération (borne mémoire, bornes sur les arcs et la longueur des chemins). L'outil fournit des données de test pour couvrir chacune des instructions, sous réserve que l'instruction soit exécutable, atteignable en respectant les conditions fixées par les paramètres, et que le temps de génération n'excède pas le timeout. Les données de test

créées sont données sous une forme textuelle. JAUT propose des états mémoires en entrée permettant de couvrir les instructions objectifs, mais, dans l'état actuel de l'outil, il ne garantit pas que ces états puissent être atteints par une séquence d'appels aux constructeurs et méthodes. Par conséquent, il ne donne pas le code permettant de lancer l'exécution du programme avec les données de test. Une telle extension est discutée en perspectives.

8.2 Architecture détaillée

Les fichiers bytecode traduits dans le langage intermédiaire, ainsi que les paramètres fournis par l'utilisateur, servent au module *generator* pour parcourir le graphe de flot de contrôle. Le module *generator* exploite le module *graph* pour guider le parcours, et le *backtrack* prolog pour revenir sur les points de choix. Au fur et à mesure du parcours, le module *generator* fait appel au module *statement* pour générer les contraintes, qui lui même utilise les modules *integer*, *reference*, *type*, *functions* et *instance* selon le type de contraintes à générer (cette relation entre ces modules et *statement* est omise sur le schéma pour ne pas le surcharger). Le module *generator* se charge également de lancer la résolution de contraintes en faisant appel au module *solver*. Le module *solver* gère des appels au solveur clp(fd) de SICStus prolog pour les contraintes portant sur les entiers. Pour les autres contraintes, le module se charge de leur mise en file, de leur réveil et de l'appel aux algorithmes de filtrage codés dans les modules *integer*, *reference*, *functions*, *instance* et *type*. Pour les contraintes gérées par le solveur, celui-ci utilise deux files de contraintes par souci d'efficacité : une file prioritaire pour les contraintes plus élémentaires (égalité de références, de types, ...) qui sont rapides à traiter et dont les déductions peuvent être très utiles aux autres contraintes, et une autre file pour les contraintes plus complexes (portant sur les fonctions, les instances, ...).

Chapitre 9

Validation expérimentale

Toutes les expérimentations furent menées sur une machine standard équipée d'un processeur Intel Pentium de fréquence 2.16 GHz, tournant sous Windows XP, avec 2.0 Go de RAM.
Les performances de JAUT sont comparées à celles de deux générateurs automatiques de données de test : Microsoft Pex et JCute. Pour générer les données de test, JCute analyse le code source Java. Microsoft Pex quant à lui analyse le code en CIL (Common Intermediate Language), le langage intermédiaire vers lequel sont compilés les programmes écrits dans les langages du framework .NET. Les programmes présentés ici en Java peuvent sans difficulté être écrits dans le langage C# du *framework* .NET. L'outil Parasoft JTest a également été testé ; les résultats, peu concluants car les taux de couverture atteints sont très faibles, ne sont pas reportés ici. Le tableau de la figure 9.1 récapitule les résultats obtenus.

Methods	#Bc	#To	JCute (cov)	JCute (sec)	JCuteR (cov)	JCuteR (sec)	Pex (cov)	Pex (sec)	JAUT (cov)	JAUT (sec)	Comp. JAUT (sec)
trityp	89	24	83%	2,20	88%	30,88	**100%**	**4,17**	100%	3132,00	-
foo	15	3	**100%**	**10,90**	75%	18,80	66%	4,51	**100%**	**0,17**	0, 15
josephus	51	3	100%	1,06	100%	1,06	100%	8,07	100%	0,36	-
josephus/m	60	5	70%	192,00	70%	**	**100%**	**8,41**	60%	**	-
Node class											
insertBefore	36	4	100%	3,02	100%	35,80	100%	4,03	100%	0,20	-
DoublyLinkedList											
pop	13	2	100%	2,16	100%	23,00	100%	0,59	100%	0,13	-
add	42	4	100%	9,19	88%	45,00	100%	6,09	100%	0,14	-
remove	31	4	100%	2,94	100%	1,89	100%	0,69	100%	0,16	-
RedBlackTree											
rotateLeft	48	6	17%	1,12	17%	25,10	83%	0,65	**100%**	**0,25**	0, 05
deleteEntry	124	18	56%	2,12	92%	5,64	83%	56,00	**100%**	**1,86**	0, 25
fixAfterDeletion	175	14	14%	1,11	7%	**	79%	92,00	**100%**	**12,60**	0, 88
fixAfterInsertion	127	11	18%	1,30	18%	28,30	55%	120,00	**100%**	**75,73**	60, 49

FIG. 9.1 – Temps et couverture avec JCute, Pex et JAUT (** : *timeout* de 3600 sec)

#Bc est le nombre d'instructions bytecode de la méthode, #To est le nombre d'objectifs de test. Les objectifs de test sont les instructions cibles des instructions de saut conditionnel, le choix de ces objectifs est discuté en 9.2.2. Pour les trois outils, JAUT, Pex et JCute, sont reportés les pourcentages de couverture obtenus, que nous avons calculés nous-mêmes sur une base commune. En effet, PEX rapporte une couverture dynamique des branches, qui ne correspond pas nécessairement au taux de couverture des branches (comme expliqué en 1.2.6.4). De plus, le nombre d'instructions du code source est souvent différent du nombre d'instructions en bytecode. C'est pourquoi nous avons calculé le nombre d'objectifs couverts plutôt que de prendre en compte les chiffres fournis par les outils. Deux modes de recherche furent utilisés pour JCute : le premier correspond à une recherche en profondeur d'abord sur l'arbre d'exécutions (deux colonnes notées JCute) tandis que le second correspond à une exploration aléatoire des chemins d'exécution (deux colonnes notées JCuteR). Nous avons également mesuré le temps d'exécution requis par JAUT pour compléter l'ensemble de test généré par PEX (colonne comp. JAUT) quand il était incomplet. Le symbole "-" figure quand cette mesure était inutile. En gras sont indiqués les meilleurs résultats obtenus pour chacun des outils.
Rappelons que notre outil ne garantit pas que les données de test qu'il génère puissent être créées par des séquences d'appels à des constructeurs et des méthodes publics. JCute et Pex le font et fournissent le code nécessaire à l'exécution de la méthode sous test avec les données d'entrée, sous la forme de cas de test respectivement en JUnit et en .NET. Par souci d'équité, tous les attributs des classes sont publics, ainsi placer un objet dans un certain état ne pose pas de problème. JCute et Pex peuvent donc générer les mêmes données de test que JAUT, et le surcoût lié à la génération des cas de test s'en trouve limité. Pex et JCute cherchent par défaut à couvrir toutes les branches du programme sous test, y compris les branches des méthodes appelées. Pour Pex, l'utilisateur peut préciser sur la couverture de quel portion code se concentrer, ce que nous avons fait pour les expériences, mais pour JCute il n'y a pas de possibilité de désactiver la couverture des branches des méthodes appelées. Le critère choisi, qui est de couvrir toutes les instructions qui sont des cibles de sauts conditionnels, se veut une base commune pour comparer les outils, même si les outils n'ont pas été conçus pour couvrir ce critère. Ainsi, Pex a pour objectif de couvrir toutes les branches, JCute également et cherche en plus à couvrir les branches des fonctions appelées. JAUT, quant à lui, cherche à couvrir des instructions difficiles à atteindre, il est un complément aux autres outils car la complexité de son modèle mémoire, qui a un coût en termes de temps d'exécution, est inutile pour couvrir des instructions facilement atteignables en générant des données d'entrée aléatoirement ou couvertes par d'autres outils.
Dans cette section sont fournies, pour chacun des programmes testés, leurs spécificités et une analyse des résultats.

9.1 Programmes avec arithmétique

Nous avons comparé le comportement des outils sur des programmes contenant des opérations arithmétiques. Le premier programme est un programme sans boucle, trityp, issu de la littérature du test logiciel. Le second programme comporte une boucle dont la satisfaction de la condition d'arrêt dépend d'opérations arithmétiques.

9.1.1 Programme trityp

`trityp` est une méthode statique qui prend trois entiers en entrée (les longueurs des côtés du triangle) et qui rend une valeur entière dépendant du type du triangle. Ce programme (listing 9.1) est souvent utilisé pour évaluer les méthodes de génération automatique de données de test car certaines instructions ont une probabilité très faible d'être exécutées, ce qui met en difficulté les générateurs aléatoires de données de test.

JAUT permet d'atteindre la couverture de chacune des branches de la méthode. Pour la majorité des branches, chacune d'entre elles est couverte en moins d'une minute. La branche L est couverte en 7 minutes, la branche G en 17 minutes et la branche I en 28 minutes. Le temps de génération élevé obtenu pour couvrir la branche I s'explique par le temps nécessaire au solveur fd de SICStus Prolog pour trouver une solution aux contraintes entières générées par JAUT, dont il a la charge. En effet, pour trouver une solution qui satisfasse un système tel que $I + J \leq K, J + K > I, I + J > K, J \neq K, I \neq K, I \neq J, K \neq 0, J \neq 0, I \neq 0$, le solveur de SICStus Prolog énumère plusieurs combinaisons de valeurs pour les variables I, J et K. Si le domaine des variables sous contraintes est grand, ce qui est notre cas puisque I, J et K modélisent des entiers sur 32 bits, la phase d'énumération est très longue ce qui explique le mauvais temps obtenu.
Sur cet exemple, la détection précoce de sous-chemins non exécutables permet de gagner du temps de génération, car cette méthode contient de nombreux chemins non exécutables. Ainsi, pour atteindre l'instruction i, JAUT pose les contraintes associées au sous-chemin partant de l'instruction i jusqu'à l'instruction a et ne passant pas par les branches C, D et E. La résolution du système est lancée, elle détecte que le chemin est non exécutable. Cette détection précoce évite la pose des contraintes qui expriment la négation de la condition $((i == 0)||(j == 0)||(k == 0))$, d'où un gain de temps. D'autant plus que des situations similaires sont plusieurs fois rencontrées, plusieurs sous-chemins non exécutables entre les instructions a et i sont parcourus (en sens inverse de l'exécution).
Les résultats obtenus avec JAUT sur ce programme sont donc mitigés : le temps nécessaire à la génération des contraintes n'est pas en cause, mais le temps de résolution par SICStus prolog des contraintes générées pour couvrir certaines instructions est trop important. Pex, qui utilise le solveur de Microsoft SMT Z3, obtient de très bons résultats sur ce programme : un ensemble de données de test permettant une couverture de 100% des branches est obtenu en 4,17 secondes. JCute quant à lui ne permet pas l'obtention d'une couverture totale : il couvre 73% des branches.
Utiliser un solveur autre que celui de SICStus pour les entiers (par exemple le solveur Z3) pourrait permettre de gagner en efficacité.

9.1.2 Programme avec boucle

Le programme du listing 9.2 présente une boucle, le nombre de dépliages nécessaires de la boucle pour atteindre l'instruction `return 1` est conditionné par l'instruction conditionnelle inst.

