Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Génération de contraintes pour le test de programmes manipulant une base de donnée

Hardenne, Romain

Award date:
2015

Awarding institution:
Universite de Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 20. Apr. 2024

Université de Namur
Faculté d'informatique
Année académique 2014–2015

Génération de contraintes pour le test de programmes manipulant une base de données

Romain Hardenne

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Wim Vanhoof

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Résumé

Dans le cycle de vie d'un projet informatique, la phase de test occupe une place importante. Cette étape cruciale peut être réalisée de différentes manières et à l'aide de nombreuses techniques. L'exécution symbolique fait partie de celles-ci et permet de générer automatiquement des données de tests. Les travaux de M. Marcozzi à propos de l'exécution symbolique y ont ajouté la dimension de la gestion des manipulations d'une base de données. Néanmoins, l'implémentation de cet algorithme n'est pas complète : seul un ensemble réduit des instructions disponibles dans le langage Java est supporté lors de l'analyse. Ce travail a pour objectif d'étendre cet ensemble d'instructions en intégrant cet algorithme à un programme bien connu réalisant déjà de l'exécution symbolique. Le choix de ce programme s'est porté sur l'outil JPF qui possède les qualités requises pour être l'objet de cette intégration. Bien que cet objectif n'ait pas été entièrement rempli, ce mémoire ouvre les voies et donne des pistes pour la résolution des problèmes rencontrés.

Mots clés : *Tests, Génération de Données de Test, Exécution Symbolique, JPF, SPF, Contraintes, Base de Données, SQL*

Abstract

Test phase is a significant part of an IT project lifecycle. Many different techniques are used to achieve this step. Symbolic execution one of them and enables the generation of test inputs in order to test a program. M. Marcozzi's researches about symbolic execution add the support of database programs. However the algorithm designed by M. Marcozzi is not complete since it only handles a subset of the Java instructions. The purpose of this paper is the integration of M. Marcozzi's algorithm into an existing tool that fully performs symbolic execution. The selected tool is JPF as it possesses all the required defined qualities. Although the objective is not completely fulfilled, this paper proposes tracks to achieve the initial goal and to resolve the encountered problems.

Keywords : *Sofware Testing, Test Case Generation, Symbolic Execution, JPF, SPF, Constraints, Database, SQL*

En préambule à ce mémoire, je souhaite remercier mon promoteur, Monsieur Wim Vanhoof pour l'intérêt qu'il a témoigné à mon projet ainsi que pour ses remarques avisées.

Je souhaite également manifester ma gratitude aux différentes personnes qui, tout au long de l'élaboration de ce mémoire, m'ont conseillé et prêté une oreille attentive : mes collègues, mes amis et ma famille.

J'aimerais spécialement remercier mon ami de longue date, Régis Grandgagnage, pour sa relecture et ses critiques constructives.

Enfin, j'ai une pensée toute particulière pour ma compagne, Claire Van Loo, qui m'a soutenu et encouragé tout au long de mon parcours.

A vous tous : merci.

Table des matières

1 Introduction **6**

2 Contexte **8**

2.1 Tester 8

2.1.1 Les niveaux de test 8

2.1.2 White-box testing VS black-box testing 10

2.1.3 Couverture de code 11

2.2 Génération de cas de tests 12

2.3 Exécution symbolique 13

2.3.1 Principe 14

2.3.2 Limitations 15

2.3.3 Exécution concolique 18

2.4 Base de données et exécution symbolique 18

2.5 Application de la nouvelle approche 19

3 Choix de l'outil : JPF **21**

3.1 Qualités et choix de l'outil 21

3.2 JPF & SPF 22

3.3 Klee 23

3.4 Pex 23

3.5 DART 23

3.6 Cute & JCute 23

4 JPF & SPF **25**

4.1 Description de JPF 25

4.2 Utilisation 26
4.3 Principe - fonctionnement 27
4.4 Configuration de JPF 28

5 Réalisation **30**
5.1 La méthode de travail 30
5.2 Préliminaires 30
5.3 JPF 31
5.3.1 JPF-core 31
5.3.2 JPF-symbc 31
5.3.3 Configuration de JPF 31
5.3.4 Exécution de JPF 32
5.4 Le code de M. Marcozzi 32
5.4.1 Installation et configuration 34
5.4.2 Extraction de la configuration 34
5.4.3 Modification du parser 35
5.5 Intégration 36
5.5.1 Création du nouveau listener 36
5.5.2 Intégration du parsing de la structure de base de données 36
5.5.3 Création d'un POC 37
5.5.4 JPF-nhandler 38
5.6 Pistes et améliorations 38
5.6.1 JPF 38
5.6.2 Code de M. Marcozzi 39

6 Conclusion **40**

Chapitre 1

Introduction

Ces dernières années, la technologie a pris de plus en plus d'ampleur dans la société, dans les entreprises et dans la vie de tous les jours. Les projets informatiques se multiplient et l'expérience en matière de gestion et de conduite de ces projets s'affine dans un effort de délivrer des produits de qualité.

Une des pratiques mises à l'honneur est la valorisation de la phase de tests des applications et programmes, et plus particulièrement des tests bas niveau. Cette tendance a fait naitre des approches nouvelles en matière de testing, et en a également remis des plus anciennes au goût du jour. Parmi celles-ci, l'exécution symbolique permet d'achever des objectifs spécifiques en matière de couverture de code.

Ce mémoire prend comme point de départ les travaux de M. Marcozzi concernant l'exécution symbolique dans le cadre du test d'applications manipulant une base de données : une nouvelle approche prometteuse permettant d'introduire la dimension des bases de données relationnelles dans l'univers de l'exécution symbolique.

Pour des raisons pratiques, l'ensemble des instructions supportées par l'algorithme de M. Marcozzi a été volontairement réduit. Parallèlement, il existe de nombreux programmes réalisant de l'exécution symbolique qui supportent la totalité des instructions du langage Java. C'est à partir de ces observations que découlent les objectifs de ce mémoire, à savoir la sélection

d'un programme existant et l'intégration de l'algorithme de M. Marcozzi dans celui-ci.

Dans un premier temps, ce document introduit le contexte dans lequel il s'inscrit et aborde les problématiques qui y seront traitées. Il développe également les concepts pertinents pour la compréhension tel que l'exécution symbolique et résume les travaux sur lesquels il est basé.

Le chapitre suivant décrit les raisons pour lesquelles l'outil JPF a été choisi comme programme pour l'intégration du code de M. Marcozzi. Les critères de sélection du programme idéal sont définis, et les candidats au titre sont brièvement présentés : les fonctionnalités offertes, leurs forces et faiblesses.

Le chapitre 4 explore JPF plus en détails au travers de sa philosophie particulière. Son fonctionnement, ses principes et ses éléments clés y sont expliqués de manière précise.

Enfin, le dernier chapitre reprend l'ensemble des étapes et avancées réalisées durant l'intégration du code de M. Marcozzi dans JPF : la création de nouveaux composants, les améliorations apportées ainsi que les obstacles rencontrés.

Chapitre 2

Contexte

2.1 Tester

Aujourd'hui, les systèmes informatiques sont de plus en plus sophistiqués, de taille et de complexité croissantes[34]. Cependant, la qualité de ces projets informatiques doit être garantie au risque que ceux-ci ne deviennent des échecs couteux. La vérification de cet attribut qu'est la qualité se concrétise par différentes pratiques lors de ce qui est communément appelé la phase de test. Cette étape importante dans le cycle de développement logiciel représente généralement près de 50% du cout et des ressources alloués au développement[41].

2.1.1 Les niveaux de test

Tester un programme, une interface utilisateur ou un projet signifie valider les spécifications énoncées lors de la phase d'analyse. Un test est la vérification du contrat entre un objet et son environnement. Il en existe deux grandes catégories : les tests manuels et les tests automatiques.

Les tests manuels sont généralement des tests exploratoires réalisés par un humain navigant à travers les écrans d'une application. Ce genre de test est relativement lent et chronophage, et se concentre sur la détection de problèmes de design d'interface ou d'utilisabilité[27].

Les tests automatiques quant à eux sont accomplis par une machine qui,

selon un contexte donné, vérifie que l'artefact testé s'exécute de manière attendue et produit les résultats escomptés. Il existe différents niveaux de tests automatiques selon le but et l'abstraction désirée. La figure 2.1 représente la pyramide simplifiée des tests automatiques[31], découpée en trois principaux étages :

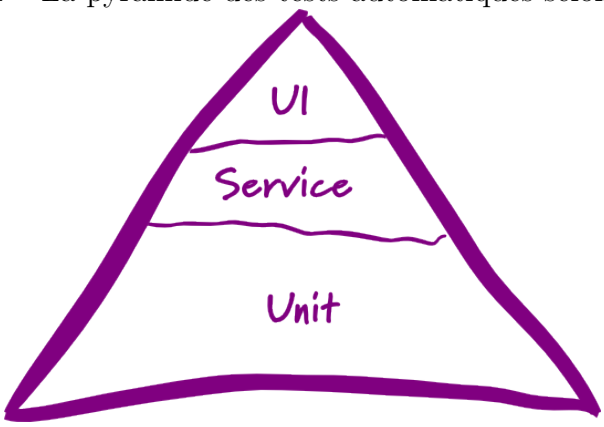
— UI : niveau interface utilisateur, voit le système comme une *black box*. Les tests automatisés d'UI vont se concentrer sur un ensemble restreint de fonctionnalités que l'utilisateur doit pouvoir réaliser dans un certain contexte et vérifier que ces tâches peuvent être accomplies correctement. Un utilisateur sans compétence particulière en informatique pourra mettre ces tests en place en s'aidant par exemple d'outils tels que *Selenium* pour les applications web[12]. Un des gros désavantages de ce niveau est qu'il est difficile de maintenir ces tests qui sont intimement liés à la partie la moins pérenne dans un système informatique[48, 30].

