

Outils pour la synthèse de tests et la maîtrise de l'explosion combinatoire.

Olivier Maury

▶ **To cite this version:**

Olivier Maury. Outils pour la synthèse de tests et la maîtrise de l'explosion combinatoire.. Génie logiciel [cs.SE]. Université Joseph-Fourier - Grenoble I, 2005. Français. NNT : . tel-00011507

HAL Id: tel-00011507
https://theses.hal.science/tel-00011507

Submitted on 31 Jan 2006

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.

Université Joseph Fourier - Grenoble I
UFR Informatique et Mathématiques Appliquées

THESE

pour obtenir le titre de

Docteur de l'Université Joseph Fourier

Discipline : Informatique

présentée et soutenue publiquement

par

Olivier Maury

le 09 décembre 2005

Outils pour la synthèse de tests et la maîtrise de l'explosion combinatoire.

Composition du Jury :

P.-Y. Cunin	**Président**
P. Le Gall	**Rapporteur**
F. Bouquet	**Rapporteur**
Y. Ledru	**Directeur de Thèse**
C. Oriat	**Co-directrice de Thèse**
L. du Bousquet	**Examinateur**

Remerciements

Lorsqu'on arrive au moment de rédiger les remcerciements on se dit que le plus gros est passé. Pourtant rédiger les remerciements n'est pas si simple car, lorsque j'y réfléchie le nombre de personnes qui m'ont permis d'arriver au bout de ma thèse est assez phénoménal. Alors bien entendu tous n'ont pas eu la même importance, malgré tout même les petites choses comptent. Rassurez-vous, je ne parlerai pas des commercants grâce à qui j'ai pu me nourrir, élément au combien vital pour terminer une thèse (entre autres).

Je remercie les rapporteurs Pascale et Fabrice pour avoir accepté d'évaluer mon travail. Je remercie tout spécialement Pascale car je ne l'avais pas revu depuis six ans. Je remercie aussi Pierre-Yves pour avoir présidé mon Jury de thèse.

Je remercie mes directeurs de thèse Yves et Catherine pour avoir accepté d'encadrer ma thèse au cours de ces cinq ans. Je remercie Yves, Catherine, Lydie, Marie-Laure et Farid pour leurs conseils scientifiques et pour leurs encouragements lors des moments difficiles.

Je remercie aussi tous les membres du laboratoire, les ingénieurs système Christiane et François, les membres du secrétariat Liliane, Marine et Pascale qui m'ont permis de réaliser ma thèse dans de bonnes conditions.

Je remercie ma compagne Carine qui a aussi bien supporté mes moments d'exaltation que mes états d'âme, qui a relu ma thèse et mes transparents.

Je remercie tous mes amis Grenoblois Aline, Cyril, Francis, Gogo, Jérome, Jean-Christophe, Jean-Philippe, Karim, Manu, Olivier, Pierre, Rémi, Romain, Sophie, Sylvia qui ont su me changer les idées et avec qui j'ai pu échanger sur des sujets aussi variés que la politique, la science, la religion, et les femmes (mais pas nécessairement les quatre sujets en même temps).

Je remercie aussi mes amis Essonniens Franck, Julien, Lucie, Marc et Stéphane avec qui j'ai peu m'évader certains week-end, ce qui m'a fait le plus grand bien.

C'est sans oublier mes parents Jacqueline et Jean-Claude qui ont toujours su être là pour moi.

J'ai une pensée spéciale pour Pierre qui a partagé un bureau avec moi pendant trois ans. Il a dû supporter mes nombreuses idées tordues et mes doutes, en plus des siens ! Doué d'une résistance hors norme, il a survécu à ces trois années et ne semble pas avoir gardé de séquelles de mon passage.

Table des matières

I Problématique et contexte **1**

1 Introduction **3**

2 Contexte général **9**
2.1 Test : quelques définitions 9
2.2 Test de conformité 11
2.2.1 Quelques définitions 12
2.2.2 Conformité 13
2.2.3 Problème de l'exhaustivité 14
2.2.4 Problème de l'oracle 16
2.3 État de l'art sur la synthèse de test 19
2.3.1 Génération de tests à partir de spécifications orientées modèles 20
2.3.2 Génération combinatoire de tests 22
2.3.3 Synthèse 26

3 Spécifications comme Oracle **27**
3.1 VDM 27
3.1.1 Fondements du langage 27
3.1.2 Exemple : le gestionnaire de processus 28
3.1.3 Relation de conformité en VDM 30
3.2 JML : un langage de spécification pour les programmes JAVA 33
3.2.1 Fondements du langage 33
3.2.2 Exemple 35
3.2.3 Vérification d'assertions à l'exécution 38
3.3 Synthèse 41

II Génération de tests combinatoire **43**

4 Langage de synthèse de tests **45**
4.1 Motivations 45
4.1.1 Projet LHUSY 45
4.1.2 Projet COTE 46

4.2 Principes du langage de synthèse de tests . . . 47
4.2.1 Spécification d'entrée . . . 48
4.2.2 Déploiement . . . 49
4.2.3 Groupes d'opérations . . . 49
4.2.4 Langage de schéma de tests . . . 50
4.2.5 Alternative et corégion pour la génération de tests . . . 56
4.3 Génération de tests concrets . . . 59
4.3.1 Génération de tests pour VDM . . . 59
4.3.2 Génération de tests Java . . . 61
4.4 Synthèse . . . 62

5 Étude de cas industrielle en Java 63
5.1 Contexte et application . . . 63
5.1.1 Contexte . . . 63
5.1.2 Étude de cas . . . 65
5.2 Approche expérimentale . . . 66
5.2.1 Revue de code et test aléatoire . . . 68
5.2.2 Génération combinatoire à partir du cahier des charges . . . 68
5.3 Campagnes de test . . . 69
5.3.1 Fonctionnalité 1 : Opérations d'ouverture et de clôture de compte . . . 69
5.3.2 Fonctionnalité 2 : création des règles d'épargne et de dépense . . . 71
5.3.3 Fonctionnalité 3 : les règles sont activées de manière périodique . . . 75
5.4 Résultats et synthèse . . . 77
5.4.1 Synthèse des problèmes trouvés . . . 77
5.4.2 Couverture du code . . . 79
5.4.3 Testabilité de l'application . . . 80
5.4.4 Utilisabilité du langage de schéma de tests . . . 82

III Maîtrise de l'explosion combinatoire 85

6 Maîtrise de l'explosion combinatoire à la génération 87
6.1 Motivations . . . 87
6.1.1 Tests redondants . . . 88
6.1.2 Tests violant la spécification . . . 89
6.2 Définition de contraintes . . . 89
6.2.1 Utilisation de séquences . . . 90
6.2.2 Utilisation de fonctions . . . 91
6.2.3 Implantation avec VDM . . . 92
6.2.4 Expérimentations . . . 92
6.3 Évolutions du langage de schéma de tests . . . 96
6.4 Programmation logique avec contraintes . . . 97
6.4.1 Évaluation de séquences complètement instanciées . . . 98

6.4.2 Utilisation de séquences non instanciées . . . 98
6.4.3 Utilisation de séquences partiellement instanciées . . . 100
6.4.4 Gestion du non-déterminisme . . . 101
6.4.5 Bilan . . . 102
6.5 Conclusions . . . 102

7 Filtrage à l'exécution **103**
7.1 Principes . . . 103
7.2 Mise en œuvre . . . 104
7.2.1 Arbre d'appels . . . 104
7.2.2 Autre manière d'exécuter les tests . . . 106
7.3 Optimisations . . . 106
7.4 Expérimentations . . . 107
7.5 Conclusions . . . 109

8 Étude de cas en VDM **111**
8.1 Cahier des charges et spécification . . . 112
8.2 Différentes implantations . . . 115
8.3 Utilisation d'une suite de tests manuelle . . . 117
8.3.1 Création des tests . . . 117
8.3.2 Exécution des tests . . . 118
8.4 Utilisation des schémas de tests . . . 118
8.4.1 Suite de "tests simples" (*Suite 1*) . . . 119
8.4.2 Suite de tests "force brutale" (*Suite 2*) . . . 122
8.4.3 Synthèse . . . 124
8.5 Filtrage . . . 126
8.5.1 Filtrage à la génération . . . 126
8.5.2 Filtrage à l'exécution . . . 130
8.6 Conclusions . . . 130

9 Conclusions et perspectives **133**

A Tobias **139**
A.1 Données quantitatives . . . 139
A.2 Création de schémas de test dans Tobias . . . 140

B Étude de cas VDM : compléments **143**
B.1 Spécifications . . . 143
B.1.1 size_min_gr . . . 143
B.1.2 size_max_gr . . . 143
B.1.3 max_nb_of_gr . . . 144
B.1.4 table2partition . . . 144
B.2 Groupes pour Tobias . . . 144

B.2.1 Groupe *LTIGr* 144
B.2.2 Groupe *LTVGr* 146
B.2.3 Groupe *LTGrosG* 146

Table des figures

1.1 Processus de génération de tests. 5

2.1 Deux exemples de systèmes de transitions étiquetées à entrées et sorties. . . 13

3.1 Vérifications de VDM-Tools. 32
3.2 Une partie de la hiérarchie des exceptions JML. 40

4.1 Métamodèle du langage de synthèse de test. 47
4.2 Étapes de la génération de tests. 60

5.1 Processus de validation. 64
5.2 Diagramme de classes de l'application bancaire. 64
5.3 Extrait de la classe Currency_src. 67
5.4 Un exemple de test pour JUnit. 72

6.1 Filtrage des tests à la génération. 92

7.1 Arbre d'appels construit à partir du schéma `NewRdySwap`. 105
7.2 Élimination des fils d'un nœud ayant entraîné une erreur. 107
7.3 Arbre d'appels modifié. 108
7.4 Arbre d'appels modifié. 109
7.5 Architecture du pilote de tests. 110

8.1 Quelques exemples de groupes d'étudiants corrects et incorrects 112

A.1 architecture de Tobias sous forme de packages 139
A.2 modèle UML de la hiérarchisation des types dans Tobias 141

Liste des tableaux

2.1 Effets de l'oracle sur la validité du test. 17
2.2 Combinatoire sur trois paramètres binaires. 25

4.1 Grammaire du langage de schémas de test. 50
4.2 Grammaire des tests abstraits. 56

5.1 Quelques métriques sur l'application bancaire. 65

7.1 Temps d'exécution et nombre de tests éliminés à l'aide des divers pilotes de test. 108

8.1 Couverture de code et détection d'erreurs des suites de tests 125
8.2 Couverture de code et détection d'erreurs de la *suite 3* 130
8.3 Temps d'exécution et nombre de tests éliminés à l'aide des divers pilotes de test. 131

Première partie

Problématique et contexte

Chapitre 1

Introduction

Ce mémoire de thèse présente une approche de génération combinatoire de tests basée sur l'élaboration de schémas de tests. Afin de réduire les inconvénients des approches combinatoires, et en particulier de celle que nous proposons, nous présentons diverses techniques de sélection de tests. Nous étudions de plus des méthodes permettant d'optimiser et donc d'accélérer l'exécution des tests. Ce chapitre présente les motivations, les résultats obtenus ainsi que le plan du manuscrit.

Contexte général du test

La qualité des produits vendus dans le commerce, qu'il s'agisse de produits alimentaires ou de haute technologie, est soumise à un grand nombre de contrôles. Les clients sont de plus en plus exigeants et désirent acheter des produits de qualité. Ce qui est vrai pour les produits de consommation grand public l'est aussi pour l'industrie du logiciel, qu'il s'agisse de jeux vidéo ou de logiciels de contrôle aérien, la qualité tient une place importante. Évidemment, le niveau de qualité dépend d'un certain nombre de paramètres dont le niveau de criticité du logiciel. Quoi qu'il en soit, les industriels doivent aujourd'hui pouvoir fournir des produits ayant une certaine qualité sous peine de céder des parts de marché à la concurrence. Cependant, les applications actuelles ont des tailles très variables pouvant aller de quelques milliers de lignes de code pour des applications embarquées à plusieurs millions de lignes de code pour des progiciels. De même, les besoins en validation se trouvent très variés. Ainsi, des applications critiques mettant en jeu des vies humaines ou ayant un fort besoin de sécurité (applications bancaires, spatiales, aéronautiques...) auront besoin d'un très haut niveau de validation ; à l'inverse, le temps consacré à la validation d'applications moins sensibles, et donc son coût, se verra nettement réduit. Le challenge pour le monde industriel et pour celui de la recherche est donc de proposer des techniques et des outils permettant soit de réduire le coût de la validation à qualité constante, soit d'augmenter la qualité de la validation à coût constant. De grands progrès ont d'ailleurs été faits au cours de ces vingt dernières années. Au sein des deux grandes familles de validation que sont la preuve et le test, nombre d'outils et de méthodes ont vu le jour permettant de vérifier la qualité d'un produit en automatisant

tout ou partie du processus.

Le test logiciel repose sur deux grandes classes que sont le test statique et le test dynamique. Le test statique repose sur une étude du programme sans que celui-ci ne soit exécuté. Le test dynamique repose lui sur une exécution du logiciel testé. L'objectif du test dynamique est de stimuler l'application à l'aide d'un ensemble fini d'entrées et d'observer, pour chaque entrée, le comportement produit. Pour pouvoir établir un verdict (*pass, fail, inconclusive*) statuant sur le résultat du test, il est nécessaire de définir, pour chaque test, quels sont les résultats attendus. Une procédure, appelée oracle, doit ensuite permettre de comparer les résultats obtenus aux résultats attendus. C'est cette même procédure qui rend le verdict du test.

Il existe de nombreux problèmes inhérents au test dynamique. Nous pouvons citer, par exemple : comment produire automatiquement des tests ? comment nous assurer de la qualité des tests produits ? comment peut-on dire que l'on a fait suffisamment de tests ?

Le test dynamique peut se découper en plusieurs catégories selon la disponibilité du code source ou d'une spécification ou encore de l'étape du cycle de vie considéré. Les problèmes posés plus haut trouvent des réponses différentes en fonction du type de test dynamique que l'on souhaite exécuter. Nous nous sommes intéressé plus particulièrement au test de conformité et, plus précisément, au problème de l'automatisation du test.

Dans ce domaine, de nombreuses recherches ont déjà eu lieu qui ont donné naissance à divers outils. Beaucoup d'approches proposées reposent sur l'utilisation de spécifications formelles. Cependant, la spécification n'est pas toujours complète ou tout simplement présente au moment où l'on doit écrire les tests. Par ailleurs, il est difficile pour ces outils de couvrir l'ensemble des primitives du langage de spécification. Enfin, lorsque la spécification est très complexe, ces outils peuvent tomber dans le piège de l'explosion combinatoire et peuvent nécessiter une intervention humaine pour les aider à générer des tests en un temps borné.

Il existe d'autres techniques de génération qui ne s'appuient pas sur la spécification comme les techniques de génération combinatoire. Ces techniques permettent de produire des tests alors que la spécification formelle est absente ou n'est utilisée que pour servir d'oracle pour le test. Toutefois, les techniques combinatoires posent deux problèmes. D'une part, c'est à l'utilisateur de choisir lui-même les valeurs de test. D'autre part, si l'utilisateur définit un trop grand nombre de valeurs possibles pour un appel de méthode, si la méthode a beaucoup de paramètres ou si ses tests comportent de nombreux appels, il va tomber sur le problème de l'explosion combinatoire du nombre de tests générés.

Travail réalisé et contributions

Nous proposons dans ce travail une approche de génération combinatoire de tests à l'aide d'un langage de synthèse de tests basé sur les expressions régulières. De cette manière nous permettons à l'utilisateur de définir et de générer ses tests même lorsqu'il n'a pas de spécification formelle à sa disposition ou que celle-ci est trop incomplète pour permettre de produire automatiquement des tests. Comme nous le montrons figure 1.1, le langage de synthèse de tests s'articule autour de la notion de groupe de méthodes. Les groupes sont des constructions permettant de rassembler sous une même étiquette un ensemble d'appels de méthodes instanciés. L'utilisateur va pouvoir écrire des schémas de tests qui font appel à ces groupes. Le dépliage des schémas vers des tests abstraits se fait en combinant les éléments du groupe. Les tests abstraits sont des séquences, non exécutables, d'appels de méthodes instanciés. Ils peuvent alors être traduits vers différentes plates-formes d'exécution. Nous avons étudié dans ce travail la génération de tests pour des programmes Java et VDM. Nous avons, de plus, utilisé les langages JML et VDM-SL comme support pour l'oracle pour, respectivement, les programmes Java et VDM. En effet, un sous-ensemble de ces deux langages de spécification est exécutable. Dès lors, lorsque nous exécutons le programme sous test, nous exécutons aussi la spécification qui nous sert de référence et donc d'oracle.

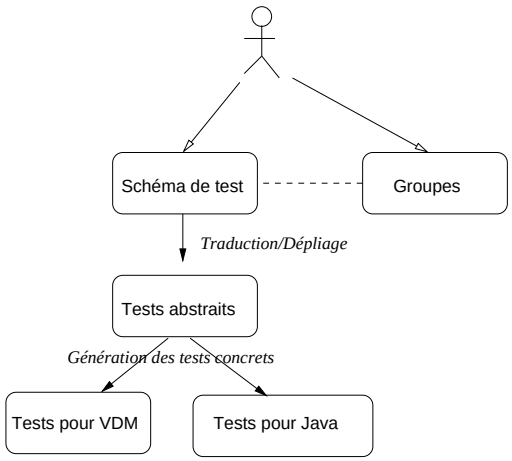


FIG. 1.1 – Processus de génération de tests.

Compte tenu des problèmes inhérents aux approches combinatoires, nous proposons différents mécanismes permettant à l'utilisateur de mieux maîtriser l'explosion combinatoire. Ces mécanismes sont définis suivant deux axes : d'une part des mécanismes permettant de contrôler le nombre de tests générés, d'autre part des mécanismes permettant de réduire le temps d'exécution en utilisant les informations recueillies lors de l'exécution des autres tests.

Les techniques de maîtrise de l'explosion combinatoire à la génération, que nous présentons ici, suivent trois axes :

1. la définition de contraintes portant sur les schémas de tests. Ces contraintes sont définies par l'utilisateur et permettent de sélectionner un sous-ensemble des tests produits par un schéma donné ;
2. de nouveaux constructeurs pour le langage de synthèse de tests. Ces constructeurs vont permettre de produire plus finement certains types de séquences de test ;
3. l'utilisation, lorsqu'elle est disponible, de la spécification formelle en couplant notre approche avec une approche de génération de tests à partir de spécifications. Nous utiliserons la programmation logique avec contraintes, à travers l'outil UCASTING, pour définir automatiquement des valeurs de tests et éliminer très tôt des tests qui ne sont pas conformes à la spécification.

Le filtrage à l'exécution permet de réduire le temps d'exécution en évitant de rejouer des séquences dont nous savons à l'avance qu'elles vont échouer (précondition, postcondition, invariant violé ou erreur de programmation). Certaines séquences ont des préfixes communs. Lorsque l'un de ces préfixes échoue, il est inutile de jouer les séquences ayant le même préfixe.

Nous validons sur deux études de cas, une industrielle et une universitaire, les principes et techniques présentés ci-dessus et dont la majeure partie a été instrumentée dans l'outil Tobias.

Plan du manuscrit

Le manuscrit se découpe en sept chapitres différents :

1. En premier lieu nous présentons un état de l'art du test de conformité. Nous présentons ainsi les différentes notions liées à ce type de test telles que : la notion de relation de conformité, d'hypothèse de test et d'oracle. De plus, nous nous intéressons aux différents outils de synthèse de test pour le test de conformité. Nous présentons des outils selon deux axes : génération à partir de spécifications orientées modèles et génération de tests combinatoire.
2. Nous présentons ensuite les deux langages de spécification que nous utilisons comme support pour l'oracle. Nous présentons en premier lieu VDM-SL, un langage de spécification pour des applications écrites en VDM. En second lieu, nous présentons JML qui permet de spécifier formellement des applications écrites en Java. Nous estimons que ces langages sont représentatifs des paradigmes de programmation classiquement utilisés dans l'industrie : actionnel et orienté objets.
3. Le troisième chapitre présente le langage de synthèse de test développé au cours du projet RNTL COTE. Ce langage permet de générer des tests de manière combinatoire, à partir d'une représentation synthétique (schéma de tests) des tests que l'on

souhaite obtenir. Ce langage permet de produire des tests exécutables aussi bien pour des programmes Java que pour des programmes VDM.

4. Le quatrième chapitre illustre l'utilisation du langage de synthèse de test sur une étude de cas Java développée par la société Gemplus. Nous proposons, à travers cette étude de cas, une méthodologie d'utilisation du langage de synthèse de test. Nous tirons, de plus, les conclusions sur l'utilisabilité du langage de synthèse de tests.
5. Nous proposons, dans le cinquième chapitre, un ensemble de techniques et d'outils permettant à l'utilisateur de mieux maîtriser le phénomène d'explosion combinatoire lié au langage de synthèse de tests. Nous proposons trois approches pour aider l'utilisateur. La première consiste à définir des contraintes portant sur les schémas. La seconde consiste à fournir d'autres opérateurs au niveau du langage de synthèse de tests. La dernière cherche à exploiter la spécification formelle lorsqu'elle est disponible.
6. Le sixième chapitre propose d'exploiter la structure des tests générés afin d'en optimiser l'exécution. L'idée est d'éviter d'exécuter des tests dont le verdict peut être connu à partir des résultats de l'exécution des autres tests.
7. Le septième chapitre illustre la mise en œuvre des différentes techniques de filtrage à travers une étude de cas VDM.

Chapitre 2

Contexte général

Le domaine du test logiciel est particulièrement vaste et riche en techniques, méthodes ou outils. Nous n'avons pas la prétention de pouvoir en faire le tour, aussi nous préférons présenter une vue très générale du test puis nous recentrer sur le sujet qui nous intéresse, à savoir le test de conformité.

Nous posons tout d'abord quelques notions élémentaires de test. À savoir, qu'est-ce que le test, à quoi cela sert-il et quelles en sont les limitations ?

Ensuite, nous faisons, plus en détail, l'état de l'art du test de conformité. Nous donnons la définition du test de conformité, exposons la notion d'hypothèse de test et le problème de l'oracle.

Enfin, nous présentons divers techniques et outils de synthèse de test. Il existe une gamme très large de techniques de synthèse de tests. Les techniques et outils que nous présentons sont centrés autour de deux axes : la génération de tests à partir de spécifications orientées modèles et la génération combinatoire de tests.

2.1 Test : quelques définitions

Dans tout développement logiciel il existe une étape de validation. Cette étape sert à vérifier que le logiciel possède certaines bonnes propriétés (application robuste, rapide, dont les fonctionnalités correspondent à un cahier des charges, etc.). Il existe plusieurs techniques permettant de valider un logiciel : la preuve, la vérification par model-checking ou encore le test. La preuve consiste à représenter formellement les propriétés attendues du système et à démontrer que ces propriétés sont vraies. Les techniques de model-checking visent à définir une représentation abstraite à partir de l'application et à vérifier que certaines propriétés sont bien présentes dans le modèle extrait de l'application et, par construction, dans l'application. Le test consiste à vérifier par l'expérimentation que l'application répond bien aux critères précédemment définis (robustesse, fonctionnalités correctement implantées, etc.). Le test peut donc s'utiliser de différentes manières : pour aider l'ingénieur à trouver les erreurs de son application ou pour certifier qu'une application répond à un certain nombre de critères identifiés par la suite de tests. Dans le premier cas, une bonne suite de tests est une suite de

tests qui trouve un maximum d'erreurs. Dans le second cas, la suite de tests sert à garantir un comportement minimum de l'application ; une bonne suite de tests est donc une suite de tests qui couvre un maximum de fonctionnalités.

Toute la problématique du test consiste à savoir quand nous pouvons nous arrêter : à quel moment sommes-nous assez complet pour dire qu'une application ne comporte pas d'erreurs ou pour dire qu'une suite de tests garantit une couverture suffisante des fonctionnalités ? Si nous prenons une simple opération ayant trois paramètres de type entier codés sur 32 bits, pour la tester de manière exhaustive, nous avons $2^{32} * 2^{32} * 2^{32} = 2^{96}$ combinaisons de valeurs de paramètres possibles. Les applications industrielles courantes comportent plusieurs milliers de fonctions ayant des paramètres pouvant eux-mêmes être des structures complexes, etc. Le nombre de tests nécessaires pour vérifier de manière exhaustive ce type d'application serait plus grand que le nombre de particules de l'univers. Il faut donc trouver des méthodes permettant d'obtenir, en un temps limité, un certain niveau de confiance dans l'application, même s'il nous est impossible d'obtenir une confiance totale. Diverses méthodes existent pour décider de l'arrêt du test. Citons, par exemple, les métriques à travers la couverture de code [Nta88, Bei90] ou encore le test de mutation [DLS78, OH96] qui vise à établir la qualité d'une suite de tests par rapport au nombre d'erreurs, artificiellement injectées, qui sont détectées par la suite de tests. Néanmoins, aucune des techniques employées pour définir un critère d'arrêt du test n'est infaillible et ne permet de garantir que l'application ne comporte pas d'erreur.

Dans le monde du test, il existe deux grandes familles :
- le test statique qui regroupe un ensemble de techniques permettant de trouver des fautes dans le logiciel sans l'exécuter (par exemple : la relecture de code) ;
- le test dynamique qui consiste à exécuter le programme et à vérifier son comportement (fonctionnalités correctement implantées, stabilité du produit, rapidité ...).

La famille qui nous intéresse dans ce document est le test dynamique. Plus précisément, nous nous focalisons sur le test de conformité. Lorsque nous avons à disposition un cahier des charges ou une spécification de l'application, nous pouvons vérifier si celle-ci est conforme à ce qui est spécifié. C'est ce que nous appelons test de conformité.

Si l'on considère spécifiquement le test dynamique, tester signifie exécuter l'application avec certaines entrées et, en fonction des réponses de l'application, décider si elle est ou non correcte. Si nous considérons un développement logiciel selon un cycle en V, à chaque niveau du développement il existe un niveau de test. Dans les couches les plus basses (développement du code source de l'application), une entrée consiste en un appel d'opération ou de méthode [1]. Dans les couches les plus hautes, il peut s'agir de vérifier que l'application s'installe correctement sur un système particulier, qu'elle supporte bien un certain degré de charge, etc. Le travail présenté dans ce manuscrit cible les couches "basses" du développement où une entrée consiste en un appel d'opération ou de fonction.

Plusieurs types d'erreurs sont possibles lors de la réalisation d'un programme. Les er-

[1] Ce type de test est communément appelé test unitaire.

reurs à l'exécution comme, par exemple, un débordement de tableau, un accès interdit à la mémoire. Il existe aussi des erreurs dites de conformité. Lorsque l'application possède une spécification, au moins sous la forme d'un cahier des charges rédigé en langue naturelle, nous pouvons effectuer du test de conformité. Cette spécification définit ce qu'est censé faire l'application, autrement dit, elle définit les comportements attendus par l'utilisateur lors de l'utilisation de l'application. Lorsque celle-ci répond d'une manière non conforme à la spécification, nous parlons d'erreur de conformité.

Le test peut aussi permettre de garantir que l'application ne contient pas de code mort. Il faut pour cela réussir à exécuter au moins une fois chaque instruction. En effet, le code mort est une portion du code de l'application qui ne pourra jamais être exécuté quelles que soient les valeurs de paramètres. Par exemple, dans `if(false) then exec1 ;` l'instruction `exec1` ne pourra jamais être exécutée. Le code mort est une mauvaise pratique de développement qui complique la maintenance des applications. Il doit être supprimé du code source.

Il existe de nombreuses techniques de tests visant à détecter ces différents types d'erreurs. Nous ne les présentons pas ici toutes en détail. Pour de plus amples informations, le lecteur peut se référer à [Bei90]. Nous pouvons regrouper ces techniques dans les deux grandes catégories de test :

- Le test en boîte noire, où le comportement interne de l'application sous test est inconnu de l'ingénieur de test. Ce type de test est aussi appelé test fonctionnel car l'ingénieur n'a accès qu'aux descriptions des fonctionnalités de l'application pour la tester.
- Le test en boîte blanche où la structure interne de l'application sous test est connue. L'ingénieur de test a ainsi accès à des informations comme la structure du code, les valeurs possibles de certaines variables du programme, etc.

Lorsque nous parlons de test en boîte noire ou en boîte blanche, nous faisons référence tant à la méthode de sélection des tests qu'à leur exécution.

Il existe une troisième catégorie de test qualifiée de test en boîte grise. Dans ce dernier cas, certains éléments de la structure peuvent être connus grâce, par exemple, à des points d'entrée ou à un certain niveau d'instrumentation de l'application sous test. Le type de test que nous présentons dans cette thèse entre dans cette troisième catégorie. En effet, nous utilisons des langages de spécification à base d'assertions. Ces assertions sont vérifiées lors de l'exécution des tests et peuvent, par exemple, porter sur les variables d'état du système ainsi que sur les paramètres des opérations. Elles agissent donc comme des points de contrôle et d'observation sans pour autant que nous ayons connaissance de la structure exacte du code de l'application sous test. Nous sommes donc bien en test dit "boîte grise".

2.2 Test de conformité

D'une manière générale, le test de conformité vise à s'assurer que l'implantation sous test est conforme à sa spécification. Ce type de test a été très étudié pour le test des protocoles et a fait l'objet de standardisations ISO [ISO91, ISO95]. L'idée principale de tous les travaux qui ont été effectués autour du test de conformité est de permettre, entre autre, la certification des protocoles à l'aide du test. Un cadre formel est défini, permettant d'exprimer la notion de conformité et les hypothèses de test. Deux problèmes importants sont ici abordés : d'une part le problème de la conformité et d'autre part le problème de l'explosion combinatoire liée à l'exhaustivité "nécessaire" à l'établissement de la conformité.

2.2.1 Quelques définitions

Avant d'aller plus avant dans l'étude de ce qu'est la conformité et du problème de la sélection des tests, nous présentons succinctement quels sont les formalismes principalement utilisés dans la littérature pour le test des protocoles. Cela permettra au lecteur de mieux appréhender les notions présentées plus loin.

Olivier Charles présente, dans sa thèse [Cha94], un large éventail des formalismes utilisés : les machines d'états finis, les systèmes de transitions étiquetées, les automates à entrées et sorties ou encore les systèmes de transitions étiquetées à entrées et sorties. Certains de ces formalismes servent à définir la sémantique de langages de plus haut niveau, par exemple, les systèmes de transitions étiquetées qui servent à définir la sémantique du langage LOTOS. Nous ne donnons que la définition des systèmes de transitions étiquetées car les autres n'apporteront rien de plus à la compréhension de ce qui suit.

Un système de transitions étiquetées (ou Labelled Transition System : LTS) est un quadruplet $< S, L, T, s_0 >$ où :
- S est un ensemble fini non vide d'états,
- L est un ensemble non vide d'interactions (étiquettes),
- T est la relation de transition définie par $T \subseteq S \times (L \cup \{\tau\}) \times S$ et,
- s_0 est l'état de départ.

Un système de transitions étiquetées à entrées et sorties (ou Input Output Labelled Transition System : IOLTS) est un cas particulier de celui présenté ci-dessus. Dans ce cas, L se décompose en deux sous-ensembles tels que $L = L_I \cup L_O \cup \{\tau\} \cup \{\delta\} \wedge L_I \cap L_O = \emptyset$ et où L_I et L_O sont respectivement l'ensemble des entrées (Input) et l'ensemble des sorties (Output). Le test fonctionnel dans le monde des protocoles a été essentiellement vu suivant un aspect "test boîte noire". Il ne s'agit pas de dire que les actions n'ont pas lieu, mais d'avoir une représentation permettant de comparer une spécification fonctionnelle et une application dont le comportement interne n'est pas toujours connu. Un modèle commun pour ces deux éléments (la spécification et l'application) doit se baser uniquement sur des éléments observables comme les entrées et les sorties. En conséquence, les actions internes ne

sont pas considérées et elles sont remplacées par des transitions τ. De plus, l'utilisateur peut vouloir spécifier quand il est normal qu'un système ne réponde pas dans un temps fini. Ce type de comportement, appelé un silence, est spécifié par une boucle $\circlearrowleft \delta$ sur le ou les états concernés. Deux exemples, tirés de [TB99], de ce genre de système sont présentés figure 2.1

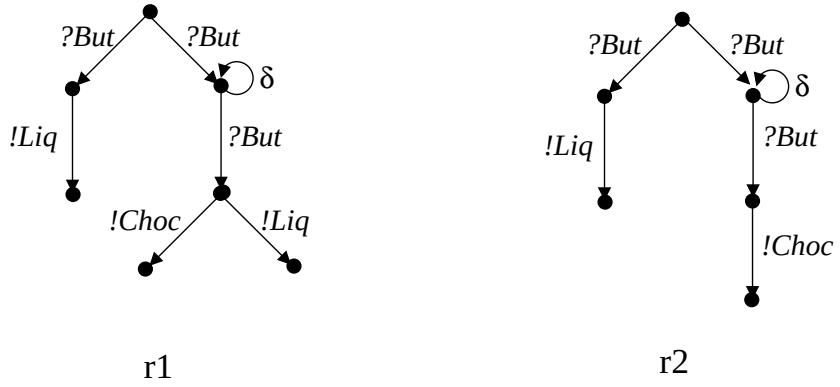


FIG. 2.1 – Deux exemples de systèmes de transitions étiquetées à entrées et sorties.

Dans cet exemple, les entrées commencent par ? et les sorties par ! ce qui nous donne : $L = \{?But, !Liq, !Choc\} \wedge L_I = \{?But\} \wedge L_O = \{!Liq, !Choc\}$. Sur ces exemples, lorsque l'on prend la transition de droite $?But$, la boucle δ sur l'état signifie que le système ne répond pas et qu'il attend que l'utilisateur actionne une nouvelle fois $?But$ pour changer d'état. Si l'utilisateur ne fait rien, le système attend.

2.2.2 Conformité

Comme nous l'avons dit précédement, le test de conformité a pour objectif de vérifier si une implantation est conforme à sa spécification. Il faut, néanmoins, nous poser la question de ce que signifie concrètement "est conforme à".

Dans [Pha94], il est dit : "Dans le modèle de comportement, les spécifications et les implantations sont représentées par une expression décrivant leurs comportements observables respectifs dans des langages de spécification formels (comme Estelle, LOTOS ou LDS), ou directement par des modèles opérationnels de ces comportements, comme les automates d'états-finis ou les systèmes de transitions. La conformité est alors définie comme une relation, dite relation d'implantation, entre les objets modélisant les implantations et ceux modélisant les spécifications."