JAUT génère deux données de test permettant une couverture totale de la méthode : i=0 et i=42, la génération prend 0,17 seconde.
Pex ne permet pas la couverture de l'instruction I. Avec la valeur par défaut des bornes en paramètre, Pex indique que certaines bornes sont atteintes. Modifier une première fois la valeur de ces bornes, comme le suggère l'outil, ne permet pas davantage d'atteindre l'instruction de retour *return 1*. Fixer la valeur des bornes à l'infini ne semble pas augmenter la couverture : aucune amélioration de la couverture n'est observée au bout d'un quart

```
public static int trityp(int i,int j, int k)
{
  int trityp ;
  if ( (i == 0) || (j == 0) || (k == 0))
      trityp = 4 ;//branche A
  else
    {
    //branche B
  trityp = 0 ;//instruction a
  if ( i == j)
      trityp = trityp + 1 ;//branche C
  if ( i == k)
      trityp = trityp + 2 ;//branche D
  if ( j == k )
      trityp = trityp + 3 ;//branche E
  if (trityp == 0)
    {
        //branche E
      if ( (i+j <= k) || (j+k <= i) || (i+k <= j))
          trityp = 4 ;//branche F
      else
          trityp = 1 ;//branche G
    }
  else
    {
    //branche H
  if (trityp > 3)
      trityp = 3 ;//instruction i, branche I
  else
    if ( (trityp == 1) && (i+j > k) )
        trityp = 2 ;//branche J
    else
      if ( (trityp == 2) && (i+k > j) )
        trityp = 2 ;//branche K
      else
        if ( (trityp == 3) && (j+k > i))
          trityp = 2 ;//branche L
        else
          trityp = 4 ;//branche M
    }
    }
  return trityp ;
}
```

Listing 9.1 – La méthode `trityp`

```
public static final int a=50;
public static int foo(int i){
         int j = 10 ;
         while (i > 1){
            j++;
            i--;
                      }
         if (j > a)//instruction inst
            return 1;
         return 0;
     }
```

Listing 9.2 – Programme avec une boucle dont la condition d'arrêt dépend d'un paramètre

d'heure.
JCute permet de couvrir toutes les instructions de la méthode en 188 secondes. Cependant, si on augmente progressivement la valeur de la constante a, on constate que le temps nécessaire à JCute pour générer un ensemble de données de test permettant de couvrir toutes les instructions croît beaucoup plus rapidement que le temps nécessaire à JAUT comme l'illustre la figure 9.2. Le surcoût pour JCute lié à la création de tests unitaires JUnit n'explique pas cet accroissement, car le nombre de données de test à générer n'augmente pas avec la valeur de a (il faut toujours deux données de test). Notre analyse est que JCute nécessite davantage d'essais de valeurs pour trouver une entrée qui atteigne l'objectif quand a augmente, et donc davantage d'exécutions du programme sous test puisque JCute utilise une exécution symbolique dynamique. Dans JAUT les contraintes sont directement traitées par le solveur de contraintes, et le *backtrack* utilisé pour le parcours est fourni par prolog d'une manière optimisée. Sur cet exemple, JAUT est le plus adapté des quatre outils testés pour générer rapidement des données de test couvrant toutes les instructions.

9.2 Programmes avec structures de données

9.2.1 Listes doublement chaînées

Le test porte sur une classe `DoublyLinkedList` qui fait appel à une classe `Node` pour représenter les maillons des listes doublement chaînées. Elles manipulent des références, en tant qu'attributs de classe, paramètres de méthodes, et dans certaines instructions conditionnelles des méthodes. Les appels de méthodes sont également nombreux. Le code de ces classes se trouve dans l'annexe A. Couvrir l'ensemble des instructions des méthodes des deux classes ne pose pas de problème aux autres outils, l'utilisation de ce programme avait surtout pour but de s'assurer du bon fonctionnement de l'implémentation des contraintes portant sur le tas.
Pour chacune des méthodes, JAUT génère un ensemble de données de test couvrant toutes les instructions en moins d'une demi-seconde. La détection rapide de sous-chemins non exécutables permet là encore, pour certaines méthodes, un gain de temps.
JCute et Pex couvrent les instructions des méthodes avec un temps de génération compris entre une et dix secondes, temps supérieur au temps nécessaire à JAUT qui nécessite moins de deux dixièmes de seconde. Il faut cependant rappeler que JCute et Pex d'une part essaient de couvrir également toutes les instructions des méthodes appelées par la méthode sous test, et d'autre part génèrent le code nécessaire à l'exécution des tests.

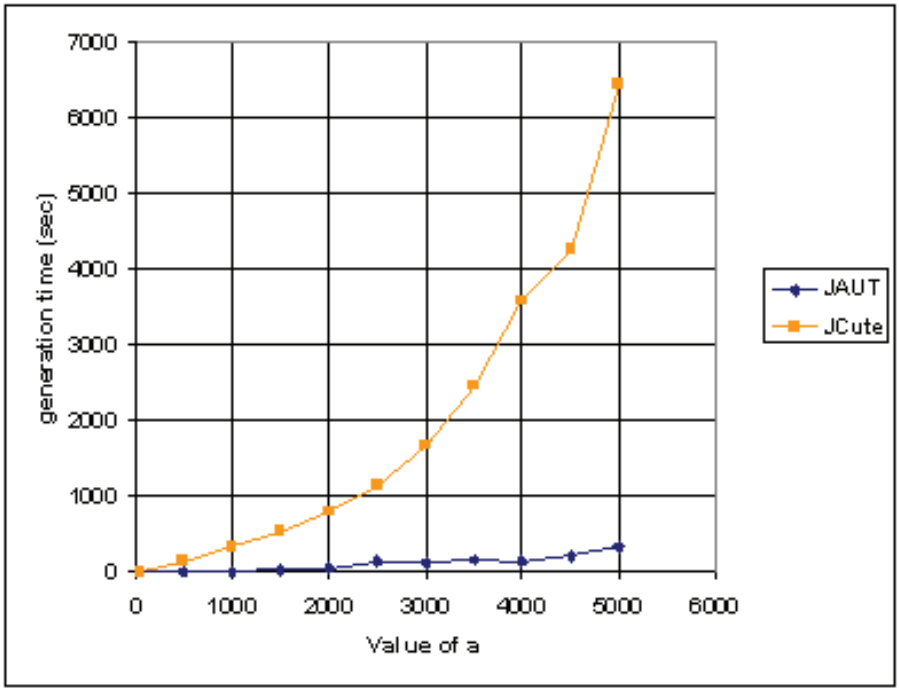


FIG. 9.2 – Variation du temps de génération des données de test pour la méthode `foo` en fonction de `a`

JAUT obtient donc de bons résultats sur cet exemple, la propagation pour les contraintes portant sur le tas se montre ici efficace.

9.2.2 Arbres colorés

Nous avons utilisé une adaptation de la librairie Java TreeMap, qui est intéressante pour évaluer les outils car ses méthodes comportent des instructions conditionnelles complexes sur les références, nécessitant un traitement des alias de références notamment pour les données en entrée de méthodes. Il a été nécessaire de l'adapter car JAUT n'est pas encore étendu pour le traitement des interfaces. Tous les attributs ont été déclarés `public` et aucun invariant de classe n'est fourni aux outils. Par conséquent, les données de test générées sont des données qui couvrent les instructions, mais il n'est pas garanti qu'ils respectent les invariants décrivant un arbre coloré. Le code complet de la librairie `RedBlackTree` avec les instructions objectifs utilisées sont données dans l'annexe B.

Les instructions objectifs sont les instructions cibles des instructions de saut conditionnel, qui sont des instructions en début de blocs de base. En effet, les choisir à la fin d'une séquence d'instructions d'un bloc de base complique très fortement la tâche car certaines méthodes appellent d'autres méthodes très complexes. Or, pour qu'une donnée de test atteigne le début d'un bloc de base mais pas la fin, il faut qu'il y ait des erreurs levées lors de l'exécution du bloc. Il paraît donc préférable de chercher à générer une donnée de test qui atteint le début du bloc, et de l'exécuter ensuite pour vérifier qu'elle atteint bien la fin

de ce bloc [1].

Sur les quatre méthodes considérées, JCute et Pex obtiennent moins de 95% de couverture des objectifs ; pour l'une de ces méthodes (la méthode `fixAfterInsertion`) le taux chute à 55% pour Pex et 18% pour JCute. JAUT se montre capable de générer des données de test comportant des structures de données complexes puisqu'il atteint 100% de couverture en moins d'une minute vingt pour chacune des méthodes. Le listing 9.3 donne un extrait des classes et d'une méthode, `rotateLeft`, et l'on prend pour instruction bytecode à couvrir celle qui correspond à l'instruction du code source notée *inst* sur le code fourni.

Ni Pex, ni JCute n'arrivent à générer d'entrée pour atteindre l'instruction *inst*. JAUT y parvient et fournit la donnée d'entrée représentée de manière schématique sur la figure 9.3.

Cependant, pour couvrir l'ensemble des objectifs des méthodes `fixAfterInsertion` et `fixAfterDeletion`, certains paramètres de la génération sont importants. Ces méthodes comportent des boucles : en raison des problèmes que cela pose pour l'exploration (détaillés en 6.1.3.2), le paramètre des arcs est fixé à 1 (chaque arc issu d'un point de choix ne peut être parcouru qu'une fois). Le paramètre sur la longueur des chemins est fixé à 150. Toutefois, si la borne de longueur fixée ne convient pas, l'outil la fait grandir progressivement. En effet, tester plusieurs valeurs de bornes n'est pas rédhibitoire en termes de temps puisque JAUT se montre rapide pour atteindre ces débuts de branches.

Sur cet exemple, utiliser JAUT permet donc de compléter l'ensemble de données de test généré par Pex et JCute. JAUT est capable de prendre en compte des contraintes sur les alias complexes, comme le prouve le cas détaillé de rotateLeft.

9.3 Limitation du parcours de graphe

9.3.1 Josephus_m

Le programme josephus est inspiré d'un programme faisant partie du folklore des problèmes de pointeurs en C. Ici, le programme josephus est modifié par l'ajout d'une structure if-then-else en fin de programme, le code est donné 9.4.