— Service : niveau intégration des composants. Un système est composé de modules ayant chacun un ensemble restreint de responsabilités et interagissant entre eux. Cet étage de la pyramide vérifie que ces blocs sont correctement agencés et sont capables de communiquer. Ces tests sont généralement de haut niveau, faciles à écrire mais parfois difficiles à maintenir tout au long du développement de l'application.

— Unit : niveau composant. Généralement écrit par le programmeur, ce genre de test vérifie une unité spécifique et de ce fait est facile à écrire et à maintenir. Cet étage est d'une importance capitale : il représente la base de la pyramide et se doit d'être solide pour supporter les étages supérieurs. Un test unitaire possède un certain nombre de qualités tels que la complétude, la rapidité ou encore le déterminisme[36].

A travers ce mémoire, nous allons principalement porter notre attention sur la base de la pyramide et plus précisément, sur l'approche *génération de tests automatiques*.

FIGURE 2.1 – La pyramide des tests automatiques selon M. Fowler



2.1.2 White-box testing VS black-box testing

Il existe deux grandes méthodes lorsqu'il s'agit d'aborder la manière de concevoir un test : l'approche black-box et l'approche white-box.

La première méthode considère l'artefact testé comme une boite noire, c'est-à-dire que la personne réalisant le test n'a aucune connaissance de la structure interne, de l'implémentation ou du fonctionnement du programme. Par contre, le testeur connait le résultat auquel le programme doit parvenir. Sur base des spécifications et des exigences, il pourra donc déterminer un ensemble de données en entrée du programme devant aboutir à un résultat attendu. Ce type de méthode peut être utilisé à tous les niveaux du testing (unitaire, intégration, acceptation) du fait de son haut niveau d'abstraction.

A l'inverse, l'approche white-box (boite blanche, aussi appelée boite transparente) requiert une connaissance approfondie des rouages du programme afin de concevoir les tests. Là où l'approche black-box se focalise sur les exigences et les spécifications, l'approche white-box se concentre sur le code du programme. Le but est de créer des tests qui aboutiront à une couverture de code élevée. Traditionnellement, ce type d'approche est réservé aux tests unitaires, mais il peut également être utilisé dans les tests d'intégration et système[44].

Les avantages du white-box testing sont :

— Détection d'erreurs dans les cas limites

— Détection de code inutilisé ou mort

— Force le développeur à réfléchir sur les détails d'implémentation (car documentation requise pour les tests)

— Facile à automatiser

Les inconvénients du white-box testing sont :

— La connaissance de l'implémentation est requise (le testeur ne peut pas être *n'importe qui*)

— Parfois complexe et couteux à créer et exécuter

— Il est parfois inenvisageable de tester chaque condition, et certaines partie du code resteront non testées

— Centré sur le logiciel tel qu'il existe (possibilité de passer à côté de fonctionnalités clés)

D'autres méthodes dérivées de ces deux premières approches ont également vu le jour, comme la méthode *gray-box* qui en mélange les concepts.

Dans la suite de ce document, l'approche white-box sera principalement utilisée.

2.1.3 Couverture de code

Une des dernières tâches à réaliser lors de la phase de test est de vérifier que les objectifs sont atteints, c'est-à-dire la validation de l'objet testé (une méthode, une fonctionnalité, un programme) comme étant conforme aux exigences. Il peut être difficile d'estimer que l'objet testé est correct et qu'il le restera peu importe les variables ayant un impact sur celui-ci. C'est là qu'intervient la notion de couverture de code.

La couverture de code est une des premières mesures quand il est question de décrire le degré avec lequel le code d'un programme est testé : plus elle est élevée, plus le programme est testé et moins il est susceptible de contenir des erreurs. C'est pour cette raison que la couverture de code est souvent utilisée comme critère de qualité et beaucoup de projets définissent un pourcentage de couverture comme exigence non fonctionnelle.

Il existe différents types de couverture dont les principaux sont [41]

— couverture de fonctions : mesurée grâce au nombre de fonctions appelées durant les tests

— couverture d'instructions : mesurée grâce au nombre d'instructions exécutées durant les tests

— couverture de branches [1] : mesurée grâce au nombre de branches empruntées durant les tests

— couverture de condition : mesurée grâce au nombre de sous-expressions booléennes évaluées à *true* et *false*

Dans la suite de ce document, un type de couverture de code sera particulièrement intéressant : la couverture des chemins d'exécution, mesurée par le nombre de chemins d'exécutions empruntés.

2.2 Génération de cas de tests

Une des difficultés rencontrées lors de l'écriture de tests unitaires est la complétude de ceux-ci. Un test unitaire valide un bout de programme avec un jeu de données en entrée. Ces données peuvent être artificiellement créées par le programmeur pour le besoin du test ou récupérées depuis l'environnement de production. Le problème avec ces pratiques est que l'exhaustivité du jeu de données n'est pas garantie : il est possible qu'un cas limite n'ait pas été repris et qu'une erreur dans le programme soit donc passée inaperçue.

La génération de cas de tests est une démarche permettant de définir un ensemble de combinaisons de données en entrées permettant de valider l'ensemble du programme testé. Il existe différentes approches selon l'objectif et les résultats voulus[28] :

— Génération de données de tests aléatoire

— Génération de données de tests orientée objectif

— Génération de données de tests intelligente

1. une branche est une des séquences d'instructions empruntée dans une structure de contrôle. Par exemple, la structure *if-else* possède deux branches : une pour le *if* et une pour le *else*.

— Génération de données de tests par chemin

Des quatre approches, les techniques basées sur la génération de données de tests par chemin sont les plus communes et considérées comme les plus prometteuses[24, 43, 47].

Bien qu'intéressantes, ces approches ne sont cependant pas la irréprochable. Les outils et techniques sont encore immatures et souffrent de nombreuses lacunes. En effet, il existe actuellement des obstacles freinant l'utilisation de telles pratiques dans le monde de l'industrie telles que l'efficacité (qui requiert énormément de ressources) et la complexité des programmes à tester[37]. Néanmoins, des avancées sont continuellement observées et certaines niches commencent à utiliser ces outils[19].

2.3 Exécution symbolique

L'exécution symbolique est une technique d'analyse de programmes. L'idée principale derrière celle-ci n'est pas récente et a été introduite dans les années 1970[16, 23, 38]. Néanmoins, elle a été mise de côté pour des raisons pratiques et ce n'est que récemment qu'elle refait surface grâce aux avancées dans différents domaines tels que la satisfaction de contraintes[25], l'utilisation d'approches dynamiques et extensibles (ex : exécution concolique)[18, 32] ainsi que l'explosion en matière de puissance de calcul et de ressources mémoire[40].

L'exécution symbolique permet de réaliser différents objectifs selon l'utilisation comme :

— la génération de données de tests pour atteindre une couverture maximale

— le test des cas limites

— la détection de problèmes de concurrence[13]

2.3.1 Principe

Le principe de l'exécution symbolique est d'utiliser des valeurs symboliques au lieu de données concrètes. Le programme analysé est considéré comme une suite d'expressions booléennes déterminant un ou autre chemin d'exécution. De cette façon, il peut être modélisé par un arbre appelé *arbre d'exécution* dont chaque branche représente un chemin d'exécution possible. Aussi, l'exécution symbolique maintient un état symbolique des variables et chacune de celles-ci est associée à une valeur symbolique. Le résultat du programme est exprimé comme une fonction de ces valeurs. Le programme est lu et l'arbre construit au fur et à mesure en explorant les chemins et en réalisant du *backtracking*[2]. Le parcours depuis la racine de l'arbre jusqu'à une feuille représente un chemin d'exécution possible, soit la séquence d'instructions sous forme de formules sans quantificateur du premier ordre. Une fois qu'un chemin est exploré (la fin du programme est atteinte ou une condition d'arrêt (ex : un *timeout*) est vérifiée), le système de contraintes est résolu afin de générer des valeurs concrètes à associer aux données en entrée pour parcourir ce chemin.

Illustrons l'exécution avec l'exemple de la figure 2.2. La méthode testée se nomme *testme*, et le point d'entrée est la méthode *main*.