Comme cela est montré dans [Tre92], il est possible de définir plusieurs relations de conformité. Tretmans met en avant le fait que, en fonction de la manière dont on spécifie son application, une même implantation pourra être ou non conforme à sa spécification. Plus précisément, en fonction du langage formel utilisé pour décrire les exigences informelles,

l'application sera ou non conforme à la spécification. Sur cette base, il définit trois types de langage, soit trois relations de conformité.

1. Le premier langage permet à l'utilisateur de décrire ce qui doit *au moins* être fait par l'implantation.
2. Le second permet de décrire ce que l'implantation doit *au plus* faire (une telle relation n'est pas vraiment viable car le programme vide sera toujours conforme).
3. Le troisième reprend la même idée que le précédent mais est plus précis puisqu'il vise à s'assurer qu'une implantation i ne peut jamais produire une sortie qui ne peut être produite par sa spécification s dans les mêmes circonstances, c'est-à-dire après la même séquence d'actions. Cette relation de conformité a ensuite été raffinée pour prendre en compte la notion de silence dans les protocoles [Tre96, TB99, Tre00]. Un silence signifie que l'implantation ne répond pas (cas d'un time-out) et n'est autorisé que si la spécification le permet. Cette relation de conformité sur les entrées/sorties, appelée **ioco**(input-output conformance relation), est formellement définie par :

   i **ioco** $s \Longleftrightarrow_{def} \forall \sigma \in Straces(s) : out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma)$
   où :
   - p **after** σ est l'ensemble des états atteignables par le système de transition p après avoir exécuté la séquence d'actions σ,
   - $out(p \textbf{ after } \sigma)$ est l'ensemble des actions de sortie qui peuvent se produire à partir d'un des états de p **after** σ,
   - $Straces(s)$ est l'ensemble des séquences d'actions d'entrée, de sortie ou de silence de la spécification s.

   i **ioco** s signifie que l'implantation i est conforme à la spécification s.

   Cette relation de conformité est illustrée figure 2.1. Considérant ces deux systèmes si $r2$ est un modèle de l'implantation et $r1$ sa spécification, nous avons $r2$ **ioco** $r1$, dans le cas contraire, $r1$ **i/oco** $r2$.

   Dans le premier cas, $out(r2 \textbf{ after } ?but) = \{!liq\} \subseteq out(r1 \textbf{ after } ?but) = \{!liq\}$ et $out(r2 \textbf{ after } ?but, \delta, ?but) = \{!choc\} \subseteq out(r1 \textbf{ after } ?but, \delta, ?but) = \{!choc, !liq\}$.

   Alors que dans le second cas, $out(r1 \textbf{ after } ?but, \delta, ?but) \nsubseteq out(r2 \textbf{ after } ?but, \delta, ?but)$. Elle établit que, dans une certaine mesure, l'implantation peut faire moins de choses que ce qui est spécifié. En fait, elle permet à l'implantation d'être déterministe là où la spécification ne l'est pas.

Il existe de nombreuses autres relations de conformité [Tre96, Led91] pour le test des protocoles. Toutes ces relations de conformité prennent au moins en compte la sémantique de la spécification. Néanmoins, souvent, la spécification seule ne définit pas le critère de conformité. Ainsi, ce critère est aussi défini en fonction du contexte, du type de système étudié et des hypothèses de test. La relation *ioco* est définie par la sémantique de la spécification sous forme d'IOLTS mais aussi par des contraintes supplémentaires qui sont posées sur les comportements observables.

2.2.3 Problème de l'exhaustivité

Une fois que nous avons identifié la relation de conformité, il reste à savoir comment l'établir. Autrement dit, comment sélectionner un ensemble fini de tests qui permette de garantir que l'application sous test est bien conforme à sa spécification ? Pour cela, deux définitions sont utiles :

1. une suite de tests est dite *valide* si et seulement si son succès permet de dire que le programme est correct, c'est-à-dire qu'elle rejette toute implantation incorrecte,
2. une suite de tests est dite *non biaisée* si et seulement si elle ne rejette aucun programme correct.

Si nous partons du principe que la suite de tests est *valide* et *non biaisée*, alors une suite de tests exhaustive permet de garantir la conformité. La création d'un ensemble exhaustif de test est évidemment impossible lorsqu'il s'agit de tester des applications industrielles. Aussi, [BGM91, Ber91] définissent un processus formel permettant d'obtenir un ensemble fini de tests ayant les mêmes propriétés qu'un ensemble exhaustif de tests.

Le principe de cette approche est de construire des données de tests à partir de spécifications formelles. Elle comporte, par ailleurs, un processus de sélection d'un ensemble de tests par raffinement successifs d'un contexte de test, garantissant les mêmes propriétés que l'ensemble de tests exhaustif.

Un contexte de test est un triplet (H, T, O) avec :
- H est un ensemble d'hypothèses portant sur le programme P,
- T est un ensemble de tests et, chaque τ appartenant à T correspond, dans notre cas, à une séquence de test comprenant les entrées et les sorties.
- l'oracle O est un prédicat partiel défini sur T qui prend en entrée n'importe quelle séquence de test τ et, si $O(\tau)$ est défini, permet, lorsque le test τ est soumis au programme P, de décider si celui-ci réussit ou échoue.

L'objectif des hypothèses de test est de réduire la taille de l'ensemble des implantations possibles (correctes et incorrectes). En effet, comme nous considérons un nombre moins important d'implantations pouvant satisfaire ou non la spécification, nous avons besoin de moins de tests pour vérifier la conformité de l'implantation sous test. Si nous faisons suffisamment d'hypothèses, un nombre raisonnable de tests suffit à garantir la conformité de l'application sous test. Il faut néanmoins être prudent car, si le programme ne satisfait pas les hypothèses posées, il n'est plus possible de la garantir.

Pour sélectionner un ensemble fini de tests garantissant les mêmes propriétés qu'un ensemble exhaustif de tests, il faut partir d'un contexte de test appelé *contexte de test canonique*. Ce contexte de test est composé d'une hypothèse de test minimale disant que le programme implante toutes les opérations spécifiées, autrement dit, les signatures du programme et de la spécification sont identiques. L'ensemble des tests initialement considéré est l'ensemble de tests exhaustif. L'oracle est défini par le prédicat partiel *undef*.

Soit $TC_1 = (H_1, T_1, O_1)$ et $TC_2 = (H_2, T_2, O_2)$ deux contextes de test tels que TC_2 raffine TC_1. Le raffinement consiste en trois points :

- Renforcement des hypothèses de tests soit, $H_2 \Rightarrow H_1$.
- Sous les hypothèses H_2, toute erreur détectée par T_1 doit être détectée par T_2.
- La relation partielle définissant l'oracle peut être construite pas à pas en augmentant progressivement son domaine de définition tel que toute erreur révélée par l'oracle de départ soit aussi révélée par l'oracle raffiné. Ainsi, sous les hypothèses H_2, $dom(O_1) \subseteq dom(O_2)$ et, $\forall \tau \in dom(O_1) \bullet O_2(\tau) \Rightarrow O_1(\tau)$.

Prenons par exemple une fonction $f(x)$ où $x \in \mathbb{Z}$. La première hypothèse $H1$, que nous pouvons faire, est que le domaine d'entrée de la fonction f possède une certaine uniformité, à savoir que le domaine de x peut se découper en dix domaines : $x \in [-Maxint..-Maxint+1]$, $x \in]-Maxint+1..-Maxint/2]$, $x \in [-1..1]$ etc. Il nous faut donc un test par domaine, soit dix tests. Afin de réduire le nombre de tests nécessaires, nous allons poser une hypothèse d'uniformité $H2$ plus forte disant que trois domaines peuvent être définis pour x, à savoir $x < -1$, $x \in [-1..1]$ et $x > 1$. Si l'hypothèse $H2$ est vraie ie, si 3 trois tests suffisent, alors l'hypothèse $H1$ est vraie.

L'objectif est d'effectuer un nombre suffisant de raffinements afin d'obtenir un contexte de test (H, T, O) dit *praticable*. Un tel contexte possède un ensemble de test T fini de taille raisonnable. De plus, pour chaque test dans T, l'oracle O est défini et décidable. Dès lors, si le programme satisfait l'ensemble des hypothèses de test H, la conformité entre le programme et sa spécification est établie par l'exécution et l'évaluation de l'ensemble des tests de T.

2.2.4 Problème de l'oracle

Nous avons rapidement défini dans la sous-section précédente ce qu'était l'oracle dans le monde du test. En toute généralité, l'oracle est l'élément qui va permettre de juger, en fonction du résultat du test, si l'implantation satisfait ou non la relation de conformité entre l'implantation et la spécification. Ainsi, dans le test de conformité, le rôle de l'oracle est de juger si les sorties produites par le logiciel sous test sont conformes aux sorties attendues. Cette vérification est coûteuse en temps puisqu'il faut vérifier pour chaque cas de test que les résultats correspondent aux résultats attendus. Un des moyens de gagner du temps est donc d'automatiser le plus possible cette tâche. Toutefois, cette automatisation n'est pas sans poser des problèmes : comment vérifier automatiquement que la sortie produite est satisfaisante ? Dans [BY01], les auteurs résument les différentes études menées pour résoudre ce problème. Au final, l'ingénieur en charge du test reste le seul vrai juge.

L'un des moyens les plus naturels pour définir un oracle automatique est l'utilisation de méthodes formelles qui soient suffisamment outillées pour permettre de comparer les résultats automatiquement. Comme nous avons pu l'établir précédemment, le test de conformité vise à s'assurer que l'implantation est conforme, suivant une certaine relation, à une spécification de référence. Cette relation de conformité est liée au formalisme utilisé pour décrire la spécification. En conséquence, l'oracle, qui sert à juger si cette relation est vérifiée ou violée, est lié au formalisme utilisé pour décrire la spécification ainsi qu'au critère de

conformité.

Problème des résultats biaisés.

Etablir un oracle automatique, valide, c'est-à-dire qui rejette toute implantation incorrecte, et non biaisée, c'est-à-dire qui accepte toute implantation correcte, est un problème très difficile. Dans [Bin99], l'auteur décrit les effets de l'oracle sur la validité du test. Reprenons son exemple :
Imaginons que l'application doive normalement effectuer l'action `x = y+z` où `y` et `z` sont deux entrées de type entier. Pour automatiser la vérification, l'oracle est supposé simuler ou calculer la même chose. Néanmoins l'oracle tout comme l'implantation peuvent contenir des erreurs. La table 2.1 reprend une partie de l'illustration[2] présente dans l'ouvrage. Nous pouvons ainsi noter qu'en fonction de l'implantation, de la spécification utilisée comme oracle, qui peut, elle aussi, être erronée, et des entrées choisies, les résultats peuvent être trompeurs voire totalement faux. Reprenons les cas présentés dans le tableau :

- Dans le premier cas, l'application et l'oracle sont corrects. Quelque soit le test choisi, nous aurons un *Pass* qui aura du sens.
- Dans le second cas, l'application est correcte alors que l'oracle est erroné. Le test qui est exécuté va lever une erreur de conformité. Nous obtenons alors un *Fail* qualifié de faussé car il est dû à une erreur dans l'oracle et non pas dans l'application.
- Dans le troisième cas, l'application contient une erreur mais cette fois, l'oracle est correct. Le test choisi permet de détecter un problème de conformité entre l'oracle et l'application. Comme l'oracle est ici correct, nous qualifions le *Fail* de correct.
- Enfin, dans le dernier cas, l'application et l'oracle sont incorrects. Le test choisi ne permet pas de lever d'erreur de conformité c'est pourquoi nous qualifions ce *Pass* de résultat faussé. Ajoutons qu'ici, même si l'oracle avait été correct, ce test n'aurait pas permis de détecter l'erreur.

Ceci nous amène donc à penser qu'accroître le nombre d'entrées ou de combinaisons d'entrée va limiter le risque de générer des résultats biaisés à cause du choix des valeurs, et par conséquence, va permettre à l'utilisateur d'accroître sa confiance dans l'application et dans l'oracle.

Problème de l'automatisation de l'oracle.

Lorsque nous utilisons une spécification formelle comme oracle, nous nous retrouvons confronté au problème de son évaluation automatique. En effet, l'oracle, pour être vraiment utile, doit pouvoir être automatiquement évalué. Ce point soulève deux problèmes liés : celui du lien entre abstraction et monde concret et, celui de l'observabilité.

Le problème du lien entre abstraction et implantation se pose lorsque la spécification utilise des types de données très abstraits comme par exemple avec les spécifications algébri-

[2]Dans [Bin99], neuf cas différents d'interaction sont définis.

Application		Oracle		Test		Résultats			Validité du test
				y	z	requis	attendu	obtenu	
Correct	`y+z`	Correct	`y+z`	5	3	8	8	8	*Pass* correct
Correct	`y+z`	Incorrect	`y*z`	5	3	8	15	8	*Fail*, mais verdict faussé
Incorrect	`y-z`	Correct	`y+z`	5	3	8	8	2	*Fail* correct
Incorrect	`y-z`	Incorrect	`y-z`	0	0	0	0	0	*Pass*, mais verdict faussé

TAB. 2.1 – Effets de l'oracle sur la validité du test.

ques. Il peut alors être difficile de faire le lien entre l'abstraction et les types de données concrets [Gau95, GA96, BY01] pour procéder automatiquement à l'évaluation de l'oracle.

Prenons par exemple un système de gestion de pile. Nous avons à notre disposition diverses opérations :

- `push` est une opération qui prend un élément en paramètre et le met au sommet de la pile,
- `pop` est une opération qui retire l'élément situé au sommet de la pile de celle-ci,
- `top` est une fonction qui renvoie l'élément situé au sommet de la pile et,
- `height` qui est une fonction qui renvoie la hauteur de la pile.

Supposons maintenant que cette pile soit concrètement implantée par un tableau d'éléments. `push` va donc placer l'élément dans le tableau à l'indice égal à la hauteur de la pile, puis va incrémenter la hauteur de 1. `pop` se contente de diminuer la hauteur de la pile de 1. Nous poserons comme postulat qu'initialement, toutes les cellules du tableau sont vides.

Soit `pop(push(2,push(3,push(4,empty))))` un test. De la spécification, nous pouvons déduire que :
`pop(push(2,push(3,push(4,empty))))` = `push(3,push(4,empty))`.
Mais que signifie, dans le monde concret, cette égalité ? En effet, la description rapide du fonctionnement des opérations nous permet de déduire qu'après ce test, le tableau contiendra l'élément 4 en position 1, l'élément 3 en position 2 et que la hauteur sera de 1. Or, est-ce réellement la même chose qu'une pile contenant un élément du point de vue de la spécification ? Nous voyons ici que l'état initial représentant une pile vide est un tableau dont toutes les cellules sont vides alors qu'après le cas de test, le tableau contient deux éléments. Il s'agit donc d'un problème de transposition d'une égalité abstraite en égalité concrète. Résoudre ce problème est loin d'être trivial et l'utilisateur doit choisir avec attention la manière dont il va traduire cette égalité dans le monde concret. Concernant notre exemple, nous pouvons évaluer cette égalité comme :

```
height(pop(push(2,push(3,push(4,empty))))) = height(push(3,push(4,empty))) ∧
top(pop(push(2,push(3,push(4,empty))))) = top(push(3,push(4,empty))) ∧
top(pop(pop(push(2,push(3,push(4,empty)))))) = top(pop(push(3,push(4,empty))))
```

Nous vérifions ainsi que les piles sont bien identiques en comparant leur hauteur et

les éléments des piles deux à deux. Nous venons de définir un *contexte observable* permettant de comparer deux piles entre elles. Cependant, ce *contexte observable* fait appel aux opérations de manipulation des piles, celles-là même que nous sommes en train de tester. En conséquence, si ces opérations contiennent des erreurs, nous risquons de fausser le verdict.

Le problème de l'observabilité [BGM91, Ber91] est finalement à l'origine des problèmes d'automatisation de l'oracle. En effet, l'accès à la structure interne de l'application, en test fonctionnel, est un point délicat. Dès lors, si nous voulons vérifier que l'exécution des opérations s'est bien déroulée, il nous faut trouver un moyen de vérifier l'état du système. Comme nous venons de l'illustrer, il est nécessaire, pour pouvoir rendre un verdict, d'utiliser les opérations fournies par le système : le fait d'instrumenter le code par des ajouts de fonctions est ici exclu car, dans la théorie proposée, l'hypothèse de test minimale est que le système et la spécification implantent le même ensemble d'opérations/fonctions. Dès lors, si une erreur est introduite dans l'une des opérations utilisée pour accéder à l'état du système, dans notre exemple : top, le verdict risque d'être faussé.

Pour conclure, l'oracle constitue, comme nous pouvons le voir, un problème difficile à résoudre. Certains problèmes comme la confiance attribuée à la spécification et/ou aux tests peuvent être partiellement résolus par une plus grande diversité de tests et de valeurs de tests. Nous voyons bien ici qu'établir la relation de conformité ne se limite pas au test de l'application. Implicitement, c'est le couple spécification/implantation qui est testé. Il est difficile d'avoir une confiance très forte dans la spécification, à moins que celle-ci n'ait fait l'objet d'un processus de validation poussé, notamment grâce à la preuve. Le problème du lien entre les deux mondes, abstrait et concret, reste entier et difficile à résoudre. C'est pourquoi nous avons choisi d'utiliser les langages de spécification VDM et JML comme support pour l'oracle. En effet, ces deux langages offrent des mécanismes et des constructions qui permettent d'automatiser ce processus de "concrétisation" puisque, pour chacun de ces langages, un sous-ensemble est directement exécutable. De plus, JML offre des mécanismes permettant à l'utilisateur de faire le lien entre un modèle et une implantation. Enfin, le problème de l'observabilité peut se résoudre de plusieurs manières. Nous en avons exposé une : le fait d'utiliser les fonctions/opérations du système sous test pour vérifier son état. Mais, si cette approche est en adéquation avec la théorie proposée, elle est très risquée. En effet, si l'une des opérations utilisées pour vérifier l'état est erronée, le verdict est faussé. Une autre façon est de penser au test dès le début de la conception du logiciel et de rendre l'application plus testable en permettant plus de transparence pour vérifier l'état du système sans trop de risques. Une troisième, que nous avons choisi d'utiliser dans cette thèse est l'utilisation de contrats. Les contrats peuvent être vus comme des points d'observation portant sur des éléments internes du système. Cette approche s'intègre parfaitement avec les deux langages de spécification que nous avons choisis pour définir l'oracle. En effet, ils possèdent tous deux des outils qui permettent d'instrumenter le code grâce à la spécification.

2.3 État de l'art sur la synthèse de test

Il existe de nombreux outils et techniques de synthèse de tests. Certains s'appuient sur des critères structurels [GBR98, Got00], d'autres s'appuient sur les spécifications algébriques [Mar95], comportementales [JM99, LG01, dBORZ99, MA00, CLRZ99] ou orientées modèles [DF93, HHS03]. Il existe aussi des techniques de génération de tests aléatoire [dBORZ99] et pour finir des techniques combinatoires [CL02b]. Nous ne présentons pas toutes ces techniques en détail. Compte tenu du fait que nous nous sommes principalement intéressé à la génération de tests pour les langages VDM et JML, nous présentons ici les techniques de génération à partir de spécifications orientées modèles et les techniques de génération combinatoire.

2.3.1 Génération de tests à partir de spécifications orientées modèles

Les outils de génération de tests à partir de spécifications orientées modèles [3] utilisent pour une large majorité la programmation logique avec contraintes ensemblistes. Cette technique est basée sur la résolution de systèmes de contraintes logiques. De manière simplifiée, les opérations sont définies sous la forme d'un système d'équations logiques liant leurs entrées et leurs sorties. Il suffit ensuite de résoudre automatiquement ledit système pour trouver des ensembles de valeurs d'entrées/sorties. Évidemment ces techniques sont loin d'être triviales. Tout d'abord il n'est pas simple de définir le système d'équations, ensuite lorsqu'on y arrive leur résolution peut s'avérer très coûteuse en temps et en ressources.

J. Dick et A. Faivre

Plusieurs outils de génération automatique de tests à partir de spécifications formelles et à l'aide de la programmation logique avec contraintes s'appuient sur les travaux de J. Dick et A. Faivre [DF93].

Dans ces travaux, les auteurs proposent, en mettant sous forme normale disjonctive une spécification écrite en VDM, de partitionner les domaines des entrées des opérations. Chaque opération se voit donc découpée en sous-opérations (une par disjonction). Comme les spécifications des opérations portent aussi sur l'état du système (entre autres à travers un invariant), l'état du système est implicitement découpé en domaines correspondant aux états atteignables à partir des sous-opérations. Il est alors possible de construire un automate d'états fini où les transitions sont les sous-opérations et les états de l'automate sont des partitions de l'état du système. Le processus de construction de l'automate n'a pas fait

[3] Nous entendons, par spécifications orientées modèles, les spécifications engendrées par des langages comme Z [Spi92], B [Abr96], VDM [Jon90] ou JML [LBR99].

l'objet d'une automatisation car il est très coûteux (explosion du nombre d'états) et que potentiellement certains états ne sont pas atteignables.

Casting

L. Van Aertryck a développé une méthode de génération de tests générique [VBL97, Van98]. Cette méthode appelée Casting permet de générer des tests fonctionnels à partir de n'importe quel formalisme qui peut être traduit sous la forme d'un ensemble de contraintes logiques. Elle a été appliquée à la méthode B (B-Casting) et à la notation UML (UML-Casting [VJ03]).

L'utilisateur doit fournir à l'outil des règles permettant de traduire le formalisme de départ en prédicats pré/post qui sont exprimés dans un langage interne à Casting. L'utilisateur fournit aussi un jeu de règles de décomposition de ces prédicats. Il peut, par exemple, utiliser la stratégie de décomposition proposée par J. Dick et A. Faivre.

Comme avec la méthode de J. Dick et A. Faivre, cette décomposition des prédicats permet de découper chaque fonction ou opération en un ensemble de sous-fonctions appelées cas de tests. Un graphe de tests est alors construit à partir de ces cas de tests. Les états représentent des instanciations symboliques de l'état du système et les transitions sont des appels instanciés des sous-fonctions.

Ce processus de construction se fait de manière itérative. À partir de l'opération d'initialisation, l'outil construit le premier état du système. Il cherche ensuite les sous-fonctions qui peuvent s'appliquer, instancie les valeurs des paramètres et, à partir de la postcondition, calcule l'état symbolique suivant. Un solveur de contraintes est utilisé pour construire ce graphe. Lorsque le solveur ne trouve pas seul la solution, il peut demander l'aide de l'utilisateur pour qu'il la lui fournisse. Lorsque certains états ne sont pas atteignables ou que le nombre d'états est trop grand, le processus de génération peut ne pas terminer. Pour résoudre ce problème, l'utilisateur peut fixer une limite de calcul soit en temps, soit en taille.

Une fois l'automate construit, un algorithme sélectionne un ensemble de chemins couvrant l'ensemble des transitions. Cet algorithme cherche à construire des séquences les plus courtes possibles afin de limiter les répétitions d'appels à chaque sous-fonction. L'outil produit ainsi des séquences de test.

BZ-TT

La méthode BZ-TT [LPU02, Peu02, LP02] permet, comme B-CASTING, de synthétiser des tests à partir de spécifications B [Abr96] ou Z [Spi92]. Tout comme Casting, cette méthode s'appuie sur la programmation logique avec contraintes ensemblistes.

L'objectif de cette méthode repose sur deux points : créer des tests permettant de passer par des états limites et, une fois dans un état limite, d'appeler une opération avec des valeurs de paramètres limites.

- Pour ce faire, l'outil partitionne le domaine des variables d'état de la spécification pour en extraire des valeurs aux limites. Ainsi, si nous avons une variable d'état x dont le domaine est partitionné en trois : $x < -1$, $x \in -1..1$ et $x > 1$, l'outil cherchera à produire des tests passant par les états où x vaut : $MININT$, -2, -1, 1, 2 et $MAXINT$. Un état limite est un état dont au moins l'une des variables d'état est assignée à une de ses valeurs limites.
- La séquence permettant d'atteindre un état limite est appelée préambule. L'opération appelée juste après avoir atteint l'état limite est appelée corps du test. Cette opération a comme particularité d'être appelée avec des valeurs de paramètres, eux aussi, aux limites. Ces valeurs limites sont les extrema et minima des partitions des domaines des variables d'entrée de l'opération testée.

Un solveur spécifique, CLPS-B [PLT00, BLP02], a été développé pour traiter les particularités du langage B. Ce solveur est utilisé pour transformer la spécification en un système de contraintes équivalent, partitionner les domaines des variables d'états de la spécification afin d'obtenir des ensembles de contraintes aux limites et, génèrer des valeurs aux limites à partir de ces partitions.

Une fois le partitionnement effectué, un parcours du graphe d'atteignabilité est effectué à la volée au fur et à mesure de son exploration, grâce au solveur CLPS-B qui permet, depuis l'état courant, de simuler l'exécution (contrainte) des opérations activables. Cette technique permet de traiter des spécifications dont le nombre d'états est très élevé et pour lesquels un calcul exhaustif s'avère impraticable.

Le problème de la détection de l'atteignabilité d'un état repose sur deux traitements appelés test partiel d'atteignabilité :

- chacun des états à atteindre doit vérifier l'invariant. Cette condition, dans le cadre d'une spécification B prouvée (dans lequel se placent les auteurs), est nécessaire pour attester de son atteignabilité, mais n'est pas suffisante ! En effet, tous les états atteignables de la spécification préservent l'invariant, mais tous les états consistants aux vues des propriétés invariantes ne sont pas forcément atteignables.
- pour pallier ce second problème, la recherche du préambule du test est bornée. Ainsi, l'exploration du graphe est bornée en terme de profondeur maximale (la séquence d'opérations constitutant le préambule est limitée à N opérations), et en terme de temps de calcul (un temps maximum est allouée pour chaque recherche de préambule).

Il n'est pas toujours possible de trouver des préambules pour tous les états aux limites (y compris pour des états aux limites pourtant atteignables). L'outil propose alors à l'utilisateur de donner une séquence d'opérations (un préambule) qui permette d'atteindre cet état.

Nous pouvons constater que les trois techniques présentées ici comprennent un oracle basé sur la spécification abstraite, qui permet la génération des tests. En revanche, il faut procéder

à la réification des tests abstraits produits au niveau d'abstraction de la spécification, en tests concrets qui seront exécutés sur le programme à tester [BL03].

2.3.2 Génération combinatoire de tests

Dans le domaine de la génération de tests, une manière simple de générer un grand nombre de tests est d'utiliser des techniques combinatoires. Ces techniques, contrairement à la génération de tests à partir de spécifications, se basent uniquement sur la signature des opérations (leurs entrées) pour générer des tests. Évidemment, elles souffrent d'un gros problème : l'explosion du nombre de tests.

JMLUnit

Basé sur le framework de test JUnit [JUn], JMLUnit [CL02b] est un outil de génération combinatoire de tests unitaires pour les programmes Java. La conception de cet outil se place dans une optique "extreme programming" ou les développeurs font des tests unitaires (test d'un appel de méthode) au fur et à mesure qu'ils développent. Comme le code source de l'application évolue continuellement, la maintenance des tests est particulièrement lourde et pénible.

L'outil propose donc de faciliter ce travail en se basant sur deux éléments. Tout d'abord, l'existence d'une spécification JML [LBR99, LBR02] qui pourra servir d'oracle pour le tests grâce à un outil permettant d'exécuter la spécification en même temps que l'application [Bho00, CL02a]. Ensuite l'outil JMLUnit offre aux utilisateurs des facilités de création de tests.

L'outil de génération de tests permet une maintenance aisée des tests en générant automatiquement des tests exécutables avec JUnit. L'utilisateur peut se concentrer sur les aspects créatifs du test limités ici à la sélection des valeurs de tests et à la définition des états initiaux. L'outil permet de définir le contexte d'appel de la méthode testée. L'utilisateur configure, par des appels de méthodes et de constructeurs, l'état dans lequel sera appelée la méthode. Il définit ensuite des tableaux de valeurs pour les différents paramètres de la méthode sous test. L'outil génère, en faisant le produit cartésien des différentes valeurs, l'ensemble des appels à la dite méthode. Il génère un fichier de test JUnit contenant tous les cas de tests ainsi générés. Néanmoins, cette méthode ne génère que des tests constitués d'un unique appel de méthode.

Lors de l'exécution des tests, la spécification JML est évaluée et sert d'oracle. Son utilisation permet de simplifier la maintenance des tests. En effet, lorsque l'on utilise JUnit seul, les valeurs attendues doivent être définies à la main. L'utilisateur doit donc écrire, pour chaque test, les valeurs attendues. De plus, lorsque la structure des résultats change, il faut corriger tous les tests précédemment écrits. L'utilisation de JML permet de centraliser les

modifications, d'éviter les oublis, et d'offrir à l'utilisateur la possibilité d'écrire plus de tests puisqu'il n'a pas à se soucier de leur maintenance.

UniTesk

UniTesk [Kul04, BKKP02, KKP02] est un framework de test qui se veut générique, c'est-à-dire applicable à n'importe quel type d'application. Il n'est pas très clair de savoir ce qui est construit à la main par l'utilisateur et ce qui est produit automatiquement par l'outil mais l'idée générale est la suivante : le framework se base sur une machine d'états finie représentant le système sous test. Des critères de couverture sont ensuite utilisés pour partitionner les domaines des entrées de l'application.

Étant donnés ces partitions et le modèle du système, l'objectif est de passer au moins une fois par chaque transition et de tirer au moins une valeur dans chaque partition. Pour cela, l'utilisateur doit construire un itérateur qui, étant donné un état, va produire un symbole, c'est-à-dire un appel de fonction. Pour construire cet itérateur, l'utilisateur doit définir toutes les entrées possibles de chaque fonction, ou donner au moins une entrée par partition. Il doit de plus définir manuellement l'algorithme de parcours de la machine d'états finie.

L'outil utilise ensuite une spécification exécutable, au moins en partie, pour produire l'oracle du test.

Cet outil possède donc un aspect combinatoire pour ce qui est de la production des entrées. Cet aspect est pondéré par le fait de vouloir exercer un nombre fini de transitions et au final obtenir une certaine couverture de l'application. Le framework utilise de plus des spécifications formelles (par exemple décrites en VDM-SL) afin de produire un oracle automatique pour le test.

Korat et TestEra

Korat [BKM02] et TestEra [MK01] sont deux outils de génération automatique de tests visant à tester des opérations manipulant des structures de données complexes en Java. Tout comme JMLUnit, Korat et TestEra adoptent une génération combinatoire de tests unitaires. L'originalité de ces deux outils réside dans le fait qu'ils construisent de manière semi-automatique des données de test qui sont des structures Java, telles qu'un arbre binaire, un tableau, etc.

TestEra et Korat utilisent tous deux un langage de spécification permettant de spécifier, pour chaque opération testée, d'une part les entrées valides, d'autre part l'oracle.

TestEra utilise le langage Alloy [JSS01] et son analyseur de contraintes ALCOA [Jac00, JSS00]. L'utilisateur spécifie en Alloy des invariants structurels qui permettent à ALCOA de générer des instances valides de la spécification et donc des structures de données. ALCOA a besoin pour cela d'une borne définie par un nombre d'atomes, c'est-à-dire un nombre fini d'éléments constituant la structure de données. Par exemple on va demander à ALCOA de

générer un arbre comportant trois nœuds différents. L'analyseur de contraintes va, à partir de cette donnée et des différentes valeurs de nœuds possibles, générer toutes les structures de données non isomorphes. Dans notre exemple, il va générer tous les arbres non isomorphes comportant trois nœuds. Ce même langage est utilisé pour spécifier les résultats attendus, autrement dit, l'oracle.

Les langages Alloy et Java étant différents, l'outil TestEra requiert une fonction de concrétisation et une fonction d'abstraction. La première permet de traduire les données d'entrée générées par ALCOA en Alloy en des instances Java utilisables par la méthode sous test. La fonction d'abstraction fait le travail inverse et permet de traduire en Alloy le résultat du test exécuté en Java. Une fois le résultat traduit en Alloy, ALCOA permet de vérifier, en fonction des paramètres d'entrée, la correction du résultat par rapport à la spécification.

Korat reprend la même idée que TestEra mais intègre le tout directement en Java. Pour cela, l'utilisateur doit décrire en Java l'invariant structurel permettant d'identifier les instances valides utilisées en paramètre. Là aussi, l'utilisateur doit borner le nombre d'atomes de la structure et préciser les valeurs que ceux-ci peuvent prendre. À partir de ces informations, Korat génère toutes les instances, non isomorphes, d'entrées valides. Par exemple, pour le test d'une fonction permettant de supprimer un élément passé en paramètre d'un arbre binaire de taille bornée, Korat génère tous les couples $(arbre, element)$ tels que *arbre* est un arbre binaire de taille bornée et *element* est présent dans l'arbre. Si Korat a à sa disposition une spécification JML de la méthode sous test, il peut vérifier que le résultat obtenu satisfait bien la spécification. Si aucune spécification n'est donnée, Korat vérifie au moins que l'invariant structurel est satisfait.

AETG

Le principe de l'outil AETG System [CDFP97] est de ne pas couvrir toutes les combinaisons possibles de tous les paramètres mais de couvrir toutes les combinaisons de couples de paramètres en un minimum de tests. AETG se base sur l'hypothèse suivante : une erreur résulte d'une combinaison d'au plus deux paramètres d'entrée. En conséquence, une suite de test est pertinente si, pour chaque valeur de chaque paramètre, celui-ci est associé au moins une fois avec chaque valeur des autres paramètres. Cette hypothèse est vérifiée si l'apparition d'une erreur ne dépend que d'une combinaison particulière de deux des paramètres et si cette erreur est indépendante des valeurs des autres paramètres. En général la mise en œuvre de cette technique se fait sur la base de considérations empiriques ou pragmatiques. Cette hypothèse permet de réduire considérablement le nombre de configurations de valeurs à prendre en compte. AETG est configurable et il permet de considérer des couples, des triplets, des n-uplets, etc.

Le tableau 2.2 décrit trois paramètres A, B et C qui peuvent chacun prendre les valeurs 0 ou 1. Il existe huit combinaisons possibles de valeurs entre ces trois paramètres. Si nous souhaitons couvrir seulement les combinaisons deux à deux des paramètres, nous pouvons

dans ce cas nous contenter des lignes 2, 3, 4 et 5. Nous aurions aussi bien pu considérer les lignes 1, 4 , 6 et 7.

	sans interactions			interactions 2 à 2		
Ligne	**A**	**B**	**C**	**AB**	**BC**	**AC**
1	0	0	0	-	-	-
2	0	0	1	00	01	01
3	0	1	0	01	10	00
4	0	1	1	-	-	-
5	1	0	0	10	00	10
6	1	0	1	-	-	-
7	1	1	0	-	-	-
8	1	1	1	11	11	11

TAB. 2.2 – Combinatoire sur trois paramètres binaires.