La fin de ce code met volontairement en évidence les différentes branches qui apparaissent au niveau du bytecode lorsque l'instruction `return ndeEnd.key == 41 && nde.key == 31` est compilée. La première boucle construit une liste circulaire de n nœuds, alors que les boucles while1 et for2 suppriment un par un les nœuds de la chaîne jusqu'à ce qu'un seul reste. La boucle for2 calcule quel nœud doit être éliminé, les nœuds éliminés se situant à intervalles réguliers fixés par le paramètre m. Pour une exécution donnée, le nombre de tours de boucle effectués dans les structures de contrôle while1 et for1 sont égaux. Pour que la condition de l'instruction I soit satisfaite, il faut que ces boucles soient dépliées exactement 40 fois.
Pour couvrir l'instruction J, JAUT utilise une recherche arrière en profondeur qui se fait "piéger" par la boucle for1. Soit le paramètre de génération *borne_arcs* qui donne le nombre maximal de fois où l'on peut passer par le même arc, et par conséquent le nombre

1. D'autres tests ont été effectués pour atteindre les instructions en fin de blocs, JAUT arrive à couvrir toutes les instructions mais il s'avère extrêmement sensible au paramètre de la longueur des chemins. Il faudrait faire davantage d'essais pour trouver la valeur de la borne de longueur à fournir par défaut, et peut-être l'adapter à la longueur du programme. Le lecteur intéressé pourra trouver tous les résultats d'expériences en annexe B

```
public final class Entry {

        public static final int BLACK=1;
        public static final int RED=0;
        public int key;
        public int value;
        public Entry left;
        public Entry right;
        public Entry parent;
        public int color;

        ...
}

public class RedBlackTree {


         public Entry root;
         public int size;//The number of entries in the tree
         public int modCount;//The number of structural modifications to the tree.

        public void rotateLeft(Entry p) {
        if (p != null) {
                Entry r = p.right;
                p.right = r.left;
                if (r.left != null)
                        r.left.parent = p;
                r.parent = p.parent;
                if (p.parent == null)
                        root = r;
                else
                        if (p.parent.left == p)
                         //inst
                        ...
                }
        }
}
```

Listing 9.3 – Extrait de classes sous test représentant des arbres colorés avec une instruction objectif notée inst

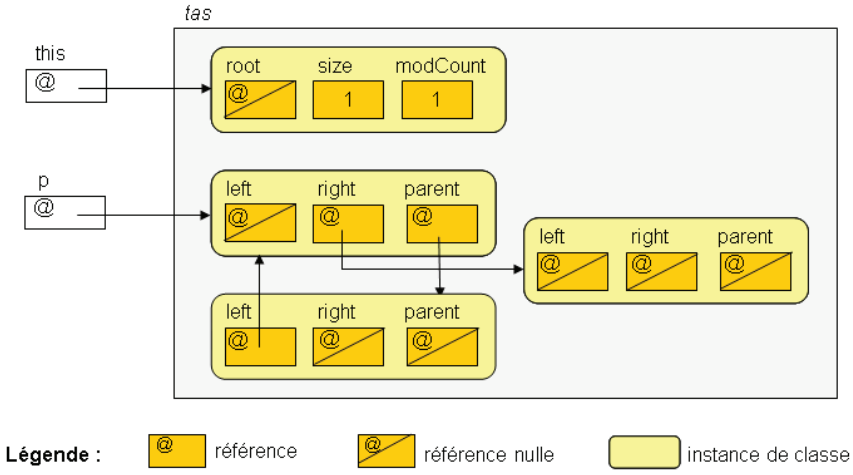


FIG. 9.3 – Forme de la mémoire en entrée générée par JAUT pour couvrir l'instruction inst de la méthode rotateLeft du listing 9.3

de fois qu'une boucle peut être dépliée dans les chemins testés : JAUT commence par explorer des chemins en dépliant uniquement la boucle $for1$, et ne revient sur son choix de ne pas déplier while1 qu'après avoir atteint la limite $borne_arcs$ pour le nombre de dépliages de for1. Puis il explore les chemins pour lesquels for1 est déplié entre 0 et $borne_ars$ fois et while1 une fois. Le processus se poursuit jusqu'à ce que while1 et for1 soient dépliés 40 fois chacun.
Pour pouvoir couvrir l'instruction J, $borne_arcs$ doit être au moins égal à 40. Plus $borne_arcs$ est grand, plus le nombre de chemins testés est important. Cependant, même en fixant $borne_arcs$ à la valeur idéale dans notre cas, c'est-à-dire 40, JAUT ne permet pas la couverture de J avant d'atteindre la fin du temps imparti : ce programme est donc problématique pour l'outil, un parcours plus adapté des chemins est nécessaire.
Ce programme pose également problème à JCute qui obtient 70% d'objectifs couverts. Pex, quand à lui, génère un ensemble de 19 données de test pour couvrir l'intégralité des instructions de la méthode en 8.41 secondes.

9.4 Conclusion sur la validation de l'approche

Ces expérimentations montrent que le modèle à contraintes est déductif, en particulier pour raisonner sur des relations d'alias complexes. JAUT arrive à couvrir des instructions que les autres outils ne couvrent pas, et les temps de génération obtenus sont encourageants. Le parcours incrémental est également bénéfique car il permet d'écarter certains sous-chemins non exécutables. Cependant, il faudrait évaluer expérimentalement la prise en compte du polymorphisme. De plus, le parcours du graphe de flot de contrôle est encore à améliorer, car atteindre certaines instructions requiert un paramétrage trop spécifique.

```
public static bool josephus_m(int n, int m)
       {
           /* ----code de josephus(int n, int m)-------------*/
           //construction d'une liste  circulaire  de longueur n
           NodeJ nde0 = new NodeJ(1);
           NodeJ nde = nde0;
           for (int i = 2; i <= n; i++) //for1
           {
               nde.next = new NodeJ(i);
               nde = nde.next;
           }
           NodeJ ndeEnd = nde;
           ndeEnd.next = nde0;
           while (nde != nde.next) //while1
           {
               for (int i = 1; i < m; i++)//for2
               {
                   nde = nde.next;
               }
               nde.next = nde.next.next;
           }
           /* ----code ajouté pour josephus_m----------------*/
          //bloc  équivalent  à  return  ndeEnd.key == 41 && nde.key == 31;
           if (ndeEnd.key == 41){ //instruction conditionnelle I
                   if(nde.key == 31)//instruction J
                       return true;

            }
           return false;
       }
```

Listing 9.4 – Code de `josephus_m`

Quatrième partie

Conclusion et perspectives

Chapitre 10

Conclusion

Nous avons proposé une nouvelle méthode de génération automatique de données de test, basée sur la programmation par contraintes, pour des programmes en bytecode Java. Cette méthode s'appuie sur un modèle à contraintes de la sémantique relationnelle du bytecode Java, qui permet de restreindre par propagation de contraintes le nombre de données d'entrée à énumérer pour couvrir une séquence d'instructions menant à l'objectif. La force de ce modèle à contraintes est sa capacité à traiter des relations d'alias complexes entre références, à gérer la méconnaissance de la forme de la mémoire en entrée, ainsi que l'utilisation de variables de type permettant de modéliser l'héritage et les appels de méthodes polymorphes. En guise d'illustration de la capacité de déduction du modèle, la figure 9.3 de la section 9.2.2 montre qu'il permet de générer des structures de données complexes, notamment cycliques, indispensables pour couvrir certaines instructions. Ceci se fait sans nécessiter l'énumération de toutes les formes de mémoire possibles car le domaine de recherche est restreint par propagation de contraintes. Des déductions sur les types sont également possibles, par exemple grâce à la modélisation de l'instruction `checkcast`.

Le parcours de graphe utilisé pour énumérer les chemins menant à l'instruction cible — donc pour choisir les séquences d'instructions à couvrir — est original. Il part de l'objectif à atteindre pour "remonter" vers le point d'entrée de la méthode, et fait régulièrement appel au solveur de contraintes pour détecter des conditions de chemins inconsistantes et couper au plus tôt l'exploration de chemins non exécutables. Différentes stratégies ont été proposées pour le rendre plus efficace : ordonnancement des prédécesseurs en présence de point de choix, bornes sur le nombre de passages par les arcs et sur la longueur des chemins parcourus.

Notre approche est complémentaire des autres méthodes de génération de données de test, et vise une couverture de 100% des instructions exécutables. Les autres méthodes nécessitent soit de faire des approximations sur les conditions de chemins, soit que l'utilisateur fournisse des indications sur l'état de la mémoire en entrée (relations d'alias), ou ne gèrent pas l'allocation dynamique. Ces limites sont dues au modèle mémoire, aux solveurs de contraintes ou aux prouveurs qu'elles utilisent. Elles ont cependant l'avantage d'être moins coûteuses, leur utilisation en premier lieu est donc indiquée. Notre méthode permet ensuite de compléter la couverture grâce à son modèle mémoire et à ses contraintes plus expressives, et à son parcours original du graphe de flot de contrôle. En effet, d'une part le modèle mémoire et les contraintes expressives permettent de modéliser des relations d'alias entre références, y compris en présence de déréférencements multiples, ou encore de déduire des relations d'alias entre références n'apparaissant pas explicitement dans le test des

conditions des instructions conditionnelles. D'autre part le parcours du graphe de flot de contrôle, en sens inverse de l'exécution, peut s'avérer plus efficace pour couvrir certaines instructions.

Notre méthode de génération automatique de données de test est implantée dans un outil, JAUT (Java Automatic Unit Testing), qui contient un solveur de contraintes développé pour mettre en œuvre notre modèle. La capacité de notre méthode à compléter la couverture de code obtenue avec d'autres méthodes de génération de tests a été confirmée par des expérimentations. JAUT permet en effet, sur certains programmes, de compléter la couverture de code obtenue avec d'autres outils tels que Pex. Ainsi, pour quatre méthodes de la classe `RedBlackTree` mettant en jeu des relations d'alias de références et des déréférencements multiples, Pex obtient moins de 83% de couverture des instructions cibles alors que notre outil permet d'atteindre 100% de couverture (cf. section 9.2.2). Les temps obtenus lors des expériences sont encourageants. Notre outil est cependant mis en défaut par certains types de programmes avec boucles en raison du mode de parcours du graphe. En effet, en présence de deux boucles successives, les chemins sont énumérés en donnant la priorité au dépliage de la première boucle, ce qui peut être inadapté pour atteindre l'instruction cible comme cela est montré sur le programme `josephus_m` décrit en section 9.3.1. L'une des priorités est maintenant de mener des expériences pour vérifier que l'outil supporte le polymorphisme avec des temps de génération raisonnables. D'autres pistes d'amélioration telles que l'extension du langage traité, ou encore l'amélioration du parcours de graphe sont discutées en perspectives.

Chapitre 11

Perspectives

Ces travaux ouvrent diverses perspectives : améliorer le parcours du graphe de flot de contrôle, étendre le langage traité, ne générer que des entrées réalistes, prendre en compte des spécifications, améliorer le traitement des appels de méthode, étendre l'approche pour couvrir d'autres critères, augmenter la puissance de déduction des opérateurs, et permettre une approche orientée but.

11.1 Améliorer le parcours du graphe de flot de contrôle

Le parcours actuel du graphe de flot de contrôle se fait en sens inverse de l'exécution, en profondeur d'abord, en fixant des bornes sur la longueur du chemin exploré (*borne de longueur*) et sur le nombre de passage par les arcs situés aux points de choix (*borne des arcs*).
Il faudrait davantage d'expériences avec l'outil JAUT pour déterminer la valeur à fixer par défaut à la borne de longueur [1], et le pas duquel l'augmenter lorsque la génération échoue et qu'elle s'est montrée limitante. Il faudrait également choisir quelle borne modifier lorsque les deux bornes se sont montrées limitantes, à moins de laisser ce choix à l'utilisateur comme le fait Pex, qui suggère de nouvelles valeurs pour les paramètres de génération.