1. lignes 17 et 18 : initialisation des données en entrées avec des valeurs symboliques. A ce moment, l'état symbolique du programme vaut $\{x \mapsto \alpha, y \mapsto \beta\}$.
2. ligne 19 : appel de la méthode à tester avec les paramètres initialisés précédemment
3. ligne 6 : la méthode *twice* est exprimée comme une fonction $f(\gamma) = 2 * \gamma$
4. ligne 6 : mise à jour de l'état symbolique du programme avec la nouvelle variable z : $\{x \mapsto \alpha, y \mapsto \beta, z \mapsto 2 * \beta\}$
5. lignes 7 et 8 : évaluation des conditions. Pour chaque condition $if(\epsilon)then[...]else[...]$, une intersection est créée : une branche de l'arbre

2. Le backtracking est un algorithme permettant de revenir en arrière. En l'occurrence, une fois un chemin exploré, le programme retourne sur ses pas afin d'emprunter un nouveau chemin, permettant ainsi le parcours de tous les chemins praticables possibles

pour quand la condition est vérifiée, et une autre branche pour quand elle ne l'est pas

6. fin du chemin : une fin de méthode est atteinte ou une erreur est rencontrée (une exception est remontée ou une assertion échoue). Cela signifie que le chemin courant est terminé, et que le système de contraintes obtenu en le parcourant est prêt à être résolu. La résolution du système de contraintes permet de vérifier que le chemin peut être satisfait par une combinaison de données en entrée, et calcule cette combinaison
7. backtracking : une fois la résolution du système de contraintes pour le chemin courant réalisé, l'exécution symbolique reprend à un croisement choisi et emprunte un nouveau chemin
8. fin de l'exécution symbolique : une fois tous les chemins parcourus ou qu'une autre condition d'arrêt est rencontrée, l'exécution symbolique prend fin. Sont alors disponibles des paires composées d'une combinaison de données en entrées et de résultats.

Pour la méthode *testme()* de la figure 2.2, le nombre de chemins possibles est 3 et ceux-ci sont représentés dans l'arbre d'exécutions de la figure 2.3. Ces chemins peuvent être explorés en passant en entrée de la méthode les combinaisons :

— {x = 0, y = 1}
— {x = 2, y = 1}
— {x = 30, y =15}

Un des buts de l'exécution symbolique est de générer ces combinaisons de données en entrée pour explorer tous les chemins d'exécutions possibles et parvenir à une couverture de code maximale.

2.3.2 Limitations

Il existe des limitations à l'exécution symbolique. En effet, un des principaux désavantages de la méthode est qu'elle ne peut produire de résultat que si le système de contraintes peut être résolu (de manière efficace) par un

FIGURE 2.2 – Exemple simple illustrant l'exécution symbolique [20]

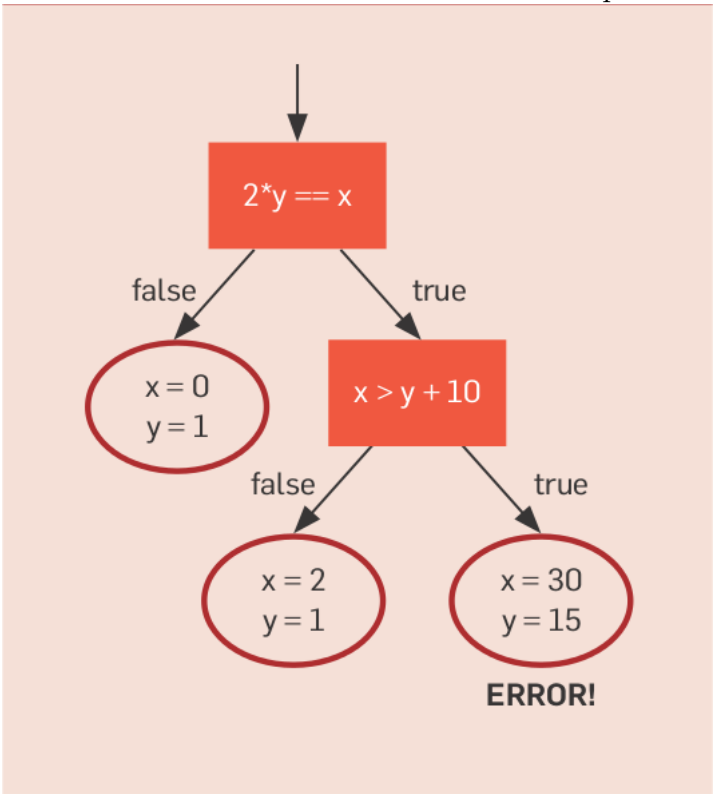
```
1  int twice (int v) {
2      return 2*v;
3  }
4
5  void testme (int x, int y) {
6      z = twice (y);
7      if (z == x) {
8          if (x > y+10) {
9              ERROR;
10         }
11     }
12 }
13
14 /* simple driver exercising testme ()
15  * with sym inputs */
16 int main() {
17     x = sym_input();
18     y = sym_input();
19     testme (x, y);
20     return 0;
21 }
```

solutionneur. Ce genre de chemin symbolique représentant un chemin d'exécution possible est typiquement produit par une boucle dont la condition de sortie est elle-même symbolique, résultant en une infinité de chemins possibles. Ce cas peut cependant être contourné en spécifiant un *timeout*[3] ou un nombre maximum d'itérations.

Un autre problème de taille rencontré est *l'explosion de chemins*. Le nombre de chemins d'exécutions possibles croit de manière exponentielle, pour chaque choix à réaliser, chaque variable à prendre en compte, chaque expression (boucles, récursion, ...). Il convient donc de mettre en place une stratégie de recherche afin de limiter le temps consacré à l'exploration. Il existe différentes stratégies[35] :

3. Dépassement de délai fixé.

FIGURE 2.3 – Arbre d'exécution de l'exemple 2.2



— basique : en profondeur ou en largeur d'abord, ces stratégies sont peu efficaces et peuvent être facilement bloquées

— *randomness* : choix au hasard de chemins. Cette stratégie est efficace, mais possède le désavantage majeur de ne pas être reproductible

— heuristique orientée couverture : l'idée est d'essayer de visiter des expressions qui n'ont pas encore été vues. Cette stratégie a pour avantage de tenter d'explorer tous les chemins d'exécutions possibles du programme.

— hybride : une combinaison de stratégies vues précédemment, comme *randomness* et l'heuristique orientée couverture qui est utilisée par KLEE [4][17]

Une troisième limitation survient lorsqu'une librairie externe ou du code

4. Klee est un outil permettant de réaliser de l'exécution symbolique ; voir infra.

système est appelé. Une solution à ce problème est de récupérer le code source de la librairie et de lui appliquer l'exécution symbolique. Néanmoins, cette pratique n'est pas toujours possible ni désirable : il peut être difficile de se procurer le code ou celui-ci peut être extrêmement complexe à analyser. Une autre solution serait de créer des modèles de codes simulant ces appels, mais cette option représente une quantité de travail non négligeable.

2.3.3 Exécution concolique

L'exécution concolique [5] est une méthode moderne qui allie l'exécution concrète à l'exécution symbolique. L'idée derrière cette combinaison est d'améliorer les performances et de combler certaines lacunes de l'exécution symbolique en y incorporant en parallèle une exécution avec des données concrètes[50].

2.4 Base de données et exécution symbolique

La plupart des logiciels manipulant de l'information interagissent à un moment ou à un autre avec une base de données [6], que ce soit pour y lire ou y écrire. De manière générale, tester un programme en prenant en compte la structure d'une base de données et la complexité des requêtes SQL [7] est non triviale. C'est encore plus vrai quand on ajoute la dimension de l'exécution symbolique.

D'un point de vue théorique, il n'est pas possible de calculer la satisfiabilité d'une requête SQL[14]. De ce fait, il n'est pas non plus possible de générer des données en entrée pour un programme possédant dans ses chemins la résolution d'une requête SQL[51].

D'un point de vue pratique, une base de données peut être vue comme un conteneur d'informations que le programme va manipuler. Ce conteneur est structuré à l'aide d'un schéma de base de données défini grâce au code

5. l'appellation provient du mot *concolic* en anglais, qui est la contraction des mots *concrete* et *symbolic*

6. Nous nous limiterons dans ce document aux bases de données relationnelles

7. Le SQL (pour *Structured Query Language*) est un langage de programmation utilisé pour interagir avec une base de données.

SQL DDL[8] : il contient la définition des tables et des contraintes d'intégrité de données (*primary key*, *check*, *foreign key*, ...). Ces éléments peuvent être traduits en des contraintes plus ou moins complexes, et les données peuvent être représentées par des valeurs symboliques[39]. Il reste à résoudre la partie interaction entre le programme et la base de données qui se concrétise par des requêtes SQL.

Deux approches différentes ont été déjà étudiées et mises en pratique concernant la gestion de requêtes SQL dans un programme[22, 42, 29].

La première approche introduit de nouvelles variables natives dans le code testé qui représentent le contenu de la base de données. Aussi, les requêtes SQL sont elles-même remplacées par du code qui agira sur ces nouvelles variables. Ces deux manipulations permettent d'obtenir du code normalisé qui peut être l'objet d'une analyse via exécution symbolique.

La deuxième approche quant à elle considère le résultat des requêtes dans le code testé comme de nouvelles variables en entrées qui peuvent être utilisées dans le système de contraintes.

Une troisième approche appelée *relational symbolic execution* permet de traduire directement les requêtes SQL en contraintes[39]. Celles-ci sont ajoutées au système qui pourra être résolu de manière classique.

2.5 Application de la nouvelle approche

Cette nouvelle approche permet de traduire les interactions avec la base de données directement en contraintes. Cette pratique a pour avantage de ne pas devoir modifier le code natif et peut gérer n'importe quelle requête DML[9].