K. Burr et W. Young [BY98] présentent l'exemple d'une méthode avec 13 paramètres pouvant tous prendre 3 valeurs différentes. L'ensemble des combinaisons représente 1 594 323 cas de test. La couverture par paires d'AETG permet de réduire le nombre de cas de test nécessaires à seulement 19 cas de test.

2.3.3 Synthèse

Nous proposons ici d'utiliser une approche de génération combinatoire de tests pour générer des séquences d'appels d'opérations. Afin de pallier le problème de l'explosion combinatoire, nous proposons aussi diverses techniques permettant de maîtriser cette explosion que ce soit à la génération ou lors de l'exécution des tests. Nous proposons d'utiliser des spécifications exécutables afin d'obtenir un oracle automatique. La génération des tests n'étant pas directement liée à la précision de la spécification, l'utilisateur pourra spécifier de manière très précise les parties critiques de son application et produire une spécification plus légère à d'autres endroits.

Nos motivations sont les suivantes :

1. Pouvoir générer des tests sans avoir besoin d'une spécification formelle, en nous basant sur l'expérience de l'ingénieur de test.
2. Utiliser une spécification exécutable comme oracle. Celle-ci n'explique pas comment calculer le résultat mais est simplement capable de dire si le résultat est correct. Ceci permet, entre autres, l'utilisation de spécifications non-déterministes qui sont souvent mal prises en compte par les méthodes de synthèse de tests à partir de spécifications. En outre, le temps de spécification sera moins important que pour utiliser des outils comme Casting ou BZ-TT. D'une part, les spécifications qu'ils prennent en entrée doivent permettre de calculer le résultat attendu. D'autre part, elles doivent leur permettre

de produire une séquence partant de l'état initial de l'application, ce qui sous-entend que la spécification doit être complète là où nous n'aurons besoin que de spécifier les parties importantes à tester.

3. La plupart des approches combinatoires (JMLUnit, Korat, AETG) ne considèrent qu'un seul appel de méthode. Ceci suppose que l'on puisse positionner arbitrairement l'état du système sous test (grâce à un constructeur offrant un haut niveau de paramétrage). Nous préférons considérer, comme BZ-TT, que certains états ne sont atteignables qu'après une séquence d'appels de méthodes.

Chapitre 3

Spécifications comme Oracle

Nous avons exposé dans le chapitre 2 la définition du test de conformité. Celle-ci repose sur l'existence d'une spécification. Dans le cas qui nous intéresse, nous utiliserons des spécifications orientées modèles ou par contrats [Mey92a]. Comme nous l'avons montré, le test de conformité a été très formalisé dans le cadre du test des protocoles. De nombreux outils ont été développés et industrialisés. Mais ces outils travaillent essentiellement à partir de spécifications comportementales définies par exemple sous forme d'IOLTS (Input Ouput Labeled Transition System).

S'il existe des outils travaillant à partir de spécifications orientées modèles comme nous l'avons montré en section 2.3, les recherches dans ce domaine restent assez récentes. C'est pourquoi nous nous sommes penché sur les possibilités d'utilisation de notre outil avec ce type de technologie. Nous présentons dans ce chapitre les langages VDM et JML. Ces deux langages, qui vont nous permettre d'illustrer nos recherches, ont été choisis, d'une part car ils se situent dans deux paradigmes différents (VDM est purement actionnel et JML est objet), et d'autre part car peu d'outils ont été développés pour ces langages, bien que les recherches autour de JML se développent aujourd'hui de plus en plus. Par ailleurs, ils permettent de répondre au problème de l'oracle que nous présentons section 2.2.4.

Nous présentons dans ce chapitre les notions de base de VDM et de JML nécessaires à la compréhension de nos travaux, ainsi qu'une description des outils permettant, à partir de spécifications décrites dans ces langages, d'obtenir un oracle automatique pour le test. Ce faisant, nous explicitons la relation de conformité pour ces deux langages.

3.1 VDM

3.1.1 Fondements du langage

VDM ou Vienna Development Method est une méthode formelle de développement de logiciel basée sur le langage de spécification orienté modèles VDM-SL [Jon90, FL98, ISO96].

Ce langage de spécification permet de spécifier l'état du système à l'aide de variables d'états. Il est possible de définir un invariant d'état permettant de spécifier une ou plusieurs propriétés invariantes du système. Cette propriété logique doit être vraie à tout moment lors de l'exécution. Chaque variable est typée. Il existe de nombreux types de base (naturels, réels, booléens ...) ainsi que les types ensemble, fonction, séquence ou composite. Le langage met à disposition tout un ensemble d'opérations définies pour ces types (union, concaténation, surcharge, etc.).

L'utilisateur a la possibilité de définir ses propres types à partir des types existants et de les contraindre par un invariant de type. Un invariant de type est une propriété qui doit être vraie pour toute valeur du type défini. Par exemple, soit le type `EntierPair` définissant un nombre entier pair et soit une variable `i` de type `EntierPair`. Toute opération portant sur `i` devra préserver la propriété que `i` est un nombre entier pair.

```
types EntierPair = nat
      inv ep == ep mod 2 = 0
```

Ici, le type `EntierPair` est un naturel positif. `inv` permet de spécifier l'invariant de type tel que tout élément `ep` de type `EntierPair` est un naturel et que le reste de sa division par 2 est égal à 0.

Outre les variables d'état, une spécification VDM comporte des opérations, qui portent sur l'état, et des fonctions. Les opérations et les fonctions sont spécifiées en termes de pré- et postconditions. La notation permet d'insérer le code exécutable des opérations, exprimé en VDM, dans la spécification. Il est ainsi possible de décrire du code au même niveau d'abstraction que la spécification. L'atelier VDMTools [Gro00] permet d'animer la spécification et de vérifier que le code ainsi écrit satisfait les invariants de types, d'état et les pré- et postconditions des opérations. Ce code ne reste néanmoins exécutable que dans l'atelier. Un module de génération de code C++ permet de traduire le code décrit en VDM-SL en C++, celui-ci pouvant alors être compilé et exécuté en dehors de l'atelier.

3.1.2 Exemple : le gestionnaire de processus

Afin d'illustrer les concepts de VDM, nous présentons ici un exemple très simplifié de gestionnaire de processus déjà présenté dans [DF93]. Le gestionnaire gère plusieurs processus. Chaque processus (type `PID`) est une chaîne de caractères non vide devant commencer par la lettre `p` (lignes 1 et 2). L'état du système est composé de trois variables : la variable `active` (ligne 3) représente le processus actif, la variable `ready` (ligne 4) représente l'ensemble des processus candidats à l'activation et `waiting` (ligne 5) l'ensemble des processus restants. Etant donné qu'il peut ne pas y avoir de processus actif à un instant donné, `active : [PID]` spécifie que la variable `active` est optionnelle. S'il n'y a aucun processus

actif, la valeur de la variable est `nil`.

```
1.  types   PID = seq of char
2.          inv pp == pp <> [ ] and pp(1) = 'p'
3.  state scheduler of active : [PID]
4.                      ready  : set of PID
5.                      waiting: set of PID
6.  inv mk_scheduler(active,ready,waiting) ==
7.              (ready inter waiting = {})
8.          and active not in set  (ready union waiting)
9.          and ((active = nil) => ready = {})
10. init S == S = mk_scheduler(nil,{},{})
11. end
```

L'invariant d'état caractérise le fait que :
- il ne peut y avoir de processus à la fois prêt et en attente (ligne 7),
- le processus actif n'est ni dans l'état prêt ni dans l'état en attente (ligne 8) : si `active` a pour valeur `nil`, la condition `active not in set ...` aura pour valeur vraie, et
- s'il n'y a pas de processus actif alors il n'y a pas de processus prêt (ligne 9). Ainsi, dès qu'un processus a fini sa période d'activité, il sera remplacé par un processus prêt, si cet ensemble n'est pas vide.

La dernière partie (`init S == ...`) de la spécification décrit l'état initial. Ici, il n'y a aucun processus actif, prêt ou en attente.

La spécification et le code des opérations du gestionnaire de processus sont présentés ci-dessous. Comme expliqué précédemment, les opérations et fonctions contiennent à la fois leur spécification en termes de pré et postconditions et leur code exécutable.

- L'opération `new` ajoute un nouveau processus à l'ensemble des processus en attente. La précondition spécifie que le nouveau processus n'est pas le processus actif et qu'il n'existe pas déjà. La postcondition spécifie que le processus passé en paramètre est ajouté à l'ensemble des processus en attente (`waiting~` représente l'état de la variable avant l'opération) et que l'état des autres variables reste inchangé.

  ```
  new : PID ==> ()
  new(pp) == (
       waiting := waiting union {pp}
  )
  pre pp <> active and pp not in set (ready union waiting)
  post waiting = waiting~ union {pp} and ready = ready~
          and active = active~
  ```

- L'opération `rdy` fait passer le processus en paramètre de l'état d'attente à l'état actif si aucun processus n'est actif et sinon de l'état d'attente à l'état prêt. La précondition spécifie que le processus passé en paramètre doit être dans l'état d'attente. La postcondition spécifie que le processus n'est plus dans l'état d'attente et, si aucun processus

n'est actif alors celui-ci devient actif et l'état des processus prêts reste inchangé. Sinon, ce processus s'ajoute à la liste des processus prêts et le processus actif reste le même.

```
rdy : PID ==> ()
rdy(qq) == (
    waiting := waiting\{qq};
    if active = nil then active := qq
    else ready := ready union {qq}
)
pre qq in set waiting
post waiting = waiting~\{qq}
     and if active~ = nil then (ready=ready~ and active = qq)
         else (ready= ready~ union {qq} and active = active~)
```

– L'opération `swap` fait passer le processus actif en attente et choisit un processus prêt pour le faire passer actif. La précondition spécifie qu'il doit y avoir un processus actif. La postcondition établit que le processus actif passe en attente et que s'il y a au moins un processus dans l'état prêt, alors le nouveau processus actif est choisi par la fonction `schedule`.

```
swap : () ==> ()
swap() == (
     dcl oldpp : PID := active;
     if ready = {} then (active := nil)
     else (pp     := schedule(ready);
           ready  := ready\{pp};
           active := pp);
     waiting := waiting union {oldpp}
)
pre active <> nil
post (if ready~ = {} then (active = nil and ready = {})
      else (active = schedule(ready~)
            and ready = ready~\{active}))
      and waiting = waiting~ union {active~}
```

```
functions
schedule : set of PID +> PID
schedule(fr) == (let pp in set fr be st true in pp)
pre fr <> { }
post RESULT in set fr
```

La fonction `schedule` va tirer un processus dans l'ensemble de processus passé en paramètre selon un algorithme non précisé. La précondition spécifie que cet ensemble ne doit pas être vide et la postcondition que le résultat de l'opération est un élément de cet ensemble.

3.1.3 Relation de conformité en VDM

Relation liée à la sémantique

Nous avons montré section 2.2.2, que la relation de conformité entre une implantation et une spécification était liée à la sémantique de la spécification.

En VDM, nous avons vu que la spécification était caractérisée par des propriétés invariantes, des préconditions et des postconditions. La sémantique du langage impose que ces propriétés soient vraies à certains moments.

L'invariant doit être vrai avant et après tout appel d'opération. De plus, en VDM, il doit être vrai avant et après toute action atomique. Cette contrainte est particulièrement forte car elle signifie qu'à l'intérieur d'une opération l'invariant doit être préservé. Ainsi, quelle qu'en soit la raison, avant ou après toute action atomique, si l'invariant est violé, il y a une erreur de conformité. Ainsi, dans l'opération `swap`, si nous inversons les deux instructions `active := pp` et `ready := ready\{pp}`, l'invariant est faux entre les deux instructions car `active` est un élément de `ready`.

Les préconditions et postconditions d'une opération doivent être vraies respectivement avant et après tout appel de la dite opération. L'appel peut être effectué par l'utilisateur ou par une autre opération du système. En VDM les préconditions ne font pas office de garde, si, lors d'un appel, la précondition est vraie, alors la postcondition doit aussi être vraie. En revanche, si, lors d'un appel, la précondition est fausse, le comportement n'est plus garanti et la postcondition n'a pas l'obligation d'être vraie. Dès lors, il est possible d'appeler une opération en dehors de sa précondition sans que la conformité du système ne soit remise en cause. Lorsque, au cours d'un test, c'est l'utilisateur qui appelle une opération en dehors de sa précondition, il n'est pas possible de conclure quant à la conformité du système. Nous parlons alors de verdict *Inconclusif*.

Lorsque l'opération op_a est appelée par une autre opération op_b, sa précondition doit être vraie au moment où elle est appelée et sa postcondition doit être vraie à sa sortie. Si la précondition de l'opération op_a est violée lors de son appel dans op_b, il y a une erreur de conformité.

Description de l'environnement VDMTools

Le caractère exécutable de certaines spécifications VDM fait qu'il est possible d'utiliser directement les pré, postconditions et invariants comme oracle. L'environnement VDMTools [Gro00] permet en effet d'évaluer ces assertions avant, pendant, et après chaque exécution d'opération. Il est ainsi possible de vérifier directement que la relation de conformité entre l'implantation et la spécification VDM correspondante n'est pas violée lors d'une exécution.

La figure 3.1 montre plus en détail les différentes vérifications faites par l'atelier VDMTools lors de l'exécution d'une opération. Nous pouvons voir sur cette figure que les vérifica-

tions initiales, c'est-à-dire avant l'exécution du code de l'opération, portent sur les paramètres d'entrée ainsi que sur l'état des variables d'état "avant l'opération" ; les vérifications finales, ie après l'exécution du code, portent sur les paramètres de sortie, les paramètres d'entrée et les états des variables d'état "avant" et "après" l'opération ; les vérifications internes, après chaque instruction présente dans le code, ne portent que sur les variables d'état et les éventuelles variables définies localement à l'opération.

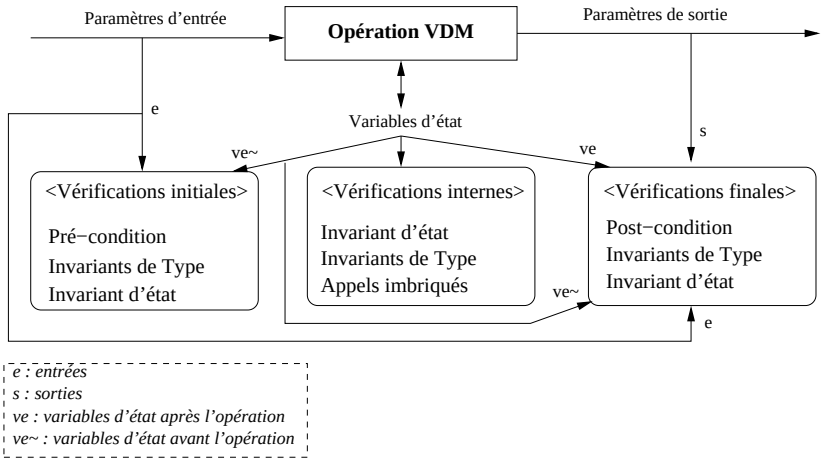


FIG. 3.1 – Vérifications de VDM-Tools.

Ainsi, lorsqu'une opération est exécutée dans l'environnement VDM, trois cas peuvent se produire :

- toutes les vérifications sont réussies. Dans ce cas, cela signifie que le comportement de l'opération, compte tenu des entrées et de l'état, est conforme au comportement spécifié. Le test délivre un verdict *pass*.
- une vérification interne ou finale échoue. L'implantation est seule responsable de l'erreur, ce qui signifie qu'elle n'est pas conforme à la spécification. Le test délivre un verdict *fail*.
- une vérification initiale échoue. Cela signifie que l'appel de l'opération se fait en dehors de la spécification. Dans ce cas, exécuter l'opération n'aidera pas à établir la relation de conformité. Le test délivre un verdict *inconclusif*.

Lorsqu'une vérification échoue, l'atelier arrête l'exécution du test au point où l'erreur a eu lieu et renvoie la nature de l'erreur.

Si l'opération fait appel à d'autres opérations, l'atelier procède récursivement. Néanmoins, toute erreur lors d'une vérification initiale de l'une de ces opérations doit conduire à un verdict *fail*. En effet, l'opération appelante est seule responsable de l'état dans lequel elle appelle une autre opération, signifiant donc que le code est erroné. Une erreur de précondition peut donc, soit conduire à un verdict *inconclusif*, soit à un verdict *fail*. Il nous faut dès lors distinguer les cas où il s'agit d'un simple appel d'opération d'un appel d'opération imbriqué pour

établir le bon verdict.

Pour bien comprendre les vérifications faites par l'atelier et leurs conséquences, nous présentons trois implantations erronées de l'opération `swap()` dont la spécification a été donnée plus haut.

```
swapWrong1 : () ==> ()
swapWrong1() == (skip)
```

`swapWrong1()` est implanté par `skip` qui ne modifie pas les variables d'état. Cette implantation n'établira jamais la postcondition de l'opération.

```
swapWrong2 : () ==> ()
swapWrong2() == (
    waiting := waiting union {active}; -- le même processus est
                                       -- à la fois actif et en attente
    if ready = {} then (active := nil)
    else (let pp in set ready in (ready := ready \ {pp};
                                  active := pp))
)
```

`swapWrong2()` ajoute le processus actif aux processus en attente puis met à jour l'état des variables `active` et `ready`. Cette implantation va casser l'invariant à l'intérieur de l'opération puisqu'un même processus ne peut être simultanément dans deux états. Cette opération va donc lever une erreur d'invariant pendant la vérification interne.

```
swapWrong3 : () ==> ()
swapWrong3() == (
    new(active);
    if ready = {} then (active := nil)
    else (let pp in set ready in (ready := ready \ {pp};
                                  active := pp))
)
```

`swapWrong3()` est une variante de l'opération précédente. Néanmoins, l'ajout du processus actif à l'ensemble des processus en attente se fait via l'appel à l'opération `new(active)`. Cet appel va violer la précondition de l'opération `new` et faire échouer la vérification interne de l'opération `swapWrong3()` entraînant alors un verdict *fail*.

3.2 JML : un langage de spécification pour les programmes JAVA

3.2.1 Fondements du langage

JML [LBR99, LBR02] est un langage de spécification orienté modèles créé pour spécifier des applications Java. JML, en adoptant une syntaxe à la JAVA, reprend un des principes d'Eiffel [Mey92b]. Il possède une sémantique et des concepts tirés des langages de spécification orientés modèles comme VDM et du langage de spécification algébrique Larch [GHG+93]. JML est, de plus, un langage en grande partie exécutable. Les spécifications JML sont définies à l'aide d'un invariant pour l'état et de pré et postconditions pour les méthodes. Il est aussi possible de définir des assertions au milieu du code Java.

Les spécifications JML s'insèrent dans le code Java sous la forme de commentaires particuliers. Un mot clef permet en effet aux outils JML d'identifier les spécifications, celles-ci étant ignorées par le compilateur Java. Il existe trois constructions possibles permettant de décrire des spécifications JML, nous ne donnons ici que celles que nous utilisons par la suite :

`/* @` *assertions JML*
`@*/`

et

`// @` *assertions JML*

Les assertions JML sont des expressions booléennes pouvant faire appel à des fonctions. Les fonctions ainsi appelées ne doivent pas contenir d'effet de bord sans quoi l'évaluation de la contrainte modifierait le comportement de la méthode spécifiée.

Etant dans un monde objet, JML intègre plusieurs types de contraintes portant sur l'état. Nous retrouvons classiquement les invariants portant sur les attributs d'instances. Mais il existe aussi des invariants dit "de classe" portant sur des attributs statiques et des contraintes d'historique permettant d'établir des contraintes entre deux états successifs d'une même instance. Il est aussi possible de spécifier une visibilité pour les invariants. Ainsi, lorsque l'invariant est :

- *public*, alors il ne peut porter que sur des attributs *public* ;
- *protected*, alors il ne peut porter que sur des attributs *public* ou *protected* ;
- *private*, alors il peut porter sur tous les attributs.

Lorsqu'il a la visibilité par défaut, c'est-à-dire lorsqu'elle n'est pas précisée, l'invariant ne peut pas porter sur des attributs *private*.

Les opérations sont spécifiées, comme pour VDM, en termes de pré- et postconditions. Dans la postcondition, il est possible, avec l'opérateur **\old**, de faire référence à la valeur

d'une expression avant l'appel de la méthode et de la comparer avec, par exemple, sa valeur après. Prenons, par exemple, la postcondition suivante :

old(`instanceA.attribut1`)>`instanceA.attribut1`, où `instanceA` est une instance de la classe `A` et où `attribut1` est un entier.

Dans cette postcondition, l'utilisateur spécifie que la valeur de l'attribut `attribut1` avant l'opération (**old**(`instanceA.attribut1`)) est strictement supérieure à sa valeur après l'opération. Il faut néanmoins être prudent. En effet, si l'utilisateur écrit :

old(`instanceA`).`attribut1`>`instanceA.attribut1`

Alors, en écrivant **old**(`instanceA`) il fait référence à l'ancienne valeur de `instanceA`. Or, en Java, `instanceA` correspond à la référence à l'instance et non à sa valeur. JML capture la valeur de la référence avant l'appel de la méthode qui, si elle n'est pas modifiée, sera la même après l'appel. La comparaison porte alors sur les attributs de la même instance au même moment.

Comme les méthodes Java peuvent lever des exceptions, JML intègre un mécanisme, appelé postcondition exceptionnelle, permettant de spécifier une postcondition pour chaque exception pouvant être levée par la méthode. Les pré-, postconditions et postconditions exceptionnelles sont respectivement définies à l'aide des mots-clés **requires**, **ensures** et **signals**. Il est par ailleurs possible de spécifier quels attributs de l'objet vont être modifiés par la méthode à l'aide du mot-clé **assignable** et, si la méthode est purement fonctionnelle, **pure** permet de spécifier que celle-ci n'a aucun effet de bord. Il faut cependant noter que **pure** et **assignable** ne sont pas vérifiés à l'exécution.

Le langage JML possède en outre les quantificateurs universel et existentiel, dont seul une forme restreinte est exécutable :

(**forall** *élément* **;** *tel que* **;** *propriété*)
(**exists** *élément* **;** *tel que* **;** *propriété*)

3.2.2 Exemple

Afin d'illustrer plus explicitement les concepts JML, nous reprenons l'exemple du gestionnaire de processus. La modélisation de ce gestionnaire en JML nous a amené à définir deux classes. Une classe représentant le type PID et une classe représentant le gestionnaire.

La classe `PID` a un attribut `id` de type *String* représentant le nom du processus. L'invariant de type défini en VDM se caractérise ici par un invariant d'instance qui spécifie que l'identifiant doit avoir au moins un caractère et que la première lettre doit être un 'p' :

```
public class PID {
    private String id;
    //@ private invariant id.length()>0 && id.charAt(0)=='p';

    ...
}
```

Le constructeur a une précondition (**requires**) forte reprenant l'invariant et précisant que le paramètre ne doit pas être `null`. La clause **assignable** précise les attributs modifiés par l'appel de méthode. La postcondition (**ensures**) spécifie la valeur de l'attribut `id` (retourné par la fonction `getId`) après l'opération, à savoir qu'il ne doit pas être `null` et qu'il doit être lexicalement égal à la chaîne de caractères passée en paramètre.

```
/*@ private normal_behavior
   requires newId != null && newId.length()>0 && newId.charAt(0)=='p';
   assignable id;
   ensures getId() != null
               && getId().equals(newId) == 0; % //equals renvoie 0
                                              si les deux chaînes sont identiques
  @*/

public PID( String newId){
       id = newId;
}
```

Selon le même modèle que pour la spécification VDM, la classe `Scheduler` est composée de trois attributs représentant respectivement le processus actif, l'ensemble des processus prêts et l'ensemble des processus en attente. JML offre la possibilité d'utiliser la classe `JMLObjectSet` pour définir un ensemble avec des opérations d'union, d'intersection, d'appartenance etc. La manipulation des ensembles se fait ici de manière fonctionnelle c'est-à-dire que, par exemple, l'insertion d'un nouvel élément dans l'ensemble ne modifie pas l'état de l'objet paramètre mais renvoie un nouvel ensemble contenant le nouvel élément.

L'inconvénient majeur de cette classe est que l'appartenance est testée avec l'opération `==` sur les éléments ce qui permet de manipuler des types simples mais ne permet pas de comparer deux instances par rapport à leur contenu : dans le cas d'instances, `==` va comparer leurs références. Nous avons donc défini une nouvelle classe `MyJMLObjectSet`. Celle-ci est basée sur `JMLObjectSet` et permet une définition plus fine de l'égalité entre deux instances. Pour ce faire, l'utilisateur définit une méthode `equals` grâce à laquelle il peut comparer l'égalité de deux instances par rapport à leurs attributs.

L'invariant spécifie les mêmes propriétés que l'invariant VDM, à savoir qu'un même processus ne peut être dans deux états en même temps et que s'il n'y a pas de processus actif alors il n'y a pas de processus prêt.

```
public class Scheduler {
    private PID active;
    private MyJMLObjectSet ready;
    private MyJMLObjectSet waiting;

    /*@ private invariant
       (ready.intersection(waiting)).isEmpty() &&
       (active != null ==> !(ready.union(waiting)).has(active)) &&
       ((active == null) ==> ready.isEmpty()==0);
    @*/
    ...
}
```

Le constructeur initialise l'état de l'objet en mettant `active` à `null` et en créant les ensembles `ready` et `waiting`. Cette méthode ne comporte pas de précondition. La postcondition spécifie que les ensembles doivent être vides et que l'attribut `active` ne doit pas avoir de valeur.

```
    /*@  private normal_behavior
       ensures active == null && ready.size() == 0 && waiting.size() == 0;
      @*/
    public Scheduler(){
        active = null;
        ready = new MyJMLObjectSet();
        waiting = new MyJMLObjectSet();
    }
```

Nous ne présentons ici que la méthode `rdyP`, la présentation des autres méthodes n'apportant rien de plus à la compréhension de JML.

La précondition impose que le processus passé en paramètre soit présent dans l'ensemble `waiting`. Nous précisons aussi les attributs pouvant être affectés par la méthode, ici les trois attributs peuvent se trouver modifiés. La postcondition est une conjonction de contraintes, néanmoins, afin de mieux identifier l'origine d'une erreur (spécification ou code), nous avons ajouté le mot clef **ensures** devant chaque sous-formule. De cette manière, nous pouvons plus facilement identifier quelle est la ligne qui a levé une erreur. Il est par ailleurs intéressant de noter la manière dont nous utilisons **\old**. Comme nous l'avons expliqué plus haut, JML ne capture pas la valeur de l'état avant l'opération mais la valeur d'une expression. Une expression peut être une formule booléenne, un entier ou tout autre formule de type simple. Si le paramètre est une instance, **\old** capture la valeur de la référence à l'instance avant l'appel de méthode. Comme nous utilisons une définition fonctionnelle des ensembles, toute opération entraîne la création d'un nouvel ensemble. Il est donc possible de comparer l'état de l'ensemble "avant" avec l'état "après" l'appel de méthode. Par exemple, l'opération `remove` renvoie un élément de type `MyJMLObjectSet` ainsi, la suppression d'un élément de l'ensemble se fait de la manière suivante `waiting = waiting.remove(pcur)`.

```
/*@  private normal_behavior
   requires waiting.has(pRdy);
   assignable waiting, ready, active;
   ensures !(waiting.has(pRdy));
   ensures (\old(waiting).remove(pRdy)).equals(waiting);
   ensures (\old(active) == null ==> active.equals(pRdy) &&
                       ready == \old(ready));
   ensures (\old(active) != null ==> active == \old(active) &&
                       ready.equals(\old(ready).insert(pRdy)));
  @*/

public void rdyP( PID pRdy){
      if(active == null) active = pRdy;
      else ready = ready.insert(pRdy);
      waiting = waiting.remove(pRdy);
  }
```

3.2.3 Vérification d'assertions à l'exécution

Relation liée à la sémantique

La relation de conformité entre un programme Java et une spécification JML est très proche de celle définie entre une implantation et une spécification VDM. La différence principale tient dans l'évaluation de l'invariant.

En JML, l'invariant doit être vrai avant et après tout appel de méthode qu'il soit imbriqué ou non. Alors qu'en VDM l'invariant doit être vrai en tout point du programme. Cette différence est importante car pour une même spécification, comme celle du gestionnaire de processus, les implantations qui satisfont la relation de conformité ne sont pas les mêmes. En fait, toute implantation qui satisfait la relation de conformité VDM satisfait la relation de conformité JML, en revanche l'inverse n'est pas vrai.

L'évaluation des pré et postconditions suit les mêmes règles que pour VDM.

Compilateur : JMLC

Le langage JML est accompagné de nombreux outils. L'outil qui nous intéresse dans ce travail est le "compilateur" JMLC [Bho00, CL02a]. Le principe est de permettre au développeur de bénéficier rapidement des avantages de la spécification. Pour ce faire, les spécifications JML sont compilées et intégrées au bytecode Java de l'application. L'application, ainsi instrumentée, peut être plus facilement testée puisque l'oracle est généré automatiquement par le compilateur.

Prenons par exemple la méthode `rdyP`. Le compilateur d'assertions va créer une nouvelle

méthode privée `internal$rdyP` et va l'insérer dans le code de l'opération `rdyP` généré automatiquement par l'outil[1]. Le code de la méthode, que nous ne donnons que partiellement, devient :

```
public void rdyP( PID pRdy){
    checkPre$rdyP(pRdy);  // vérifie la précondition de la méthode
    checkInv$rdyP();  // vérifie l'invariant
    boolean rac$ok = true;
    try{
        internal$rdyP(pRdy);  // procède à l'appel de la méthode
        checkPost$rdyP(pRdy);  // vérifie la postcondition de la méthode
    } catch( JMLEntryPreconditionError e){  // une précondition a été violée dans
                                            le code de l'application
        rac$ok = false;
        throw new JMLInternalPreconditionError(e);  // permet de savoir
                            // si l'erreur est survenue dans le cas d'un appel imbriqué
    }
    ...
     catch( Throwable e){  // permet de récuperer les exceptions
                                levées ''normalement'' par la méthode
         try{
             checkExceptionalPost$rdyP(pRdy, e);  // pour vérifier
                                                  // les postconditions exceptionnelles
        } catch( JMLAssertionError e1){
                    rac$ok = false;
                    throw e1;
             }
    } finally{
        if(rac$ok){
             checkInv();  // vérifie l'invariant après l'appel de la méthode
        }
    }
}
```

Exceptions

Comme nous pouvons le remarquer ici, JMLC rajoute de nouveaux appels de méthode pour vérifier la précondition et la postcondition de la méthode ainsi que l'invariant avant et après l'appel effectif de la méthode. Si une assertion JML est violée, la procédure `check` correspondante lève une exception. L'outil ajoute aussi des captures d'exceptions pouvant être levées et propose un traitement particulier pour chacune.

La figure 3.2 montre la structure des exceptions levées par JML. Les classes en italique représentent des classes abstraites, les autres étant des classes concrètes.

[1] nous avons utilisé la version 4.0 de JML

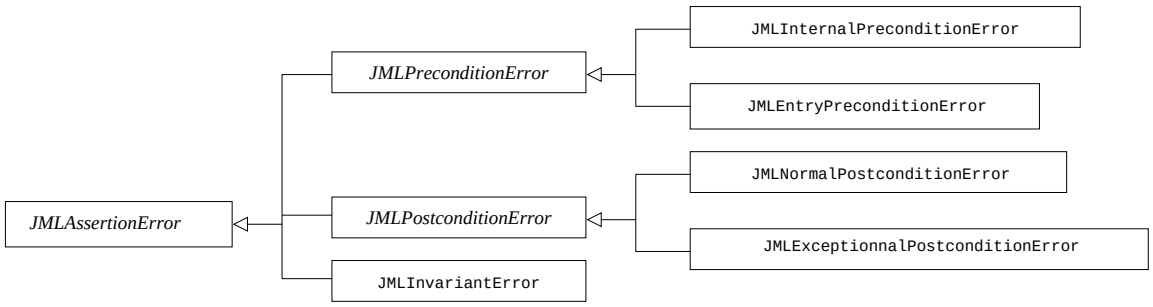


FIG. 3.2 – Une partie de la hiérarchie des exceptions JML.

On trouve ainsi cinq grandes classes d'exceptions, correspondant chacune à un type de contrainte. Les erreurs de pré et postconditions se découpent en deux sous catégories. Pour les préconditions, JML distingue deux types d'erreurs :

- à l'entrée : ce qui correspond à l'exception levée, par exemple, par l'appel `checkPre$rdyP(pRdy)` ;
- internes : ce cas de figure se présente lorsque la méthode invoquée fait appel dans son code à une méthode et invoque celle-ci en dehors de sa précondition. La méthode invoquée lève alors une *JMLEntryPreconditionError* qui est capturée par la méthode appelante, laquelle lève alors une *JMLInternalPreconditionError*.

De même, les erreurs de postcondition se décomposent en deux catégories :

- les erreurs de postconditions normales : celles-ci sont levées lors de la violation de l'une des clauses **ensures**.
- les erreurs de postconditions exceptionnelles : nous avons vu section 3.2.1, que l'utilisateur a la possibilité de spécifier un comportement particulier attendu lorsqu'une méthode lève une exception (clause **signals**) particulière. Si la contrainte, spécifiée dans ce cas, n'est pas respectée, JML lève une erreur de postcondition exceptionnelle.

Pour finir, les erreurs d'invariants sont signalées par l'exception `JMLInvariantError`.

Verdicts

Tout comme pour VDM, l'exécution de tests avec JML mène à trois verdicts différents :

- *pass*. Ce verdict est rendu si l'application a répondu en accord avec sa spécification c'est-à-dire que tous les éléments suivants sont vérifiés : précondition, invariant en entrée et en sortie, postconditions normale et exceptionnelle, assertions et appels imbriqués.
- *inconclusif* n'est rendu que si l'utilisateur appelle la méthode en dehors de sa précondition. Comme pour VDM, on établit ce verdict car l'appel de la méthode n'est plus garanti. Il reste néanmoins un comportement suspect pouvant venir d'un appel de méthode dans un état instable non détecté par l'invariant.
- *fail* est rendu dans tous les autres cas.

Autres outils utilisant de JML

JMLC n'est pas le seul outil utilisant JML. Il existe d'autres outils qui sont capables d'utiliser des clauses non exécutables de JML (certaines assertions utilisant les quantificateurs existentiels ou universels, les assertions `pure` etc.). Ce sont les outils qui effectuent des vérifications statiques à partir des spécifications JML [BCC+05].

Par exemple, *ESC/Java* [FLL+02], initialement développé par Compaq, effectue des vérifications à la compilation plus élaborées que la vérification de type. L'outil vérifie certaines autres propriétés comme le dépassement de tableau, les appels effectués sur des objets non instanciés, etc. *ESC/Java* peut exploiter un sous-ensemble du langage JML. Les spécifications ainsi écrites permettent à l'outil de produire des analyses plus fines. Par exemple, la précondition d'une opération garantit certaines propriétés sur l'utilisation de ses variables à l'intérieur du corps de la méthode. De même l'outil pourra vérifier, dans une certaine mesure, si une opération risque d'être appelée en dehors de sa précondition et prévenir l'utilisateur le cas échéant.