Le parcours en profondeur des chemins est mis en défaut sur le programme `josephus_m`, où deux boucles successives dans le programme doivent être dépliées exactement 40 fois chacune. D'autres heuristiques pourraient être envisagées, par exemple en introduisant de l'aléatoire dans le choix des prédécesseurs.

Il serait également possible de comparer entre eux les états mémoires sous contraintes (EMC) générés en un point de programme, cela pourrait éviter de passer plusieurs fois par le même état en arrêtant l'exploration du chemin courant. Se pose alors la question du choix de l'opérateur de comparaison des états :
- un opérateur d'égalité ;
- un opérateur d'inclusion de l'état courant dans un état déjà rencontré, mais en ce cas la précision supplémentaire qu'avait l'état courant est perdue ;
- un opérateur d'inclusion d'un état déjà rencontré dans l'état courant, mais en ce cas les comportements du programme sont sous-approximés.

Comparer les états semble coûteux, peut-être davantage que de repasser par les mêmes états. Une étude approfondie de la question serait nécessaire.

1. Actuellement la borne de longueur vaut par défaut 150.

11.2 Étendre le langage traité

Le langage traité par notre modèle nécessite d'être étendu afin de pouvoir tester une plus grande classe de programmes.

11.2.1 Gestion des nombres flottants

Les nombres flottants posent problème pour la génération de données de test, car leur arithmétique n'est pas exacte. Dans Pex, un solveur de contraintes sur les nombres rationnels est utilisé, mais cela introduit une approximation car l'arithmétique des flottants est différente de celle des rationnels (et de celle des réels). La prise en compte sans approximation des opérations sur les nombres flottants nécessite de développer des solveurs de contraintes dédiés, respectant la sémantique de l'arithmétique sur les flottants. Un tel solveur pour les flottants qui respectent le standard *IEEE-754* est par exemple décrit dans [BGM06] et est utilisé notamment dans les outils Euclide et InKa.
L'arithmétique des flottants utilisée dans Java utilise un sous-ensemble du standard *IEEE-754*. Les instructions de saut conditionnel portant sur les flottants (instructions `fcmpg`, `fcmpl`,`dcmpg`, `dcmpl`) ont quant à elles une sémantique propre au bytecode Java. Une solution est donc d'adapter le solveur de flottants décrit dans [BGM06] pour le bytecode, une étude plus approfondie serait nécessaire pour savoir si cela est réalisable. L'architecture de JAUT devrait permettre un ajout assez aisé du solveur.

11.2.2 Traitement des tableaux

Notre modèle doit être étendu pour gérer les tableaux. En Java les tableaux sont des objets, leur longueur est connue. Les tableaux peuvent être représentés dans le tas de manière similaire aux objets, en faisant un parallèle entre les cases d'un tableau et les attributs des objets. Cependant, une différence non négligeable entre la manipulation des tableaux et celle des objets doit être prise en considération : pour un objet, les instructions d'accès et de modification des attributs (`getfield` et `putfield`) désignent l'attribut à manipuler via un indice d'attribut qui est une constante, alors que l'accès ou la modification d'un tableau peut se faire via un index variable dépendant des données d'entrée de la méthode. Ainsi, l'accès à un élément d'un tableau serait représenté par un opérateur à contraintes $atload(H, Index, Ref, Val)$ où H est le tas, Ref la référence qui désigne le tableau auquel on accède (qui a pour domaine un ensemble d'adresses du tas possibles pour le tableau), $Index$ l'index de l'élément auquel on accède dans le tableau qui est une variable entière positive, et Val la valeur à laquelle on accède. Les règles de déductions seraient donc plus nombreuses que pour l'opérateur équivalent pour l'accès à un attribut, $getfield(H, idAttr, Ref, Val)$, puisque $Index$ est une variable et $idAttr$ une constante.
Pouvoir modéliser de manière exacte la sémantique de l'accès à un élément d'un tableau, et de la modification d'un tel élément via un index variable, est l'un des avantages du pouvoir d'expression offert par la programmation par contraintes. Les approches n'utilisant pas la programmation par contraintes considèrent en général autant de cas que de valeurs possibles pour l'index, ce qui multiplie le nombre de chemins à analyser.
En résumé, l'extension de notre méthode aux tableaux requiert l'écriture d'opérateurs à contraintes spécifiques qui s'inspirent des algorithmes utilisés pour la manipulation des objets, cela ne pose pas de difficultés théoriques grâce à la flexibilité de la programmation par contraintes.

11.2.3 Autres extensions

Notre modèle ne traite pas actuellement les interfaces, les attributs statiques, les modificateurs de visibilité, les opérateurs bit à bit et les exceptions.

Le traitement des **interfaces** ne devrait pas poser de problème. Un objet dont le type déclaré est une interface a pour types possibles toutes les classes de la famille de l'interface. Dans notre approche, l'utilisateur doit lister tous les fichiers bytecode utilisables pour la génération, et donc tous les types pouvant être utilisés. Pour pouvoir tester une méthode définie dans une interface ou ayant un paramètre de type interface, l'utilisateur doit fournir au moins une classe de la famille de l'interface.

La prise en compte des **attributs statiques** ne devrait pas être une difficulté. Un attribut statique est un attribut partagé entre toutes les instances de la classe où il est défini. Dans notre modèle, chacun des attributs statiques définis dans les classes pourrait être représenté par une variable de type primitif (VTPR). Les instructions de manipulation de ces variables se modélisent plus facilement que les instructions de manipulation des variables d'instances (les attributs non statiques) : quand on accède à un attribut non statique, la référence de l'objet dont on accède à l'attribut peut être une variable, alors qu'il n'y a pas d'ambiguïté pour l'accès à un attribut statique.

Notre modèle ne gère pas encore les **modificateurs de visibilité** (`public`, `private` ou `protected`) et traite toutes les classes, tous les attributs et toutes les méthodes comme étant `public`. Il faudrait inclure des règles liées aux modificateurs de visibilité.

Les **opérateurs bit à bit** (`and`, `xor`, `shift`) restent à modéliser.

Enfin la prise en compte des **exceptions** est nécessaire. En effet, certaines instructions qui ne font pas partie des gestionnaires d'exceptions ne peuvent être atteintes qu'après lancement et récupération d'une exception, car certains programmeurs exploitent les exceptions pour différencier des cas. Cela nécessitera d'étendre la sémantique modélisée qui est actuellement une sémantique sans erreurs.

11.3 Génération de séquences d'initialisation

Actuellement, l'état de la mémoire en entrée généré (objets présents, valeur des variables d'instances et des paramètres) n'est pas forcément un état accessible. Il faut garantir que chacun des objets de l'état d'entrée puisse être créé par une séquence d'appels à un constructeur et des méthodes publics, que l'on appelle ici *séquence d'initialisation*. En ce cas l'entrée est dite réaliste, et peut être générée depuis n'importe quelle autre classe externe. Le choix de ne générer que des entrées réalistes semble discutable, car les méthodes sous test peuvent être appelées à l'intérieur même de leur classe et il faut aussi s'assurer de leur bon comportement dans ce cas là. Le mieux serait d'avoir plusieurs modes, selon le type de test souhaité.

Pour ne générer que des entrées réalistes, comme le nombre de séquences d'initialisation est infini, il faut sélectionner un ordre dans lequel essayer ces séquences. Ainsi, il paraît plus pertinent de considérer d'abord les séquences formées uniquement d'un constructeur, puis des séquences formées d'un constructeur et de modifieurs (*setter*), puis des appels plus

complexes, en augmentant progressivement la longueur de la séquence d'appels.
Une possibilité est alors de générer un état de la mémoire en entrée (une donnée de test) pour la méthode sous test, puis de poser cet état de la mémoire comme étant la mémoire à obtenir en sortie de séquence d'initialisation, d'énumérer les chemins des séquences en générant les contraintes associées, et de résoudre le système. Grâce au modèle à contraintes, la mémoire à obtenir en sortie de séquence d'initialisation va contraindre les valeurs possibles pour la mémoire en entrée de séquence, évitant ainsi l'énumération de tous les états mémoires possibles en entrée. L'avantage de générer d'abord un état de la mémoire pour la méthode sous test et de le poser ensuite comme sortie de la séquence d'initialisation est que le nombre d'objets en mémoire est connu, ce qui guide la création de la séquence d'initialisation (le nombre d'appels aux constructeurs est connu).
Une autre possibilité serait de ne pas instancier la mémoire en entrée de la méthode sous test, et de parcourir les chemins des séquences d'initialisation avant de lancer l'énumération, cette fois sur la mémoire en entrée de séquence d'initialisation. Le problème est la méconnaissance possible du nombre d'objets à créer, la question reste à étudier.

L'avantage de la détermination d'une séquence d'initialisation est de rendre ensuite aisée la création de scripts pour lancer l'exécution de la méthode de test avec les données de test générées.

11.4 Prendre en compte des spécifications

De nombreuses méthodes de génération de données de test prennent en compte des spécifications (méthodes implantées dans PathCrawler, JCute et Pex), de diverses manières. Il serait intéressant de prendre en compte les spécifications dans le cadre de la programmation par contrat, c'est-à-dire si l'on fait l'hypothèse que les méthodes sont toujours appelées en respectant les contraintes données par la spécification (invariants de classe et pré-conditions) [2].
Premièrement, les **invariants de classe** indiquent les contraintes que doivent respecter les instances de la classe. Par exemple, pour les arbres colorés, un invariant peut imposer l'alternance de couleurs d'un niveau de l'arbre à l'autre, ou encore l'équilibrage de l'arbre. Ces invariants peuvent s'écrire dans le langage source comme c'est le cas pour JCute ou pour Pex, qui exploitent une méthode `repOk` répondant vrai si l'instance respecte l'invariant de classe, faux autrement. L'avantage est que la traduction en contraintes de l'invariant se fait de la même façon que celle du code sous test. Les invariants peuvent aussi s'écrire plus classiquement dans un langage de spécifications, il faudrait définir quel langage de spécification serait le plus adapté (JML par exemple) et traduire les expressions du langage de spécification sous forme de contraintes. Quel que soit le mode d'expression des invariants retenu, il faudrait déterminer à quel moment exploiter ces invariants : après la génération d'un objet pour vérifier qu'il satisfait bien l'invariant, ou avant pour guider la génération.
De manière similaire aux invariants de classe, des **pré-conditions** permettraient de guider la génération des entrées en fonction de l'utilisation prévue de la méthode.

Ensuite, si la spécification intègre des **post-conditions**, celles-ci peuvent servir d'oracles. Il est alors envisageable de générer automatiquement un script qui comprend, pour chacune des données de test générées, une séquence d'initialisation, l'exécution de la méthode sous test avec la donnée de test, et la vérification de la post-condition en guise d'oracle.