Deux étapes composent cette technique :

1. Traitement du DDL : la structure de la base de données est lue, analysée et chaque table est convertie en une relation mathématique

8. Du code SQL DDL (pour *Data Definition Language*) est le code de définition de structure de données.

9. Le langage DML (pour *Data Manipulation Language*) est un sous-ensemble des commandes (lecture et écriture) permettant d'interagir avec une base de données.

2. Traitement des requêtes DML : chaque requête est modélisée en une opération relationnelle à partir des relations mathématiques issues de l'analyse du DDL

L'algorithme introduit donc de nouvelles contraintes au fur et à mesure que des requêtes sont rencontrées lors de l'exécution symbolique.

Cette approche prometteuse est néanmoins encore limitée. En effet, l'algorithme proposé par M. Marcozzi[39] supporte l'analyse de DDL, de code Java et SQL relativement simple.

Chapitre 3

Choix de l'outil : JPF

3.1 Qualités et choix de l'outil

La première partie de ce travail consiste à choisir l'outil adéquat pour intégrer l'algorithme de M. Marcozzi. Cet outil doit posséder certaines qualités :

— Java : le code de base est écrit en Java et lit du Java basique. Afin de faciliter la tâche d'intégration de l'algorithme de M. Marcozzi, l'outil devra pouvoir lire du code Java et proposer un moyen d'interagir avec du code écrit dans ce langage.

— Modulaire : l'objectif du travail étant d'intégrer une nouvelle fonctionnalité dans un programme existant, l'outil choisi se devra d'être facilement extensible.

— Adapté : l'outil doit offrir les fonctionnalités de base permettant de réaliser de l'exécution symbolique.

— Documenté : la documentation de l'outil devra comprendre des explications concernant son utilisation et son extension.

— Open source et sous licence libre : l'outil pourra être récupéré et modifié à des fins de recherche.

— Mature (optionnel) : l'outil aura fait ses preuves dans le monde académique et/ou industriel.

La recherche a été réalisée sur base des outils et programmes les plus connus et utilisés.

Le choix s'est porté sur JPF & SPF étant donné qu'il possède toutes les qualités requises énoncées ci-dessus.

3.2 JPF & SPF

JPF (*Java Path Finder*) est un outil extrêmement customisable de vérification de *Java bytecode* développé par le *NASA Ames Research Center*[8]. Il est *open source* depuis 2005 sous licence NOSA 1.3[10].

JPF est un vérificateur de modèle relativement général : il offre des possibilités de configuration intéressantes permettant une optimisation selon le but recherché, comme par exemple la stratégie de recherche et de choix de chemins.

SPF (*Symbolic Path Finder* fait partie du projet JPF et permet de réaliser de l'exécution symbolique. Etant donné que SPF travaille avec du *bytecode*, il lui est possible d'analyser du code Java ainsi que n'importe quel autre modèle traduit en *bytecode*, comme des diagrammes UML[19].

Une des grandes forces de JPF est qu'il a été construit pour être hautement modulable et configurable. Cette approche permet de pouvoir greffer au programme principal un ou plusieurs modules appelés extensions. Une extension utilise les fonctionnalités du programme principal pour réaliser certaines opérations et vérifications spécifiques. Parmi les extensions disponibles, on pourra noter :

— jpf-core : le module core indispensable utilisé par les autres extensions

— jpf-symbc : l'extension permettant de réaliser de l'exécution symbolique

— jpf-concolic : l'extension permettant de réaliser de l'exécution concolique

3.3 Klee

Klee[6], le successeur d'EXE[21], est un outil permettant de réaliser de l'exécution symbolique avec comme résultat une grande couverture de test et la capacité à trouver des erreurs. Il est basé sur l'infrastructure du compilateur LLVM[7] et est donc écrit en *C++*. Il est notamment utilisé pour tester les utilitaires GNU COREUTILS[17] où il a fait ses preuves en achevant une couverture de plus de 94% et en trouvant de nombreux bugs ayant échappé aux programmeurs pendant plus de 15 ans. Depuis 2009, le projet a ouvert son code et est utilisé dans des secteurs très diversifiés tels que les réseaux sans fil[46] et les jeux en ligne[15].

3.4 Pex

Pex[11] est l'outil développé par Microsoft et implémente l'exécution symbolique dynamique. Pex est spécialisé pour le langage *.NET*, mais supporte également d'autres langages comme le *C#*, le *VisualBasic* ou le *F#*.

Pex est disponible sous licence académique et commerciale, et est intégré à la suite *Microsoft Visual Studio 2010* pour le développement de programmes *.NET*.

3.5 DART

DART[33], pour *Directed Automated Random Testing* est l'outil d'exécution symbolique qui cible les programmes écrits en langage de programmation *C*. L'objectif de DART est de détecter les erreurs classiques comme le *crash* de programme, la violation d'assertions ou les boucles infinies. DART n'est actuellement pas disponible au public.

3.6 Cute & JCute

Cute et JCute[49] sont des moteurs de génération de tests unitaires pour le langage *C* et le langage Java, respectivement. Grâce à l'exécution concolique,

l'outil permet de générer une suite de tests unitaires garantissant une haute couverture de code tout en offrant des fonctionnalités comme la gestion du *multi-threading*.

JCute a notamment été exécuté sur certaines librairies de *Java* (*java.util* dans sa version *1.4*) et a permis de détecter et corriger plusieurs bugs.

JCute est disponible gratuitement à des fins de recherche et d'éducation[3].

Chapitre 4

JPF & SPF

4.1 Description de JPF

JPF (*Java Path Finder*) est avant tout un vérificateur de modèles développé par la NASA[8]. Il est actuellement très utilisé dans le monde académique, mais émerge également dans l'industrie. Fujitsu a, par exemple, étendu JPF pour le test d'applications web en mettant au point une fonctionnalité permettant de gérer de manière efficace des entrées de type chaine de caractères[9].

Dès le début du projet JPF, une des principales préoccupations de l'équipe en charge était que JPF serve plus qu'un seul objectif, et qu'il soit capable de réaliser toute sorte de tâches. De ce fait, une qualité de JPF d'être hautement extensible et adaptable. En fonction des besoins il est possible de configurer JPF de différentes manières afin d'obtenir les résultats désirés :

— Ajout d'extension : activation d'un module spécialisé dans un traitement spécifique. Par exemple, activer l'extension jpf-symbc permet de réaliser de l'exécution symbolique sur l'objet testé.

— Configuration du traitement : les règles que JPF et ses composants respecteront lors de l'exécution. La configuration de JPF est réalisée à différents niveau et passe de la définition des *timeouts* à la sélection du générateur de choix.

— Ajustement des sorties : la forme sous laquelle JPF présentera ses ré-

sultats. Il peut s'agir d'imprimer le système de contraintes ou juste le résultat de sa résolution, le test unitaire généré ou encore une combinaison de plusieurs présentations.

4.2 Utilisation

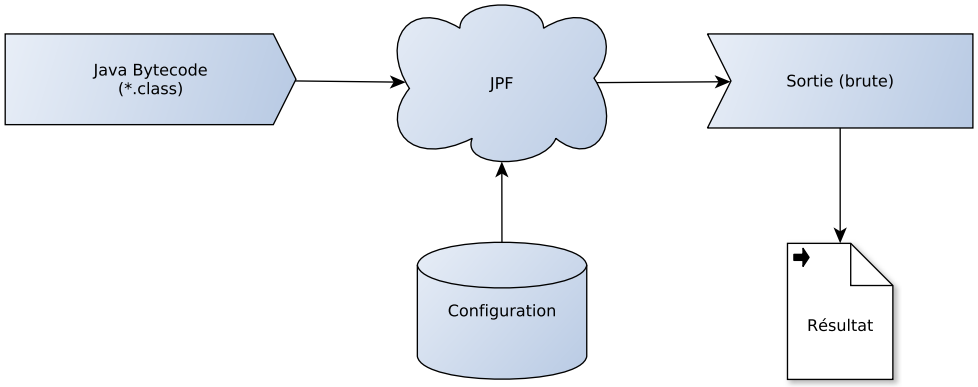
L'utilisation de JPF est relativement complexe et demande un certain à maitriser, que ce soit pour la configuration et le lancement d'une exécution ou la lecture et récupération des rapports. En effet, JPF ne propose pas d'interface ni pour la gestion de la configuration et de l'exécution, ni pour l'interprétation des résultats (graphiques, navigation dans les résultats intermédiares, ...)[45].

Voici les grandes étapes pour l'utilisation de JPF :

1. Choisir la cible : cette première étape a pour but d'identifier le code qui sera testé. Il est possible de rendre l'objet du test "*JPF-aware*" en ajoutant des annotations sur des ressources Java (classes, méthodes, champs, ...) afin que JPF vérifie certaines propriétés lors de l'exécution (pré- et post-conditions par exemple).
2. Préparer la cible : JPF réalisant son analyse sur du Java bytecode, il convient de compiler les classes nécessaires au test et de rendre disponibles les éventuelles librairies et ressources utilisées.
3. Préparer JPF et ses extensions : dans un troisième temps, il s'agit de déterminer le travail que JPF va accomplir, et comment il va le réaliser. Cette étape particulièrement complexe et importante nécessite donc de configurer JPF et les extensions requises pour satisfaire l'objectif.
4. Exécuter JPF sur la cible : démarrage de l'exécution de JPF.
5. Récupérer les résultats : lors de l'exécution, JPF imprime un certain nombre d'informations diverses (configurable lors de l'étape 2). Ces résultats sont généralement assez bruts.
6. Interprétation des résultats : selon les extensions et la configuration utilisées, les résultats générés demandent à être interprétés et travaillés.