JACK [BRL03] produit des obligations de preuve à partir du code source Java d'un programme complété par des spécifications JML. L'outil utilise ensuite un démonstrateur automatique pour vérifier les obligations de preuve. Lorsque l'outil n'arrive pas à résoudre l'obligation de preuve, il peut se faire assister de l'utilisateur qui guide alors le processus de preuve. L'objectif de JACK est de fournir un outil utilisable par tout développeur Java souhaitant vérifier la correction de son application.

JML-TT [BDLU05] est un animateur de spécification JML. L'outil exécute donc la spécification de manière symbolique. Pour ce faire, il transforme la spécification JML en un format intermédiaire qui sert d'entrée au solveur CLPS-BZ. Lorsque l'utilisateur anime la spécification, l'outil produit une trace qui peut ensuite être sauvée sous la forme d'un cas de test Java, permettant alors de vérifier dynamiquement si l'implantation est conforme à la spécification.

3.3 Synthèse

Nous avons présenté dans ce chapitre deux langages de spécification : VDM-SL et JML. VDM-SL offre un support de spécification pour des programmes décrits en VDM et JML offre un support de spécification pour des programmes écrits en Java. Ces deux langages de spécification utilisent des assertions de type pré et postcondition pour les opérations et invariant pour l'état. JML étant dédié à Java, il permet de prendre en considération les aspects objets de ce langage. Cette particularité complique un certain nombre de choses : relation entre l'état "avant" et l'état "après" difficile à exprimer car on travaille souvent sur des références et non pas sur des valeurs, problèmes liés à l'héritage ne permettant pas toujours de bien savoir quelle méthode et donc quelle spécification est prise en compte, etc.

Néanmoins, ces deux langages ont la particularité d'être en partie exécutable, grâce,

notamment, aux outils VDM Tools et JMLC. Ainsi, les spécifications décrites en VDM-SL ou JML peuvent servir d'oracle automatique pour le test de programmes écrits en VDM ou en Java.

Nous avons aussi introduit la notion de verdict *Inconclusif*. Ce type de verdict est rendu lorsque l'utilisateur appel une opération en dehors de sa précondition. En effet, comme c'est ici l'utilisateur qui est responsable du mauvais appel et non l'application, et comme le comportement spécifié n'est valable que si la précondition est vérifiée, il n'est pas possible de conclure sur un verdict tranché *pass* ou *fail*.

Deuxième partie

Génération de tests combinatoire

Chapitre 4

Langage de synthèse de tests

Le langage de synthèse de tests a été développé dans le cadre du projet RNTL COTE[1], en collaboration avec P. Bontron [Bon05]. Ce projet visait à intégrer les outils UMLAUT [HJGP99, LG01] et TGV [JM99] avec l'atelier de modélisation UML Objecteering[2] afin de fournir un environnement complet de spécification et de génération de tests de composants, à partir d'une spécification UML [UML99].

Nous présentons dans ce chapitre ce langage qui s'articule autour de trois éléments :

- la spécification de l'application sous test, définie sous la forme d'un diagramme de classes ;
- la notion de groupe d'opérations, permettant de définir un ensemble d'appels à l'aide d'une seule étiquette ;
- la notion de schéma de tests servant à définir de manière synthétique un ensemble de séquences de test.

Nous présentons, annexe A, l'outil Tobias [BMdB+01a] qui est un environnement de création de tests basé sur ce langage.

4.1 Motivations

Deux projets sont à l'origine de la création du langage de synthèse de tests : le projet LHUSY et le projet COTE.

4.1.1 Projet LHUSY

Le projet LHUSY [dBMJ01] était un projet regroupant la société Gemplus et l'IRISA. Il visait à comparer une approche manuelle de production de tests, en vigueur chez Gemplus, pour application Java Card, à une approche automatique basée sur les outils UMLAUT et TGV [3] développés à l'IRISA.

[1] http ://www.irisa.fr/COTE

[2] http ://www.softeam.com

[3] TGV est le fruit d'une coopération entre l'IRISA et VERIMAG.

Ce projet a permis de démontrer que l'utilisation des outils UMLAUT/TGV permettait d'accroître la productivité de l'ingénieur de test. Il a, de plus, montré que cet outil n'était pas sans faiblesses. En effet, TGV a besoin que l'utilisateur définisse un objectif de test. À partir de cette objectif de test et d'une description comportementale de l'application, TGV génère un test abstrait sous la forme d'une séquence d'opérations. Pour générer plusieurs tests satisfaisant les critères de l'objectif de test, l'utilisateur peut être amené à relancer plusieurs fois l'outil. De plus, la description comportementale de l'application sous test pouvant être non-déterministe, l'outil est amené à faire des choix de manière aléatoire. À cause de cet élément, l'utilisateur ne sait pas si l'outil a produit tous les tests qui satisfont les critères de l'objectif de test.

Les conclusions du projet ainsi que les travaux d'H. Martin [Mar01] montrent que, pour pallier à ce problème, l'utilisateur est amené à décrire de nombreux objectifs de tests similaires pour guider l'outil dans son processus de génération. Ils montrent, de plus, que ce processus d'écriture d'objectifs de test peut être en partie automatisé.

4.1.2 Projet COTE

Le projet COTE était un projet RNTL [4] précompétitif regroupant trois industriels : Softeam, Gemplus et France Télécom R&D ainsi que deux laboratoires universitaires : l'IRISA et le LSR.

L'objectif premier du projet COTE était de produire un environnement de création de tests de composants permettant de générer des tests indépendamment de leur plate-forme d'exécution en utilisant une approche MDA [BGMR03, OMG] (Model Driven Architecture). C'est la notation UML qui a été choisie pour définir un cadre unifié pour la spécification et le développement de tests. Le langage TeLa [PJH+01] a été créé pour permettre la description des tests abstraits à l'aide de la notation UML. Ces tests peuvent ensuite être traduits vers l'une des trois plates-formes utilisées pour ce projet :EJB, .net et CORBA. Le langage TeLa a été intégré à un profil UML [5], développé afin de permettre la description des tests dans l'atelier de modélisation Objecteering.

Un second objectif du projet était d'intégrer les outils UMLAUT et TGV afin d'automatiser la production des tests abstraits. Le couple UMLAUT/TGV fonctionne à partir d'une spécification UML et d'un ensemble d'objectifs de tests définis par l'utilisateur. La spécification est donnée sous la forme de diagrammes de classes et d'états-transitions et sert d'entrée à l'outil UMLAUT qui va piloter TGV. Les objectifs de tests sont définis dans le langage O-TeLa [BMdB+01b] qui est, tout comme TeLa, basé sur la notation UML.

[4] Réseau National des Technologies du Logiciel.

[5] Ces travaux ont contribué, par l'intermédiaire des sociétés Softeam et Gemplus, aux réflexions du groupe de travail de l'OMG chargé de définir un profil UML pour le test [OMG02].

Partant des conclusions du projet LHUSY, nous avons décidé de définir un langage de synthèse d'objectifs de test. Ce langage permet, à partir d'une description synthétique, de produire un ensemble d'objectifs de test. Lors de l'utilisation du langage pour synthétiser des objectifs de tests, nous avons constaté qu'il était possible de le modifier légèrement pour générer directement des tests abstraits. C'est ce dernier langage, servant à produire des tests abstraits, que nous présentons ici.

4.2 Principes du langage de synthèse de tests

Le langage de synthèse de tests utilise trois éléments : la signature des opérations, des groupes qui permettent de rassembler sous une même étiquette un ensemble d'opérations et des schémas de tests décrivant un ensemble de tests.

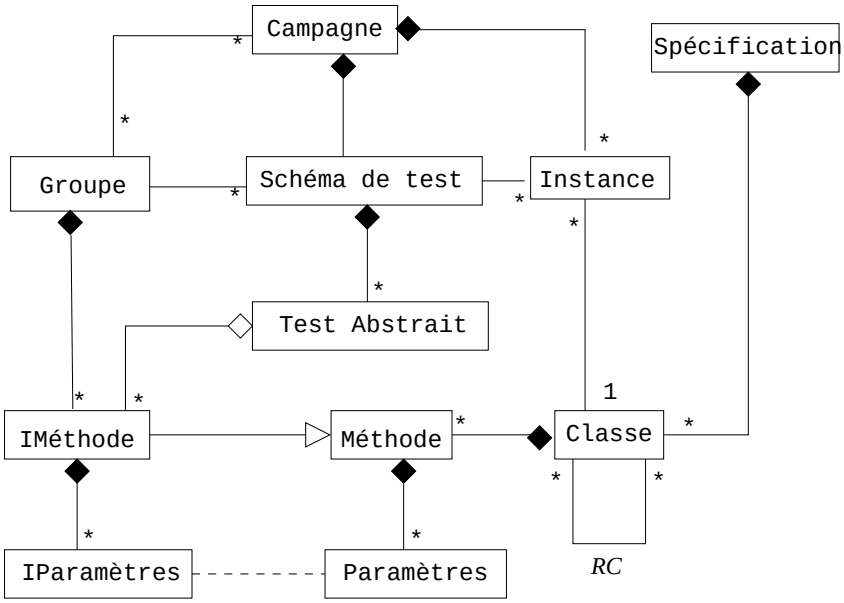


FIG. 4.1 – Métamodèle du langage de synthèse de test.

La figure 4.1 présente le métamodèle du langage de synthèse de tests. Celui-ci s'articule autour des notions de groupe et de schéma de tests.

Pour définir les groupes et les schémas, nous avons besoin d'une spécification définie par un ensemble de classes. Ces classes sont composées d'une ou plusieurs méthodes pouvant avoir plusieurs paramètres. De même, il peut exister des relations entre les classes. Ces relations nous donnent des informations sur le fait qu'une classe peut appeler directement une méthode de telle autre classe.

À partir de ces éléments de spécification, l'utilisateur peut définir des groupes. Les groupes sont des ensembles composés de méthodes dont les paramètres sont instanciés, représentées

par la classe `IMéthode`. Les méthodes instanciées raffinent `Méthode` en permettant de stocker différentes valeurs de paramètres. L'utilisateur associe un ensemble de valeurs possibles à chaque paramètre appartenant aux méthodes du groupe, représenté par la classe `IParamètre`.

Le reste est structuré sous la forme de campagnes de tests. La campagne permet de définir un déploiement (un état) particulier de l'application. Cet état est défini ici par une instanciation spécifique des classes de l'application sous test. Il constitue l'état initial du test.

C'est aussi à l'intérieur des campagnes que sont définis les schémas de tests. Un schéma de tests est une description synthétique caractérisant un ensemble de tests abstraits. Le schéma génère les tests abstraits en fonction des groupes qu'il utilise et en fonction de l'état initial associé à la campagne à laquelle il est rattaché.

Les tests abstraits sont des séquences d'appels de méthodes instanciés (`IMéthode`). Comme ces séquences ne sont pas directement exécutables sur une plate-forme de test Java, VDM ou autre, nous les considérons comme abstraits. L'utilisateur pourra ensuite, pour une séquence de test donnée, choisir de produire un test exécutable en Java ou en VDM.

4.2.1 Spécification d'entrée

Le langage de synthèse de tests ne requiert que la signature des opérations. Dans le monde objet, cette signature comporte la classe de l'opération et la signature classique de toute opération à savoir le nom de l'opération et les paramètres avec leur type.

Cette information peut être extraite de diverses manières (spécification UML, analyse de code...). Dans le cas du projet RNTL COTE, nous la récupérons à partir du diagramme de classes.

Ces éléments se retrouvent dans notre métamodèle (figure 4.1) : les classes sont en relation avec des méthodes qui sont composées de paramètres.

Soit C l'ensemble des classes. C_i représente une classe et contient un ensemble de méthodes M_{C_i} qui appartiennent à cette classe. Chaque méthode m appartenant à M_{C_i} possède une liste de paramètres P_m de type T_{P_m}.

Soit R_C l'ensemble des relations entre classes. Ainsi, $(C_i, C_j) \in R_C$ signifie qu'il existe une relation entre C_i et C_j.

Si nous reprenons l'exemple du gestionnaire de processus Java, nous avons deux classes :

- la classe PID qui a une seule méthode :

  public `PID(` *String* `pName)` *//constructeur*
- la classe *Scheduler* qui a les méthodes :

  public `Scheduler()` *//constructeur*

  public *void* `newP(` *PID* `pNew)`

  public *void* `rdyP(` *PID* `pRdy)`

public *void* `swap()`

4.2.2 Déploiement

Le déploiement de l'application consiste à configurer l'état initial du système sous test. Étant dans un monde objet, il est nécessaire de définir, à partir des classes, quelles instances entreront en jeu lors du test. Les instances définies à ce niveau serviront de paramètres pour les appels de méthodes qui seront générés dans la campagne de test. L'utilisateur définit donc un ensemble I d'instances tel que : $\forall inst \in I, Classe(inst) \in C$. $Classe$ est une fonction qui renvoie la classe de l'instance passée en paramètre.

Pour chaque campagne de tests, l'utilisateur définit comment son application est déployée. Par exemple, dans le cas du gestionnaire de processus, nous pouvons définir le déploiement suivant : $I_1 = \{p1 : PID, p2 : PID, p3 : PID, sch1 : Scheduler\}$ où nous définissons trois instances de PID et une instance du $Scheduler$.

4.2.3 Groupes d'opérations

Si nous regardons un ensemble de séquences de tests, nous pouvons trouver certaines similitudes. L'idée est de factoriser ces similitudes sous la forme de groupes. Comme nous sommes dans une phase de conception de tests, nous devons construire les groupes en fonction des tests que l'on souhaite produire. Nous cherchons donc à identifier les méthodes qui peuvent avoir des contextes d'appel similaires. Ce que nous appelons contexte d'appel est en fait l'état du système à partir duquel on appelle la méthode. Cet état peut être le résultat d'une séquence d'appels d'opérations.

Un groupe G représente donc un ensemble de méthodes m, chacune appartenant à des classes de C. Comme nous cherchons à produire des tests exécutables, l'utilisateur doit spécifier les valeurs des paramètres des méthodes de G de sorte que l'on puisse créer des appels de méthodes instanciés à partir du groupe G. Ainsi, pour chaque méthode $m_G \in G$, nous définissons, pour chaque paramètre p_{m_G} de m_G un ensemble de valeurs v.

Nous avons donc :
$G = \{m_G(p_1, \ldots, p_n) | m_G$ est une méthode ayant n paramètres $\wedge$ p_i est un ensemble de valeurs $\}$

Ainsi, dans l'exemple du gestionnaire de processus, nous pouvons définir les groupes suivants :

$newRdyP12 = \{new(x), rdy(x)\}$ avec $x = \{p1, p2\}$

$RdySwapP13 = \{rdy(y), swap()\}$ avec $y = \{p1, p2, p3\}$

Nous avons vu, dans le métamodèle, que le langage est structuré en campagnes. Les groupes peuvent être rattachés à une campagne. Lorsque c'est le cas, les méthodes qui peuvent être utilisées pour le définir sont celles des instances définies dans le déploiement. Nous avons fait ce choix en partant du principe qu'un groupe défini dans une campagne a pour but d'être adapté à la campagne en question. Il ne doit donc pas être possible de définir,

dans une campagne, des groupes dont les méthodes portent sur des classes non instanciées. Si un tel cas arrivait, l'ajout de la méthode dans le groupe ne serait pas pris en compte lors de la génération des séquences abstraites ; il serait donc inutile.

4.2.4 Langage de schéma de tests

Le langage de schéma de tests permet, à partir des éléments définis ci-dessus, de décrire de manière synthétique un ensemble de séquences abstraites pouvant ensuite être traduites en séquences de test exécutables.

Grammaire du langage

Le langage se compose de quatre opérateurs qui sont le séquencement, l'alternative, la co-région et la boucle. Ces quatre opérateurs permettent de composer les éléments de base du langage que sont les appels de méthodes. Nous présentons, table 4.1 la grammaire abstraite du langage de schéma de tests.

Schéma	: :=	*Séquence* ‖ *Schéma*
	|	*Séquence*
Séquence	: :=	*Structure* ; *Séquence*
	|	*(Schéma)* ; *Séquence*
	|	*Structure*
Structure	: :=	*Boucle*
	|	*Corégion*
	|	*Appel*
Corégion	: :=	<*Appel*>*Corégion*
	|	<*Appel*><*Appel*>
Boucle	: :=	*Appel*^entier..entier
	|	(*Schéma*)^entier..entier
Appel	: :=	inst !inst.groupe

TAB. 4.1 – Grammaire du langage de schémas de test.

L'alternative , représentée par le mot clé "‖" exprime un choix non-déterministe entre deux séquences d'appels.

Le séquencement , représenté par le mot clé ";", est l'opération de base des schémas de test.

La corégion , représentée par le mot clef "<>" est un mécanisme permettant de représenter le caractère parallèle d'une exécution. Si nous avons par exemple $< app1 >< app2 >< app3 >$, cela signifie que nous n'imposons pas d'ordre dans la transmission des appels de méthode *app1*, *app2* et *app3*.

Les boucles permettent de répéter plusieurs fois un même appel de méthode ou un même schéma dans un intervalle donné. Si nous avons par exemple : $app\ ^{\wedge 2..4}$, cela signifie que nous avons deux, trois ou quatre appels successifs à *app*.

Instances et groupes inst correspond aux instances. Dans un schéma de tests, une instance peut être clairement identifiée ou remplacée par "*". Dans ce cas, nous remplaçons l'étoile par toutes les instances pouvant correspondre aux méthodes qui sont associées au groupe lors de la génération des séquences de tests.

L'alternative et la corégion sont des cas à part. En effet, ces constructions peuvent assez naturellement s'interpréter par une génération de tests non-déterministes. Nous reviendrons sur ce point à la section 4.2.5.

À partir des instances définies pour la campagne de tests, des groupes et du langage de schéma de tests, nous pouvons définir divers schémas :

```
S1 = MTC!sch1.newRdyP12^2..2 ; MTC!sch1.rdySwapP13^2..3
S2 = MTC!sch1.newRdyP12 ; <MTC!sch1.newRdyP12><MTC!sch1.rdySwapP13> ;
     MTC!sch1.rdySwapP13^2..2
S3 = MTC!sch1.newRdyP12 ; (MTC!sch1.newRdyP12 || MTC!sch1.rdySwapP13) ;
     MTC!sch1.rdySwapP13^2..2
```

Dans ces schémas, nous voyons apparaître l'instance `MTC` (Main Test Component). Cette instance est une instance générique désignant le testeur ou le programme qui exécutera les tests.

Le schéma `S1` est un cas typique de schéma de tests utilisé pour générer des séquences de test. Les schémas `S2` et `S3` sont définis ici pour illustrer la manière dont nous pouvons utiliser les constructeurs d'alternative et de corégion lors de la génération de tests.

Génération des tests à partir de `S1`

$newRdyP12 = \{new(x), rdy(x)\}$ avec $x = \{p1, p2\}$
$RdySwapP13 = \{rdy(y), swap()\}$ avec $y = \{p1, p2, p3\}$

Le premier schéma, `S1`, fait deux appels au groupe $newRdyP12$ suivis de deux à trois appels au groupe $rdySwapP13$. Si nous suivons la sémantique du langage de schéma de

tests, nous devons déplier les différents groupes. Ainsi, `MTC !sch1.newRdyP12` va produire quatre appels différents : `MTC !sch1.new(p1)`, `MTC !sch1.new(p2)`, `MTC !sch1.rdy(p1)` et `MTC !sch1.rdy(p2)`. De même, `MTC !sch1.rdySwapP13` va produire les quatres appels suivants : `MTC !sch1.rdy(p1)`,
`MTC !sch1.rdy(p2)`, `MTC !sch1.rdy(p3)` et `MTC !sch1.swap()`.

La boucle, bornée par les entiers 2..2, portant sur l'appel au groupe $newRdyP12$ va produire 16 séquences différentes. En effet, comme nous avons deux appels au groupe en question et que le groupe, une fois déplié, produit quatre appels, nous avons $4 * 4$ combinaisons.

La boucle, bornée par les entiers 2..3, portant sur le groupe $rdySwapP13$ va produire 80 appels différents : $16(4 * 4)$ appels sont générés par la succession de deux appels au groupe, auxquels il faut ajouter les 64 $(4 * 4 * 4)$ appels générés par la succession de trois appels au groupe.

Comme les deux boucles sont en séquence, le nombre de tests générés est la combinaison des ensembles générés à partir des deux boucles. Ce qui nous donne $16 * 80 = 1280$ tests.

Voici quelques exemples de tests abstraits générés :

```
seq_11 = MTC!Sch1.new(p1) ; MTC!Sch1.new(p1) ; MTC!Sch1.rdy(p1) ;
         MTC!Sch1.rdy(p1)
seq_12 = MTC!Sch1.new(p1) ; MTC!Sch1.new(p1) ; MTC!Sch1.rdy(p1) ;
         MTC!Sch1.rdy(p2)
...
seq_1i = MTC!Sch1.new(p2) ; MTC!Sch1.rdy(p2) ; MTC!Sch1.swap() ;
         MTC!Sch1.rdy(p1)
...
seq_1k = MTC!Sch1.rdy(p1) ; MTC!Sch1.new(p1) ; MTC!Sch1.rdy(p1) ;
         MTC!Sch1.rdy(p3) ; MTC!Sch1.rdy(p1)
...
```

Fonction de traduction

Soit T une fonction de traduction prenant en entrée un schéma de tests S ainsi qu'un déploiement I et retournant un ensemble Seq_a de séquences abstraites dont la grammaire est donnée table 4.2. Ces séquences abstraites peuvent ensuite être traduites en objectifs de test sous la forme d'iolts ou, en séquences de test VDM ou Java. Compte tenu des différents langages cibles possibles, les opérateurs de corégion et d'alternative ne sont pas directement interprétés à la génération. Ceci nous permet de leur attribuer différentes sémantiques en fonction du langage ciblé.

1. $T_{Schema}(\textbf{Seq1} \parallel \textbf{Sch1}) = \{s1 \| s2 | s1 \in T_{Sequence}(\text{Seq1}) \wedge s2 \in T_{Schema}(\text{Sch1})\}$
2. $T_{Schema}(\textbf{Seq1}) = T_{Sequence}(\text{Seq1})$
3. $T_{Sequence}(\textbf{Struct1} \ ; \textbf{Seq1}) = \{s1; s2 | s1 \in T_{Structure}(\text{Struct1}) \wedge s2 \in T_{Sequence}(\text{Seq1})\}$
4. $T_{Sequence}((\textbf{Sch1}); \textbf{Seq1} \ ) = \{(s1); s2 | s1 \in T_{Schema}(\text{Sch1}) \wedge s2 \in T_{Sequence}(\text{Seq1})\}$
5. $T_{Sequence}(\textbf{Struct1}) = \{s1 | s1 \in T_{Structure}(\text{Struct1})\}$
6. $T_{Structure}(\textbf{Bou1}) = T_{Boucle}(\text{Bou1})$

7. $T_{Structure}(\mathbf{Cor1}) = T_{Coregion}(\text{Cor1})$
8. $T_{Structure}(\mathbf{App1}) = T_{Appel}(\text{App1})$
9. $T_{Coregion}(<\mathbf{appel1}><\mathbf{appel2}>) = \{< a1 >< a2 > |a1 \in T_{Appel}(appel1) \wedge a2 \in T_{Appel}(appel2)\}$
10. $T_{Coregion}(<\mathbf{appel1}>\mathbf{Cor1}) = \{< a1 > c1 | a1 \in T_{Appel}(appel1) \wedge c1 \in T_{Coregion}(Cor1)\}$

Les premières règles appliquent récursivement le dépliage des schémas, des séquences, des structures et des co-régions. Les règles 11 et 12 traitent le cas des boucles en utilisant la fonction *itere* (voir plus loin). La règle 13 correspond au dépliage d'un groupe.

11. $T_{Boucle}(\mathbf{App1}\hat{}b_{inf}..b_{sup}) = itere(T_{Appel}(\text{App1}), b_{inf}, b_{sup})$
12. $T_{Boucle}((\mathbf{S1})\hat{}b_{inf}..bsup) = itere(\{(s1)|s1 \in T_{Schema}(S1)\}, b_{inf}, b_{sup})$
13. $T_{Appel}(\mathbf{Appelant\ !Appelé.Groupe}) = degroupe(\text{Appelant, Appelé, Groupe})$ Appelé et Appelant peuvent soit être une instance de I soit *****. Dans le dernier cas, cela signifie que toutes les instances de I doivent être considérées lors de la génération.

– Soit $NomMethodes : Groupe \rightarrow NomMethode$ une fonction qui prend un groupe en paramètre et renvoie l'ensemble des noms de méthode du groupe.
– Soit $Methodes : Groupe \times NomMethode \rightarrow Methodes_G$ une fonction qui prend en paramètre un groupe et un nom de méthode et renvoie la signature de la méthode de nom $NomMethode$ appartenant au groupe passé en paramètre.
– Soit $Parametres : Methodes_G \rightarrow P_{m_G}$ une fonction qui prend, en paramètre, la signature d'une méthode appartenant à un groupe et renvoie un tuple contenant les ensembles de valeurs définies pour cette méthode.
– Soit Π le produit cartésien entre n ensembles.

Par exemple, soit $g1 = foo(x, y), bar(x)$ avec $x = \{a, b, c\}$ et $y = \{1, 2\}$:

1. $NomMethodes(g1)$ renvoie l'ensemble de noms de méthodes $\{foo, bar\}$
2. $Methodes(g1, foo)$ renvoie la méthode $foo(\{a, b, c\}, \{1, 2\})$
3. $Parametres(foo(\{a, b, c\}, \{1, 2\}))$ renvoye le tuple $(\{a, b, c\}, \{1, 2\})$
4. $\Pi(\{a, b, c\}, \{1, 2\})$ renvoie $\{(a, 1), (a, 2), (b, 1), (b, 2), (c, 1), (c, 2)\}$

Ces quatre premières fonctions servent de base à la génération des tests puisqu'elles permettent de déplier le contenu des groupes. Les trois fonctions suivantes permettent de déplier les boucles d'appels et d'instancier les appels de méthodes en fonction des instances définies dans la campagne de tests.

– Soit `itere(`S_Seq, b_inf, b_sup`)` :$ensembledeSeq_a \times borne \times borne \rightarrow ensembledeSeq_a$ une fonction qui permet de déplier les boucles. Cette fonction va ainsi générer toutes les séquences composées de *b_inf* à *b_sup* éléments de l'ensemble passé en paramètres. Cette fonction s'appuie sur la fonction `genere(`S_Seq, nb`)` :$ensembledeSeq_a \times entier \rightarrow ensemble$ *de Seq_a* qui, pour un nombre *nb* donné passé en paramètre et un ensemble de séquences *S_Seq* va générer toutes les combinaisons de *nb* séquences de *S_Seq*. Par exemple, `itere({a,b},1,2)` va renvoyer l'ensemble {"a","b","a ;a","a ;b","b ;b","b ;a"}.

– Soit `degroupe` :*instance* × *instance* × *Groupe* → *ensemble de Seq_a* une fontion qui va permettre, d'instancier tous les appels définis par un groupe donné en paramètre. Cette fonction prend en paramètre une instance de départ *iDep*, une instance d'arrivée *iDes* et un groupe *G*. Elle va générer toutes les instanciations de méthode pour chaque méthode et chaque valeur de paramètre contenues dans *G*. Nous effectuons donc, pour une méthode donnée, la combinaisons de toutes ses valeurs de paramètres. À partir de cet ensemble de méthodes dont les paramètres sont instanciés, la fonction va construire les appels de méthodes en fonction de *iDes* et *iDep*, en respectant les règles suivantes :

  1. chaque appel de méthode s'effectue sur une instance dont la classe possède la dite méthode ;
  2. chaque appel de méthode s'effectue à partir d'une instance de classe en relation avec l'instance d'arrivée ;
  3. lorsque *iDep* (respectivement *iDes*) est égal à `*`, on prend en considération toutes les instances définies dans la campagne comme étant potentiellement une instance de départ (respectivement d'arrivée).

```
genere : ensemble de Seq_a × entier → ensemble de Seq_a
genere(S_Seq,nb) =
 début
   si nb = 1
   alors
         /**
          * Lorsque nb vaut 1, cela signifie qu'il n'y a pas de répétition
          * donc qu'il n'y a pas de combinaisons entre les éléments de
          * l'ensemble de départ. On le renvoie donc tel quel.
          */
         retourner S_Seq
   sinon
         /**
          * nb ne pouvant être négatif où nul, il faut effectuer toutes
          * les combinaisons de nb éléments de l'ensemble S_Seq.
          * Ces combinaisons se font par séquencement entre chaque
          * séquence de S_Seq et toutes les combinaisons de nb-1 éléments
          * de l'ensemble S_Seq
          */
         retourner {"s1;s2"| s1 ∈ S_Seq ∧ s2 ∈ genere(S_Seq, nb − 1)}
 fin

itere : ensemble de Seq_a × borne × borne → ensemble de Seq_a
itere(S_Seq, b_inf, b_sup) =
 début
    SRes = {}
    pour i allant de b_inf à b_sup
       /**
```

```
        * Nous générons toutes les combinaisons de i éléments
        * parmi les éléments de l'ensemble de départ
        */
       SRes = SRes ⋃ genere(S_Seq, i)
    retourner SRes
fin

degroupe : instance × instance × Groupe → ensemble de Seq_a
degroupe(iDep, iDes, G) =
début
//On commence par déplier les appels à partir du groupe
  appels = {m(v1, v2, ..., vn) | m ∈ NomMethodes(G)
                               & (v1, v2, ..., vn) ∈ Π(Parametres(Methodes(G, m)))}

//Puis, en fonction de la valeur des instances de départ et d'arrivée
//on définit les ensembles concernés.
  si iArr = *
  alors iArrEns = I //Si l'instance d'arrivée n'est pas précisée,
                    //on prend toutes les instances définies au niveau
                    //du déploiement.
  sinon si iDes ∈ I
  alors iArrEns = {iDes} //Sinon on crée un ensemble contenant
                         //pour seul élément, l'instance passée en paramètre.
  sinon retourner erreur

  si iDep = * //On procède de la même façon pour l'instance de départ.
  alors iDepEns = I
  sinon si iDes ∈ I
  alors iDepEns = {iDep}
  sinon retourner erreur

/**
  * On définit maintenant tous les couples d'instances départ!arrivée
  * à partir des ensembles précédemment construits.
  * Ces couples ne sont construits que pour les instances dont
  * les classes sont en relation: (Classe(dep), Classe(des)) ∈ R_C
  */
  dep_arr = {dep!des | ∀ dep ∈ iDepEns,
                       ∀ des ∈ iArrEns,
                               (Classe(dep), Classe(des)) ∈ R_C }

/**
  * On concatène tous les appels de méthodes, créés à partir du groupe
  * passé en paramètre aux couples d'instances départ!arrivée
  * à partir de l'ensemble dep_arr et tel que:
  * la méthode appelée appartient bien à la classe de l'instance d'arrivée.
```

```
  */
  appelsInst = {id!ia.app | ∀ app ∈ appels,
                            ∀ id!ia ∈ dep_arr,
                                      appartient(app, Classe(ia))}
  retourner appelsInst
fin
```

TestAbstrait	: :=	*Séquence* ‖ *TestAbstrait*
	|	*Séquence*
Séquence	: :=	*Structure* ; *Séquence*
	|	*(TestAbstrait)* ; *Séquence*
	|	*Structure*
Structure	: :=	*Corégion*
	|	*Appel*
Corégion	: :=	<*Appel*>*Corégion*
	|	<*Appel*><*Appel*>
Appel	: :=	inst !inst.méthode(*Paramètres*)
	|	inst !inst.méthode()
Paramètre	: :=	valeur
	|	*Paramètres*,valeur

TAB. 4.2 – Grammaire des tests abstraits.

Nous présentons table 4.2 la grammaire des tests abstraits. Cette grammaire est très proche de la grammaire des schémas de tests. Nous pouvons constater que les boucles ont disparu ainsi que les groupes. Tous les appels de méthode sont instanciés. En revanche, la corégion et l'alternative restent présentes. Comme nous allons le voir juste après, ces constructions peuvent être traduites de différentes manières. Il est important de noter que ces deux éléments doivent être traduits avant la génération des tests concrets. Seules les séquences constituées uniquement d'appels instanciés sont ensuite traduites en tests exécutables.

4.2.5 Alternative et corégion pour la génération de tests

Nous avons vu que les constructions alternative et corégion n'étaient pas traduites lors de la génération des tests abstraits [6]. Lorsque nous allons produire des tests concrets, ces

[6]Ces deux constructions ont été initialement définies pour produire des objectifs de tests pour l'outil TGV. Leur sémantique étant définie au moment de la génération de l'objectif de test TGV, nous pouvons envisager une autre sémantique lorsque nous souhaitons générer des tests exécutables.

constructions devront être traduites en code exécutable. Nous proposons dans cette section plusieurs traductions pour ces opérations. Cependant, nous n'avons pas eu l'occasion de vérifier par l'expérience la pertinence de ces deux opérateurs pour la génération de tests.

La traduction la plus intuitive que l'on a de la corégion est d'encapsuler chaque appel de la corégion dans un thread séparé. Chaque thread s'exécute alors de manière parallèle. L'alternative pourrait elle se traduire par un test comportant plusieurs chemins, le choix de prendre un chemin plutôt qu'un autre se faisant de manière non-déterministe lors de l'exécution. Néanmoins, il n'est pas raisonnable de produire des tests qui sont non-déterministes. En effet, il est important, pour l'ingénieur en charge du test, de pouvoir dégager des séquences d'appels qui conduisent systématiquement à une erreur. Si le cas de test est non-déterministe, plusieurs exécutions du même cas de test peuvent conduire à des résultats différents. Bien entendu, ce cas peut arriver lorsque l'application est elle-même non-déterministe, mais, dans la mesure du possible nous pensons qu'il est toujours préférable d'obtenir des comportements reproductibles afin de permettre l'identification et la correction de l'erreur.

Les schémas `S2` et `S3` ont été définis afin d'illustrer ces deux mécanismes. Nous proposons ici une interprétation pour ces deux constructions. La traduction de ces constructions peut se faire à plusieurs niveaux. Nous pouvons partir des tests abstraits contenant les constructions non interprétées et les interpréter lors d'une seconde passe. Nous pouvons aussi inclure l'interprétation de ces constructions directement dans la fonction de traduction. Pour finir, nous pouvons traduire ces constructions au moment de produire les tests concrets.