2. A l'opposée de la programmation défensive qui teste toute les entrées pour vérifier qu'elles sont correctes par rapport à la spécification.

11.5 Améliorer le traitement des appels de méthode

Les appels de méthode accroissent généralement le nombre de chemins possibles. Diverses techniques pourraient être envisagées pour accélérer le traitement de ces appels.

Dans le cadre de mon stage de Master de recherche, nous avons proposé une méthode pour accélérer la prise en compte des appels de méthode dans l'outil InKa. Le principe est de créer un nouveau système de contraintes pour chacune des méthodes appelées, les systèmes de contraintes de la méthode appelée et de la méthode appelante ne partagent que les variables modélisant les paramètres et les valeurs de retour. Cette idée vient du constat que pour certains programmes, peu de variables sont partagées entre la méthode appelante et la méthode appelée. La résolution de contraintes est accélérée en effectuant la propagation de contraintes sur les contraintes liées à la méthode appelée, jusqu'à ce qu'aucune déduction ne soit plus possible, avant de revenir au système de contraintes de la méthode appelante, qui bénéficie alors des déductions effectuées sur les domaines des variables partagées entre les systèmes.

PathCrawler exploite les spécifications des méthodes appelées, sous la forme de couples pre-post, à la place du code de ces méthodes, dans le but de réduire le nombre de chemins à considérer, car la spécification comporte souvent moins de chemins que le programme lui-même. Il serait intéressant d'utiliser la même approche dans notre outil lorsque de telles spécifications sont disponibles.

11.6 Étendre l'approche pour couvrir d'autres critères

Au niveau du bytecode Java, les décisions (par exemple `if (a && (b || c))`) sont décomposées en test de conditions élémentaires (`if(a)`), le test de critères comme le critère MC/DC est donc difficile à mettre en œuvre car la forme qu'avaient les décisions dans le code source est perdue. Les critères basés sur le flot de contrôle utilisés au niveau du code source ne le sont donc pas tous pour le code machine, du moins dans les travaux dont nous avons connaissance. Les critères les plus couramment utilisés pour le bytecode Java sont ceux de **toutes les instructions**, **toutes les branches** et **tous les chemins**. Citons toutefois les travaux de Staats [Sta09] qui proposent au développeur d'exprimer au niveau du code source Java des obligations de couverture pour le critère MC/DC, afin de les couvrir ensuite en opérant au niveau du bytecode (avec l'outil JPF-SE).
Notre approche peut être aisément étendue pour le critère de couverture de toutes les branches, en débutant les chemins explorés par la branche cible (car les chemins explorés le sont dans le sens inverse de celui de l'exécution). Des critères de tous les chemins relaxés pourraient être couverts, soit en bornant la longueur des chemins explorés, soit le nombre de dépliages des boucles, ce qui nécessiterait néanmoins une étude du graphe pour détecter ces boucles.

Pour le critère de couverture basé sur le flot des données (**couverture des couples *def-use***), la difficulté est d'identifier les couples à couvrir. Pour les paramètres, cela semble assez simple en identifiant les instructions qui définissent ou qui lisent la valeur contenue dans un registre. Mais pour les variables d'instance, cela est beaucoup plus difficile car les objets en mémoire ne sont pas toujours connus lors du parcours des chemins. De même des accès au contenu d'un tableau via des index variables rendent la détermination des couples à couvrir plus difficile. Étudier les travaux existants sur la détermination de ces couples est

donc nécessaire. Une fois un couple d'instructions *def-use* trouvé, notre approche devrait permettre de chercher un chemin entre ces instructions sans redéfinition de la variable concernée.

11.7 Rendre les opérateurs à contraintes sur le tas plus déductifs en enrichissant le modèle mémoire

Les opérateurs *getfield* et *putfield* permettent d'accéder ou de modifier un attribut d'une instance de classe. Considérons la séquence d'instructions suivante :

```
getfield #2 ;
...
getfield #2 ;
```

Supposons qu'elle se traduise en contraintes par le système
{*getfield(H1,2,Ref1,Val1)*, ..., *getfield(H2,2,Ref2,Val2)*}.
Si l'opérateur à contraintes $getfield(H1, 2, Ref1, Val1)$ est examiné alors que le tas $H1$ ne contient aucune instance de classe compatible avec le type de $Ref1$, l'opérateur $getfield$ serait plus déductif s'il ajoutait dans $H1$ une instance dont le type serait compatible avec celui de $Ref1$, par exemple à l'adresse i. Supposons que pour les mêmes raisons l'opérateur $getfield(H2, 2, Ref2, Val2)$ soit amené à ajouter une instance compatible avec le type de $Ref2$ à l'adresse j (différente de i par construction). Si à un moment ultérieur la contrainte $Ref1 =_{vtpr} Ref2$ figure dans le système de contraintes, comme $Ref1$ vaut i et $Ref2$ vaut j la contrainte échoue, du moins avec le modèle mémoire actuel. Pour permettre aux opérateurs $getfield$ et $putfield$ de rajouter une instance dans le tas quand aucune des instances présentes n'est compatible avec l'opérateur (à cause de son type ou du domaine de l'attribut), une solution serait d'utiliser des classes d'adresses, c'est-à-dire, en reprenant l'exemple, de pouvoir exprimer que i et j sont dans la même classe, et donc que les instances qui y sont stockées sont égales.

11.8 Exploiter le modèle de mémoire dans une approche orientée but

Le modèle à contraintes proposé pour le bytecode Java est utilisé par une méthode de génération de données de test orientée chemins. Or le modèle a été pensé pour pouvoir être exploité également par une méthode orientée but. Cela nécessite de faire des unions d'états mémoires sous contraintes (EMC) en certains points de programmes où plusieurs chemins se rencontrent, le modèle devrait permettre d'exprimer de telles unions.
Pour savoir en quels points faire des unions d'EMC et comment réfuter certaines branches non exécutables, l'approche orientée but décrite dans [GBR98, Got09], utilisée dans les outils InKa et Euclide, exploite la structure du programme donnée par la syntaxe du code source. Ainsi une structure *if-then-else* se traduit, dans la modélisation utilisée par ces méthodes, par un opérateur à contraintes *ite*, qui essaie de montrer que l'une des deux branches *then* ou *else* est non exécutable, et qui fait, s'il n'y arrive pas, l'union des EMC provenant des deux branches. Au niveau du bytecode, les structures de contrôle ne sont plus repérables syntaxiquement, il est nécessaire de se baser sur le graphe de flot de contrôle pour trouver des motifs (boucles par exemple) et choisir en quels points faire les unions de mémoire.

Annexes

Annexe A : classes Node et DoublyLinkedList

La classe `Node` représente un nœud d'une liste doublement chaînée, la classe `DoublyLinkedList` sert à manipuler de telles listes.

```
/**
 * Basic doubly linked list node interface.
 */
public class Node {
  public Node next;
  public Node prev;

  public Node getNext() { return next; }
  public Node getPrev() { return prev; }

  public void setNext(Node newNext) { next = newNext; }
  public void setPrev(Node newPrev) { prev = newPrev; }

  /**
   * Unlink this node from it's current list ...
   */
  public void unlink()
  {
    if (getNext() != null)
      getNext().setPrev(getPrev());
    if (getPrev() != null)
      getPrev().setNext(getNext());

    setNext(null);
    setPrev(null);
  }

  /**
   * Link this node in, infront of nde (unlinks it's self
   * before hand if needed).
   * @param nde the node to link in before.
   */
  public void insertBefore(Node nde)
  {
    // Already here ...
    if (this == nde) return;
    if (getPrev() != null)
      unlink();
    // Actually insert this node ...
```

```
    if (nde == null) {
      // empty lst ...
      setNext(this);
      setPrev(this);
    } else {
      setNext(nde);
      setPrev(nde.getPrev());
      nde.setPrev(this);
      if (getPrev() != null)
        getPrev().setNext(this);
    }
  }
}
```

Listing 11.1 – Code Java de la classe `Node` qui représente un nœud d'une liste doublement chaînée

```
public class DoublyLinkedList {

  public Node head;
  public int size;

  public DoublyLinkedList() {}

  /**
   * Returns the number of elements currently in the list .
   */
  public int getSize() { return size; }

  /**
   * Removes all elements from the list .
   */
  public void empty() {
    while(size > 0) pop();
  }

  /**
   * Get the current head element
   * @return The current ' first ' element in list .
   */
  public Node getHead() { return head; }
  /**
   * Get the current tail element
   * @return The current 'last ' element in list .
   */
  public Node getTail() { return head.getPrev(); }

  /**
   * Moves <tt>nde</tt> to the head of the list (equivilent to
   * remove(nde); add(nde); but faster .
   */
  public void touch(Node nde) {
    if (nde == null) return;
    nde.insertBefore(head);
    head = nde;
  }
```

```
public void add(int index, Node nde) {
  if (nde == null) return;
  if (index == 0) {
    // This makes it the first element in the list .
    nde.insertBefore(head);
    head = nde;
  } else if (index == size) {
    // Because the list is circular this
    // makes it the last element in the list .
    nde.insertBefore(head);
  } else {
    Node after = head;
    while (index != 0) {
      after = after.getNext();
      index--;
    }
    nde.insertBefore(after);
  }
  size++;
}

/**
 * Adds nde to the head of the list .
 * In perl this is called an 'unpop'. <tt>nde</tt> should
 * not currently be part of any list .
 * @param nde the node to add to the list .
 */
public void add(Node nde) {
  if (nde == null) return;
  nde.insertBefore(head);
  head = nde;
  size++;
}

/**
 * Removes nde from the list it is part of (should be this
 * one, otherwise results are undefined). If nde is the
 * current head element, then the next element becomes head,
 * if there are no more elements the list becomes empty.
 * @param nde node to remove.
 */
public void remove(Node nde) {
  if (nde == null) return;
  if (nde == head) {
    if (head.getNext() == head)
      head = null; // Last node...
    else
      head = head.getNext();
  }
  nde.unlink();
  size--;
}

/**
 * Removes 'head' from list and returns it . Returns null if list is empty.
 * @returns current head element, next element becomes head.
 */
public Node pop() {
  if (head == null) return null;
```

```
    Node nde = head;
    remove(nde);
    return nde;
  }

  /**
   * Removes 'tail' from list and returns it. Returns null if list is empty.
   * @returns current tail element.
   */
  public Node unpush() {
    if (head == null) return null;

    Node nde = getTail();
    remove(nde);
    return nde;
  }



  /**
   * Adds nde to tail of list
   */
  public void push(Node nde) {
    nde.insertBefore(head);
    if (head == null) head = nde;
    size++;
  }

  /**
   * Adds nde to head of list
   */
  public void unpop(Node nde) {
    nde.insertBefore(head);
    head = nde;
    size++;
  }
}
```

Listing 11.2 – Code Java de la classe `DoublyLinkedList` qui représente une liste doublement chaînée

Les codes de ces classes sont inspirés d'un code sous licence Apache version 2.0. Le code se trouve à l'adresse http ://www.java2s.com/Open-Source/Java-Document/Graphic-Library/batik/org/apache/batik/util/DoublyLinkedList.java.htm, la licence sur http ://www.apache.org/licenses/LICENSE-2.0.