Par exemple, un test unitaire pourra être créé à partir des données ou du code récupéré à l'étape précédente.

FIGURE 4.1 – Fonctionnement de JPF



4.3 Principe - fonctionnement

Le fonctionnement est intrinsèquement lié à sa configuration et aux extensions utilisées. De ce fait, il est compliqué d'en brosser une image exhaustive. Cette section va donc s'attarder sur les principes généraux et le fonctionnement de base du coeur de JPF, à savoir le module *jpf-core*.

Le module *jpf-core* est en réalité une machine virtuelle pour Java bytecode s'exécutant elle-même dans la traditionnelle machine virtuelle de Java. Un ensemble d'instructions spécifiques y est disponible. Par exemple, le type *choix d'exécution* permet à JPF de détecter les points de choix de l'application testée, c'est-à-dire là où l'exécution peut emprunter un chemin ou un autre. JPF offre de base des composants de choix d'exécution comme le choix au hasard, mais permet également à l'utilisateur de spécifier son propre type de sélecteur de chemin.

Lorsque JPF rencontre un choix et fait appel au sélecteur de chemin, le contexte du programme est enregistré dans une machine à état. Ce mécanisme permet de restaurer autant de fois qu'il est nécessaire un contexte particulier lors du *backtracking*. Ce dernier est réalisé si JPF détecte des si-

militudes entre l'état dans lequel se trouve le programme et un de ceux de la machine à états et permet de continuer l'exécution avec des chemins encore inexplorés. Cette combinaison de fonctionnalités lutte activement contre un des principaux problèmes de l'exécution symbolique, à savoir l'explosion du nombre de chemins.

Chaque instruction exécutée peut être interceptée par un *listener*. Ce composant est en charge de la vérification d'un attribut du programme et a accès au contexte courant de l'exécution. Lors de la configuration d'une exécution de JPF, il est possible de spécifier un ensemble de *listeners* (ainsi que les données nécessaires pour le traitement de ceux-ci) qui réaliseront une série de tâches.

Le module *jpf-core* met à disposition un ensemble de *listeners* de base permettant de réaliser certaines analyses. Par exemple, le *listener PrecisRaceDetector* peut être utilisé pour détecter des problèmes de concurrence en vérifiant que les variables statiques d'un programme sont lues et modifiées de manière correctes (utilisation du mot clef *synchronize* ou d'un autre mécanisme). D'autres *listeners* peuvent être utilisés conjointement pour vérifier la possibilité de *deadlocks* ou encore le lancement d'exceptions non vérifiées (exemple : *assert*).

Un dernier concept de JPF intéressant à développer est celui de *publisher*. Son rôle est de générer des rapports spécifiques à une tâche. Plusieurs *publishers* peuvent être configurés pour agir en parallèle selon le résultat voulu.

4.4 Configuration de JPF

La configuration de JPF est une étape très importante qui peut s'avérer complexe. En effet, il existe quatre niveaux de configuration :

1. Site : il s'agit du fichier de configuration ayant le plus haut niveau de JPF. C'est dans celui-ci que sont spécifiés les informations concernant JPF et ses composants ainsi que les modules disponibles et installés. Il est généralement placé dans le *home*, tel que *${user.home}/.jpf/site.properties*.
2. Projet : ce fichier contient la configuration par défaut pour un module

particulier. Typiquement, on y retrouvera des informations concernant les *classpaths* (de JPF et natif) [1] ainsi que d'autres propriétés nécessaires à la bonne exécution du module. Ce fichier de configuration doit être placé *module_root/jpf.properties*.

3. Application : chaque programme qui sera l'objet d'une exécution par JPF est accompagnée d'un fichier de configuration spécifiant les propriétés qui seront utilisées ainsi que les tâches que JPF devra accomplir. C'est dans ce fichier que les différents *listeners* et leur configuration sont ajoutés. Il contient également les informations concernant l'objet testé.
4. Ligne de commande : il est enfin possible d'étendre ou de surcharger les propriétés présentes dans les fichiers mentionnés ci-dessus dans la ligne de commande d'invocation de JPF.

Ces différents niveaux font de JPF un outil extensible et adaptable afin de pouvoir répondre aux besoins des utilisateurs.

1. JPF étant une machine virtuelle exécutant du code Java, deux classpaths distincts sont donc nécessaires : un pour JPF lui-même et un autre pour l'application qu'il exécute.

Chapitre 5

Réalisation

5.1 La méthode de travail

Le projet a été découpé en différentes phases afin de faciliter sa réalisation : le code source de M. Marcozzi d'une part et le celui de JPF d'autre part. Les codes n'étant pas maitrisés et relativement complexes, nous avons décidé d'adopter une démarche agile et de procéder par étapes. Chacune de celles-ci possède un objectif clair et simple.

La démarche suivra donc les étapes suivantes :

— Préliminaires : installation de l'environnement de travail

— JPF : prise en main, installation et tests

— Code de M. Marcozzi : prise en main, installation et tests

— Intégration : incorporer le travail de M. Marcozzi dans JPF

— Pistes à suivre : les perspectives futures

5.2 Préliminaires

La première étape du projet consiste en la mise en place d'un environnement de travail adéquat. Celui-ci contient l'ensemble des programmes nécessaires à tout niveau du projet ainsi que la récupération du code source de JPF et du code de M. Marcozzi.

Pour JPF, le code source est disponible librement sur un *repository* Mercurial `http://babelfish.arc.nasa.gov/hg/jpf`.

Concernant le code de M. Marcozzi, il nous a été remis sous forme d'archive.

Le travail sera réalisé sur une plateforme Ubuntu 14.04.2 LTS avec une version de Java 1.8.0_25.

5.3 JPF

5.3.1 JPF-core

La première opération consiste en la récupération du code du module *jpf-core*. Ce module est la base de JPF et celui-ci ne peut s'exécuter sans lui. Une fois le code source disponible localement, le module est compilé à l'aide de l'outil *Ant*[1] [1] afin de rendre le fichier *JAR* [2] *jpf-core_root/build/jpf.jar* est disponible. C'est cet exécutable Java qui sera utilisé pour démarrer une analyse JPF.

5.3.2 JPF-symbc

Le module *jpf-symbc* est un module permettant de réaliser de l'exécution symbolique. Tout comme *jpf-core*, il doit être récupéré et compilé (en utilisant *Ant*) afin de rendre le fichier *JAR jpf-symbc_root/build/jpf-symbc.jar* est disponible. Cet exécutable Java sera utilisé si un *listener* réalisant de l'exécution symbolique est défini dans la configuration de l'analyse JPF.

5.3.3 Configuration de JPF

Lors de la configuration de JPF, nous avons modifié le fichier *Site* 5.1 en ajoutant les informations concernant *jpf-core* et *jpf-symbc*. Nous avons

1. Ant est un outil développé par Apache permettant notamment la compilation de code source à l'aide d'un fichier de configuration (*build.xml*)
2. Un fichier JAR (pour *Java ARchive*) est une archive Java qui peut être exécutée.

également adapté les fichiers du niveau *"Projet"* afin de mettre à jour certaines propriétés comme les *classpaths*[3].

FIGURE 5.1 – Fichier de configuration JPF *site.properties*

```
# JPF site configuration

jpf-core = ${user.home}/workspace/MIHD/memoire/spf/jpf-
    core
jpf-nhandler = ${user.home}/workspace/MIHD/memoire/spf/
    jpf-nhandler
jpf-symbc = ${user.home}/workspace/MIHD/memoire/spf/jpf
    -symbc


extensions=${jpf-core}, ${jpf-nhandler}, {jpf-symbc}
```

5.3.4 Exécution de JPF

Afin de vérifier que l'installation et la configuration de JPF et de ses composants ait été réalisées correctement, nous avons démarré une exécution sur une classe de test. En effet, chaque module de JPF fournit une batterie d'exemples permettant de tester l'ensemble des concepts et configurations mis à disposition par JPF.

Par exemple, la classe de démonstration *NumericExample.java* 5.2 et son fichier de configuration 5.3 se trouvant dans le module *jpf-symbc* permettent de réaliser le test de l'exécution symbolique sur un bout de code simple.

Le résultat de cette exécution est disponible en annexe 1.

5.4 Le code de M. Marcozzi

L'application de M. Marcozzi réalise de l'exécution symbolique sur de petits programmes en incorporant la dimension base de données. Actuellement,

3. Le *classpath* est une propriété utilisée par la machine virtuelle Java qui représente les chemins d'accès au répertoires contenant les classes Java.

FIGURE 5.2 – Code de démo pour *jpf-symbc* : *NumericExample.java*

```
package demo;

public class NumericExample {

    public static int test(int a, int b) {
        int c = a / (b + a - 2);
        if (c > 0) {
            System.out.println(">0");
        } else {
            System.out.println("<=0");
        }
        return c;
    }

    public static void main(String[] args) {
        test(0, 0);
    }

}
```

cette application réalise les opérations suivantes :

1. Lecture et parsing du fichier testé : ce fichier doit posséder un format bien spécifique, à savoir la description de la structure de la base de données (DDL) suivi d'une ou plusieurs méthodes contenant du code Java.
2. Analyse du code : réalisation de l'exécution symbolique sur l'ensemble du programme.
3. Vérification de la satisfiabilité du système : pour chaque chemin possible, traduction de celui-ci en contraintes pour le solutionneur (en l'occurrence le produit de Microsoft nommé Z3[26]).
4. Génération des résulats : pour chaque chemin (possible ou non), enregistrement des données en entrées et résultats (retour de méthode ou erreur).