Nous avons choisi de faire cette traduction lors d'une seconde passe produisant des tests abstraits. En faisant ce choix nous gardons la possibilité d'implanter d'autres manières de traduire ces constructions. Comme nous produisons des tests abstraits, ils pourront être traduits vers diverses plates-formes d'exécution. Nous gardons de cette manière une certaine flexibilité quant à l'utilisation de ces constructions et aux sémantiques que nous pouvons leur donner.

Utilisation de la corégion : `S2`

```
S2 = MTC!sch1.newRdyP12 ; <MTC!sch1.newRdyP12><MTC!sch1.rdySwapP13> ;
     MTC!sch1.rdySwapP13^2..2
```

$newRdyP12 = \{new(x), rdy(x)\}$ avec $x = \{p1, p2\}$
$RdySwapP13 = \{rdy(y), swap()\}$ avec $y = \{p1, p2, p3\}$

Comme chaque groupe se déplie en 4 appels, nous avons 4 appels pour le premier groupe du schéma, puis $4 * 4$ appels pour la corégion, puis $4 * 4$ appels pour la boucle de deux appels au groupe $rdySwapP13$. Comme tous ces éléments sont en séquence, nous générons toutes les combinaisons. Le nombre de tests générés par `S2`, est donc de $4 * (4 * 4) * (4 * 4) = 1024$.

Voici quelques exemples de tests abstraits générés :

```
seq_21 = MTC!Sch1.new(p1) ; <MTC!Sch1.new(p1)><MTC!Sch1.rdy(p1)> ;
```

```
        MTC!Sch1.rdy(p1) ; MTC!Sch1.rdy(p1)
seq_22 = MTC!Sch1.new(p1) ; <MTC!Sch1.new(p1)><MTC!Sch1.rdy(p1)> ;
        MTC!Sch1.rdy(p1) ; MTC!Sch1.rdy(p2)
...
seq_2i = MTC!Sch1.new(p1) ; <MTC!Sch1.rdy(p1)><MTC!Sch1.rdy(p2)>;
        MTC!Sch1.swap() ; MTC!Sch1.rdy(p1)
...
seq_2k = MTC!Sch1.new(p1) ; <MTC!Sch1.rdy(p2)><MTC!Sch1.rdy(p1)> ;
        MTC!Sch1.swap() ; MTC!Sch1.rdy(p1)
...
seq_2n = MTC!Sch1.rdy(p1) ; <MTC!Sch1.new(p1)><MTC!Sch1.rdy(p1)> ;
        MTC!Sch1.rdy(p3) ; MTC!Sch1.rdy(p1)
...
```

Si l'utilisateur décide de mettre plusieurs appels dans une corégion, nous comprenons qu'il n'attache pas d'importance à l'ordre dans lequel les appels sont effectués. Nous pouvons donc ordonner de manière aléatoire les appels d'opérations et produire ainsi une séquence originale. Si la séquence est produite aléatoirement, son exécution est déterministe (sauf si l'application n'est pas déterministe).

Par exemple, lors de la production du test concret de *seq_2i*, nous pouvons générer l'une des deux séquences abstraites suivantes :

```
seq_2ia = MTC!Sch1.new(p1) ; MTC!Sch1.rdy(p1) ; MTC!Sch1.rdy(p2) ;
        MTC!Sch1.swap() ; MTC!Sch1.rdy(p1)
seq_2ib = MTC!Sch1.new(p1) ; MTC!Sch1.rdy(p2) ; MTC!Sch1.rdy(p1) ;
        MTC!Sch1.swap() ; MTC!Sch1.rdy(p1)
```

En effectuant certains choix, nous pouvons être amené à générer plusieurs fois les mêmes tests (ie des tests syntaxiquement identiques). La manière de traiter ce genre de cas n'est pas évidente. Par exemple, si nous générons la séquence *seq_2ib*, lorsque nous allons traiter la séquence *seq_2k*, nous pouvons générer l'une des séquences suivantes :

```
seq_2ka = MTC!Sch1.new(p1) ; MTC!Sch1.rdy(p2) ; MTC!Sch1.rdy(p1) ;
        MTC!Sch1.swap() ; MTC!Sch1.rdy(p1)
seq_2kb = MTC!Sch1.new(p1) ; MTC!Sch1.rdy(p1) ; MTC!Sch1.rdy(p2) ;
        MTC!Sch1.swap() ; MTC!Sch1.rdy(p1)
```

Si la séquence générée est *seq_2ka*, nous aurons généré deux fois la même séquence. Nous pouvons envisager alors un autre tirage ou décider de ne générer qu'une seule séquence au lieu de deux.

Utilisation de l'alternative : `S3`

```
S3 = MTC!sch1.newRdyP12 ; (MTC!sch1.newRdyP12 || MTC!sch1.rdySwapP13) ;
     MTC!sch1.rdySwapP13^2..2
```

Le schéma S3 est très semblable au schéma S2 pour ce qui est de sa structure et donc du nombre de tests générés. En effet, à la place de la corégion nous avons une alternative. Or, ces deux constructeurs se déplient de la même manière, l'outil produira donc aussi 1024 séquences de test abstraites à partir de S3.

Voici quelques exemples de tests abstraits générés :

```
seq_31 = MTC!Sch1.new(p1) ; (MTC!Sch1.new(p1)|MTC!Sch1.rdy(p1)) ;
         MTC!Sch1.rdy(p1) ; MTC!Sch1.rdy(p1)
seq_32 = MTC!Sch1.new(p1) ; (MTC!Sch1.new(p1)|MTC!Sch1.rdy(p1)) ;
         MTC!Sch1.rdy(p1) ; MTC!Sch1.rdy(p2)
...
seq_3i = MTC!Sch1.new(p2) ; (MTC!Sch1.rdy(p2)|MTC!Sch1.swap()) ;
         MTC!Sch1.rdy(p1)
...
seq_3k = MTC!Sch1.rdy(p1) ; (MTC!Sch1.new(p1)|MTC!Sch1.rdy(p1)) ;
         MTC!Sch1.rdy(p3) ; MTC!Sch1.rdy(p1)
...
```

L'alternative est une construction exprimant un choix non-déterministe. Là encore, nous ne pouvons pas nous permettre de générer un test qui exprimerait ce non-déterminisme. Néanmoins, si l'utilisateur utilise cette construction pour générer des tests, nous pouvons l'interpréter comme le fait qu'il préfère laisser l'outil faire le choix à sa place. En conséquence, lorsque nous avons une alternative, au moment de générer les tests concrets, nous pouvons tirer aléatoirement l'appel ou la séquence d'appel qui sera effectué.

Par exemple, lors de la production du test concret de *seq_31*, nous pouvons générer, au choix, l'une des deux séquences suivantes :

```
seq_31a = MTC!Sch1.new(p1) ;  MTC!Sch1.rdy(p1) ; MTC!Sch1.rdy(p1) ;
          MTC!Sch1.rdy(p1)
seq_31b = MTC!Sch1.new(p1) ; MTC!Sch1.new(p1) ; MTC!Sch1.rdy(p1) ;
          MTC!Sch1.rdy(p1)
```

Tout comme pour la corégion, nous pouvons nous retrouver à générer deux fois la même séquence. Le même choix s'offre alors à nous, à savoir, regénérer une nouvelle séquence ou supprimer la séquence redondante. Dans certains cas, le choix s'imposera de lui-même puisqu'il peut arriver que toutes les possibilités existent déjà.

Une fois que les séquences de test abstraites sont générées, il reste à générer les tests concrets qui seront effectivement exécutés sur l'application.

4.3 Génération de tests concrets

Nous illustrons ici comment, à l'aide du prototype Tobias et du langage de synthèse de tests, nous pouvons générer des séquences de test concrètes. Comme l'outil produit des

séquences de test abstraites, il nous faut les traduire vers le langage dans lequel les tests seront exécutés.

La figure 4.2 montre les différentes étapes de génération à partir d'un schéma de tests. Nous pouvons voir que le schéma de tests est déplié une première fois grâce à la fonction

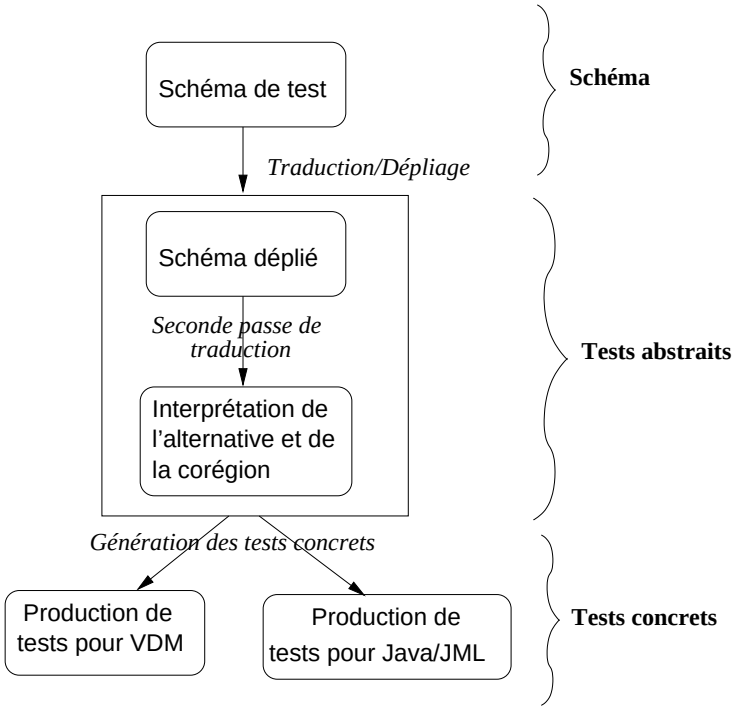


FIG. 4.2 – Étapes de la génération de tests.

de traduction définie plus haut. Les tests abstraits produits peuvent encore faire appel aux mécanismes de corégion ou d'alternative. Donc, avec une seconde passe, nous traduisons ces éléments de sorte que les tests abstraits produits ne soit plus que de simples séquences d'appels de méthodes. Ces séquences peuvent alors être concrétisées en VDM ou en Java [7].

4.3.1 Génération de tests pour VDM

En VDM l'initialisation du système se fait grâce à une opération spéciale `init`. VDM, contrairement à d'autres langages comme Java, n'a pas besoin de charger un état. L'opération `init` sert à définir les éventuelles valeurs de départ des variables d'état. Ainsi, les différentes valeurs qui sont spécifiées dans le déploiement ne servent pas directement à déployer l'application mais permettent d'instancier certains paramètres. Par exemple, nous pouvons définir l'instanciation suivante pour les différentes variables utilisées dans le langage de synthèse de tests :

[7] D'autres études sont en cours pour traduire ces séquences vers d'autres langages tels que le C/C++.

```
p1 = "p1"
p2 = "p2"
p3 = "p3"
Sch1 = ""
```

Lorsque nous allons produire les tests VDM, les paramètres `p1`, `p2`, et `p3` seront remplacés respectivement par les valeurs `"p1"`, `"p2"`, et `"p3"` dans les appels d'opération.

Les tests concrets sont des séquences d'appels d'opérations séparés par le caractère ','. A chaque début d'un cas de test, il faut faire appel à l'opération `init` permettant de restaurer l'état initial. Par ailleurs, VDM n'étant pas un langage orienté objet, les instances `MTC` ou `Sch1` ne sont plus considérées ici. Elles disparaissent donc des appels. L'atelier VDMTools permet de jouer de tels tests directement. Nous aurons ainsi un fichier contenant les tests produits à partir du schéma `S1`. En voici un extrait :

```
"Cas de test 1", init, new("p1"), new("p1"), rdy("p1"), rdy("p1"),
"Cas de test 12", init, new("p1"), new("p1"), rdy("p1"), rdy("p2"),
...
"Cas de test 1i", init, new("p2"), rdy("p2"), swap(), rdy("p1"),
...
"Cas de test 1k", init, rdy("p1"), new("p1"), rdy("p1"), rdy("p3"), rdy("p1"),
...
```

Les chaînes de caractères entre guillemets (`"`), comme `"Cas de test 1"`, ne sont pas interprétées par l'atelier, sauf lorsqu'il s'agit du paramètre d'une opération.

Les appels d'opérations sont exécutés les uns à la suite des autres. Il s'avère que le fichier d'entrée est limité en taille et ne peut excéder un certain nombre d'appels. L'élaboration d'un pilote de test nous a permis de passer outre cette limitation (cf Chapitre 7).

4.3.2 Génération de tests Java

Les tests que nous générons pour Java sont exécutés à l'aide du framework de test JUnit. Avant l'exécution de chaque cas de test, JUnit exécute une méthode permettant d'initialiser le système, c'est-à-dire de créer les instances qui vont être utilisées pour le test. L'utilisateur doit donc définir de quelle manière les instances vont être créées. Rappelons qu'il a préalablement défini le type de ces objets (`p1 :PID`, `p2 :PID` etc.) dans la partie déploiement de l'application. Par exemple, pour l'exemple du gestionnaire de processus, l'utilisateur peut définir les instanciations suivantes :

```
p1 = new PID("p1");
p2 = new PID("p2");
p3 = new PID("p3");
Sch1 = new Scheduler();
```

La classe de test peut maintenant être générée. En premier lieu, nous déclarons sous la forme d'attributs de la classe, les différentes variables définies dans le déploiement :

```
public class TestS1 extends TestCase {

  PID p1;
  PID p2;
  PID p3;
  Scheduler Sch1;
```

Nous générons ensuite la méthode `setUp` qui est appelée avant chaque cas de test et qui permet d'initialiser l'état pour chaque test :

```
  protected void setUp(){
    p1 = new PID("p1");
    p2 = new PID("p2");
    p3 = new PID("p3");
    Sch1 = new Scheduler();
  }
```

Pour finir, chaque cas de test est stocké dans une méthode qui lui est propre :

```
...
  public void testSequence_11(){
     try{
       Sch1.new(p1);
       Sch1.new(p1);
       Sch1.rdy(p1);
       Sch1.rdy(p1);
    } catch( JMLEntryPreconditionError e$) {
        ...
    } catch( JMLAssertionError e$){
        ...
    }
...
```

L'outil JUnit identifie les cas de tests en récupérant les méthodes de la classe de test qui commencent par `test`. Puis, il les exécute les uns à la suite des autres en faisant en séquence un appel à la méthode `setUp()`, un appel à la méthode contenant le cas de test et un appel à la méthode `tearDown()` qui permet de mettre le système dans un état stable après le cas de test. Cette dernière méthode est, elle aussi, définie par l'utilisateur. Étant donné que nous réinitialisons l'application avant chaque test, nous n'avons pas eu à utiliser cette méthode.

4.4 Synthèse

Nous proposons ici une technique permettant de structurer les tests et d'amplifier la créativité de l'ingénieur de test. La définition de campagnes de tests permettant de mettre en œuvre différentes instanciations de l'application ainsi que la définition de groupes de méthodes permettent à l'utilisateur de structurer ses tests. Les schémas de tests permettent

à l'utilisateur de définir simplement et rapidement de nombreux tests. La structuration des données de tests lui permet d'adapter rapidement les tests à ses besoins : si une valeur ne convient pas, il lui suffit de la changer au niveau du groupe concerné et de regénérer les tests. La génération combinatoire va amplifier sa créativité en produisant des tests auxquels il n'aurait pas pensé. Nous offrons ici une chaîne complète de production de tests permettant à l'utilisateur de produire des tests exécutables. Aujourd'hui nous n'avons ciblé que les langages VDM et Java mais nous pouvons facilement imaginer traduire les tests abstraits dans d'autres langages comme le C ou le C++.

Cependant, de par la nature même du fonctionnement de cette technique, nous tombons sur le problème de l'explosion combinatoire du nombre de tests. Si cet élément n'est pas correctement maîtrisé, l'utilisateur risque de générer un trop grand nombre de tests les rendant inexploitables (temps d'exécution et d'analyse trop long).

Afin d'éviter ce problème, il faut, pour chaque schéma de tests, réussir à trouver le bon compromis entre le nombre de valeurs de paramètres, la longueur des séquences que l'on souhaite générer, le nombre de méthodes par groupe et le nombre d'itérations de chaque appel de groupe. Il n'y a malheureusement pas de recettes miracles, c'est à l'utilisateur de faire les bons choix de test. Nous montrons comment il est possible d'utiliser cette approche de manière intelligente chapitre 5. De plus, nous proposons , chapitres 6 et 7, des techniques permettant à l'utilisateur de mieux contrôler le nombre de tests générés et d'optimiser le temps d'exécution des tests en fonction de l'exécution d'autres tests.

Chapitre 5

Étude de cas industrielle en Java

Ce chapitre traite une étude de cas effectuée lors de la collaboration entre la société Gemplus et le laboratoire LSR au cours du projet COTE. Depuis plusieurs années déjà, la société Gemplus s'est intéressée au développement d'applications pour carte à puce (smartcard) à l'aide de méthodes formelles et notamment à l'aide de la méthode B [Abr96]. Gemplus s'est également intéressé au langage JML afin de valider des applications embarquées sur carte à puce à l'aide de techniques de preuve implantées dans l'outil Jack [BRL03].

La preuve permet de garantir la correction d'une application par rapport à sa spécification formelle. Toutefois, le caractère indécidable des logiques utilisées empêche le solveur de résoudre automatiquement toutes les obligations de preuve. De plus, l'activité de preuve ne permet de garantir que la conformité de l'application vis-à-vis de sa spécification formelle, or, le cahier des charges informel est rarement entièrement décrit par la spécification formelle. Dans un tel contexte, d'autres activités de validation, comme le test, doivent être utilisées.

Ce chapitre présente les résultats de notre activité de test, effectuée à l'aide des outils développés au sein de notre laboratoire, dont l'outil Tobias présenté dans cette thèse.

Ce chapitre se découpe en quatre parties. En premier lieu, nous présentons le contexte de l'étude ainsi que l'application testée. Puis, nous présentons notre démarche expérimentale. Nous définissons ensuite les différentes campagnes de tests qui ont été crées. Pour finir, nous présentons les résultats obtenus.

5.1 Contexte et application

5.1.1 Contexte

L'application que nous utilisons dans ce chapitre est une application bancaire simplifiée, fournie par Gemplus. Cette application a été initialement conçue dans le but d'évaluer l'outil Jack. Elle a été développée en Java et spécifiée à l'aide du langage JML. Elle nous a été fournie dans le cadre du projet RNTL COTE afin de valider l'outil Tobias et le principe des schémas

de tests.

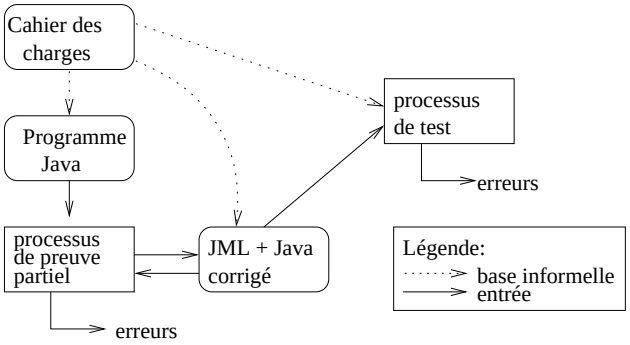


FIG. 5.1 – Processus de validation.

Nous montrons figure 5.1 les différentes phases du processus de développement de l'application. Le programme Java a été développé à partir du cahier des charges informel. Elle a ensuite fait l'objet d'un premier processus de validation basé sur des techniques de preuve. Afin de réaliser cette étape, une spécification JML a été écrite. Plusieurs erreurs ont pu être détectées et corrigées grâce à cette méthode. Cette activité, effectuée par Gemplus, est en partie décrite dans [BRL03]. Au cours du processus de preuve, cinq des huits classes de l'application (cf. figure 5.2) ont été validées et 944 des 1055 obligations de preuve ont été automatiquement prouvées.

La seconde étape de validation, qui est présentée ici, est basée sur le test et sur l'utilisation des outils Tobias et Jartège, un outil de test aléatoire développé dans l'équipe [Ori02]. Nous n'avions pas connaissance des résultats obtenus par l'équipe de Gemplus lors de cette phase de test. En particulier, nous ne savions pas quelles parties de l'application avaient été prouvées, et nous avions comme matériel à notre disposition le code source de l'application en Java, les spécifications JML ainsi qu'un ensemble d'exigences informelles.

5.1.2 Étude de cas

L'application bancaire fournie est une petite application (500 lignes de code Java) qui permet de gérer des transferts d'argent entre comptes. L'administrateur (BankOfficer) peut créer des comptes (Account). L'utilisateur (Customer) peut consulter le solde de ses comptes et effectuer des transferts d'un compte sur un autre. L'utilisateur peut aussi enregistrer des règles de transfert afin de programmer des transferts périodiques. Ces règles peuvent être de deux types : dépense (SpendingRule) ou épargne (SavingRule). L'application inclut de plus un convertisseur de monnaie.

Cette étude de cas est une version très simplifiée d'une application réelle. L'application réelle tourne sur un serveur qui est connecté à plusieurs terminaux pour carte à puce. La gestion des terminaux n'a pas été prise en compte afin de simplifier l'étude de cas.

Le programme de l'application est structuré en huit classes (cf. figure 5.2) parmi lesquelles :

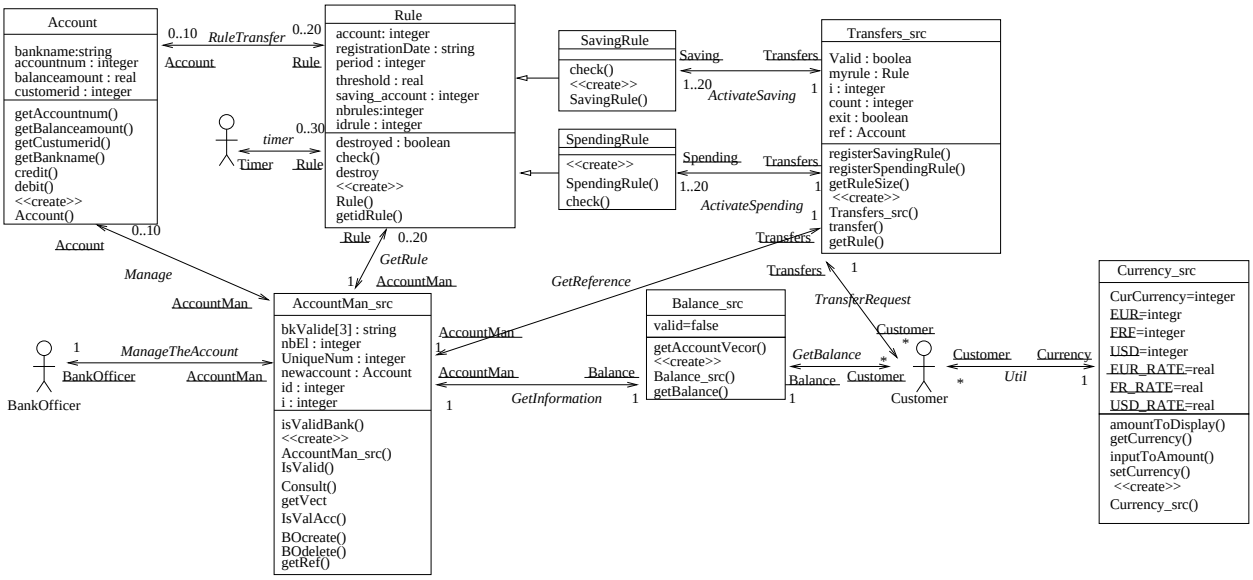


FIG. 5.2 – Diagramme de classes de l'application bancaire.

- une classe représentant les comptes (Account),
- une classe permettant de créer et de supprimer des comptes (AccountMan_src),
- une classe permettant de définir des règles d'épargne et de dépense servant à transférer automatiquement de l'argent d'un compte sur un autre en fonction de déclencheurs prédéfinis (Transfers_src),
- une classe permettant à l'utilisateur de visualiser le solde de ses comptes (Balance_src),
- et le convertisseur de monnaie (Currency_src).

Les trois classes restantes définissent les principes des règles de transfert.

La table 5.1 présente diverses mesures en termes de nombre de lignes de code. Nous pouvons constater que le nombre de lignes de code JML est parfois plus important que le nombre de lignes de code de Java. Ceci est dû au fait que les spécifications ont été écrites en vue d'effectuer des preuves et au fait que, bien que l'équipe de Gemplus possède une expertise dans le développement de spécification formelles, elle utilisait JML pour la première fois.

Nous présentons figure 5.3 un exemple de code Java augmenté de sa spécification JML. Ce code représente un morceau de la classe Currency_src. Celui-ci a été réduit pour des raisons de clarté : il ne prend en compte que les monnaies Franc Français (FRF) et Euro (EUR). Ces deux monnaies sont représentées par les valeurs entières 1 et 2. La valeur 0 est utilisée pour définir le code "monnaie non définie". Le taux de change entre les Francs et l'Euro est fixé (1 EUR = 6.55957 FRF). La méthode `amountToDisplay` convertit un montant en Francs dans une monnaie choisie par l'utilisateur à l'aide de la méthode `setCurrency`.

Comme nous l'avons expliqué chapitre 3 section 3.2.1, il existe deux manières de décrire une spécification JML : soit entre `/*@` et `@*/`, soit en commençant la ligne par `//@`. Les spécifications doivent se trouver juste avant la déclaration de l'élément spécifié (méthode ou

attribut). Les spécifications JML adoptent, de plus, une approche "par contrats" basée sur trois types d'assertions : invariant, préconditions et postconditions.

L'invariant lignes 19-20 figure 5.3 spécifie que l'attribut `CurCurrency` doit avoir une valeur toujours comprise entre 0 et 2. Deux autres invariants lignes 13 et 17 expriment le fait que les taux de change EUR_RATE et FRF_RATE ont une valeur figée, respectivement à 6.55957 et à 1.

Les préconditions définies dans cette application sont pour la plupart toujours vraies comme nous pouvons le constater lignes 22 et 38 figure 5.3. En effet, le caractère critique de l'application bancaire et la possibilité de comportements frauduleux de la part de certains utilisateurs obligent à avoir un style de programmation défensif. Les préconditions tiennent compte de cet état de fait en étant spécifiées à "vrai". De cette manière, il est clairement indiqué que les méthodes du système doivent être capables de gérer n'importe quel type d'entrée. En revanche, en cas de comportement anormal, la méthode doit retourner un message d'erreur ou une exception, sauf lorsqu'il est explicitement indiqué que la méthode ne doit pas lever d'exception.

La postcondition de la méthode `setCurrency` (lignes 41 à 43, figure 5.3) spécifie que l'attribut `CurCurrency` doit avoir une valeur égale à 0, 1 ou 2. La spécification de l'application complète contient une large variété de postconditions allant de la postcondition "true" (voir la postcondition de la méthode `amountToDisplay` ligne 23, figure 5.3), à des postconditions plus complexes faisant référence aux variables d'état, mettant en relation les états finaux et initiaux (caractérisés par `\old`) et utilisant les quantificateurs universel et existentiel sur des éléments de type ensemble ou vecteur.

La clause `exsures`, figure 5.3 ligne 23, permet de spécifier que la méthode `amountToDisplay` ne doit jamais lever d'exception.

Classes	Lignes Java	Lignes JavaDoc	Lignes JML
Transfer_src	116	34	150
AccountMan_src	105	51	236
Currency_src	93	20	28
Balance_src	64	38	58
Spending_rule	40	33	42
Saving_rule	40	33	42
Rule	40	22	23
Account	30	20	36
Total	518	251	615

TAB. 5.1 – Quelques métriques sur l'application bancaire.

```
1   /** Copyright (c) 2002 GEMPLUS group.
     *  All Rights Reserved.
     *----------------------------------------
     *  Project name:  COTE  - Case Study -
5    *  Version 1.0 1/9/2002
     *-------------------------------------  */
    package banking;

    public class Currency_src {
10    private static final int EUR = 1;
      //@ private invariant EUR == 1;
      private static final float EUR_RATE=6.55957f;
      //@ private invariant EUR_RATE == 6.55957f;
      private static final int FRF = 2;
15    /*@ private invariant FRF == 2; @*/
      private static final float FRF_RATE =1.0f;
      /*@ private invariant FRF_RATE ==1.0f; @*/
      private /*@ spec_public */ int CurCurrency;
      /*@ private invariant (CurCurrency >=0
20                         && CurCurrency <3); @*/

    /*@ requires true;
      @ exsures (Exception e) false;  @*/
      public String amountToDisplay(float amount){
25      float rate;
        String toDisplay;
        switch (CurCurrency) {
          case (EUR) : rate = EUR_RATE; break;
          case (FRF) : rate = FRF_RATE; break;
30        default : rate = 1;
        }
        toDisplay = (Double.toString
            (Math.round((amount/rate)*100)/100));
        System.out.println("aToD = " + toDisplay);
35      return Double.toString(toDisplay);
      }

    /*@ requires true;
      @ modifies CurCurrency ;
40    @ ensures  CurCurrency >= 0 ;
      @ ensures  CurCurrency == 0
                 || CurCurrency == 1
                 || CurCurrency == 2 ;
      @ exsures (Exception e) false;  @*/
45    public void setCurrency(String s) {
        if (s == null) {CurCurrency = 0; return;}
        if (s.compareToIgnoreCase("FRF")==0)
               CurCurrency = FRF;
        else if (s.compareToIgnoreCase("EUR")==0)
50             CurCurrency = EUR;
        else {CurCurrency = 0;}
    } }
```


FIG. 5.3 – Extrait de la classe Currency_src.

5.2 Approche expérimentale

L'objectif premier de cette étude était de valider les outils développés au sein de notre équipe, et en particulier l'outil Tobias articulé autour du langage de schéma de tests. Gemplus nous a fourni les spécifications informelles de l'application, les spécifications JML et le code source Java de l'application.

Notre but était donc de trouver des erreurs dans l'application qui nous a été fournie, à l'aide de ces outils. Plus précisément, il s'agissait de trouver des incohérences entre le code de l'application, la spécification JML et la spécification informelle. Nous avons identifié trois cas :

- Le code n'est pas conforme aux spécifications JML. Ce type d'erreur est détecté pendant le test lorsqu'une assertion JML est violée ou lorsque qu'une exception Java n'est pas rattrapée par l'application.
- Les spécifications JML ne sont pas conformes aux spécifications informelles mais sont conformes au code Java. Un tel type d'erreur ne peut être détecté que grâce à une analyse humaine des spécifications JML ou du résultat de l'exécution des tests.
- Les spécifications informelles et JML, ainsi que le code de l'application sont cohérents mais le comportement observé n'est pas en accord avec le "bon sens". Par exemple, c'est le cas lorsque certains cas importants ont été omis du cahier des charges informel, comme nous pouvons le voir section 5.4 dans le paragraphe "*Comportement contre-intuitif*".

Le travail de validation a été effectué par deux équipes sur une période de trois jours. La première équipe a effectué une revue de code et a utilisé un outil de génération de tests aléatoires. La deuxième équipe a utilisé l'outil Tobias de génération combinatoire de tests basé sur le langage de schéma de tests (chapitre 4). En plus, de l'évaluation automatique des assertions JML, nous avons effectué une analyse de l'exécution des tests. Cette analyse manuelle nous a permis de détecter les cas où l'application était conforme à la spécification JML mais pas au cahier des charges informel ou au "bon sens" de l'ingénieur.

5.2.1 Revue de code et test aléatoire

La première équipe a utilisé le test aléatoire. Pour cela, l'équipe a utilisé un outil de génération aléatoire de tests pour des programmes Java spécifiés en JML.

Néanmoins, avant d'utiliser cet outil, l'équipe a effectué une revue de code. La revue de code consiste à lire le code source de l'application afin de trouver des erreurs. Lorsque des erreurs ont été trouvées, des tests ont été écrits à la main afin de les illustrer. Ici, la revue de code a permis de détecter des cas suspects nécessitant un effort de test plus important.

Jartege [Ori02] (Java Random Test Generator) est un outil, développé au sein du LSR, qui permet de générer dynamiquement des tests pour des applications Java spécifiées en JML. L'outil produit à la volée des appels de méthode, choisis de manière aléatoire, tout en

utilisant la spécification pour d'une part produire des appels qui satisfont la spécification, et d'autre part avoir un oracle automatique. De plus, l'outil offre des mécanismes permettant de guider la génération (utilisation de poids sur les classes et les méthodes, raffinement des fonctions de génération au cas par cas, contrôle du nombre d'instances créées etc.).

5.2.2 Génération combinatoire à partir du cahier des charges

Nous avons utilisé l'outil Tobias permettant de manipuler le langage de synthèse de tests. Pour cela, nous sommes partis du cahier des charges informel fourni par Gemplus. Nous en avons déduit un ensemble d'exigences. Par exemple, les transferts doivent être effectués si le montant est correctement défini et si les comptes existent et sont différents. De plus, un transfert doit modifier les deux comptes spécifiés, et ne pas modifier les autres comptes.

Nous avons pris une à une ces exigences fonctionnelles et avons défini un ou plusieurs schémas de tests permettant de les vérifier. Un des défauts de notre approche et que nous n'utilisons pas, comme c'est le cas pour les outils type BZ-TT ou Casting, la spécification formelle lorsqu'elle est disponible pour juger de la pertinence de nos tests à l'aide d'un critère de couverture de la spécification. C'est donc l'ingénieur qui, seul, décide si les tests produits couvrent suffisament la spécification ou s'il est nécessaire d'en produire d'autres. Par contre, nos tests étant construits à partir du cahier des charges, il est plus facile d'établir la traçabilité entre les tests et les exigences de haut niveau.

5.3 Campagnes de test

Dans cette section, nous montrons comment nous avons utilisé l'outil Tobias et le langage de synthèse de tests. Plutôt que de tenter d'être exhaustif, nous cherchons ici à montrer une méthodologie qui permet d'éviter en partie les problèmes liés à cette approche combinatoire.

Comme nous l'avons exposé dans la section précédente, la conception des schémas de tests a été guidée par les fonctionnalités à tester. Chaque fonctionnalité fait l'objet d'une campagne de tests. Une fonctionnalité a souvent été découpée en sous-propriétés faisant l'objet d'un schéma de tests. Nous ne détaillons pas ici tous les schémas de tests qui ont été créés. Nous en avons extrait quelques-uns afin de montrer, d'une part la démarche employée, d'autre part comment utiliser le langage de schéma de tests.