Annexe B : classes Entry et RedBlackTree

La classe `Entry` représente un nœud d'un arbre coloré, la classe `RedBlackTree` permet de manipuler de tels arbres. Les instructions objectifs sont indiquées dans le code. Les objectifs de type A représentent des instructions cibles de sauts conditionnels, les objectifs de type B sont positionnés en fin des blocs de base.

```
/* Adaptation de la librairie Sun TreeMap */

public final class Entry {

 public static final int BLACK=1;
 public static final int RED=0;
 public int key;
 public int value;
 public Entry left;
 public Entry right;
 public Entry parent;
 public int color;

 /**
   * Make a new cell with given key, value, and parent
   */
public Entry(int key, int value, Entry parent) {
  this.key = key;
  this.value = value;
  this.parent = parent;
 }

 /**
   * Returns the key.
   * @return the key
   */
public int getKey() {
  return key;
 }

 /**
   * Returns the value associated with the key.
   * @return the value associated with the key
   */
public int getValue() {
  return value;
 }
```

```
 /**
  * Replaces the value currently associated with the key with the given value.
  * @return the value associated with the key before this method was called
  */
 public int setValue(int value) {
  int oldValue = this.value;
  this.value = value;
  return oldValue;
 }
}
```

Listing 11.3 – Code Java de la classe `Entry` qui représente une entrée d'un arbre binaire équilibré rouge-noir

```
/* Adaptation de la librairie Sun TreeMap */

public class RedBlackTree {

  public Entry root;
  public int size;//The number of entries in the tree
  public int modCount;//The number of structural modifications to the tree.

  public static Entry parentOf(Entry p) {
    return (p == null ? null: p.parent);
  }
  public static int colorOf(Entry p) {
    return (p == null ? Entry.BLACK : p.color);
  }

  public static Entry leftOf(Entry p) {
    return (p == null) ? null: p.left;
  }

  public static Entry rightOf(Entry p) {
    return (p == null) ? null: p.right;
  }

  public static void setColor(Entry p, int c) {
    if (p != null)
      p.color = c;
  }

  public void deleteEntry(Entry p) {
    modCount++;
    size--;
    //If strictly internal, copy successor's element to p and then make p
    //point to successor.
    if (p.left != null //OBJ A1
        && p.right != null) {//OBJ A2
      Entry s = successor (p);
      p.key = s.key;
      p.value = s.value;
      p = s;//OBJ B1
    }// p has 2 children

    //Start fixup at replacement node, if it exists.
    Entry replacement =
```

```
    (p.left != null ? //OBJ A3
       p.left //OBJ B2
       : //OBJ A4
         p.right //OBJ B3 );
         if (replacement != null) {//OBJ A5
           // Link replacement to parent
           replacement.parent = p.parent;
           if (p.parent == null)//OBJ A6
             root = replacement;//OBJ B4
           else //OBJ A7
             if (p == p.parent.left)//OBJ A8
               p.parent.left = replacement;//OBJ B5
             else//OBJ A9
               p.parent.right = replacement;//OBJ B6
           // Null out links so they are OK to use by fixAfterDeletion.
           p.left = p.right = p.parent = null;

           // Fix replacement
           if (p.color == Entry.BLACK)//OBJ A10
             fixAfterDeletion(replacement);//OBJ B7
         } else //OBJ A11
           if (p.parent == null) { // return if we are the only node. OBJ A12
             root = null;//OBJ B8
           }else{ //No children. Use self as phantom replacement and unlink. OBJ A13
             if (p.color == Entry.BLACK)//OBJ A14
               fixAfterDeletion(p);//OBJ B9
             if (p.parent != null) {//OBJ A15
               if (p == p.parent.left)//OBJ A16
                 p.parent.left = null;//OBJ B10
               else //OBJ A18
                 if (p == p.parent.right)//OBJ A17
                   p.parent.right = null;//OBJ B11
               p.parent = null;//OBJ B12
             }//OBJ B13
           }
}

/**
 * Returns the successor of the specified Entry, or null if no such.
 */
static Entry successor(Entry t) {
  if (t == null)
    return null;
  else
    if (t.right != null) {
      Entry p = t.right;
      while (p.left != null)
        p = p.left;
      return p;
    } else {
      Entry p = t.parent;
      Entry ch = t;
      while (p != null && ch == p.right) {
        ch = p;
        p = p.parent;
      }
      return p;
    }
}
```

```
public void fixAfterDeletion(Entry x) {
  while (x != root //OBJ A1
      && colorOf(x) == Entry.BLACK) {//OBJ A2
    if (x == leftOf(parentOf(x))) {//OBJ A3
      Entry sib = rightOf(parentOf(x));
      if (colorOf(sib) == Entry.RED) {//OBJ A4
        setColor(sib, Entry.BLACK);
        setColor(parentOf(x), Entry.RED);
        rotateLeft(parentOf(x));
        sib = rightOf(parentOf(x));//OBJ B1
      }
      if (colorOf(leftOf(sib)) == Entry.BLACK &&//OBJ A5
          colorOf(rightOf(sib)) == Entry.BLACK) {//OBJ A6
        setColor(sib, Entry.RED);
        x = parentOf(x);//OBJ B2
      } else {//OBJ A7
        if (colorOf(rightOf(sib)) == Entry.BLACK) {//OBJ A8
          setColor(leftOf(sib), Entry.BLACK);
          setColor(sib, Entry.RED);
          rotateRight(sib);
          sib = rightOf(parentOf(x));//OBJ B3
        }
        setColor(sib, colorOf(parentOf(x)));
        setColor(parentOf(x), Entry.BLACK);
        setColor(rightOf(sib), Entry.BLACK);
        rotateLeft(parentOf(x));
        x = root;//OBJ B4
      }
    }else {//OBJ A9
      Entry sib = leftOf(parentOf(x));
      if (colorOf(sib) == Entry.RED) {//OBJ A8
        setColor(sib, Entry.BLACK);
        setColor(parentOf(x), Entry.RED);
        rotateRight(parentOf(x));//OBJ B5
      }
      if (colorOf(rightOf(sib)) == Entry.BLACK //OBJ A10
          && colorOf(leftOf(sib)) == Entry.BLACK) {//OBJ A11
        setColor(sib, Entry.RED);
        x = parentOf(x);//OBJ B6
      }else{//OBJ A12
        if (colorOf(leftOf(sib)) == Entry.BLACK) {//OBJ A13
          setColor(rightOf(sib), Entry.BLACK);
          setColor(sib, Entry.RED);
          rotateLeft(sib);
          sib = leftOf(parentOf(x));//OBJ B7
        }
        setColor(sib, colorOf(parentOf(x)));
        setColor(parentOf(x), Entry.BLACK);
        setColor(leftOf(sib), Entry.BLACK);
        rotateRight(parentOf(x));
        x = root;//OBJ B8
      }//OBJ B9
    }//OBJ B10
  }
  setColor(x, Entry.BLACK);//OBJ B11
}

public void fixAfterInsertion(Entry x) {
  x.color = Entry.RED;
  while (x != null //OBJ A1
      && x != root //OBJ A2
```

```
      && x.parent.color == Entry.RED) {//OBJ A3
    if (parentOf(x) == leftOf(parentOf(parentOf(x)))) {//OBJ A4
      Entry y = rightOf(parentOf(parentOf(x)));
      if (colorOf(y) == Entry.RED) {//OBJ A5
        setColor(parentOf(x), Entry.BLACK);
        setColor(y, Entry.BLACK);
        setColor(parentOf(parentOf(x)), Entry.RED);
        x = parentOf(parentOf(x));//OBJ B1
      } else {//OBJ A6
        if (x == rightOf(parentOf(x))) {//OBJ A7
          x = parentOf(x);
          rotateLeft(x);//OBJ B2
        }
        setColor(parentOf(x), Entry.BLACK);
        setColor(parentOf(parentOf(x)), Entry.RED);
        rotateRight(parentOf(parentOf(x)));//OBJ B3
      }
    } else {//OBJ A8
      Entry y = leftOf(parentOf(parentOf(x)));
      if (colorOf(y) == Entry.RED) {//OBJ A9
        setColor(parentOf(x), Entry.BLACK);
        setColor(y, Entry.BLACK);
        setColor(parentOf(parentOf(x)), Entry.RED);
        x = parentOf(parentOf(x));//OBJ B4
      } else {//OBJ A10
        if (x == leftOf(parentOf(x))) {//OBJ A11
          x = parentOf(x);
          rotateRight(x);//OBJ B5
        }
        setColor(parentOf(x), Entry.BLACK);
        setColor(parentOf(parentOf(x)), Entry.RED);
        rotateLeft(parentOf(parentOf(x)));//OBJ B6
      }
    }
  }
  root.color = Entry.BLACK;//OBJ B7
}

public void rotateLeft(Entry p) {
  if (p != null) {//OBJ A1
    Entry r = p.right;
    p.right = r.left;
    if (r.left != null)//OBJ A2
      r.left.parent = p;//OBJ B1
    r.parent = p.parent;
    if (p.parent == null)//OBJ A3
      root = r;//OBJ B2
    else //OBJ A4
      if (p.parent.left == p)//OBJ A5
        p.parent.left = r;//OBJ B3
      else //OBJ A6
        p.parent.right = r; //OBJ B4
    r.left = p;
    p.parent = r;//OBJ B5
  }//OBJ B6
}

public void rotateRight(Entry p) {
  if (p != null) {
    Entry l = p.left;
    p.left = l.right;
```

```
      if (l.right != null) l.right.parent = p;
      l.parent = p.parent;
      if (p.parent == null)
        root = l;
      else if (p.parent.right == p)
        p.parent.right = l;
      else p.parent.left = l;
      l.right = p;
      p.parent = l;
    }
  }

}
```

Listing 11.4 – Code Java de la classe `RedBlackTree` qui représente des arbres binaires équilibrés rouge-noir