FIGURE 5.3 – Fichier de configuration JPF pour le code *NumericExample.java*

```
target=demo.NumericExample
classpath=${jpf-symbc}/build/examples
sourcepath=${jpf-symbc}/src/examples
symbolic.method = demo.NumericExample.test(sym#sym)

symbolic.dp=coral
listener = .symbc.SymbolicListener

symbolic.debug=on

search.multiple_errors=true
```

5.4.1 Installation et configuration

Nous avons suivi les étapes suivantes afin d'installer et la configurer de l'application :

1. Décompression et compilation des fichiers sources
2. Installation des librairies et composants nécessaires à l'exécution (comme le solutionneur Z3)
3. Configuration de l'application : chemins vers les composants et spécifications du programme à tester
4. Exécution de l'application sur programme de test.

Lors de l'exécution des tests, des problèmes de compatibilité entre plateformes ont été rencontrés. En effet, l'application initialement récupérée était dépendante du système d'exploitation Windows et ce travail est réalisé dans un environnement Linux. Des adaptations ont été réalisées pour rendre l'application compatible avec n'importe quel système d'exploitation.

5.4.2 Extraction de la configuration

Initialement, l'exécution de l'application se réalisait avec des valeurs définies dans le code. En effet, toute la configuration d'une exécution était codée

en dur, ce qui rendait l'utilisation du programme peu pratique et inefficace (obligation de recompiler pour changer une propriété). De plus, cette pratique est contraire à la philosophie de JPF qui se veut hautement extensible et configurable.

Dès lors, nous avons apporté des modifications au code de M. Marcozzi afin d'extraire tout ce qui concernait la configuration dans un fichier de propriétés (comme JPF). Cette manipulation a permis de faciliter l'utilisation du programme et de le préparer à l'intégration à JPF.

5.4.3 Modification du parser

A l'origine, le programme développé par M. Marcozzi réalisait son analyse sur une structure de base de données accompagnée d'un programme. Pour des raisons pratiques, ces deux entités devaient se trouver dans un même fichier et suivre une syntaxe spécifique afin de pouvoir être lues par le *parser*. Celui-ci lisait donc d'une part la structure de base de données et d'autre part le programme à tester.

Le *parser* de M. Marcozzi a été créé en combinant le générateur d'analyseur syntaxique *JFlex*[4] et le générateur de *parser* pour Java CUP[2].

Un des objectifs de ce travail étant de laisser à JPF la partie réalisant la lecture de programme, le *parser* de M. Marcozzi devait être adapté : il devait être scindé et les deux opérations rendues indépendantes l'une de l'autre.

Pour ce faire, la partie du *parser* concernant la lecture de la structure de la base de données a été extraite et placée dans un second *parser*. Cette modification a également nécessité quelques adaptations dans le code afin de gérer la double opération.

Alors que l'exécution de l'application requérait un seul fichier en entrée (portant l'extension **.sdb*), deux fichiers sont maintenant nécessaires : un contenant la structure de la base de données (portant l'extension **.sdb*) et un autre contenant le programme à tester (portant l'extension **.spgm*). Cette pratique est d'autant plus appréciable qu'un fichier de structure peut être utilisé par plusieurs programmes, et inversement.

5.5 Intégration

L'aboutissement de cette étape est d'obtenir un composant JPF capable de réaliser de l'exécution symbolique sur un programme manipulant une base de données en utilisant l'algorithme de M. Marcozzi. A l'origine, l'application de M. Marcozzi ne supporte qu'un ensemble restreint des instructions disponibles en Java et requiert une syntaxe spécifique (du fait de l'implémentation du *parser*).

5.5.1 Création du nouveau listener

Dans JPF, il existe deux types de *listener* appelés *SearchListener* et *VMListener*, qui étendent eux même l'interface *JPFListener*. Ces deux interfaces servent des buts différents mais pas mutuellement exclusifs. *SearchListener* permet de surveiller l'état de JPF lors de la phase de recherche et d'exploration. Quant à *VMListener*, il contient de nombreuses méthodes permettant notamment d'obtenir le contexte précédant l'exécution d'une instruction ou celui d'après exécution.

JPF offre également des *adapters* comme *PropertyListenerAdapter* : ces classes abstraites implémentent les interfaces intéressantes et permettent de surcharger uniquement les méthodes nécessaires à la réalisation de l'objectif du *listener* créé.

Pour des raisons pratiques, nous avons réalisé l'intégration dans le module *jpf-symbc* sous la forme d'un nouveau *listener* nommé *DatabaseListener* et héritant de *PropertyListenerAdapter*. En effet, cette pratique permet de bénéficier des avantages et de la configuration déjà existante de *jpf-symbc* ainsi qu'un accès direct aux classes et méthodes utilitaires pour réaliser l'exécution symbolique.

5.5.2 Intégration du parsing de la structure de base de données

La seconde étape dans le processus d'intégration a été d'ajouter le parsing de la structure de base de données dans le nouveau *listener*. Pour ce

faire, Nous avons ajouté à la configuration de JPF une nouvelle propriété représentant le chemin vers le fichier à *parser*.

L'opération en elle-même est réalisée dans le constructeur du *listener* :

1. Récupération de la configuration : nécessaire pour connaitre le fichier de structure de base de données à *parser*
2. Parsing du fichier : création de l'objet symbolisant la base de données et gestion des erreurs
3. Récupération de l'objet base de données et stockage dans une variable de classe du *listener* afin qu'il soit disponible lors de l'exécution

Afin de rendre les futures modifications du *parser* disponibles directement lors de la compilation, nous avons également modifié le fichier de configuration *Ant* pour faire en sorte que la génération des classes du *parser* soit réalisée en même temps que la compilation du module.

5.5.3 Création d'un POC

L'étape suivante a été de créer un POC [4] afin de vérifier que JPF supporte correctement les différentes librairies utilisées. L'objectif de cette étape était de développer un programme simple manipulant une base de données.

Nous avons exploré deux possibilités : l'utilisation de *mocks* [5] et l'utilisation des classes réelles du package *java.sql*.

La première option a d'abord été mise en place du fait de sa simplicité : création des classes (*mocks*) implémentant les différentes interfaces nécessaires (*java.sql.Statement*, *java.sql.Connection*, ...). Un appel de méthode de ces classes retourne *null* ou la valeur 0. L'exécution de JPF sur ce type de programme utilisant des *mocks* n'a pas soulevé d'erreur. Cependant, cette pratique s'est révélée insuffisante car JPF, lors de l'exécution symbolique, récupérait ces constantes (0 ou *null*) comme résultats des appels aux méthodes.

4. Un POC (pour Proof Of Concept) est un prototype incomplet créé afin de démontrer la faisabilité d'une méthode ou d'une idée.

5. En informatique, un *mock* est une imitation d'un objet réel dont le comportement est contrôlé. Cette pratique est généralement utilisée lors de la phase de tests.

La seconde option consiste en l'utilisation des classes réelles du package *java.sql*. Cependant, de nombreuses erreurs ont été rencontrées lors des tests de ce type de code. En effet, en réalisant l'exécution symbolique sur le programme de test, JPF descend dans les appels et explore les classes du package *java.sql*. Des exceptions de JPF sont remontées, signifiant qu'il ne parvient pas à trouver de solution pour gérer le code natif 5.4.

FIGURE 5.4 – Exception lancée par JPF lors de l'exécution d'un programme utilisant des classes du package *java.sql*

```
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.UnsatisfiedLinkError: cannot find native java.util.zip.ZipFile.initIDs
at java.util.zip.ZipFile.initIDs(no peer)
at java.util.zip.ZipFile.<clinit>(ZipFile.java:87)
at sun.net.www.protocol.jar.JarFileFactory.get(JarFileFactory.java:84)
at sun.net.www.protocol.jar.JarURLConnection.connect(JarURLConnection.java:122)
at sun.net.www.protocol.jar.JarURLConnection.getInputStream(JarURLConnection.java:150)
at java.net.URL.openStream(URL.java:1038)
[...]
```

5.5.4 JPF-nhandler

Jpf-nhandler[5] est un module permettant de déléguer automatiquement les appels de certaines méthodes de l'objet testé depuis JPF vers la JMV hôte. Plus simplement, ce module permet de générer à la volée du code afin que JPF puisse comprendre et gérer l'appel de méthodes natives.

L'utilisation du module *jpf-nhandler* nous a permis de résoudre certains problèmes liés aux limitations de JPF 5.4. Néanmoins, le module lui-même est encore en développement actif et n'est pas sans limitation. En effet, *jpf-nhandler* ne gère actuellement pas les objets dont au mois une partie de leur état est gardé nativement. C'est pourquoi, malgré l'utilisation du module, JPF ne peut garantir un *backtracking* correct et soulève des erreurs.