5.3.1 Fonctionnalité 1 : Opérations d'ouverture et de clôture de compte

La première propriété qui a été identifiée concerne les opérations d'ouverture et de clôture des comptes. Cette propriété très générale a été découpée en sous-propriétés. L'une de ces sous-propriétés est que la création d'un compte, considérée de manière isolée, doit se passer

correctement. Plus précisément, nous cherchons à montrer que "si les paramètres correspondent au comportement nominal, le compte est créé avec les attributs adéquats, et sinon, le compte n'est pas créé." Voici la spécification informelle de l'opération de création de compte telle qu'elle nous a été fournie :

```
public BOcreate ( cust_id in integer, bn in string, balance in real )
       return integer
```

"*Le gestionnaire de compte s'assure de l'unicité de l'identifiant de compte. Ceci est réalisé par l'attribut UniqueNum toujours croissant. On fait l'hypothèse que le nombre de comptes créés est inférieur à la capacité d'un Integer. Il faut ensuite tester que la banque est une banque référencée dans une liste de banques possibles. Attention, cette notion n'existe pas dans la spécification où il est possible de créer une banque fictive. Il faut ensuite vérifier que le solde est positif. On prend comme acquis que l'identifiant rentré par le BankOfficer est correct (positif ou nul) et fait partie des identifiants utilisables par un customer. En retour, on obtient si tout va bien le numéro de compte. Au cas où l'une des conditions n'est pas respectée, il faut générer un message d'erreur ou un code d'erreur. Ce sera soit : 01 erreur balance, soit l'exception BadParameterException (01), 02 erreur banque, soit l'exception BadParameterException (02).*"

Nous avons alors défini deux groupes. Le premier groupe, `BOcreate_Gr`, nous permet de créer un compte. Le second groupe, `GetBalance_Gr`, nous permet de consulter le solde des comptes. Nous avons instancié ces groupes de la façon suivante :

```
BOcreate_Gr = {AccountMan_src.BOcreate(cust_id, bn, balance)} avec :
              cust_id ∈ {0, 1, 12}
              bn ∈ {"bnp", "laposte"}
              balance∈ {0.0f, 123.5f, -1234.5f, 1.0f, Float.POSITIVE_INFINITY,
                              Float.MAX_VALUE}
GetBalance_Gr = {Balance_src.getBalances(cust_id)} avec
               cust_id ∈ {0, 1, 12}
```

Le choix des paramètres est guidé par l'envie d'essayer des valeurs correctes (123.5f, 1.0f, "bnp"), des valeurs incorrectes (-1234.5f, "poste"), "poste" n'étant pas une banque enregistrée dans l'application, des valeurs suspectes (0.0f) et des valeurs limites (MAX_VALUE, POSITIVE_INFINITY). Concernant les numéros de client, nous avons fait l'hypothèse qu'aucun traitement particulier était fait sur la valeur du numéro. En revanche, nous avons considéré intéressant de vérifier le fonctionnement de l'application si les numéros de clients ne sont pas consécutifs. Nous avons donc choisi une valeur potentiellement invalide (0), suivie de deux valeurs correctes non consécutives (1 et 12). Le comportement attendu est que les comptes sont créés lorsque les valeurs sont toutes correctes.

Nous devons aussi définir le déploiement de notre application pour pouvoir générer ces tests. Nous avons donc créé deux instances. Une instance, `CentralBancaire`, de la classe `AccountMan_src` et une instance, `Solde`, de la classe `Balance_src`. Afin de pouvoir générer

le code Java correspondant, nous devons définir de quelle manière nous créons ces deux instances. Nous avons donc défini les constructions suivantes dans Tobias :

```
CentralBancaire = new AccountMan_src();
Solde           = new Balances_src(CentralBancaire);
```

Le schéma qui a été créé fait appel aux instances déployées.

```
TestUneCreation = CentralBancaire.BOcreate_Gr ; Solde.GetBalance_Gr
```

Ce schéma génère 162 cas de tests. Un exemple de test généré pour Java/JUnit est montré figure 5.4. Aucune erreur n'a été relevée lors de l'exécution, aussi nous avons décidé, ponctuellement, d'injecter une erreur dans le code de l'application afin de vérifier la qualité des tests générés.

La méthode `BOcreate` fait appel, pour créer un nouveau compte, au constructeur de la classe `Account` en lui passant la valeur du paramètre `balance` en argument. Nous avons effectué cette modification de sorte que l'argument passé soit égal à `balance-1`. Cette suite de tests a alors produit 9 *fails* sur les 162 tests exécutés. C'est la précondition du constructeur de la classe `Account` qui a soulevé cette erreur. La chose la plus surprenante est que seuls les tests entraînant la création d'un compte au solde négatif (cas où le paramètre `balance` vaut 0) détectent cette erreur. De plus, la méthode `GetBalances()` ne nous permet pas de nous assurer directement de la valeur effective du solde d'un compte nouvellement créé. En effet, celle-ci nous renvoie un triplet contenant diverses informations dont le solde. Mais, comme ce triplet est encodé à l'intérieur d'un objet de type Vector en Java, nous ne pouvons pas afficher simplement son contenu. Il nous faudrait en effet parcourir les éléments du vecteur afin de les afficher. Ceci nous amène à deux conclusions :

- La spécification de l'application n'est pas assez précise pour détecter cette erreur.
- L'application n'est pas facilement testable sur ce point.

Nous avons donc ajouté une nouvelle méthode dans la classe `Balances_src`, appelée `PrettyPrintBalance` et prenant en paramètre un entier correspondant à un numéro de client.

Grâce aux mécanismes offerts par le langage de synthèse de tests et par l'outil Tobias, nous avons pu changer directement la méthode contenue dans le groupe `GetBalance_Gr` pour la remplacer par `PrettyPrintBalance` puis regénérer les tests correspondants. L'exécution de ces nouveaux tests et l'analyse des traces produites grâce à cette nouvelle méthode nous a également permis de constater que des comptes pouvaient être créés avec un solde égal à POSITIVE_INFINITY ce qui correspond à une situation suspecte.

```
GetBalance_Gr = {Balance_src.PrettyPrintBalance(cust_id)} avec
                cust_id ∈ {0, 1, 12}
```

5.3.2 Fonctionnalité 2 : création des règles d'épargne et de dépense

L'utilisateur a la possibilité d'enregistrer différentes règles d'épargne et de dépense. Plusieurs paramètres entrent en compte lors de la création d'une règle et une règle n'est créée

```
1   import java.util.* ;
    import java.lang.* ;
    import java.io.* ;
    import junit.framework.TestCase;
5   import org.jmlspecs.jmlrac.runtime.*;
    import banking.*;

    public class TestCreationSimple extends TestCase {
        AccountMan_src CentralBancaire;
10      Balances_src Solde;

        /**
         * setUp() est une méthode appelée avant chaque
         * appel à une méthode de test.
15       * Nous l'utilisons donc pour réinitialiser le système
         * sous test en recréant les instances requises.
         */
        protected void setUp(){
            CentralBancaire=new AccountMan_src();
20          Solde=new Balances_src(CentralBancaire);
        }

        /**
         * testSequence_0() correspond à un cas de test.
25       * Ici on crée un compte avec pour valeur 0, "bnp" et
         * FLOAT.MAX_VALUE. Puis on affiche récupère le solde
         * à l'aide de la méthode getBalances(x).
         * Une exception est levée si une assertion JML est violée.
         */
30      public void testSequence_0(){
            try {
                CentralBancaire.BOcreate(0, "bnp", Float.MAX_VALUE);
                Solde.getBalances(0);
            } //precondition violée
35          catch(JMLEntryPreconditionError e$) {
                System.out.println(e$.getMessage());
            } //postcondition, invariant, précondition interne violée
            catch(JMLAssertionError e$){
                int l$ =JMLChecker.getLevel();
40              JMLChecker.setLevel(JMLOption.NONE);
                try { //demande à JUnit d'afficher un fail.
                    junit.framework.Assert.fail("\n\t"+e$.getMessage());
                } finally {
                    JMLChecker.setLevel(l$);
45              }
            }
        }
    }
```

FIG. 5.4 – Un exemple de test pour JUnit.

que si les paramètres satisfont certaines conditions. Voici les spécifications informelles des opérations permettant d'enregistrer des règles d'épargne (saving rule) et de dépense (spending rule) :

```
public registerSavingRule(date in string, account in integer, threshold in real,
                          saving_account in integer, period in integer)
       return integer
```

"*Cette méthode permet d'enregistrer une nouvelle règle de transfert automatique d'un compte (account) vers un compte d'épargne (saving_account) si le compte dispose d'une balance supérieure à un seuil (threshold). Cette règle doit être testée suivant une certaine périodicité (period). Une fois que la règle est enregistrée, elle sera exécutée dans t0+period. Si un des paramètres n'est pas conforme alors, soit une exception de type BadParameterException (type) est lancée, soit un code d'erreur est retourné (type). Le seuil est positif ou nul, sinon le code d'erreur 04 est retourné. Si la période est nulle ou négative le code d'erreur 06 est retourné. Si account ou saving_account ne sont pas des comptes valides le code d'erreur 03 est retourné.*"

```
public registerSpendingRule(date in string, account in integer, threshold in real,
                            saving_account in integer, period in integer)
       return integer
```

"*Cette méthode permet d'enregistrer une nouvelle règle de transfert automatique d'un compte d'épargne (saving_account) vers un compte courant (account). Si le montant du solde du compte désigné par la variable account est inférieure à la valeur de la variable threshold et si le solde du compte désigné par la variable saving_account le permet, une somme d'argent est transférée du deuxième compte vers le premier. Le montant de cette somme doit faire en sorte que le solde du compte désigné par account soit égal à la valeur de la variable threshold. Toutefois, si le solde du compte désigné par la variable saving_account ne permet pas que le solde du compte désigné par la variable account soit égal à la valeur de la variable threshold alors le transfert n'a pas lieu. Cette règle doit être évaluée à chaque instant correspondant aux intervalles de temps period, exprimé en secondes. Dès que la règle est enregistrée, elle est activée à t0+period. Le seuil est positif ou nul, sinon le code d'erreur 04 est retourné. Si account ou saving_account ne sont pas des comptes valides si, le code d'erreur 03 est retourné. Si la période est négative, le code 06 est retourné.*"

Dans un premier temps, nous avons choisi de tester les cas pouvant poser problème pour ces deux règles. Pour éviter les problèmes d'explosion combinatoire dus au nombre important de paramètres, nous avons effectué un découpage par type d'erreur. Nous avons donc produit un schéma de tests permettant de vérifier qu'une règle n'est pas enregistrée si le seuil indiqué est incorrect.

La trame qui nous a servi à créer ce schéma est la suivante : un transfert doit s'effectuer entre deux comptes différents. Ces deux comptes peuvent appartenir à la même personne ou à deux personnes différentes et être situés dans la même banque ou des banques différentes.

Il nous faut, de plus, vérifier que la règle a bien été créée lorsqu'elle doit l'être : lorsque les comptes de départ et d'arrivée sont valides (qu'ils existent), que la période (positive) et la date (chaîne représentant bien une date) sont valides ainsi que le montant (positif). Nous devons vérifier, dans le cas contraire, qu'elle n'a pas été créée. Lorsque la règle est enregistrée, nous voulons qu'elle soit exécutée et pour cela nous avons fixé une période la plus courte possible. Nous souhaitons alors vérifier le solde des comptes pour nous assurer que les transferts s'effectuent correctement. À partir de là, nous avons créé différents groupes :

```
BOcreate_Gr1 = {AccountMan_src.BOcreate(cust_id, bn, balance)} avec :
               cust_id ∈ {1}
               bn ∈ {"bnp"}
               balance∈ {100}
BOcreate_Gr2 = {AccountMan_src.BOcreate(cust_id, bn, balance)} avec :
               cust_id ∈ {1,2}
               bn ∈ {"bnp", "ce" }
               balance∈ {100}
RegisterRuleSeuil_Gr1 = {
               Transfers_src.registerSavingRule(date, account, threshold1,
                                                saving_account, period),
               Transfers_src.registerSpendingRule(date, account, threshold2,
                                                  saving_account, period)
                         } avec :
               date ∈ {"foo"}
               account ∈ {11}
               threshold1 ∈ {-12.0f, 0.0f, 0.1f, 99.7f, 100.7f, 150.0f,
                             325.0f}
               threshold2 ∈ {-12.0f, 0.0f, 0.1f, 99.7f, 100.0f, 100.1f,
                             150.0f, 325.0f}
               saving_account ∈ {12}
               period ∈ {1}
Print1 = {Balance_src.PrettyPrintBalances(cust_id)} avec
                cust_id ∈ {1}
Print2 = {Balance_src.PrettyPrintBalances(cust_id)} avec
                cust_id ∈ {2}
```

Comme aujourd'hui nous ne possèdons pas de moyen simple, dans le langage de schéma de tests, pour récupérer les valeurs retournées par les appels de méthode, nous devons les anticiper. Ainsi, comme nous savons que le premier compte créé aura comme numéro "11" et le second comme numéro "12", nous les avons directement injecté dans les valeurs possibles des paramètres `account` et `saving_account`. Il est évident qu'il est important de fournir un mécanisme permettant de récupérer simplement des résultats de fonctions pour les réinjecter dans des appels ultérieurs [1]. Ce manque nous fait perdre en flexibilité car, si le code de l'application change, nous devons penser à changer les valeurs définies dans les groupes, et en

[1] Il est néanmoins possible de contourner ce problème en définissant, pour le test, des méthodes type "set" et "get" permettant, respectivement, de stocker le résultat d'un appel de méthode et de le récupérer.

puissance d'expression.

Les valeurs qui ont été définies pour les seuils visent à vérifier que seuls les seuils négatifs ou nuls sont invalides et n'entraînent pas l'enregistrement de la règle. Nous vérifions de plus que, lorsque le seuil le permet (0.1f et 99.7f pour threshold1 ; 100.1f, 150.0f pour threshold2) la règle est activée et se comporte correctement et que lorsque le seuil ne le permet pas (100.7f, 150.0f et 325.0f pour threshold1 ; 0.1f, 99.7f, 100.0f et 325.0f pour threshold2) la règle n'est pas activée. Les valeurs choisies sont pour une part des valeurs aux limites (0.1f, 100.0f, 100.1f) et pour une part des valeurs non limites (99.7f, 150.0f, 325.0f).

Étant dans une nouvelle campagne de tests, nous avons défini un nouveau déploiement :

```
CentralBancaire   = new AccountMan_src();
CentralTransferts = new Transfers_src(CentralBancaire);
Solde             = new Balances_src(CentralBancaire);
```

Voici maintenant le schéma permettant de tester le comportement des créations de règles :

```
TestCreationRegleSeuil = CentralBancaire.BOcreate_Gr1 ;
                         CentralBancaire.BOcreate_Gr2 ;
                         CentralTransferts.RegisterRuleSeuil_Gr1 ;
                         CentralTransferts.printGetRulesSize()[2] ;
                           Solde.Print1 ; Solde.Print2
```

Ce schéma de tests permet de générer 60 cas de tests (28 pour `SavingRule` et 32 pour `SpendingRule`). Lors de l'exécution des tests, plusieurs postconditions ont été violées. Ces erreurs de postconditions étaient dûes à des erreurs d'arrondi sur les nombres réels. Le fait d'effectuer, sur les nombres flottants, des opérations dans des ordres différents (voir section 5.4.1) entre le code et la spécification a entraîné des différences au niveau des résultats observés et attendus.

5.3.3 Fonctionnalité 3 : les règles sont activées de manière périodique

Lorsqu'une règle est enregistrée, nous devons définir sa période d'activation. Il n'est toutefois pas possible avec notre outil de vérifier que la période est bien respectée, néanmoins, nous pouvons nous assurer que les règles sont activées à plusieurs reprises.

Avant de définir les groupes, le déploiement et les schémas de tests, nous définissons quel(s) scénario(s) nous voulons tester. L'idée ici est de créer trois comptes. Pour tester l'activation répétée d'une règle d'épargne, un compte A va épargner de l'argent sur un compte B. Un compte C va transférer de l'argent sur le compte A (avec la méthode `transfer()`) à plusieurs reprises. Si la règle est périodique, à la fin de l'exécution, la somme d'argent déposée sur A devrait se retrouver sur B.

[2]Tout comme pour le schéma `TestUneCreation` nous avons défini une nouvelle fonction permettant d'afficher des informations complétementaires. Ici, il s'agit d'afficher le nombre de règles créées.

```
BOcreate_Gr1 = {AccountMan_src.BOcreate(cust_id, bn, balance)} avec :
               cust_id ∈ {1}
               bn ∈ {"bnp"}
               balance ∈ {100}
BOcreate_Gr2 = {AccountMan_src.BOcreate(cust_id, bn, balance)} avec :
               cust_id ∈ {1}
               bn ∈ {"ca"}
               balance ∈ {1000000}
RegisterSaveRule_Gr1 =
               {Transfers_src.registerSavingRule(date, account, threshold1,
                                                 saving_account, period),
               } avec :
               date ∈ {"foo"}
               account ∈ {12}
               threshold1 ∈ {101}
               saving_account ∈ {13}
               period ∈ {1}
Transfert_Gr1 = {Transfers_src.transfer(from_account, to_account, amount)}
               avec :
               from_account ∈ {11}
               to_account ∈ {12}
               amount ∈ {20}
Print1 = {Balance_src.PrettyPrintBalances(cust_id)} avec
                cust_id ∈ {1}
```

Afin d'éviter d'être parasité par des erreurs d'arrondis, nous avons ici choisi de ne faire que des transferts avec des valeurs entières. Le groupe `BOcreate_Gr2` va nous permettre de créer le compte C, le groupe `BOcreate_Gr1` va nous permettre de créer les comptes A et B. Là encore, nous avons anticipé l'ordre de création des comptes et avons choisi de créer le compte C en premier. Nous savons donc que celui-ci aura le numéro 11 et que A et B auront pour numéros 12 et 13. La règle d'épargne sera donc effectuée entre les comptes 12 et 13 avec un seuil fixé à 101 de sorte qu'elle ne s'exécutera qu'une fois le compte A approvisionné. Pour approvisionner A, nous avons créé le groupe `Transfert_Gr1`. Celui-ci permet d'effectuer un transfert du compte 11 vers le compte 12 d'un montant de 20. Le compte 11 est créé avec un solde très important, pour éviter qu'un transfert ne soit bloqué faute de fonds.

Au moment où nous définissons nos groupes, nous avons déjà une idée du schéma de tests que nous allons écrire. Pour nous permettre d'observer la périodicité, il nous faut un test qui comporte un séquence assez longue de transferts entre C et A. Nous devons donc utiliser le mécanisme de boucles disponible dans le langage. Pour éviter de faire exploser le nombre de tests, nous devons utiliser un nombre restreint de valeurs de tests. C'est pourquoi, chaque paramètre n'est instancié que par une seule valeur. Il pourrait pourtant être intéressant de tester le comportement de l'application lorsque nous faisons des transferts de différentes valeurs.

Nous allons utiliser le même déploiement que dans la campagne de tests précédente, à

savoir :

```
CentralBancaire   = new AccountMan_src();
CentralTransferts = new Transfers_src(CentralBancaire);
Solde             = new Balances_src(CentralBancaire);
```

Nous pouvons alors écrire le schéma de tests permettant de tester la périodicité des règles.

```
TestPeriodeSave = CentralBancaire.BOcreate_Gr2; CentralBancaire.BOcreate_Gr1^2..2;
                  CentralTransfert.RegisterSaveRule_Gr1;
                  (CentralTransfert.Transfert_Gr1; Solde.Print1)^500..500
```

Ce schéma utilise le mécanisme des boucles mais ne va générer qu'un seul test. Nous créons les deux comptes A et B grâce à deux appels au groupe `BOcreate_Gr1`. Celui-ci va créer deux comptes différents pour le même utilisateur, dans la même banque et avec le même solde. Nous enregistrons ensuite la règle d'épargne. Pour finir, nous produisons une suite de 500 séquences d'appels transfert/affichage du compte. Le fait d'afficher le solde du compte après chaque transfert doit nous permettre de nous faire une idée de la périodicité d'exécution de la règle d'épargne mais, nous n'avons aucune garantie sur le respect de cette période de temps.

Le test produit par ce schéma a permis de détecter une erreur dans la postcondition de la méthode `check` associée à `SavingRule`. Ces erreurs d'exécution sont dues à l'absence de section critique pour l'accès aux comptes. Ainsi, un transfert peut s'effectuer entre A et B en même temps que C approvisionne A. Il y a donc un accès en écriture sur le compte A par deux processus, ce qui fausse son solde.

Nous avons, par ailleurs, pu observer que la règle d'épargne était activée plusieurs fois au cours du test. En revanche, nous ne pouvons pas garantir avec ces tests que la périodicité de '1s' est bien respectée.

5.4 Résultats et synthèse

5.4.1 Synthèse des problèmes trouvés

Au total nous avons écrit 17 schémas de tests pour 7 fonctionnalités. Nous avons généré en tout 1241 cas de tests soit 40 000 lignes de code Java pour JUnit. Il nous a fallu 6 personnes-jour pour analyser la spécification, produire les schémas de tests, exécuter les tests et analyser les traces. Nous avons découvert 16 erreurs ou situations suspectes. Les deux équipes réunies, nous avons trouvé 18 erreurs ou situations suspectes différentes.

- Nous appelons "erreurs" les cas où une exception est levée, soit directement par le code Java, soit parce qu'une assertion JML a été violée.
- Nous appelons "situation suspecte" les cas où la spécification formelle et le code ont les mêmes comportements mais ne correspondent pas au cahier des charges ou vont à l'encontre du "bon sens" de l'ingénieur.

La table suivante liste les erreurs, leur type ainsi que la manière dont elles ont été découvertes. Les erreurs 3, 10, 13 et 14 ont été corrigées lors de la revue de code et avant la phase de test aléatoire afin de faciliter cette étape.

	equipe 1		*equipe 2*		
Err.	Revue de code	Test aléatoire	Tobias	Type d'erreur	Method de detection
1			X	limite	oracle humain
2			X	limite	oracle humain
3	X		X	flotant	rev code + JML or.
4		X	X	flotant	JML oracle
5		X	X	flotant	JML oracle
6		X	X	flotant	JML oracle
7			X	postcondition	JML oracle
8			X	postcondition	JML oracle
9		X	X	conception	JML oracle
10	X			conception	revue de code
11			X	limit	oracle humain
12			X	limit	oracle humain
13	X		X	design	rev code+ex. Java
14	X			postcondition	revue de code
15		X	X	multiple*	exception Java
16			X	contre-intuitif	oracle humain
17			X	contre-intuitif	oracle humain
18			X	flotant	oracle humain

*erreur de precondition, sous specification, ou erreur de conception

Les 18 erreurs peuvent se classer comme suit :

Erreurs d'approximations

Il y a 5 cas dûs à des erreurs d'approximation sur les nombres flottants (erreurs 3, 4, 5, 6 et 18). Les nombres flottants sont utilisés pour représenter le solde des comptes. Les erreurs ont été détectées lorsque la postcondition et le code calculaient la même valeur mais de manière différente. Par exemple, dans la classe `SavingRule_src`, la postcondition de la méthode `check()` est :

```
savingRef.balanceamount == \old(savingRef.balanceamount)+
                           \old(accountRef.balanceamount)-threshold
```

le code de la méhode, pour sa part, est :

```
savingRef.balanceamount = savingRef.balanceamount+
                          (accountRef.balanceamount-threshold)
```

Le résultat diffère car, lorsque l'on utilise des nombres flottants, les expressions $(x+y)-z$ et $x+(y-z)$ ne sont pas égales. En effet, à cause de leur précision limitée, l'opération + sur les flottants n'est pas associative.

Erreurs aux limites

Il y a 4 cas qui sont des erreurs aux limites (erreurs 1, 2, 11 et 12). Par exemple, la spécification JML autorise l'enregistrement de transferts avec une période de 0 alors que le cahier des charges l'interdit.

Il est dit dans la spécification informelle qu'il n'y a pas de limite de crédit. Nous avons donc fait le test de créditer un compte avec la constante Java POSITIVE_INFINITY. Le fait que l'application et la spécification autorisent un tel crédit nous semble malgré tout suspect.

Postconditions erronées

Trois erreurs sont en fait des postconditions erronées dues à des omissions de l'état "`\old(...)`" (erreurs 7, 8 et 14). Par exemple, l'erreur 14 est due à une assertion où la nouvelle valeur d'un attribut était égale à elle-même (`a==a`). L'expression correcte est `a==\old(a)`, disant que la valeur de l'attribut n'a pas changé [3]. Ce genre d'erreur de spécification est un exemple type d'erreur que nous ne pouvons pas détecter avec notre approche boîte noire, puisque cette assertion est toujours vraie.

Erreur de conception

Les erreurs 9, 10 et 13 ont été classifiées comme des erreurs de conception. Un attribut est public au lieu d'être privé (erreur 10). Il est possible de donner le même identifiant à deux comptes différents si deux gestionnaires de compte sont créés (erreur 9). L'application bancaire utilise des processus, mais aucune section critique n'est définie lorsqu'ils doivent accéder à un compte (erreur 13).

Comportement contre-intuitif

Les erreurs 16 et 17 sont dues à des comportements contre-intuitifs. Il est possible d'effacer un compte alors qu'une règle d'épargne ou de dépense porte sur ce compte. Ceci n'a été spécifié ni informellement ni formellement. Il n'est donc pas possible de conclure quant à la conformité de l'application sur ce point. Nous pouvons facilement imaginer que la suppression d'un compte sensé recevoir de l'argent grâce à une règle d'épargne peut poser des conflits si la règle n'a pas d'abord été désactivée. Par exemple, le transfert peut s'effectuer sur un compte qui n'est plus sensé exister mais dont l'instance Java reste active en mémoire, ce qui entraîne la disparition de la somme transférée.

[3] Cette propriété peut aussi s'exprimer en JML grâce au mot-clef `\not_modify`.

Erreur entrant dans plusieurs catégories

L'erreur 15 tombe dans plusieurs catégories. La méthode `inputToAmount` de la classe `Currency_ src` a un paramètre qui est une chaîne de caractères sensée représenter un nombre flottant. Certains tests ont fait appel à cette méthode avec un paramètre incorrect provoquant la levée d'une exception Java lors de l'exécution. Le fait que la chaîne de caractères doive représenter un nombre flottant n'a été spécifiée ni dans le cahier des charges ni dans la spécification JML. Cette erreur peut provenir :

- d'une précondition inadéquate (la précondition présente ne fait pas mention de la forme requise pour le paramètre),
- d'une spécification insuffisante (la spécification informelle ne fait pas mention de la forme requise pour le paramètre) ou,
- d'une erreur de conception (le paramètre aurait dû être de type flottant).

5.4.2 Couverture du code

Il est toujours difficile de savoir quand arrêter le processus de test. Bien que nous ayons trouvé plusieurs erreurs dans cette petite application, nous ne savons pas si nous les avons toutes trouvées. Les ingénieurs de Gemplus nous ont cependant confirmé qu'ils ne connaissaient pas d'autres erreurs. Afin d'avoir un retour sur la qualité des suites de tests, nous avons évalué la couverture de code de l'application.

Il est à noter que nous n'avions pas d'outil de couverture de test à notre disposition lors de la phase d'élaboration des tests. Nous avons utilisé l'outil JCoverage [JCo] plusieurs mois après l'expérience et l'avons utilisé sur les suites de tests générées avec les outils Jartège (test aléatoire) et Tobias. Voici les résultats :

Couverture des instructions		
Classes	Test suite aléatoire	Test suite combinatoire
Account	87%	87%
AccountMan_src	86%	91%
Balances_src	77%	71%
Currency_src	98%	88%
Rule	96%	96%
SavingRule	100%	100%
SpendingRule	100%	100%
Transfers_src	86%	90%

L'analyse de cette couverture nous montre que nous ne couvrons pas 100% des instructions. Nous avons clairement oublié des cas lors de la production des tests avec l'approche combinatoire. Par exemple, nous avons oublié de tester des règles de transfert avec des valeurs négatives.

Nous avons aussi pu identifier :

– du code mort : plusieurs méthodes sont définies comme étant privées (et donc n'apparaissent pas dans les interfaces publiques de leurs classes) mais ne sont jamais appelées par d'autres méthodes au sein de leur classe.
– une méthode publique de la classe `Transfers_src` qui n'est pas déclarée dans la spécification informelle (même nom de méthode qu'une des méthodes déclarées mais possédant des paramètres différents). Comme nous sommes parti de la spécification informelle pour construire les scénarios dans Tobias, nous n'avons pas prévu d'inclure des tests pour cette méthode.

Si l'on compare les deux taux de couverture, nous pouvons remarquer que le taux de couverture produit par les tests aléatoires est sensiblement supérieur à celui produit par le test combinatoire. L'oubli de certaines valeurs ou méthodes explique en grande partie cette différence. Malgré tout, nous avons pu constater que les tests produits avec l'outil de génération combinatoire ont permis de trouver plus d'erreurs. Ceci est dû au fait que nous avons choisi des valeurs de test de manière plus fine que l'outil de tests aléatoires.

Bien que le critère de couverture utilisé soit assez simple, il nous a permis de mettre le doigt sur certains oublis et certains problèmes de l'application. Nous avons bien conscience de la limite de cette approche. Si la couverture de test permet de voir certains défauts d'une suite de tests (lorsque l'application est peu couverte ou qu'il y a de gros trous dans la couverture), elle ne nous permet pas d'avoir la certitude que nous avons détecté tous les problèmes (même avec un taux de 100%, et même avec un critère plus fin que la couverture d'instructions).

5.4.3 Testabilité de l'application

La testabilité est l'évaluation de la facilité à tester une application. Ainsi, plus la testabilité est grande, plus la phase de test est facile (ou peu coûteuse). Des spécifications incomplètes ou ambiguës ou encore un code très complexe ont tendance à diminuer la testabilité de l'application. Au cours de cette expérience, nous avons identifié des problèmes de testabilité à quatre niveaux.

Spécifications informelles

Certaines propriétés de la spécification informelle sont impossibles à vérifier avec nos outils. Par exemple, les règles de transfert sont exécutées avec une périodicité définie par l'utilisateur. Il a été possible de vérifier que les règles ont été périodiquement exécutées, mais il n'a pas été possible de vérifier que les transferts se sont effectués à la période indiquée. Ce genre de propriétés temps réel ne peut pas s'exprimer en JML.

Spécifications JML incomplètes

Les spécifications JML nous ont servi d'oracle. Si les postconditions ou invariants sont sous-spécifiés, il est difficile de détecter automatiquement certaines erreurs lors du test.

Par exemple, lorsqu'une règle d'épargne est enregistrée avec des paramètres incorrects, un code erreur est retourné par la méthode. Ceux-ci sont définis dans la spécification informelle mais pas dans la spécification JML (bien qu'il soit possible de le faire). Nous n'avons donc pas pu tester si le code erreur retourné était correct. Les codes d'erreur sont spécifiés en JML dans toutes les classes sauf `Transfers_src`.

De même, la méthode `amountToDisplay` de la classe `Currency_src` ne possède pas de postcondition. Nous n'avons donc pas pu vérifier automatiquement que le montant affiché était correct.

On pourrait compléter la spécification afin, par exemple, de vérifier automatiquement que le caractère périodique des règles est correctement implanté.

Spécification JML trop proche du code

Dans un modèle de développement classique, la spécification est normalement définie avant d'écrire le code de l'application. Ici, la spécification JML a été écrite après le code, comme c'est le cas pour BZ-TT où la spécification est écrite spécifiquement pour la phase de validation. Certaines postconditions ont été fortement inspirées du code. Il est même parfois tentant de copier simplement des morceaux de code dans la spécification en remplaçant "=" par "==" et en ajoutant le mot-clé "\old" à certains endroits. Malheureusement, ces pratiques nous empêchent de détecter des erreurs dans l'application que ce soit par la preuve ou par le test. Il est donc important de faire l'effort d'exprimer les postconditions de manière différente et, si possible, plus abstraite. Ceci rend de plus la spécification plus robuste aux éventuelles évolutions du code de l'application.

Code

Certaines fonctionnalités auraient facilité le test de l'application. Par exemple, il est possible d'enregistrer une règle de transfert, mais il n'existe pas de moyen simple pour en suspendre ou en effacer. La notion de compte existe, mais il n'existe pas de méthode publique permettant de récupérer directement une instance de compte.

5.4.4 Utilisabilité du langage de schéma de tests

Nous avons pu illustrer, sur cette étude de cas, une manière d'utiliser le langage de synthèse de tests. Nous avons montré qu'il est possible d'écrire des schémas permettant de générer des tests trouvant des erreurs, tout en évitant de tomber dans le piège de l'explosion combinatoire. L'utilisation de notre approche nous a, en effet, permis de détecter 16 des 18 erreurs trouvées lors de l'évaluation.

Afin de maîtriser l'explosion combinatoire lors de l'écriture des schémas de tests, nous avons découpé les fonctionnalités à tester en sous-catégories. Puis, pour chaque sous-catégorie, nous avons cherché les méthodes qui devaient être utilisées et de quelle manière elles devaient l'être. Nous avons donc imaginé diverses séquences intéressantes sans nous attacher encore

aux valeurs. Pour chaque schéma, nous avons choisi un nombre de valeurs de tests en fonction de la taille des séquences qui nous intéressaient. En effet, plus la séquence est longue plus le nombre de tests générés va avoir tendance à exploser. Nous devons donc, lorsque nous souhaitons travailler sur de longues séquences, réduire le nombre de valeurs de tests. Ensuite, le choix des valeurs a été effectué suivant l'approche catégorie et partitions [Nta98]. Ainsi, en choisissant intelligemment les valeurs de tests, nous avons pu contrôler le nombre de tests produits. Les schémas de test ainsi écrits ont permis de capturer le savoir-faire de l'ingénieur de test dans une forme plus abstraite et plus compacte.

D'un point de vue industriel, la possibilité d'exprimer des tests très abstraits (les schémas) à partir desquels sera généré un large ensemble de tests exécutables, a été considéré comme très intéressant. Tout d'abord il est moins cher d'écrire et de maintenir un petit ensemble de tests abstraits que de nombreux tests exécutables. Ensuite, le dépliage systématique des schémas par Tobias permet de produire des séquences qui n'étaient pas imaginées par l'utilisateur.

Une équipe de Gemplus a utilisé notre outil et le processus défini dans le projet COTE pour produire des tests pour cette même application [Bon05]. Les ingénieurs sont partis des exigences fonctionnelles de leur application et ont défini une première série de 36 séquences abstraites permettant de vérifier ces exigences. À partir de ces séquences, ils ont défini une première série de 14 schémas utilisant 14 groupes permettant de les générer. Puis, ils ont cherché à minimiser le nombre de schémas en optimisant le nombre de groupes et sont arrivés à 5 schémas utilisant 6 groupes. Ce rassemblement de groupes a permis de créer des groupes plus variés et donc à générer de nouvelles séquences. Grâce à ces schémas, les ingénieurs ont pu générer 1900 séquences. Suite à cette expérience, les ingénieurs ont pu nous faire part des faiblesses de l'outil. Ainsi, il ressort que :

- Tobias ne prenait pas en compte les retours de méthode. Lors d'un test, l'ingénieur de test peut souhaiter récupérer une valeur en retour d'une méthode pour y faire appel lors de la suite du cas de test. Tobias ne propose pas de telles manipulations pour deux raisons. La première vient simplement du fait que Tobias ne gère pas les retours de méthode, on ne spécifie que des appels de méthodes dans les schémas de test. La deuxième raison vient du fait que Tobias ne stocke pas les retours d'un appel de méthode dans une variable pour l'utiliser plus tard dans le test.
- Tobias ne permet pas de définir un groupe à partir d'autres groupes. Cette approche, qui consiste à encapsuler des ensembles de groupe dans des groupes, permet de définir des schémas de test avec différents niveaux de granularité. Imaginons par exemple que l'on veuille créer un groupe avec deux fonctions `add` et `sub`, dont la spécification est d'additionner et de soustraire des entiers. Si on considère que ces méthodes appartiennent déjà aux groupes `g_add` et `g_sub`, l'utilisateur est obligé de dupliquer la saisie des valeurs des paramètres, qui a déjà été faite dans ces deux groupes, pour créer un troisième groupe. La composition des groupes permettrait de mieux structurer les méthodes et de réduire le temps de saisie des paramètres.