OBJECTIF	Numéro de l'instruction bytecode	borne mémoire	borne des arcs	borne de longueur	temps CPU (sec)
OBJ A1	27	10	-	-	0.08
OBJ A2	34	10	-	-	0.16
OBJ A3	64	10	-	-	0.08
OBJ A4	71	10	-	-	0.16
OBJ A5	80	10	-	-	0.06
OBJ A6	95	10	-	-	0.09
OBJ A7	103	10	-	-	0.14
OBJ A8	114	10	-	-	0.11
OBJ A9	125	10	-	-	0.08
OBJ A10	156	10	-	-	0.22
OBJ A11	164	10	-	-	0.09
OBJ A12	171	10	-	-	0.06
OBJ A13	179	10	-	-	0.09
OBJ A14	187	10	-	-	0.08
OBJ A15	199	10	-	-	0.11
OBJ A16	210	10	-	-	0.06
OBJ A17	221	10	-	-	0.11
OBJ A18	232	10	-	-	0.08
OBJ B1	56	10	-	-	0.17
OBJ B2	68	10	-	-	0.14
OBJ B3	72	10	-	-	0.16
OBJ B4	100	10	-	-	0.14
OBJ B5	122	10	-	-	0.23
OBJ B6	130	10	-	-	0.25
OBJ B7	161	10	-	120	8.67
OBJ B8	176	10	-	-	0.19
OBJ B9	189	10	-	120	9.02
"	"	"	-	150	27.86
"	"	"	-	200	201.25
OBJ B10	218	10	-	-	0.19
OBJ B11	237	10	-	-	0.16
OBJ B12	242	10	-	-	0.25
OBJ B13	245	10	-	-	0.14

FIG. 11.1 – Résultats pour la génération automatique de données de test pour la méthode `deleteEntry` de la classe `TreeMap`

OBJECTIF	Numéro de l'instruction bytecode	borne mémoire	borne des arcs	borne de longueur	temps CPU (sec)
OBJ A1	8	10	1	-	0.25
OBJ A2	16	10	1	-	0.27
OBJ A3	27	10	1	-	0.30
OBJ A4	42	10	1	-	0.30
OBJ A5	82	10	1	-	0.25
OBJ A6	93	10	1	-	0.27
OBJ A7	106	10	1	-	0.27
OBJ A8	117	10	1	-	0.31
OBJ A9	186	10	1	-	0.27
OBJ A10	201	10	1	120	1.30
OBJ A10	"	"	1	150	5.33
OBJ A10	"	"	1	175	16.98
OBJ A10	"	"	1	200	28.19
OBJ A11	241	10	1	-	0.28
OBJ A12	252	10	1	-	0.28
OBJ A13	265	10	1	120	0.75
"	"	"	"	150	2.91
"	"	"	"	175	8.45
"	"	"	"	200	21.33
OBJ A14	276	-	1	120	0.47
"	"	"	"	150	1.30
"	"	"	"	175	4.25
"	"	"	"	200	12.97
OBJ B1	70	10	1	120	fail
"	"	"	"	180	14.90
"	"	"	"	200	40.71
OBJ B2	103	10	1	-	0.23
OBJ B3	142	10	1	120	fail
"	"	"	"	150	fail
"	"	"	"	160	52.70
"	"	"	"	165	93.65
"	"	"	"	170	1372.52
"	"	"	"	180	timeout atteint (1 heure)
OBJ B4	183	10	1	-	0.47
OBJ B5	229	10	1	120	fail
"	"	"	"	180	84.00
"	"	"	"	200	158.83
OBJ B6	262	10	1	-	0.39
OBJ B7	301	10	1	120	fail
OBJ B7	301	10	1	150	fail
OBJ B7	301	10	1	160	54.76
OBJ B7	301	10	1	180	timeout atteint (1 heure)
OBJ B8,B9	342	10	1	120	0.39
OBJ B10	345	10	1	120	0.17
OBJ B11	350	10	1	-	0.17

FIG. 11.2 – Résultats pour la génération automatique de données de test pour la méthode `fixAfterDeletion` de la classe `TreeMap`

OBJECTIF	Numéro de l'instruction bytecode	borne mémoire	borne des arcs	borne de longueur	temps CPU (sec)
OBJ A1	9	10	1	-	0.16
OBJ A2	17	10	1	-	0.16
OBJ A3	24	10	1	-	0.15
OBJ A4	44	10	1	120	3.92
"	"	"	1	150	15.36
"	"	"	1	175	96.08
OBJ A5	62	10	1	120	1.17
"	"	"	1	150	4.39
OBJ A6	97	10	1	120	4.30
"	"	"	1	150	17.14
OBJ A7	108	10	1	120	1.67
"	"	"	1	150	8.52
OBJ A8	151	10	1	120	2.03
"	"	"	1	150	7.75
OBJ A9	166	10	1	120	2.27
"	"	"	1	150	9.00
OBJ A10	204	10	1	120	2.22
"	"	"	1	150	8.69
OBJ A11	215	10	1	120	1.17
"	"	"	"	150	4.41
"	"	"	"	180	17.75
"	"	"	"	200	40.50
OBJ B1	94	10	1	120	fail
"	"	"	"	150	4.23
"	"	"	"	180	24.91
"	"	"	"	200	58.85
OBJ B2	115	10	1	120	fail
"	"	"	"	150	29.41
"	"	"	"	180	74.67
"	"	"	"	200	163.42
OBJ B3	148	10	1	120	fail
"	"	"	"	150	4.00
"	"	"	"	180	22.14
"	"	"	"	200	52.77
OBJ B4	201	10	1	150	4.37
OBJ B5	222	10	1	120	fail
"	"	"	"	150	7.55
"	"	"	"	180	31.80
"	"	"	"	180	79.28
OBJ B6	255	10	1	120	fail
"	"	"	"	150	3.92
"	"	"	"	180	21.94
OBJ B7	266	10	1	120	1.25
"	"	"	"	150	4.12
"	"	"	"	180	16.41

FIG. 11.3 – Résultats pour la génération automatique de données de test pour la méthode `fixAfterInsertion` de la classe `TreeMap`

OBJECTIF	Numéro de l'instruction bytecode	borne mémoire	borne des arcs	borne de longueur	temps CPU (sec)
OBJ A1	4	10	-	-	0.00
OBJ A2	24	10	-	-	0.03
OBJ A3	47	10	-	-	0.06
OBJ A4	55	10	-	-	0.03
OBJ A5	66	10	-	-	0.05
OBJ A6	77	10	-	-	0.08
OBJ B1	29	10	-	-	0.03
OBJ B2	52	10	-	-	0.06
OBJ B3	74	10	-	-	0.03
OBJ B4	82	10	-	-	0.05
OBJ B5	92	10	-	-	0.05
OBJ B6	95	10	-	-	0.02

FIG. 11.4 – Résultats pour la génération automatique de données de test pour la méthode `rotateLeft` de la classe `TreeMap`

Bibliographie

[APV07] S. Anand, C. S. Pasareanu, and W. Visser. JPF-SE : A symbolic execution extension to java pathfinder. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4424 of *Lecture Notes in Computer Science*, pages 134–138. Springer, 2007.

[APV09] S. Anand, C. S. Pasareanu, and W. Visser. Symbolic execution with abstraction. *STTT*, 11(1) :53–67, 2009.

[BDHTH+09] B. Botella, M. Delahaye, S. Hong-Tuan-Ha, N. Kosmatov, P. Mouy, M. Roger, and N. Williams. Automating structural testing of c programs : Experience with pathcrawler. In *ICSE Workshop on Automation of Software Test, AST '09.*, pages 70–78, May 2009.

[Ber07] A. Bertolino. Software testing research : Achievements, challenges, dreams. In *Proceedings of Future of Software Engineering (FOSE'07) at 29th International Conference on Software Engineering*, pages 85–103. IEEE Computer Society, 2007.

[BGM06] B. Botella, A. Gotlieb, and C. Michel. Symbolic execution of floating-point computations. *Softw. Test, Verif. Reliab*, 16(2) :97–121, 2006.

[BH08] S. Bardin and P. Herrmann. Structural testing of executables. In *ICST*, pages 22–31. IEEE Computer Society, 2008.

[Bin99] R. V. Binder. *Testing Object-Oriented Systems : Models, Patterns, and Tools*. Object Technology Series. Addison Wesley, 1999.

[CBG07] F. Charreteur, B. Botella, and A. Gotlieb. Modelling dynamic memory management in constraint-based testing. In *TAIC-PART (Testing : Academic and Industrial Conference)*, Windsor, UK, Sep. 2007.

[CBG09] F. Charreteur, B. Botella, and A. Gotlieb. Modelling dynamic memory management in constraint-based testing. *Journal of Systems and Software (JSS)*, 82(11) :1755–1766, 2009.

[CG08] F. Charreteur and A. Gotlieb. Raisonnement à contraintes pour le test de bytecode java. In *quatrièmes Journées Francophones de Programmation par Contraintes(JFPC'08)*, pages 11–20, Nantes, France, Juin 2008.

[CG10] F. Charreteur and A. Gotlieb. Constraint-based test inputs generation for java bytecode, 2010. En cours de soumission.

[CGP+06] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe : automatically generating inputs of death. In *CCS '06 : Proceedings of the*

13th ACM conference on Computer and communications security, pages 322–335, New York, NY, USA, 2006. ACM.

[CGP+08] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE : Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur*, 12(2), 2008.

[CJRZ02] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. Stg : a symbolic test generation tool. In *(Tool paper) Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02). Volume 2280 of LNCS*. Springer-Verlag, 2002.

[DGD07a] T. Denmat, A. Gotlieb, and M. Ducassé. An abstract interpretation based combinator for modelling while loops in constraint programming. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 241–255. Springer, 2007.

[DGD07b] T. Denmat, A. Gotlieb, and M. Ducassé. Improving constraint-based testing with dynamic linear relaxations. In *ISSRE '07 : Proceedings of the The 18th IEEE International Symposium on Software Reliability*, pages 181–190, Washington, DC, USA, 2007. IEEE Computer Society.

[DLR06] X. Deng, J. Lee, and Robby. Bogor/kiasan : A k-bounded symbolic execution for checking strong heap properties of open systems. In *ASE*, pages 157–166. IEEE Computer Society, 2006.

[DM03a] J. Doyle and C. Meudec. Automatic structural coverage testing of java bytecode. In DSSE Technical Report, editor, *AVoCS'03 : Proceedings of the 3rd workshop on Automated Verification of Critical Systems*, Southampton (UK), 2003.

[DM03b] J. Doyle and C. Meudec. Ibis : an interactive bytecode inspection system, using symbolic execution and constraint logic programming. In *PPPJ '03 : Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 55–58, New York, NY, USA, 2003. Computer Science Press, Inc.

[dMB08] L. de Moura and N. Bjørner. Z3 : An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Berlin, 2008. Springer-Verlag.

[DRH07] X. Deng, Robby, and J. Hatcliff. Kiasan/kunit : Automatic test case generation and analysis feedback for open object-oriented systems. *Testing : Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*, pages 3–12, 10-14 Sept. 2007.

[FK96] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1) :63–86, January 1996.

[FM99] S. N. Freund and J. C. Mitchell. A formal framework for the java bytecode language and verifier. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 147–166, 1999.

[GBR98] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *ISSTA*, pages 53–62, 1998.

[GBW06] A. Gotlieb, B. Botella, and M. Watel. Inka : Ten years after the first ideas. In *19th International Conference on Software & Systems Engineering and their Applications (ICSSEA'06)*, Paris, France, Dec. 2006.

[GDB05] A. Gotlieb, T. Denmat, and B. Botella. Constraint-based test data generation in the presence of stack-directed pointers. In David F. Redmiles, Thomas Ellman, and Andrea Zisman, editors, *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, pages 313–316. ACM, 2005.