5.6 Pistes et améliorations

5.6.1 JPF

Actuellement, l'obstacle principal est la réalisation de l'exécution symbolique lorsque le programme testé utilise des classes du package *java.sql*. C'est

donc le premier problème à solutionner. Une piste serait de déléguer les appels des méthodes problématique à un plus haut niveau dans la hiérarchie :

1. Centraliser les appels au package *java.sql* dans une classe utilitaire
2. Déléguer les appels à cette classe grâce au module *jpf-nhanlder*
3. Modéliser la base de données et modifier le code généré à l'étape 2 afin que JPF puisse réaliser son exécution symbolique sur ce code

Néanmoins, cette dernière étape étant complexe et chronophage, cette piste n'a pu être explorée.

Une amélioration serait de déplacer le code dans une module séparé afin de réduire le couplage entre les composants de base de JPF et les nouvelles fonctionnalités. Cette pratique serait de surcroit dans le courant de la philosophie de JPF, à savoir hautement extensible et modulable.

5.6.2 Code de M. Marcozzi

Pour le moment, l'algorithme de M. Marcozzi utilise sa propre représentation du programme, tel qu'il est *parsé*. L'algorithme est intimement lié à la structure du programme. Cependant, cette structure est différente dans JPF : les instructions et la manière de les représenter ne sont pas toujours compatibles avec celles de M. Marcozzi et il existe également de nombreuses instructions définies par JPF qui sont tout simplement inexistantes dans le code de M. Marcozzi. Une tâche future à accomplir serait donc de lister les différences entre les instructions d'une part, et de rendre l'algorithme indépendant de la structure du programme d'autre part.

Aussi, le programme de M. Marcozzi génère des contraintes spécifiques au solutionneur Z3. Cependant, JPF n'est pas lié à un solutionneur en particulier. Il conviendrait donc d'implémenter un générateur de contraintes compatible avec les solutionneurs utilisés par JPF.

Chapitre 6

Conclusion

A travers ce mémoire, nous avons pu voir l'importance d'une phase de test efficace. La qualité d'un programme est intrinsèquement liée à celle-ci. Dès lors, la mise en application de nouvelles méthodes pérennes représente un défi important.

Les objectifs de ce mémoire étaient de sélectionner un programme permettant la mise en oeuvre de l'exécution symbolique et d'y intégrer le nouvel algorithme de génération de contraintes pour des programmes manipulant des bases de données conçu par M. Marcozzi.

Dans un premier temps, nous avons défini les critères de sélection de ce programme. A partir de ceux-ci, nous avons choisi le programme le plus adapté pour l'intégration parmi une liste composées des 6 plus grands programmes réalisant de l'exécution symbolique. L'outil possédant toutes les qualités désirées, JPF fut sélectionné pour la suite de l'intégration.

Ensuite, nous avons pris le temps d'installer, configurer et maitriser JPF d'une part, et l'application de M. Marcozzi d'autre part. Nous avons également modifié le code de cette dernière afin de la rendre plus modulable, configurable et compatible avec JPF.

Enfin, nous avons ajouté à JPF un nouveau composant permettant de réaliser le *parsing* de la structure de la base de données utilisée dans le programme à tester. L'intégration des autres fonctionnalités du programme de M. Marcozzi n'a pas pu être réalisée du fait des erreurs remontées par JPF.

Cet ouvrage clarifie la situation, explore divers chemins et donne des nouvelles pistes pour des travaux futurs. En effet, même si ce mémoire ne remplit que partiellement les objectifs initiaux, il a permis de révéler certains obstacles et de mettre à jour des problématiques qui étaient originellement inconnues. Il constitue, dès lors, une base solide pour tout qui désirerait se plonger de manière plus poussée dans la Génération de contraintes pour le test de programmes manipulant une base de données.

Bibliographie

[1] Apache ant. http://ant.apache.org/. Accessed : 2015-08-10.

[2] Java cup : Lalr parser generator for java. http://www2.cs.tum.edu/projects/cup/. Accessed : 2015-08-23.

[3] Jcute. https://github.com/osl/jcute. Accessed : 2015-07-04.

[4] Jflex lexical analyzer generator. http://jflex.de/. Accessed : 2015-08-23.

[5] Jpf-nhandler extension for jpf. https://bitbucket.org/nastaran/jpf-nhandler. Accessed : 2015-07-30.

[6] Klee. http://klee.github.io/. Accessed : 2015-07-04.

[7] Llvm. http://llvm.org/. Accessed : 2015-07-04.

[8] Nasa ames research center. http://www.nasa.gov/centers/ames/home/index.html. Accessed : 2015-07-05.

[9] Nfujitsu develops technology to enhance comprehensive testing of java programs. http://www.fujitsu.com/global/about/resources/news/press-releases/2010/0112-02.html. Accessed : 2015-07-12.

[10] Nosa 1.3 license. http://javapathfinder.sourceforge.net/NOSA-1.3-JPF.txt. Accessed : 2015-07-05.

[11] Pex. http://research.microsoft.com/en-us/projects/pex/. Accessed : 2015-07-04.

[12] Selenium hq. http://www.seleniumhq.org/. Accessed : 2015-06-28.

[13] Symbolic pathfinder. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc. Accessed : 2015-06-28.

[14] S. Abiteboul, R. Hull, and V. Vianu, editors. *Foundations of Databases : The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[15] Darrell Bethea, Robert A. Cochran, and Michael K. Reiter. Server-side verification of client behavior in online games. *ACM Trans. Inf. Syst. Secur.*, 14(4) :32 :1–32 :27, December 2008.

[16] R.S. Boyer, B. Elspas, and K.N Levitt. A formal system for testing and debugging programs by symbolic execution. *SIGPLAN*, 10 :234–245, 1975.

[17] C. Cadar, D. Dunbar, and D. Engler. Klee : Unassisted and automatic generation of high-coverage tests for complex systems programs. *USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, 2008.

[18] C. Cadar and D. Engler. Execution generated test cases : How to make systems code crash itself (invited paper). In *Proceedings of SPIN'05*, 2005.

[19] C. Cadar and Godefroid. Symbolic execution for software testing in practice : Preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE '11*, pages 1066–1071, 2011.

[20] C. Cadar and K. Sen. Symbolic execution for software testing : Three decades later. *Commun. ACM*, 82-90, 2013.

[21] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe : Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[22] M. Y. Chan and S. C. Cheung. Testing database applications with sql semantics. In *In Proceedings of the 2nd International Symposium on Cooperative Database Systems for Advanced Applications*, pages 363–374. Springer, 1999.

[23] L.A Clarke. A program testing system. In *In Proceedings of the 1976 Annual Conference*, pages 488–491, 1976.

[24] P. D. Coward. Symbolic execution systems a review. *Softw. Eng. J.*, pages 229–239, 1988.

[25] L. De Moura and N. Bjørner. Satisfiability modulo theories : introduction and applications. *Commun. ACM*, 54 :69–77, 2011.

[26] Leonardo de Moura and Nikolaj Bjørner. Z3 : An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[27] B. Duma. Human machine interaction ihdcm030. Technical report, Department of Computer Science, University of Namur, 2015.

[28] Jon Edvardsson. A survey on automatic test data generation, 1999.

[29] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 151–162, New York, NY, USA, 2007. ACM.

[30] V. Englebert. Software architecture engineering : technologies and methods ihdcm033. Technical report, Department of Computer Science, University of Namur, 2015.

[31] M. Fowler. Test pyramid. `http://martinfowler.com/bliki/TestPyramid.html`, 2012. Accessed : 2015-06-26.

[32] P. Godefroid, N. Klarlund, and K. Sen. Dart : Directed automated random testing. In *Proceedings of SPIN'05*, 2005.

[33] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart : Directed automated random testing. *SIGPLAN Not.*, 40(6) :213–223, June 2005.

[34] N. Habra. Software engineering ihdcm031. Technical report, Department of Computer Science, University of Namur, 2015.

[35] M. Hicks, M. Hammer, J Foster, and S. Strickland. Program analysis and understanding. Technical report, Department of Computer Science, University of Maryland, 2013.

[36] A. Hunt and D. Thomas. *Pragmatic Unit Testing in Java with JUnit*. Pragmatic Programmers. The Pragmatic Programmers, 2003.

[37] R.B. Jones. *Symbolic Simulation Methods for Industrial Formal Verification*. Springer US, 2002.

[38] J.C King. Symbolic execution and program testing. *Commun. ACM*, 19 :385–394, 1976.

[39] M. Marcozzi, W. Vanhoof, and J-L. Hainaut. Relational symbolic execution of sql code for effective testing of database programs. Technical report, September 2014.

[40] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38, 1965.

[41] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[42] Kai Pan, Xintao Wu, and Tao Xie. Guided test generation for database applications via synthesized database interactions. *ACM Trans. Softw. Eng. Methodol.*, 23(2) :12 :1–12 :27, April 2014.

[43] C. S. Pasareanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transf.*, pages 339–353, 2009.

[44] Roger S. Pressman. *Software Engineering : A Practitioner's Approach*. McGraw-Hill Higher Education, 5th edition, 2001.

[45] Corina S. Păsăreanu and Neha Rungta. Symbolic pathfinder : Symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 179–180, New York, NY, USA, 2010. ACM.

[46] Raimondas Sasnauskas, Olaf L, Muhammad Hamad Alizai, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle. Kleenet : Discovering in-

sidious interaction bugs in wireless sensor networks before deployment, 2010.