Les notions de composition de groupes et de gestion de retours de méthode n'ont pas encore

été intégrées à l'outil Tobias. En effet, la mise en place de ces deux notions nécessiterait de profonds changements dans le langage de synthèse de tests et ainsi que dans l'outil Tobias qui le met en œuvre.

À côté de ces faiblesses, cette expérience a mis en évidence deux points forts de l'outil :

1. sa productivité : à partir de 5 schémas, on génère plus de 50 fois le nombre de tests prévus dans le plan de test.
2. sa capacité de structuration de ce grand nombre de tests.

Pour finir, la seule expertise de l'ingénieur peut s'avérer insuffisante à maîtriser le problème de l'explosion combinatoire. Aussi, nous proposons dans la suite de ce travail des techniques fournissant d'autres armes à l'ingénieur pour contrôler ce facteur.

Troisième partie

Maîtrise de l'explosion combinatoire

Chapitre 6

Maîtrise de l'explosion combinatoire à la génération

Le problème de l'explosion combinatoire exposé chapitre 4 section 4.4, pose une réelle difficulté quant à l'utilisation du langage de synthèse de tests. Plusieurs approches sont envisageables pour maîtriser cette explosion lors de la génération des tests.

Dans un premier temps, nous présentons comment il est possible de définir des contraintes au niveau des schémas de tests. Ces contraintes sont des formules logiques qui sont vérifiées pour chaque séquence générée à partir du schéma. Si, pour une séquence de test donnée, la contrainte est satisfaite alors la séquence est conservée, sinon, elle est supprimée de la suite de tests. Nous montrons que l'utilisation des contraintes permet de mieux maîtriser l'ensemble des tests générés et comment elles sont un moyen d'exprimer formellement des hypothèses de test.

Ensuite, nous étudions la possibilité d'étendre le langage de schémas de tests afin d'offrir à l'utilisateur des mécanimes permettant de ne générer qu'un ensemble réduit de tests. Le cas précédent se fait en deux étapes : tout d'abord les séquences de test sont générées puis elles sont filtrées. Il s'agit ici d'exercer plus de contrôle directement au moment de la génération.

Enfin, lorsqu'une spécification formelle est disponible, nous l'utilisons pour filtrer les tests et ne sélectionner que ceux qui vérifient la spécification. Pour cela, nous utilisons la programmation logique avec contraintes en complément du langage de schéma de tests. La programmation logique avec contraintes nous permet de filtrer les tests à partir d'une transformation de la spécification en un système de contraintes. Nous montrons, de plus, comment il est possible de combiner les deux approches pour filtrer les tests tout en déchargeant l'utilisateur de la sélection des valeurs de paramètres.

6.1 Motivations

Nous avons montré que l'un des problèmes de Tobias réside dans la trop grande quantité de tests qui peuvent être générés. Une manière de pallier ce problème est de filtrer les tests lors de la génération. Parmi tous les tests générés par Tobias, certains conduisent à

un verdict inconclusif car ils violent la précondition d'une ou plusieurs opérations, d'autres peuvent être considérés comme redondants.

Filtrer les tests à la génération revient, d'une certaine manière, à exprimer des hypothèses de test (voir chapitre 2, section 2.2.3). En effet, en éliminant certains tests, l'utilisateur fait le pari que ceux-ci ne permettront pas de détecter des erreurs supplémentaires. Considérons l'exemple du gestionnaire de processus et les séquences suivantes :

- `seq1 = MTC!Sch1.new("p1")`
- `seq2 = MTC!Sch1.new("p2")`
- `seq3 = MTC!Sch1.rdy("p2")`

La question que nous devons nous poser est : quels sont les tests que nous pouvons éliminer ? Il existe, selon nous, deux catégories de tests qui peuvent être éliminés dans le cadre du test de conformité, à savoir : les tests redondants et les tests violant la spécification. Nous explicitons quels sont les tests concernés par chacune de ces catégories et exposons la manière de les éliminer.

6.1.1 Tests redondants

Dans la majorité des cas, un test exhaustif n'est pas possible. Il faut donc trouver des moyens pour réduire l'ensemble des tests à un ensemble fini, tout en espérant que celui-ci sera suffisant pour trouver des erreurs. Les tests qui ne sont pas pris en compte doivent donc être considérés comme redondants avec les tests sélectionnés. En ce sens, l'ensemble sélectionné doit avoir le même potentiel de détection d'erreurs que l'ensemble de départ. L'expression d'hypothèses de test [Ber91, BGM91, Gau95, Pha94] permet de ne considérer qu'un ensemble fini de tests. Les hypothèses de test définissent donc des critères de redondance.

L'expression formelle de ces hypothèses, par exemple, sous la forme de contraintes, permettrait donc de réduire le nombre de tests a priori, c'est-à-dire, avant de les exécuter sur l'application sous test. La technique que nous avons mise en œuvre et que nous présentons dans ce chapitre rejoint ce point. En effet, nous offrons à l'utilisateur la possibilité d'exprimer des contraintes sur les schémas de tests. Lorsque nous définissons une nouvelle contrainte, nous devons penser aux tests qui vont la vérifier. Ce faisant, la contrainte exprime de manière plus ou ou moins directe une propriété que l'on estime suffisante pour le test. Autrement dit, cette contrainte est une hypothèse de test.

Nous pouvons de plus considérer certains tests comme équivalents. Ayant accès au code de l'application, nous pouvons voir que les séquences de test `seq1` et `seq2` sont équivalentes. En effet, l'implantation ne fait pas de traitement particulier en fonction de la valeur du paramètre de l'opération `new` et donc le fait de considérer `"p2"` à la place de `"p1"` ne change rien. De même si nous cherchons à tester une séquence de deux `new`, tester `MTC!Sch1.new("p1"), MTC!Sch1.new("p2")` ou `MTC!Sch1.new("p2"), MTC!Sch1.new("p1")` revient au même. Néanmoins, pour résoudre ce problème dans le cas général, il faut pouvoir résoudre le problème de l'équivalence de programmes. Or l'équivalence de programmes est un problème

indécidable. Donc, identifier des tests équivalents du point de vue de l'implantation sous test n'est pas possible. En revanche, rien n'interdit de faire certaines hypothèses de test afin d'éliminer certains cas.

Par exemple, nous faisons l'hypothèse que le séquencement des appels de `new` est commutatif. Pour tester une suite d'appel à `new`, il n'est pas nécessaire de tester toutes leurs combinaisons. Cette hypothèse se définit par la contrainte suivante : lorsque nous faisons trois fois appel à l'opération `new`, les trois appels sont différents et, étant donné l'ordre lexicographique comme relation d'ordre sur les processus, les appels sont dans l'ordre croissant. Avec une telle contrainte, des séquences comme `MTC!Sch1.new("p1") ; MTC!Sch1.new("p2") ; MTC!Sch1.new("p3")` sont autorisées alors que des séquences comme `MTC!Sch1.new("p1") ; MTC!Sch1.new("p3") ; MTC!Sch1 .new("p2")` ne le sont pas. Nous éliminons donc des tests en exprimant, par une contrainte, une hypothèse de test.

6.1.2 Tests violant la spécification

Lorsque nous générons des tests à partir d'un schéma, nous ne prenons pas la spécification formelle en considération. Il arrive donc que certains tests générés violent les préconditions de certaines opérations, comme par exemple la troisième séquence de test (`seq3`). Ce type de test peut ne pas être désiré par l'utilisateur.

Ici aussi l'utilisation de contraintes peut s'avérer utile. Prenons par exemple la contrainte suivante : pour chaque appel d'opération `rdy` il existe un appel à l'opération `new` avec la même valeur de paramètre. Cette contrainte va permettre d'éliminer les cas comme `seq3`.

Une autre façon de procéder consiste à utiliser la spécification pour éliminer les tests qui vont la violer. Les techniques basées sur la programmation logique avec contraintes apportent une solution. Nous pouvons ainsi procéder de la même manière que les outils Casting [VBL97, Van98] ou BZ-TT [Peu02, LPU02].

D'autres techniques sont envisageables comme l'utilisation d'outils permettant d'animer la spécification, ou encore la preuve. Ces techniques permettent d'évaluer au niveau de la spécification si les tests ont un sens. L'utilisation de tels outils dans le cadre de nos travaux n'a pas été étudiée en détail. Il faudrait évaluer dans quelle mesure ces approches sont utilisables sur de grandes suites de tests et notamment voir les problèmes qu'elles peuvent poser en termes de : complexité de mise en œuvre, temps d'exécution et interaction avec l'utilisateur.

6.2 Définition de contraintes

Reprenons le cas du gestionnaire de processus. Supposons que nous ayons créé un schéma générant, entre autres, les trois séquences données ci-dessous.

```
MTC!Sch1.rdy("p1")
```

```
MTC!Sch1.new("p1"); MTC!Sch1.new("p2"); MTC!Sch1.rdy("p1")
MTC!Sch1.new("p1"); MTC!Sch1.new("p1"); MTC!Sch1.rdy("p1")
```

Si l'ingénieur ne veut pas tester la robustesse de ses opérations en les exécutant en dehors de leur précondition, les séquences comme la première et la troisième présentées ici ne lui sont pas utiles. Dans ce cas, les seuls tests intéressants sont ceux où nous appelons l'opération `rdy` sur un processus existant et où nous ne faisons appel à l'opération `new` qu'une fois pour chaque nouveau processus. Il est possible d'exercer un tel contrôle en définissant, par exemple, de nouvelles valeurs ou de nouveaux schémas plus précis (et donc plus longs). Par exemple, soit $RdyGr = \{rdy(x)\}$ avec $x = \{"p1", "p2", "p3"\}$, nous pouvons écrire le schéma suivant :

```
S1 = MTC!Sch1.new("p1"); MTC!Sch1.new("p2");
     MTC!Sch1.new("p3"); MTC!Sch1.RdyGr
```

Ce schéma permet de générer des séquences de tests telles que l'on ne fait appel à `rdy` que sur des processus existants. Néanmoins, cette solution n'est pas complètement satisfaisante car elle entraîne la génération de tests déjà existants ou va nécessiter la production d'un grand nombre de schémas, limitant l'intérêt de l'approche. Il nous semble plus intéressant de fournir à l'utilisateur un mécanisme de sélection à base de contraintes.

Considérons les définitions suivantes :

- une séquence Seq d'opérations est définie de la façon suivante : $Seq = [a_1; a_2; ...; a_n]$ avec $a_i = op_i(v_{i1}, ..., v_{ik})$ où op_i est le nom de l'opération et v_{ij} est la valeur de son $j^{ème}$ paramètre,
- la signature de l'opération op_i est définie de la façon suivante : $op_i(p_{i1}, ..., p_{ik})$ où p_{ij} est son $j^{ème}$ paramètre,
- nous appelons op_{p_i} le $i^{ème}$ paramètre de l'opération op et $T_{op_{p_i}}$ son type.

Les contraintes sont définies au niveau des schémas de test.

Chaque schéma de tests Sc caractérise un ensemble Σ de séquences d'opérations.

Une contrainte C prend une séquence de test en entrée et renvoie un booléen. L'objectif de la contrainte est de pouvoir exprimer des propriétés sur les valeurs des paramètres des différents appels d'opération de Seq. De plus, elle doit pouvoir renvoyer un résultat pour toute séquence $Seq \in \Sigma$. Dans une séquence de test, une même opération peut être appelée plusieurs fois tout comme il est possible qu'une opération soit appelée dans certaines séquences et pas dans d'autres. La contrainte doit donc pouvoir tenir compte de ces deux cas de figure pour renvoyer un résultat quelle que soit la séquence passée en paramètre.

Étant donnée une contrainte C définie sur un schéma de tests Sc, l'ensemble Σ_C des tests effectivement générés est : $\Sigma_C = \{Seq \in \Sigma | C(Seq)\}$.

6.2.1 Utilisation de séquences

Nous avons dit que les contraintes étaient définies au niveau des schémas de test. Un schéma de tests est composé de groupes d'opérations. L'utilisateur doit identifier les paramètres des opérations qui vont entrer en jeu dans la contrainte. Pour chaque paramètre op_{p_i} de l'opération op, il crée une variable $V_{op_{p_i}}$ de type `seq of` $T_{op_{p_i}}$. Par exemple, un paramètre de type `PID` est lié à une variable de type séquence de `PID`.

Le fait de définir la contrainte sur des séquences de valeurs permet de l'évaluer quelle que soit la séquence d'opérations passée en entrée. Ainsi, si nous reprenons l'exemple donné plus haut, avec V_{new} la variable de type `seq of PID` rattachée au paramètre de l'opération `new` et V_{rdy} de type `seq of PID` rattachée au paramètre de l'opération `rdy`, nous obtenons :

- pour la première séquence V_{new} =`[]` et V_{rdy} =`["p1"]`. Nous pouvons ici constater que lorsque l'opération `new` n'est pas présente dans la séquence, la variable, qui lui est associée et qui sert dans la contrainte, a quand même une valeur : la séquence vide,
- pour la seconde séquence V_{new} =`["p1" ;"p2"]` et V_{rdy} =`["p1"]`
- pour la troisième séquence V_{new} =`["p1" ;"p1"]` et V_{rdy} =`["p1"]`. Le fait que l'on appelle l'opération `new` deux fois avec la même valeur se retrouve dans la séquence de valeurs générée. Il nous suffit, par exemple, de compter le nombre d'éléments différents pour nous rendre compte que ce cas ne nous intéresse pas.

6.2.2 Utilisation de fonctions

Le principe ici est d'identifier la place de chaque appel d'opération. En utilisant une fonction, nous pouvons associer à chaque valeur de paramètre sa place dans la séquence d'appels d'opérations. Ceci va nous permettre d'exprimer des contraintes plus fines en prenant en compte la position des appels d'opérations.

Soit l'opération op, pour chaque paramètre op_{p_i} concerné par la contrainte, nous définissons une fonction : $V_{op_{p_i}} : nat \rightarrow T_{op_{p_i}}$. Chaque élément du domaine caractérise une position dans la séquence et chaque élément du codomaine une valeur de paramètre. Le nombre d'appels est donc caractérisé par la cardinalité du domaine de la fonction et l'ensemble des valeurs différentes par son codomaine. S'il n'y a pas d'appel d'opération alors la fonction a un domaine vide.

Ainsi, si nous reprenons l'exemple donné plus haut, avec V_{new} la variable de type $nat \rightarrow$ `PID` rattachée au paramètre de l'opération `new` et V_{rdy} de type $nat \rightarrow$ `PID` rattachée au paramètre de l'opération `rdy`, nous obtenons :

- pour la première séquence V_{new} ={} et V_{rdy} ={`1|->"p1"`}. Nous pouvons ici constater que lorsque l'opération `new` n'est pas présente dans la séquence, la variable, qui lui est associée et qui sert dans la contrainte, a pour valeur la fonction vide,
- pour la seconde séquence V_{new} ={`1|->"p1", 2|->"p2"`} et V_{rdy} ={`3|->"p1"`}

– pour la troisième séquence V_{new} ={`1|->"p1", 2|->"p1"`} et V_{rdy} ={`3|->"p1"`}. Le fait que l'on appelle l'opération `new` deux fois avec la même valeur se retrouve dans le fait que le codomaine de V_{new} contient un élément alors que son domaine en contient deux. Il nous suffit, par exemple, de compter la différence entre la cardinalité du domaine et du codomaine pour nous rendre compte que ce cas ne nous intéresse pas.

6.2.3 Implantation avec VDM

Nous aurions pu élaborer un langage complet d'expression de contraintes dont des bases ont été posées dans [Beg02]. Néanmoins nous préférons, pour des raisons de rapidité d'implantation, utiliser le langage VDM comme support d'expérimentations. En effet, il est relativement simple de définir en VDM des fonctions booléennes sur des séquences de valeurs. Cet avantage constitue une base particulièrement riche pour exprimer des contraintes. L'idée générale est donc d'exploiter la puissance du langage ainsi que les outils associés pour filtrer les séquences de test générées à partir des schémas de tests.

Comme montré figure 6.1, pour chaque séquence de test, Tobias construit l'appel de la fonction booléenne VDM rattachée au schéma correspondant et l'exécute de manière transparente pour l'utilisateur dans l'atelier VDMTools. En fonction de la réponse de celui-ci, Tobias garde ou supprime la séquence.

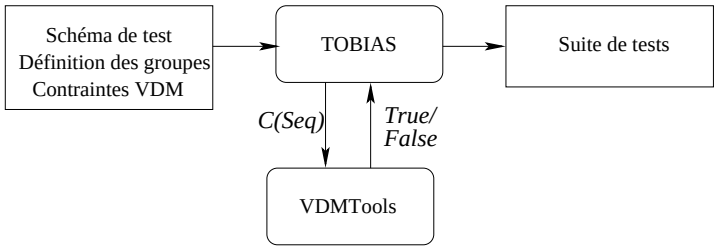


FIG. 6.1 – Filtrage des tests à la génération.

6.2.4 Expérimentations

L'exemple que nous présentons ci-dessous porte sur le gestionnaire de processus. Il a pour but d'illustrer les propos mis en avant dans la section précédente, mais ne prétend pas apporter d'information sur la méthodologie d'utilisation du langage de schéma de tests. Notre objectif est de montrer comment, de manière simple, il est possible de sélectionner un ensemble réduit de tests à partir d'un ensemble plus important et ce, à l'aide de contraintes. Nous illustrons aussi cette approche chapitre 8, section 8.5, sur une étude de cas plus im-

portante.

Soit le groupe $newrdyP12 = \{new(x), rdy(x)\}$ avec $x = \{"p1","p2"\}$. Prenons par exemple le schéma suivant :

```
NewRdySchema = MTC!Sch1.newrdyP12^1..4
```

Ce schéma se décompose en 340 séquences de test faisant appel aux opérations `new(x)` et `rdy(y)`. Parmi toutes ces séquences, certaines sont invalides du point de vue de la spécification, d'autres peuvent, sous certaines hypothèses, être considérées comme redondantes, par exemple :

- la séquence `MTC!Sch1.new("p1") ; MTC!Sch1.new("p1")` est invalide et,
- si nous faisons l'hypothèse que l'ordre dans lequel les `new` sont effectués n'a pas d'importance, des séquences comme :
  `MTC!Sch1.new("p1") ; MTC!Sch1.new("p2")`
  `MTC!Sch1.new("p2") ; MTC!Sch1.new("p1")`
  sont redondantes et peuvent donc être filtrées.

Évaluation de l'utilisation de séquences

Soit `xx` la variable rattachée au paramètre de `new` et `yy` la variable rattachée au paramètre `rdy`. Considérons les séquences suivantes, générées par le schéma `NewRdySchema` :

```
seq_a = MTC!Sch1.new("p1"); MTC!Sch1.new("p1"); MTC!Sch1.rdy("p1");
        MTC!Sch1.rdy("p2")
seq_b = MTC!Sch1.new("p2"); MTC!Sch1.new("p1"); MTC!Sch1.rdy("p1");
        MTC!Sch1.rdy("p2")
seq_c = MTC!Sch1.new("p1"); MTC!Sch1.rdy("p1"); MTC!Sch1.new("p2")
```

Pour ces cas de test, nous construisons respectivement :
- pour `seq_a` : `xx = ["p1","p1"]` et `yy = ["p1","p2"]`.
- pour `seq_b` : `xx = ["p2","p1"]` et `yy = ["p1","p2"]`.
- pour `seq_c` : `xx = ["p1","p2"]` et `yy = ["p1"]`.

Soit la contrainte suivante :

C0 : `card(elems(xx))=len(xx)`

Cette contrainte vérifie que le nombre d'éléments différents de `xx` est égal à la longueur de la séquence contenue dans `xx`. De cette façon, on s'assure qu'on ne crée pas deux fois le même processus. La séquence `MTC!Sch1.new("p1") ; MTC!Sch1.new("p1") ; MTC!Sch1.rdy("p1") ; MTC!Sch1.rdy("p2")` tombe dans ce cas de figure et est rejetée par la contrainte. Ainsi, avec cette seule contrainte, le nombre de séquences passe de 340 à 190.

Trois autres contraintes peuvent être exprimées sur ce schéma :

C1 : `card(elems(yy))=len(yy)`.
Cette contrainte, comme la précédente, sert à éliminer les cas où l'on essaie de mettre deux fois le même processus dans l'état `ready`. Dans le cas présent, comme nous n'effectuons pas d'opération `swap` dans ces séquences, la précondition de `rdy` sera violée entraînant un verdict *inconclusif*. Par ailleurs, cette contrainte réduit le nombre de séquences de 190 à 64.

C2 : `elems(yy) subset elems(xx)`.
Cette contrainte élimine les cas où l'on appelle `rdy` sur un processus qui n'a pas été ou ne sera pas créé dans la séquence. Par exemple :
`MTC!Sch1.new("p2") ; MTC!Sch1.rdy("p1")`).
Là encore cette contrainte permet d'éliminer des séquences de test violant la précondition de `rdy`. Le nombre de séquences passe maintenant de 64 à 44.

C3 : `forall i,j in set inds xx & i<j => string_lt`[1]`(xx[i],xx[j])`.
Cette contrainte compare deux à deux les valeurs stockées dans la séquence `xx`. Pour tout couple d'indices différents dans la séquence `xx`, si la première est positionnée avant la seconde dans la séquence alors elle doit être plus petite, au sens lexicographique du terme, que la seconde. Elle permet donc d'éliminer les permutations d'appels de l'opération `new` comme la séquence :
`MTC!Sch1.new("p2") ; MTC!Sch1.new("p1")`
ou encore
`MTC!Sch1.new("p2") ; MTC!Sch1.rdy("p2") ; MTC!Sch1.new("p1") ;`
`MTC!Sch1.rdy("p1")`.
Cette contrainte exprime l'hypothèse de test suivante : l'ordre dans lequel les `new` sont effectués n'a pas d'importance. Le nombre de tests passe maintenant de 44 à 25. À noter que cette dernière contrainte inclut la contrainte *C0*.

Sur les 25 séquences restantes, 11 conduisent à un verdict *pass* et 14 à un verdict *inconclusif*.

Comme les contraintes portent sur des séquences n'enregistrant que les valeurs des paramètres, l'ordre des appels d'opération n'est pas pris en compte. C'est pourquoi il reste autant de tests conduisant à un verdict *inconclusif*. Comme par exemple :
`MTC!Sch1.new("p1") ; MTC!Sch1.rdy("p1") ; MTC!Sch1.rdy("p2") ;`
`MTC!Sch1.new("p2")`.
Cette séquence respecte les quatre contraintes définies ci-dessus : `xx=["p1", "p2"]` et `yy=["p1", "p2"]`. Nous pouvons facilement vérifier *C0* et *C1*. De même, `yy` est bien un sous ensemble de `xx`, donc *C2* est vraie. Pour finir, l'ordre lexicographique est respecté, donc la contrainte *C3* est aussi vraie.

[1] `string_lt` est une fonction qui prend en entrée deux séquences de caractères et qui renvoie vrai si la première est plus petite que la seconde dans l'ordre lexicographique.

Évaluation de l'utilisation de fonctions

Reprenons l'exemple exposé dans la section précédente où maintenant `xx` et `yy` ne sont plus de type `seq of PID` mais de type `map nat to PID`. Les contraintes précédentes s'expriment maintenant de la façon suivante :

- la contrainte *C0*, qui exprime que toutes les valeurs des appels de l'opération `new` sont deux à deux différentes, s'écrit maintenant `card(dom xx)=card(rng xx)`,
- la contrainte *C1*, qui exprime que toutes les valeurs des appels de l'opération `rdy` sont deux à deux différentes, s'écrit maintenant `card(dom yy)=card(rng yy)`,
- la contrainte *C2*, qui exprime que `rdy` ne porte que sur des processus créés à partir du schéma, peut maintenant s'écrire en tenant compte de l'ordre des appels d'opération :
  ```
  forall oprdy in set dom yy & exists opnew in set dom xx &
  yy(oprdy) = xx(opnew) and oprdy > opnew
  ```
  Pour chaque élément du domaine de `yy`, c'est-à-dire pour chaque appel de l'opération `rdy`, il doit exister un appel à une opération `new`, antérieure à l'opération `rdy` considérée et avec un paramètre identique à celui de l'opération `rdy`.
- la contrainte *C3*, qui exprime que l'ordre des appels de l'opération `new` n'a pas d'importance, s'écrit maintenant :
  ```
  forall i,j in set dom xx & i<j => string_lt(xx(i),xx(j)).
  ```

Au final, le nombre de tests filtrés à l'aide de ces quatre contraintes nous permet de supprimer les 14 séquences conduisant à un verdict *inconclusif*. Il ne reste que 11 séquences.

L'utilisation de fonctions nous permet donc de prendre en compte cette dimension supplémentaire qu'est l'ordre d'appel des opérations. Néanmoins, l'expression de contraintes s'en retrouve complexifiée.

Bilan

Nous avons exposé un certain nombre de pistes pouvant permettre de résoudre les problèmes exposés en début de chapitre. L'expression de contraintes permet d'exercer un contrôle supplémentaire sur les tests qui sont générés. Il est ainsi possible d'éviter la génération de tests violant la spécification. C'est aussi un moyen de sélectionner des tests qui peuvent sembler plus intéressants que d'autres à un moment donné du processus de test. Une contrainte caractérise alors une ou plusieurs hypothèses de test faites par l'ingénieur.

Le fait de conserver la position des appels d'opérations comme information supplémentaire complique rapidement les contraintes écrites par l'utilisateur. Cela permet néanmoins d'exercer un meilleur contrôle sur les tests générés et permet donc une sélection plus fine. Il s'agit alors de trouver le bon compromis entre la complexité de la contrainte et la finesse de la sélection. Plus les contraintes sont complexes plus elles sont difficiles à écrire, et donc plus elles peuvent se trouver erronées, et plus leur exécution prend du temps.

Par ailleurs, les expériences que nous avons menées nous amènent aussi à penser que les contraintes les plus usuelles sont très simples à écrire. Elles n'utilisent seulement que quelques éléments du langage et tiennent en quelques lignes. Leur écriture ne nécessite pas une grande connaissance de VDM. En effet, comme les contraintes portent sur des séquences d'éléments, les plus courantes sont de s'assurer qu'une séquence est bien sous-séquence d'une autre, qu'elles ont le même nombre d'éléments, ou encore que le nombre d'éléments différents d'une séquence est égal à la taille de la séquence. Il nous est donc possible de créer une bibliothèque des contraintes les plus usuelles afin d'assister l'ingénieur dans sa démarche. Ainsi, l'ingénieur, même non expert en VDM, peut, à partir d'une connaissance partielle du langage, écrire des contraintes particulièrement utiles.
De plus, celui-ci garde à sa disposition toute la puissance de VDM pour définir des contraintes plus complexes portant, par exemple, sur des types de données structurés. L'utilisateur reste donc maître de la complexité d'utilisation de cette fonctionnalité.

6.3 Évolutions du langage de schéma de tests

Pour éviter de générer des tests qui sont ensuite éliminés par filtrage, nous proposons d'intégrer certaines contraintes au langage de schémas de test. Comme nous avons pu le constater précédemment, certaines des contraintes que nous écrivons sont simples. De telles contraintes peuvent facilement s'intégrer au langage de schéma de tests. Le filtrage à l'aide de contraintes se fait en deux étapes. La première consiste à générer l'ensemble des tests correspondant au schéma. Puis, pour chaque séquence, nous vérifions si elle satisfait la contrainte associée au schéma. L'idée ici est de réduire dès la première étape la taille de la suite de tests. Le fait d'avoir un ensemble de test plus réduit, dès la première étape, va nous permettre d'utiliser un plus grand nombre de valeurs de tests en diminuant le risque d'explosion combinatoire. Néanmoins, quelles que soient les améliorations apportées au langage de schéma de tests, nous pourrons toujours trouver des contraintes utiles qui ne s'exprimeront pas dans le langage. Il ne s'agit donc pas de trouver un paliatif à l'utilisation de contraintes mais bien de rendre les deux approches complémentaires.

Nous proposons ainsi trois nouveaux opérateurs d'itération :

1. $<> l..m$ qui impose que tous les appels d'une même opération, qui sont engendrés par l'itération, ont tous des valeurs de paramètres différentes.
2. $<_R l..m$ qui permet, pour chaque appel d'une même opération, de prendre une nouvelle valeur dans l'ordre croissant défini par la relation d'ordre R,
3. $>_R l..m$ qui permet, pour chaque appel d'une même opération, de prendre une nouvelle valeur dans l'ordre décroissant défini par la relation d'ordre R,

Le premier cas. Une des contraintes que nous avons exprimée en section 6.2.4 consiste à éliminer tous les tests où nous appelons plusieurs fois les mêmes opérations avec des valeurs de

paramètre identiques comme `new("p1") ; new("p1")`. Ces contraintes (*C0* et *C1*) peuvent s'exprimer directement dans le schéma de tests, de la manière suivante :

```
NewRdySchema2 = exMTC!Sch1.newrdyP12<>1..4
```

Dans les deux derniers cas, nous avons besoin d'une relation d'ordre. Une manière flexible de définir la relation d'ordre est de la faire porter sur une représentation arborescente de chaque appel. La relation d'ordre se définit alors comme une relation entre deux arbres n-aires.

Chaque appel peut se définir comme un arbre n-aire, de profondeur 1, dont la racine est le nom de la méthode et les feuilles sont les valeurs des paramètres de la dite méthode. A partir d'un tel arbre, l'utilisateur peut définir toutes sortes de relations. Par exemple, il peut choisir de définir une relation d'ordre, basée sur l'ordre lexicographique, en concaténant la racine de chaque arbre avec ses paramètres et en comparant les deux chaînes (voir l'exemple ci-dessous). Il peut aussi se contenter de prendre l'ordre lexicographique des paramètres ou plus simplement d'ordonner les appels en fonction du nombre de paramètres. A l'inverse, rien ne lui interdit de définir une relation d'ordre plus complexe prenant en considération le type des paramètres. Comme il s'agit ici de filtrage à la génération, il sera difficile de définir une relation d'ordre sur des types structurés dont on ne connait pas aujourd'hui l'état au moment de la génération du test. Nous voyons donc que cette représentation laisse une certaine latitude à l'utilisateur dans la définition de la relation d'ordre.

L'arbre t se définit de la manière suivante à partir d'un appel de méthode :
$t = (nomMethode, args)$
$args = (v1, ..., vn)$
Si la méthode n'a pas de paramètres, $args = ()$
Soit les groupes $newP13 = \{new(x)\}$ et $rdySwapP13 = \{rdy(x), swap()\}$
avec $x = \{"p1", "p2", "p3"\}$. Soit le schéma :

```
NewRdySwapSchema = MTC!Sch1.newP13^3..3; MTC!Sch1.rdySwapP13^1..4
```

L'objectif de ce schéma est de charger un état avec trois processus, puis de tester différentes combinaisons d'activation des processus. Il va permettre de générer 9180 séquences de tests ($3^3 * (4 + 4^2 + 4^3 + 4^4)$). Nous avons néanmoins fait comme hypothèse que l'ordre, dans lequel nous effectuons la création des processus, n'avait pas d'importance. Le fait de générer toutes les permutations d'appels de l'opération `new` ne nous intéresse donc pas. Nous pouvons donc écrire le schéma suivant :

```
NewRdySwapSchemaOrd = newP13<R3..3; rdySwapP13^1..4
```

avec :
$R : tree * tree \rightarrow bool$

$R(t1, t2) == return(string_lt(aplat(t1), aplat(t2)))$ où $string_lt$ est une relation d'ordre sur les chaînes de caractères.

$aplat : tree \rightarrow string$

$aplat((m, (v1, ..., vn))) == return(concatene(m, concatene(v1, concatene(...vn)...)))$

Dès lors, le nombre de tests générés, en une passe, n'est plus que de 340 séquences. En effet, le sous schéma $\texttt{newP13}^{<_R 3..3}$ ne génère plus qu'une seule séquence.

6.4 Programmation logique avec contraintes

Nous avons montré comment l'utilisation de contraintes et l'amélioration du langage de schéma de tests, permettaient, de manière complémentaire, de réduire le nombre de tests générés. La spécification peut, elle aussi, contribuer à limiter le nombre de tests générés. Une des techniques permettant d'utiliser la spécification dans le processus de génération de tests est la programmation logique avec contraintes. Nous étudions de manière théorique, dans cette section, comment nous pouvons l'utiliser en complément du langage de schéma de tests. Pour cela, nous continuons de nous appuyer sur l'exemple du gestionnaire de processus. L'article [MLdB03] présente les expérimentations que nous avons menées avec l'outil UCASTING [VJ03] sur un autre exemple.

L'une des causes principales de l'explosion combinatoire vient du fait que nous générons, pour chaque opération, toutes les combinaisons de paramètres et toutes les combinaisons d'appels d'opérations. Parmi toutes ces séquences, certaines peuvent violer la spécification.

Par ailleurs, un autre intérêt de la programmation logique avec contraintes est qu'il est possible de l'utiliser pour sélectionner automatiquement les valeurs des paramètres des opérations. Nous pouvons donc spécifier des schémas de tests sans avoir besoin de donner toutes les valeurs des paramètres. Deux approches sont alors envisageables : soit l'utilisateur ne spécifie aucune valeur de paramètre, soit il en spécifie certaines qui lui semblent importantes et laisse le soin au solveur d'instancier les paramètres qui lui semblent moins critiques. Dans les deux cas, les tests générés sont conformes à la spécification.

6.4.1 Évaluation de séquences complètement instanciées

Cette première approche rejoint ce que nous faisons déjà à l'aide des contraintes. Il s'agit de générer toutes les séquences de tests puis de vérifier leur validité du point de vue de la spécification. Cette technique va permettre d'éliminer les tests conduisant obligatoirement à un verdict inconclusif. Dans l'exemple du gestionnaire de processus, la précondition de l'opération `rdy` impose que le processus passé en paramètre de l'opération soit dans l'état `waiting` et, celle de l'opération `new`, que le processus passé en paramètre ne soit pas déjà présent dans le système. Si nous reprenons les séquences générées à partir du schéma `NewRdySchema`, toutes les séquences où nous effectuons un `rdy` sur un processus inexistant

et celles où nous effectuons au moins deux `new` avec le même paramètre seront supprimées.