[GKS05] P. Godefroid, N. Klarlund, and K. Sen. DART : directed automated random testing. *ACM SIGPLAN Notices*, 40(6) :213–223, June 2005.

[God07] P. Godefroid. Compositional dynamic test generation. In Martin Hofmann and Matthias Felleisen, editors, *POPL*, pages 47–54. ACM, 2007.

[Got09] A. Gotlieb. Euclide : A constraint-based testing framework for critical c programs. In *ICST '09 : Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, pages 151–160, Washington, DC, USA, 2009. IEEE Computer Society.

[GPR07] G.Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference java bytecode verifier. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 125–140. Springer, 2007.

[JJ04] C. Jard and T. Jéron. Tgv : theory, principles and algorithms, a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Software Tools for Technology Transfer (STTT)*, 6, October 2004.

[JL07] E. Jaffuel and B. Legeard. LEIRIOS test generator : Automated test generation from B models. In *B 2007 : Formal Specification and Development in B, 7th International Conference of B Users, Besançon, France, January 17-19, 2007, Proceedings*, volume 4355 of *Lecture Notes in Computer Science*, pages 277–280. Springer, 2007.

[Kin76] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7) :385–394, July 1976.

[KPV03] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer, 2003.

[KS98] K.Marriot and P. Stuckey. *Programming with Constraints : An Introduction.* The MIT Press, 1998.

[KS05] S. Khurshid and Y. L. Suen. Generalizing symbolic execution to library classes. In Michael D. Ernst and Thomas P. Jensen, editors, *Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'05, Lisbon, Portugal, September 5-6, 2005*, pages 103–110. ACM, 2005.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison Wesley Longman, Inc., second edition, April 1999.

[MA00] B. Marre and A. Arnould. Test sequences generation from LUSTRE descriptions : GATeL. In *ASE*, page 229, 2000.

[MLK04] R. A. Müller, C. Lembeck, and H. Kuchen. A symbolic java virtual machine for test case generation. In M. H. Hamza, editor, *IASTED Conf. on Software Engineering*, pages 365–371. IASTED/ACTA Press, 2004.

[MMWG08] P. Mouy, B. Marre, N. Williams, and P. Le Gall. Generation of all-paths unit test with function calls. In *ICST*, pages 32–41. IEEE Computer Society, 2008.

[MS01] J. D. McGregor and D. A. Sykes. *A Practical Guide to Testing Object-Oriented Software*. Addison-Wesley Object Technology Series. Addison-Wesley Professional, 2001.

[PMB+08] C. S. Pasareanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. R. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In Barbara G. Ryder and Andreas Zeller, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, pages 15–26. ACM, 2008.

[Qia99] Zhenyu Qian. A formal specification of java virtual machine instructions for objects, methods and subrountines. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 271–312. Springer, 1999.

[RdBJ00] V. Rusu, L. du Bousquet, and T. Jéron. An approach to symbolic test generation. In *International Conference on Integrating Formal Methods (IFM'00)*, LNCS 1945, pages 338–357. Springer Verlag, November 2000.

[SA06] K. Sen and G. Agha. Cute and jcute : Concolic unit testing and explicit path model-checking tools. In *18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423. Springer, 2006. (Tool Paper).

[SH99] B. Schätz and F. Huber. Integrating formal description techniques. In Jeanette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99—Formal Methods, Volume II*, volume 1709 of *Lecture Notes in Computer Science*, pages 1206–1225. Springer, 1999.

[Siv04] I. A. Siveroni. Operational semantics of the java card virtual machine. *Journal of Logic and Algebraic Programming*, 58(1–2) :3–25, 2004.

[SMA05] K. Sen, D. Marinov, and G. Agha. Cute : a concolic unit testing engine for c. In *ESEC/FSE-13 : Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272, New York, NY, USA, 2005. ACM Press.

[Sta09] M. Staats. Towards a framework for generating tests to satisfy complex code coverage in java pathfinder. In *Proceedings of the First NASA Formal Methods Symposium (NFM'09)*, pages 116–120. NASA Ames Research Center, 2009.

[TdH08] N. Tillmann and J. de Halleux. Pex-white box test generation for.NET. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008.

[TS06] N. Tillmann and W. Schulte. Unit tests reloaded : Parameterized unit testing with symbolic execution. Technical Report MSR-TR-2005-153, Microsoft Research (MSR), March 2006.

[UPL06] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing, rapport technique. 2006.

[VPK04] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with java pathfinder. In *ISSTA '04 : Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–107, New York, NY, USA, 2004. ACM Press.

[VPP06] W. Visser, C. S. Pasareanu, and R. Pelánek. Test input generation for java containers using state matching. In *ISSTA '06 : Proceedings of the 2006 international symposium on Software testing and analysis*, pages 37–48, New York, NY, USA, 2006. ACM Press.

[Wey79] E. J. Weyuker. Translatability and decidability questions for restricted classes of program schemas. *SIAM Journal on Computing*, 8(4) :587–598, November 1979.

[WLPO00] G. Wimmel, H. Lötzbeyer, A. Pretschner, and O.Slotosch. Specification based test sequence generation with propositional logic. *Softw. Test, Verif. Reliab*, 10(4) :229–248, 2000.

[WMMR05] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler : Automatic generation of path tests by combining static and dynamic analysis. In Mario Dal Cin, Mohamed Kaâniche, and András Pataricza, editors, *Dependable Computing - EDCC-5, 5th European Dependable Computing Conference, Budapest, Hungary, April 20-22, 2005, Proceedings*, volume 3463 of *Lecture Notes in Computer Science*, pages 281–292. Springer, 2005.

[WN04] Mouy P. Williams N., Marre B. On-the-fly generation of k-path tests for c functions. In *Automated Software Engineering, 2004. Proceedings. 19th International (ASE'04)*, pages 290–297, 2004.

[YPA06] J. Yuan, C. Pixley, and A. Aziz. *Constraint-Based Verification*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

Table des figures

1.1 Un programme, son graphe de flot de contrôle et une exécution symbolique de l'un de ses chemins . . . 19
1.2 Méthode de parcours des chemins pour l'exécution symbolique dynamique . 23
1.3 Tableau comparatif des différentes approches de génération de test basées sur le code (1/2) . . . 31
1.4 Tableau comparatif des différentes approches de génération de test basées sur le code (2/2) . . . 32

2.1 Illustration de la propagation de contraintes . . . 36
2.2 Un exemple de traitement possible de la contrainte $X > Y$ . . . 36
2.3 Les différents états possibles d'une contrainte . . . 37
2.4 Illustration de l'énumération . . . 38

3.1 Extrait d'un état de la mémoire avant le test de la condition `(p.next==p)` tel que la condition soit satisfaite . . . 43
3.2 Mémoires générées et contraintes créées au cours de l'interprétation de la méthode `deplaceY` . . . 49
3.3 Exemple d'indétermination de la forme de la mémoire . . . 51

5.1 Traitement de la contrainte d'égalité d'une référence et d'une adresse . . . 63
5.2 Traitement de la contrainte d'inégalité d'une référence et d'une adresse . . . 63
5.3 Traitement de la contrainte d'égalité de références . . . 64
5.4 Traitement de la contrainte d'égalité de types . . . 64
5.5 Traitement de la contrainte d'inégalité de références . . . 65
5.6 Génération des mémoires et création des contraintes selon l'instruction interprétée . . . 66
5.7 Traitement de la contrainte de vérification de type . . . 69
5.8 Traitement de la contrainte *new* . . . 70
5.9 Traitement de la contrainte d'égalité d'instances . . . 72
5.10 Traitement de la contrainte $getfield$ . . . 73
5.11 Traitement de la contrainte $putfield$ . . . 75
5.12 Traitement de la contrainte d'appel de méthode virtuelle . . . 77

6.1 Regroupement par blocs des instructions du GFC . . . 80
6.2 Exemple de graphe de flot de contrôle avec boucle . . . 82
6.3 Illustration de la notion de niveau pour les objets lors de l'énumération . . . 83

7.1 Tableau comparatif des différentes approches de génération de test basées sur le code, avec notre approche (1/2) . . . 88

7.2 Tableau comparatif des différentes approches de génération de test basées sur le code, avec notre approche (2/2) 89

8.1 Architecture de l'outil JAUT 93

9.1 Temps et couverture avec JCute, Pex et JAUT 96
9.2 Variation du temps de génération des données de test pour la méthode `foo` en fonction de `a` 101
9.3 Forme de la mémoire en entrée générée par JAUT 104

11.1 Résultats pour la génération automatique de données de test pour la méthode `deleteEntry` de la classe `TreeMap` 129
11.2 Résultats pour la génération automatique de données de test pour la méthode `fixAfterDeletion` de la classe `TreeMap` 130
11.3 Résultats pour la génération automatique de données de test pour la méthode `fixAfterInsertion` de la classe `TreeMap` 131
11.4 Résultats pour la génération automatique de données de test pour la méthode `rotateLeft` de la classe `TreeMap` 132

Listings

3.1 Code Java de la classe `Coord` qui modélise des coordonnées 44
3.2 Code machine de la méthode `deplaceY` de la classe `Coord` 45
9.1 La méthode `trityp` 99
9.2 Programme avec une boucle dont la condition d'arrêt dépend d'un paramètre 100
9.3 Extrait de classes sous test représentant des arbres colorés 103
9.4 Code de `josephus_m` 105
11.1 Code Java de la classe `Node` 119
11.2 Code Java de la classe `DoublyLinkedList` 120
11.3 Code Java de la classe `Entry` 123
11.4 Code Java de la classe `RedBlackTree` 124

Résumé

La vérification des programmes est indispensable pour maintenir un certain niveau de qualité et de fiabilité. Le test est à ce jour le moyen de vérification des logiciels le plus utilisé dans l'industrie. La programmation par contraintes est vue comme un moyen efficace pour automatiser la génération de données de test. Dans cette thèse nous proposons une modélisation par contraintes de la sémantique du bytecode Java, ainsi qu'une méthode, basée sur cette modélisation, pour générer automatiquement des données de test. Notre modèle à contraintes de la sémantique d'un programme en bytecode Java permet de faire des déductions efficaces, y compris en présence de structures de données complexes ou d'héritage. En particulier, l'utilisation de variables de type permet de prendre en compte l'héritage et les appels de méthodes polymorphes. Notre méthode de génération de données de test exploite le modèle à contraintes pour couvrir des instructions particulières du programme sous test. Elle se base sur un parcours en arrière du graphe de flot de contrôle pour énumérer des chemins menant aux instructions cibles. Elle est en particulier adaptée à la couverture d'instructions non couvertes par les autres méthodes de génération de données de test. Enfin cette méthode est mise en application dans un prototype, JAUT (Java Automatic Unit Testing). Les expériences montrent que le prototype permet d'augmenter la couverture des instructions obtenue avec les autres outils disponibles.