[47] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy, SP '10, IEEE Computer Society*, pages 317–331, 2010.

[48] A. Scott. Introducing the software testing ice-cream cone (anti-pattern). `http://watirmelon.com/2012/01/31/introducing-the-software-testing-ice-cream-cone/`, 2012. Accessed : 2015-06-28.

[49] Koushik Sen and Gul Agha. Cute and jcute : Concolic unit testing and explicit path model-checking tools. In *In CAV*, pages 419–423. Springer, 2006.

[50] Koushik Sen, Darko Marinov, and Gul Agha. Cute : A concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, 30(5) :263–272, September 2005.

[51] M. Veanes, N. Tillmann, and J. de Halleux. Qex : Symbolic sql query explorer. Technical Report MSR-TR-2009-2015, October 2009. Updated January 2010.

Table des figures

2.1 La pyramide des tests automatiques selon M. Fowler 10
2.2 Exemple simple illustrant l'exécution symbolique [20] 16
2.3 Arbre d'exécution de l'exemple 2.2 17

4.1 Fonctionnement de JPF 27

5.1 Fichier de configuration JPF *site.properties* 32
5.2 Code de démo pour *jpf-symbc* : *NumericExample.java* 33
5.3 Fichier de configuration JPF pour le code *NumericExample.java* 34
5.4 Exception lancée par JPF lors de l'exécution d'un programme utilisant des classes du package *java.sql* 38

1 Résultats générés par l'exécution de *jpf-symbc* sur le code *NumericExample.java* 48

FIGURE 1 – Résultats générés par l'exécution de *jpf-symbc* sur le code *NumericExample.java*

```
$ ./jpf-core/bin/jpf src/examples/demo/NumericExample.jpf
Running Symbolic PathFinder ...
symbolic.dp=coral
symbolic.string_dp_timeout_ms=0
symbolic.string_dp=none
symbolic.max_pc_length=2147483647
symbolic.max_pc_msec=0
symbolic.min_int=-1000000
symbolic.max_int=1000000
symbolic.min_double=-8.0
symbolic.max_double=7.0
JavaPathfinder core system v8.0 (rev 25+) - (C) 2005-2014 United States Government. All
     rights reserved.


============================================================ system under test
demo.NumericExample.main()

============================================================ search started: 8/15/15 3:51 PM
numeric PC: constraint # = 1
((a_1_SYMINT + b_2_SYMINT) - CONST_2) != CONST_0 -> true

### PCs: total:1 sat:1 unsat:0

## Warning: Non Linear Integer Constraint (only coral can handle it)%NonLinInteger% (
     a_1_SYMINT / ((a_1_SYMINT + b_2_SYMINT) - CONST_2)) <= CONST_0 &&
((a_1_SYMINT + b_2_SYMINT) - CONST_2) != CONST_0
numeric PC: constraint # = 2
%NonLinInteger% (a_1_SYMINT / ((a_1_SYMINT + b_2_SYMINT) - CONST_2)) <= CONST_0 &&
((a_1_SYMINT + b_2_SYMINT) - CONST_2) != CONST_0 -> true

### PCs: total:2 sat:2 unsat:0

## Warning: Non Linear Integer Constraint (only coral can handle it)%NonLinInteger% (
     a_1_SYMINT / ((a_1_SYMINT + b_2_SYMINT) - CONST_2)) > CONST_0 &&
((a_1_SYMINT + b_2_SYMINT) - CONST_2) != CONST_0
numeric PC: constraint # = 2
%NonLinInteger% (a_1_SYMINT / ((a_1_SYMINT + b_2_SYMINT) - CONST_2)) > CONST_0 &&
((a_1_SYMINT + b_2_SYMINT) - CONST_2) != CONST_0 -> true

### PCs: total:3 sat:3 unsat:0

>0
## Warning: Non Linear Integer Constraint (only coral can handle it)%NonLinInteger% (
     a_1_SYMINT / ((a_1_SYMINT + b_2_SYMINT) - CONST_2)) > CONST_0 &&
((a_1_SYMINT + b_2_SYMINT) - CONST_2) != CONST_0
numeric PC: constraint # = 2
%NonLinInteger% (a_1_SYMINT / ((a_1_SYMINT + b_2_SYMINT) - CONST_2)) > CONST_0 &&
((a_1_SYMINT + b_2_SYMINT) - CONST_2) != CONST_0 -> true

************Summary***************
PC is:constraint # = 2
%NonLinInteger% (a_1_SYMINT[-50] / ((a_1_SYMINT[-50] + b_2_SYMINT[17]) - CONST_2)) >
     CONST_0 &&
((a_1_SYMINT[-50] + b_2_SYMINT[17]) - CONST_2) != CONST_0
Return is:  (a_1_SYMINT[-50] / ((a_1_SYMINT[-50] + b_2_SYMINT[17]) - CONST_2))
**********************************
<=0
## Warning: Non Linear Integer Constraint (only coral can handle it)%NonLinInteger% (
     a_1_SYMINT / ((a_1_SYMINT + b_2_SYMINT) - CONST_2)) <= CONST_0 &&
((a_1_SYMINT + b_2_SYMINT) - CONST_2) != CONST_0
numeric PC: constraint # = 2
%NonLinInteger% (a_1_SYMINT / ((a_1_SYMINT + b_2_SYMINT) - CONST_2)) <= CONST_0 &&
((a_1_SYMINT + b_2_SYMINT) - CONST_2) != CONST_0 -> true
```

```
************Summary***************
PC is:constraint # = 2
%NonLinInteger% (a_1_SYMINT[0] / ((a_1_SYMINT[0] + b_2_SYMINT[0]) - CONST_2)) <= CONST_0
      &&
((a_1_SYMINT[0] + b_2_SYMINT[0]) - CONST_2) != CONST_0
Return is:  (a_1_SYMINT[0] / ((a_1_SYMINT[0] + b_2_SYMINT[0]) - CONST_2))
*********************************
numeric PC: constraint # = 1
((a_1_SYMINT + b_2_SYMINT) - CONST_2) == CONST_0 -> true

### PCs: total:4 sat:4 unsat:0

numeric PC: constraint # = 1
((a_1_SYMINT + b_2_SYMINT) - CONST_2) == CONST_0 -> true

Property Violated: PC is constraint # = 1
((a_1_SYMINT[15] + b_2_SYMINT[-13]) - CONST_2) == CONST_0
Property Violated: result is  "java.lang.ArithmeticException: div by 0..."
***************************

====================================================== error 1
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.ArithmeticException: div by 0
        at demo.NumericExample.test(NumericExample.java:24)
        at demo.NumericExample.main(NumericExample.java:34)


====================================================== snapshot #1
thread java.lang.Thread:{id:0,name:main,status:RUNNING,priority:5,isDaemon:false,
     lockCount:0,suspendCount:0}
  call stack:
        at demo.NumericExample.test(NumericExample.java:24)
        at demo.NumericExample.main(NumericExample.java:34)


====================================================== Method Summaries
Inputs: a_1_SYMINT,b_2_SYMINT

demo.NumericExample.test(-50,17)  --> Return Value: 1
demo.NumericExample.test(0,0)  --> Return Value: 0
demo.NumericExample.test(15,-13)  --> "java.lang.ArithmeticException: div by 0..."
Inputs: a_1_SYMINT,b_2_SYMINT

demo.NumericExample.test(-50,17)  --> Return Value: 1
demo.NumericExample.test(0,0)  --> Return Value: 0
demo.NumericExample.test(15,-13)  --> "java.lang.ArithmeticException: div by 0..."

====================================================== Method Summaries (HTML)
<h1>Test Cases Generated by Symbolic JavaPath Finder for demo.NumericExample.test (Path
     Coverage) </h1>
<table border=1>
<tr><td>a_1_SYMINT</td><td>b_2_SYMINT</td><td>RETURN</td></tr>
<tr><td>-50</td><td>17</td><td>Return Value: 1</td></tr>
<tr><td>0</td><td>0</td><td>Return Value: 0</td></tr>
<tr><td>15</td><td>-13</td><td>"java.lang.ArithmeticException: div by 0..."</td></tr>
</table>
<h1>Test Cases Generated by Symbolic JavaPath Finder for demo.NumericExample.test (Path
     Coverage) </h1>
<table border=1>
<tr><td>a_1_SYMINT</td><td>b_2_SYMINT</td><td>RETURN</td></tr>
<tr><td>-50</td><td>17</td><td>Return Value: 1</td></tr>
<tr><td>0</td><td>0</td><td>Return Value: 0</td></tr>
<tr><td>15</td><td>-13</td><td>"java.lang.ArithmeticException: div by 0..."</td></tr>
</table>

====================================================== results
error #1: gov.nasa.jpf.vm.NoUncaughtExceptionsProperty "java.lang.ArithmeticException:
     div by 0  at demo.N..."

====================================================== statistics
elapsed time:       00:00:00
states:             new=5,visited=0,backtracked=5,end=2
search:             maxDepth=3,constraints=0
choice generators:  thread=1 (signal=0,lock=1,sharedRef=0,threadApi=0,reschedule=0),
     data=2
heap:               new=366,released=20,maxLive=344,gcCycles=3
instructions:       3136
max memory:         77MB
loaded code:        classes=60,methods=1205

====================================================== search finished: 8/15/15 3:51 PM
```