Le principal inconvénient de cette approche est que, comme pour le filtrage avec contraintes, il est nécessaire de générer toutes les séquences de tests avant de les filtrer. En revanche, cette méthode offre l'avantage d'éviter d'exprimer les contraintes, puisqu'elles se trouvent dans la spécification. Elle coute néanmoins plus cher en temps d'exécution puisque l'ensemble des propriétés de chaque opération est évaluée.

6.4.2 Utilisation de séquences non instanciées

À l'inverse de la première approche, nous proposons ici de ne pas déplier complètement les séquences pour les filtrer.

Génération de séquences conformes à la spécification

Par exemple, la séquence `rdy(x)` va nécessairement violer la précondition de `rdy` quelle que soit la valeur de `x`. Il existe donc des séquences dont on sait qu'elles ne peuvent satisfaire la spécification quelles que soient les valeurs des paramètres. Nous proposons donc d'éliminer au plus tôt ces séquences, en ne dépliant pas le schéma de tests suivant les valeurs des paramètres. Ainsi, le schéma `NewRdySchema` n'est que partiellement déplié [2]. Ce qui nous donne :

1. `new(_)` *[1]
2. `rdy(_)`
3. `new(_); new(_)` *
4. `new(_); rdy(_)` *
5. `rdy(_); new(_)`
6. `rdy(_); rdy(_)`
7. `new(_); new(_); new(_)`
8. `new(_); new(_); rdy(_)` *
9. `new(_); rdy(_); new(_)` *
10. `new(_); rdy(_); rdy(_)`
11. `rdy(_); new(_); new(_)`
12. `rdy(_); new(_); rdy(_)`
13. `rdy(_); rdy(_); new(_)`
14. `rdy(_); rdy(_); rdy(_)`
15. `new(_); new(_); new(_); new(_)`
16. `new(_); new(_); new(_); rdy(_)`
17. `new(_); new(_); rdy(_); new(_)`
18. `new(_); new(_); rdy(_); rdy(_)` *
19. `new(_); rdy(_); new(_); new(_)`

[2] Étant donné que les instances n'ont pas de réelle utilité ici, nous les omettons afin de faciliter la lecture.
[1] '*' identifie les séquences qui peuvent être instanciées avec `"p1"` ou `"p2"` comme valeurs de paramètre.

20.`new(_); rdy(_); new(_); rdy(_) *`
21.`new(_); rdy(_); rdy(_); new(_)`
22.`new(_); rdy(_); rdy(_); rdy(_)`
23.`rdy(_); new(_); new(_); new(_)`
24.`rdy(_); new(_); new(_); rdy(_)`
25.`rdy(_); new(_); rdy(_); new(_)`
26.`rdy(_); new(_); rdy(_); rdy(_)`
27.`rdy(_); rdy(_); new(_); new(_)`
28.`rdy(_); rdy(_); new(_); rdy(_)`
29.`rdy(_); rdy(_); rdy(_); new(_)`
30.`rdy(_); rdy(_); rdy(_); rdy(_)`

Ainsi, au lieu d'analyser 340 séquences, nous n'en analysons que 30. Parmi celles-ci, 23 ne peuvent pas être instanciées avec seulement `"p1"` et `"p2"` comme valeurs de paramètres sans violer la précondition de certaines des opérations impliquées. Nous pouvons maintenant instancier les sept dernières séquences, ce qui nous donne $2 + 4 + 4 + 8 + 8 + 16 + 16 = 58$ séquences que nous filtrons ce qui nous donne 18 séquences valides. Nous avons donc évalué, en tout, 88 séquences au lieu des 340 initialement générées.

Génération de valeurs

Plutôt que de laisser l'ingénieur choisir ses valeurs de tests, précédemment `"p1"` et `"p2"`, nous pouvons laisser le solveur libre de choisir ces valeurs. Dans le cas du gestionnaire de processus, il est néanmoins nécessaire de lui préciser l'ensemble des processus qui peuvent exister. Du point de vue de la spécification VDM, nous pouvons considérer un ensemble infini de processus. Néanmoins, vu la taille des séquences à traiter, un ensemble de quatre éléments est suffisant. Dans cet exemple simple, il est facile de déterminer le nombre de processus. Mais, dans le cas général, ce n'est pas un problème simple. Cependant, on peut noter que nous pouvons nous appuyer sur la taille des séquences (ce qui est un avantage par rapport à des outils basés sur la programmation logique avec contraintes, comme BZ-TT). Ainsi, le solveur travaille à partir d'un nouveau type `PID` défini par l'ensemble `PID = {"p1", "p2", "p3", "p4"}`.

Considérant ce nouveau type, et le schéma `NewRdySchema` tel qu'aucune valeur de paramètre n'a été spécifiée, nous pouvons utiliser un solveur pour instancier les séquences issues du schéma. Le schéma est donc déplié. Nous obtenons le même ensemble de séquences que précédemment. Sur cet ensemble de séquences, certaines ne peuvent pas être instanciées et sont donc supprimées, ce qui représente 18 séquences. Il reste donc 12 séquences qui peuvent être instanciées. Suivant les stratégies implantées dans le solveur, une même séquence peut être instanciée de diverses façons [VBL97, LPU02]. Dans ce cas, nous pouvons obtenir, pour une séquence non instanciée donnée, plusieurs séquences instanciées. Nous ne considérons pas

de stratégie particulière pour notre exemple. Nous obtenons donc une séquence instanciée par séquence donnée au solveur. Par exemple, la séquence :
`new(_) ; rdy(_) ; new(_) ; new(_)`
devient `new("p1") ; rdy("p1") ; new("p2") ; new("p3")`.
Nous pouvons voir ici le bénéfice de l'approche car l'utilisateur est aidé par l'outil dans le choix des valeurs.

6.4.3 Utilisation de séquences partiellement instanciées

Dans certains cas, il peut être intéressant de générer des tests avec des valeurs bien spécifiques pour certains paramètres. L'utilisateur peut, dans ce cas, spécifier les valeurs qui lui semblent intéressantes et laisser le soin au solveur de trier et générer les valeurs des autres paramètres. Ce cas de figure est un mélange des deux précédents. Il permet d'exercer un contrôle plus précis sur les tests générés en fixant d'office certaines valeurs.

L'exemple, qui est utilisé ici, ne permet pas de bien illustrer l'intérêt de cette approche. Dans l'exemple qui a été présenté dans [MLdB03], nous utilisons des opérations portant sur des entiers compris entre 0 et 5. Lors de son utilisation, l'outil UCASTING [VJ03] générait souvent des valeurs extrêmes 0 ou 5. Nous avons donc utilisé des séquences partiellement instanciées, où certains paramètres prenaient des valeurs intermédiaires, tout en laissant le soin à l'outil de générer les valeurs non spécifiées. Nous avons ainsi contraint l'outil dans le choix des valeurs de test et obtenu d'autres tests intéressants.

6.4.4 Gestion du non-déterminisme

Les spécifications, en fonction du niveau d'abstraction, peuvent s'avérer non-déterministes là où les implantations sont, elles, déterministes. Lorsqu'une spécification est non-déterministe, plusieurs implantations différentes peuvent la satisfaire. Par exemple, dans le gestionnaire de processus, la spécification de l'opération `swap` est non-déterministe car elle ne précise pas la manière dont est choisi le processus qui va être activé. Différents algorithmes d'ordonnancement peuvent donc satisfaire cette spécification, bien qu'ils soient différents. En conséquence, une même séquence de test peut, en fonction de l'implantation, conduire à des états différents.

Nous considérons deux types de non-déterminisme : le non-déterminisme angélique et démoniaque. Le premier part du principe que les choses vont bien se passer. Le solveur va donc chercher s'il existe au moins une solution satisfaisant la contrainte associée à la séquence d'appels d'opérations. Dans le second cas, le solveur n'acceptera que les séquences qui vérifient la spécification, quel que soit le résultat des opérations non-déterministes. Ainsi, quelle que soit l'implantation, les tests seront toujours valides, du point de vue de la spécification.

Lorsqu'on considère un non-déterminisme angélique, il peut arriver qu'une séquence générée soit valide du point de vue de la spécification mais que son exécution entraîne un verdict *inconclusif*. Prenons par exemple la séquence suivante :

```
new("p1"); new("p2"); new("p3"); rdy("p1"); rdy("p2"); rdy("p3");
```

```
swap(); swap(); rdy("p2")
```

Cette séquence crée trois processus. Le premier est activé dès l'appel `rdy("p1")` puis, les deux suivants sont mis dans l'état prêt. La spécification de l'opération `swap` est non-déterministe, nous ne savons donc pas quel processus sera activé en premier. Une fois que nous avons effectué ces deux appels d'opérations, deux états sont alors envisageables :

```
Etat1:
waiting="p1", "p2"
ready=
active="p3"

Etat2:
waiting="p1", "p3"
ready=
active= "p2"
```

Si nous passons cette séquence dans un solveur de contrainte où le non-déterminisme est considéré comme démoniaque, alors elle sera rejetée. En effet, l'un des deux états entraîne la violation de la précondition de la dernière opération (`"p2"` dans l'état `active`) et la séquence ne peut pas se terminer.

Si, par contre, le non-déterminisme est considéré comme angélique, la séquence sera acceptée. En fonction des choix faits dans l'implantation de l'opération `swap`, l'exécution de cette séquence entraînera un verdict *inconclusif*.

6.4.5 Bilan

L'utilisation de la programmation logique avec contraintes offre donc plusieurs avantages. Tout d'abord, elle évite à l'utilisateur d'écrire des contraintes afin de filtrer ses tests abstraits. Elle permet, aussi, de sélectionner les tests avant leur instanciation : les séquences qui, quelles que soient les valeurs des paramètres, ne sont jamais possibles du point de vue de la spécification sont immédiatement rejetées. Ce filtrage très en amont permet de réduire efficacement le nombre de tests ensuite instanciés. Nous avons aussi vu qu'elle permettait de générer des valeurs de tests, aidant ainsi l'utilisateur dans cette tâche. Pour finir, le filtrage s'effectue à partir de la spécification et non pas de l'implantation.

En revanche, le fait que les langages de spécification, par exemple, JML dans l'outil JML-TT ou OCL dans UCasting, ne soient pas entièrement couverts par ces outils, limite leur domaine d'application. Avec notre approche, seule l'oracle peut poser problème. En effet, si certaines parties de la spécification ne sont pas exécutable, il ne peut y avoir, à ces endroits, d'oracle automatique. C'est alors à l'utilisateur d'évaluer si le résultat du test est conforme ou non à la spécification (formelle et/ou cahier des charges). Ceci n'est pas sans poser des problèmes de temps d'analyse lorsque les suites de tests sont grandes.

6.5 Conclusions

Nous avons présenté trois techniques nous permettant de sélectionner un ensemble plus réduit de tests. Certaines de ces techniques se rejoignent, comme, par exemple, l'utilisation de nouveaux itérateurs et l'utilisation de contraintes. La troisième vise à prendre en compte la spécification formelle lorsqu'elle est disponible. L'utilisateur a ainsi un choix assez large d'outils permettant de mieux maîtriser le nombre de tests produits.

Nous avons de plus montré que notre approche pouvait se combiner avec d'autres techniques. L'utilisation d'outils couvrant plus complètement certains langages de spécification et permettant notamment d'utiliser des structures de données plus complexes permettrait de mieux évaluer l'intérêt d'un tel couplage, qui nous semble, de prime abord, prometteur.

Une quatrième piste reste à explorer : il s'agit d'utiliser des algorithmes, assurant une combinatoire partielle (par exemple, toutes les combinaisons deux à deux), tels que réalisés dans l'outil AETG présenté section 2.3.2.

Chapitre 7

Filtrage à l'exécution

Nous avons vu que l'utilisateur avait à sa disposition un mécanisme de filtrage des tests à la génération. Néanmoins, même en utilisant cette fonctionnalité, il peut rester des tests dont l'exécution n'apportera rien de plus. L'idée est donc d'identifier ces tests afin de ne pas les jouer. De cette manière, nous espérons gagner du temps lors de l'exécution et surtout alléger la phase de dépouillement. En effet, une fois que les tests ont été joués, l'ingénieur doit identifier les tests ayant levé une erreur afin de mettre en évidence les bogues du programme pour que le développeur puisse les corriger.

7.1 Principes

Le caractère systématique de la génération des tests dans Tobias fait qu'ils sont souvent très similaires. C'est grâce à cette similarité des tests que nous allons pouvoir optimiser leur exécution. Par exemple, reprenons le gestionnaire de processus présenté chapitre 3 et considérons les séquences suivantes :

```
seq1 = MTC!Sch1.new("p1"); MTC!Sch1.rdy("p1"); MTC!Sch1.new("p2");
       MTC!Sch1.rdy("p2")
seq2 = MTC!Sch1.new("p1"); MTC!Sch1.rdy("p1"); MTC!Sch1.new("p3");
       MTC!Sch1.rdy("p3")
```

Imaginons qu'il y ait un problème avec l'appel `rdy("p1")`. Deux cas sont alors à considérer :

1. une erreur de conformité (verdict *fail*), détectée, par exemple, par la violation d'une postcondition ou de l'invariant. Nous devons arrêter ce test au niveau de l'erreur, l'exécution de la deuxième séquence produira la même erreur si les opérations sont déterministes. Nous pouvons, dans ce cas, optimiser le temps d'exécution en évitant de jouer ce deuxième test. En revanche, si les opérations se comportent de manière non-déterministe, il peut être intéressant de jouer la seconde séquence car rien ne dit que le résultat sera le même.

2. une erreur de précondition. Nous nous retrouvons dans une configuration similaire au cas précédent. Néanmoins, il n'est pas forcément nécessaire d'exécuter l'opération pour évaluer sa précondition. Comme nous le verrons plus loin, ce dernier point permettra certaines optimisations plus efficaces.

Partant de ce constat, plusieurs optimisations sont réalisables. Les travaux menés par M. Kessis [Kes03] vont dans ce sens en proposant diverses optimisations pour l'exécution de séquences de tests pour des programmes Java.

7.2 Mise en œuvre

L'exécution des tests nécessite souvent, pour des raisons pratiques, un pilote de tests. Celui-ci a pour but de transmettre le verdict du test, de mettre en évidence ceux qui détectent des erreurs et ceux qui n'en détectent pas. Son utilisation facilite ainsi l'analyse des résultats. Si nous prenons comme référence JUnit [JUn] qui est un pilote de test pour les programmes Java, nous pouvons constater qu'aucune optimisation n'a été prévue. C'est-à-dire que l'outil n'essaie pas de retirer des informations des tests précédemment joués. Ceci se justifie bien évidemment par le fait qu'il n'a, *a priori*, aucune information sur la structure des tests qu'il doit jouer puisque ceux-ci sont des méthodes pouvant utiliser toute la puissance du langage Java. Dans Tobias, les tests sont nécessairement des séquences. De plus, nous avons vu que les tests générés par Tobias comportaient de nombreuses similitudes, exploitables pour optimiser l'exécution des tests.

7.2.1 Arbre d'appels

Nous construisons à partir d'une suite de tests, un arbre (n-aire) d'appels d'opérations. C'est sur cet arbre que nous allons pouvoir effectuer des optimisations. Chaque nœud de l'arbre d'appel contient un appel d'opération et peut être de deux types : "fin de séquence" (noté *fs*) ou non. Toutes les feuilles de l'arbre correspondent au dernier appel de la séquence correspondant au chemin qui y mène. Elles sont donc toutes de type *fs*. Certains nœuds internes de l'arbre peuvent aussi être de type *fs*. En effet, certaines séquences générées par l'utilisateur peuvent se trouver être des préfixes de séquences plus importantes. Même si le fait de générer de tels tests peut être discutable, l'arbre d'appel ne doit pas déformer la suite de tests générée par l'ingénieur. Nous devons donc pouvoir retrouver, en parcourant l'arbre, l'ensemble des séquences générées. En outre, la racine de l'arbre correspond à l'opération d'initialisation. Cette opération vise à configurer l'état initial du test. Par exemple, en VDM, l'opération d'initialisation va consister en l'appel à l'opération `init` qui va initialiser le système. Cette opération configure l'état du système dans l'état de départ spécifié par l'utilisateur et remet à zéro l'environnement de test de VDM. En Java, elle va consister en la création des objets sous test.

Considérons le groupe suivant ainsi que le groupe `newrdyP12` défini chapitre 6, section 6.2.4.

Soit le groupe $newrdy3swap = \{new(x), rdy(x), swap()\}$ avec $x = \{"p3"\}$.
Soit `NewRdySwap` un nouveau schéma. Celui-ci se décompose en 4420 séquences de test. Là encore il ne s'agit pas de présenter des aspects méthodologiques liés à l'utilisation de l'outil Tobias mais d'illustrer le fonctionnement des outils ou principes développés autour de l'outil.

```
NewRdySwap = newrdyP12^1..4 ; newrdy3swap^0..2
```

Parmi toutes les séquences générées, considérons les séquences suivantes :

```
new("p1")
rdy("p1")
...
rdy("p1"); new("p1")
rdy("p1"); new("p1"); new("p2")
...
new("p1"); rdy("p1")
new("p1"); rdy("p1"); rdy("p2")
new("p1"); rdy("p1"); rdy("p2") new("p3")
new("p1"); rdy("p1"); rdy("p2") swap()
new("p1"); rdy("p1"); swap()
new("p1"); rdy("p1"); new("p1");
new("p1"); rdy("p1"); new("p1"); swap();
...
```

Nous présentons figure 7.1 l'arbre d'appels correspondant aux séquences présentées ci-dessus et générées à partir du schéma `NewRdySwap`. Il s'agit donc d'un arbre incomplet.

Il est intéressant de noter que chaque nœud de l'arbre représente de manière implicite un état du système. Cet état est donné par la séquence d'appels d'opérations qui a conduit jusqu'à ce nœud, l'appel contenu dans celui-ci compris.

7.2.2 Autre manière d'exécuter les tests

L'exécution classique d'une suite de tests se fait séquence par séquence. Comme les séquences de tests générées par Tobias peuvent être des préfixes d'autres séquences, plutôt que de recommencer l'exécution depuis l'état initial, nous poursuivons l'exécution jusqu'à atteindre une feuille. Par exemple, les quatre séquences suivantes :

```
new("p1")
new("p1"); rdy("p1")
new("p1"); rdy("p1"); rdy("p2")
new("p1"); rdy("p1"); rdy("p2") new("p3")
```

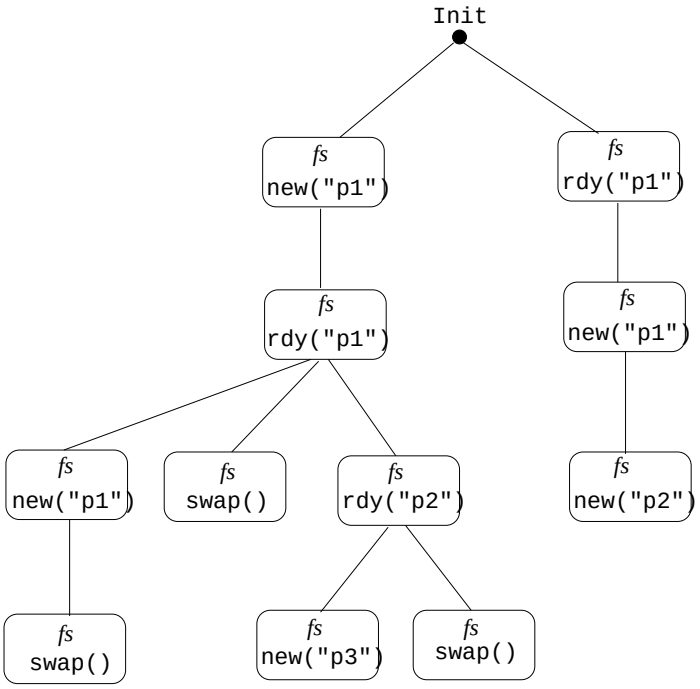


FIG. 7.1 – Arbre d'appels construit à partir du schéma `NewRdySwap`.

sont normalement exécutées les unes après les autres. Cela signifie que nous allons exécuter quatre fois l'opération `new("p1")`, trois fois l'opération `rdy("p1")` et deux fois l'opération `rdy("p2")`. En tout, nous aurons exécuté dix appels d'opérations. À partir du moment où aucun problème ne se produit, il n'est pas rentable de recommencer plusieurs fois les mêmes choses. Ainsi, en exécutant "seulement" la dernière séquence, nous obtenons le même résultat qu'en ayant exécuté, à la suite, les quatre séquences. Une simple exécution des opérations en faisant un parcours de l'arbre en profondeur d'abord nous permet d'aboutir à ce résultat.

7.3 Optimisations

Lorsqu'une (sous-) séquence entraîne une erreur, rien ne sert de la rejouer car il serait nécessaire d'omettre le(s) appel(s) erroné(s) ce qui dénaturerait le sens du test.

Plusieurs stratégies sont alors envisageables [Kes03].

1. Chaque suite de tests est représentée sour la forme d'un arbre. Lorsqu'un nous exécutons une séquence de test, nous parcourons l'arbre. Lorsqu'un appel d'opération, contenu dans un nœud de l'arbre, engendre une erreur, il nous suffit de couper toutes les branches qui lui sont rattachées. Par exemple, dans la séquence d'opérations `new("p1") ; rdy("p1") ; rdy("p2")`, l'appel `rdy("p2")` viole la précondition de l'opération `rdy`. En coupant les branches rattachées à ce nœud, comme montré figure 7.2,

nous évitons d'exécuter plusieurs fois ce préfixe invalide.

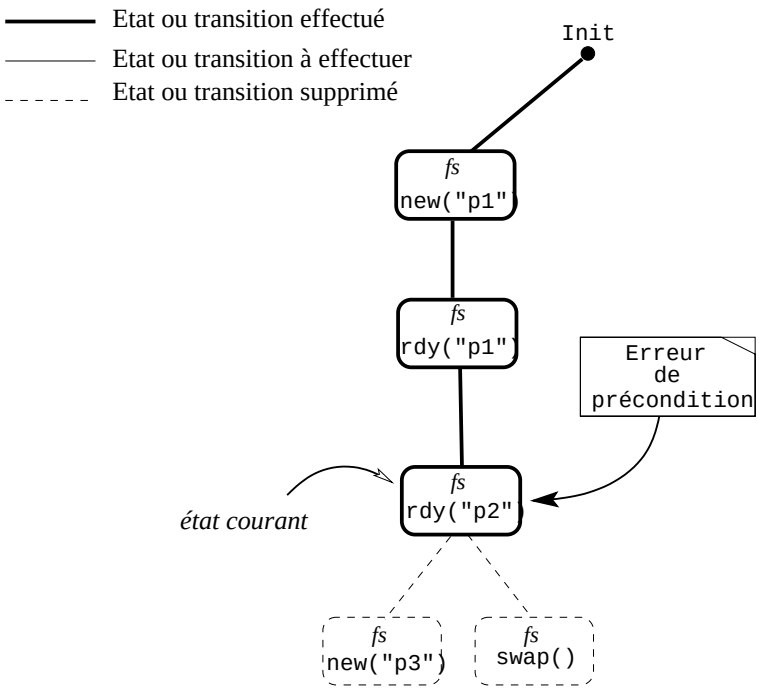


FIG. 7.2 – Élimination des fils d'un nœud ayant entraîné une erreur.

2. Considérant que chaque nœud représente un état du système, certaines vérifications peuvent se faire a priori. Si nous considérons l'ensemble des séquences commençant par `new("p1") ; rdy("p1")`, nous pouvons, lors de l'exécution évaluer toutes les préconditions des opérations qui sont les fils directs du nœud `rdy("p1")`. En effet, les préconditions ne modifient pas l'état du système rien n'interdit leur évaluation. Sur notre exemple, les séquences :
   `new("p1") ; rdy("p1") ; new("p1")` et
   `new("p1") ; rdy("p1") ; rdy("p2")`
   violent la précondition de la dernière opération. En conséquence, nous effectuons deux fois la séquence `new("p1") ; rdy("p1")`. En évaluant au plus tôt les valeurs des préconditions, nous pouvons éviter d'exécuter cette séquence inutilement, comme illustré figure 7.3.
3. De même, lorsque certaines opérations ou fonctions ne modifient pas l'état du système, nous pouvons évaluer la précondition des opérations qui leur sont rattachées. Nous avons, figure 7.4, modifié les tests précédemment générés de sorte que certaines des séquences fassent appel à une opération `print` qui permet d'afficher l'état du système. Cette opération ne modifie pas l'état du système. Il est donc possible d'explorer le graphe plus en profondeur pour évaluer certaines préconditions. En effet, l'état en sortie du nœud `print` est le même que l'état en sortie du nœud courant.

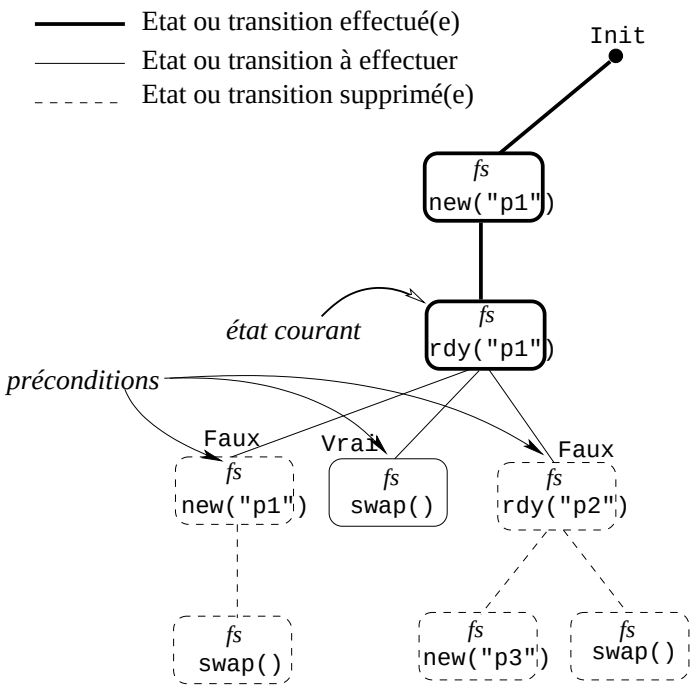


FIG. 7.3 – Arbre d'appels modifié.

7.4 Expérimentations

La figure 7.5 montre comment le pilote de tests est interfacé avec l'outil Tobias et l'environnement VDM. Il prend en entrée une suite de tests, la transforme en arbre et, exécute les séquences contenues dans l'arbre opération par opération. Notre implantation ne réalise pas les optimisations liées à l'évaluation des préconditions des opérations suivant un état donné (techniques 2 et 3).

Nous avons fait certaines expérimentations dont les résultats sont donnés table 7.1.

Nous avons exécuté les tests générés à partir des schéma `NewRdySchéma` (donné chapitre 6, section 6.2.4) et `NewRdySwap` (donné section 7.2 de ce chapitre). Ces deux suites de tests génèrent chacune 340 et 4420 séquences de test.

Les résultats, donnés table 7.1, montrent les avantages de l'utilisation du mécanisme de filtrage à l'exécution. On y constate une réduction très significative du nombre de tests effectivement exécutés et du temps d'exécution. Ces résultats sont en grande partie dus à la structure même des tests. En effet, de nombreuses séquences sont sous-séquences d'autres séquences. En conséquence, l'arbre contenant tous les tests comporte beaucoup de branchements pour une profondeur assez faible. Ainsi, l'élimination d'un nœud intermédiaire permet de couper un nombre important de branches et donc de tests. De plus, il y a un grand nombre de tests éliminés par rapport à l'ensemble des tests générés ce qui accélère d'autant le temps

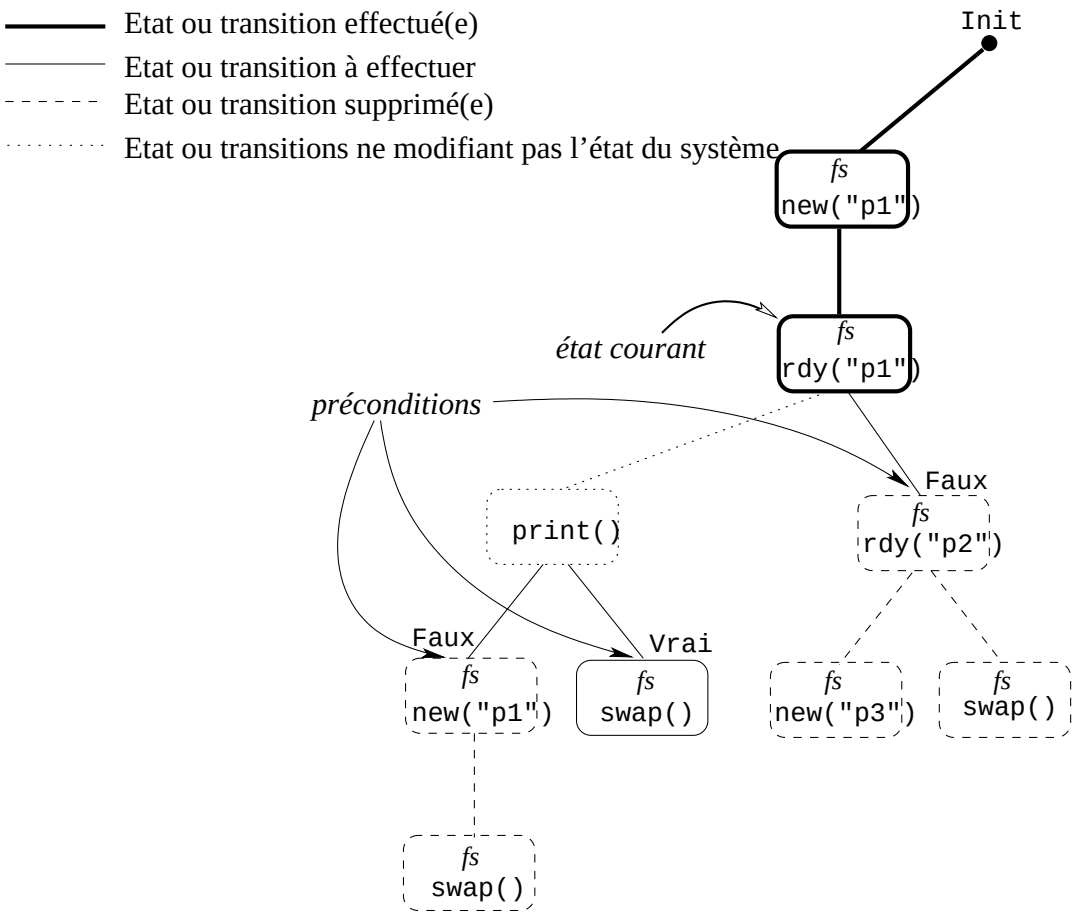


FIG. 7.4 – Arbre d'appels modifié.

d'exécution. Les expériences menées sur l'étude de cas présentée chapitre 8 et données section 8.5 confirment son utilité, même si le temps gagné est moins significatif que ce qui est montré ici.

D'autres expérimentations sont présentées dans [LdBMB04]. Elles sont effectuées sur l'étude de cas, présentée chapitre 5, développée par la société Gemplus en Java et spécifiée en JML. Elles montrent que l'utilisation de ces techniques de filtrage améliore le temps d'exécution des tests de 50% à 90%. Bien évidemment, cela dépend, là encore, de la structure de l'arbre de test et donc des tests générés.

7.5 Conclusions

Le filtrage à l'exécution présenté ici, a montré plusieurs avantages. Tout d'abord, cette technique est entièrement automatique et sûre : on ne risque pas ici de supprimer des tests utiles. De plus, nous avons pu constater des gains de temps parfois très significatifs et peu de surcoût.

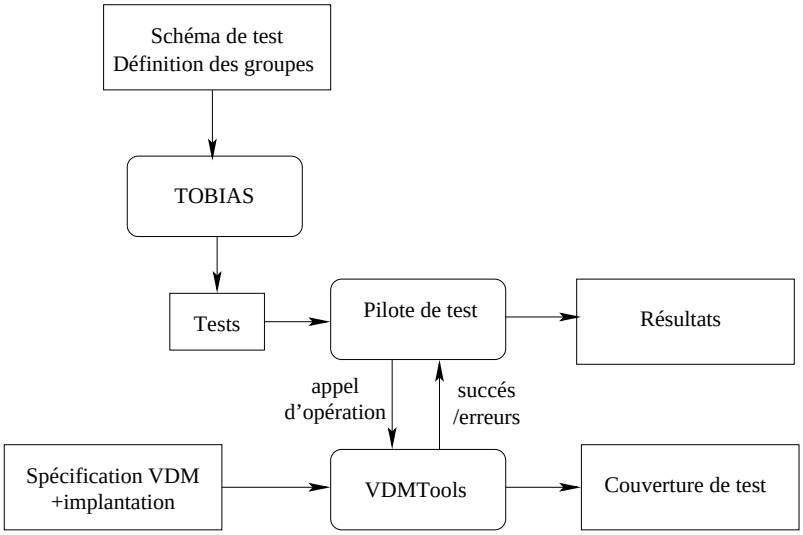


FIG. 7.5 – Architecture du pilote de tests.

Suite de tests	Nb de Tests	Optimisation à l'exécution	Temps total d'exécution	Nb de tests non joués
`NewRdySchéma`	340	aucune	$\sim$ 6s	0
		complète	1s 591ms	322
`NewRdySwap`	4420	aucune	$\sim$ 110s	0
		complète	7s 508ms	4318

TAB. 7.1 – Temps d'exécution et nombre de tests éliminés à l'aide des divers pilotes de test.

En revanche, seule la première des trois techniques a été implantée. Nous ne pouvons donc pas vérifier le gain que les deux autres propositions pourraient apporter. En effet, elles permettent d'anticiper la violation de préconditions, évitant donc de réexécuter plusieurs fois le même préfixe. Néanmoins, si l'arbre possède beaucoup de branches, le coût d'exécution de chaque précondition risque de ne pas être négligeable lorsque peu de cas sont exclus par ce biais : les préconditions sont alors évaluées plusieurs fois. Si, en revanche, nous étions capable d'évaluer chaque précondition une seule et unique fois en extrayant celles-ci [1], les techniques 2 et 3 donneraient au pire le même résultat et au mieux amélioreraient encore le temps d'exécution.

[1]Par exemple, en utilisant la programmation par aspects, nous pourrions faire évaluer la précondition par le pilote de tests et non plus à l'intérieur de l'appel de méthode comme c'est le cas avec JML.

Chapitre 8

Étude de cas en VDM

Nous avons introduit jusqu'à présent un outil et des techniques permettant d'une part, d'améliorer la qualité des jeux de tests et d'autre part, d'accélérer le temps d'exécution. Nous présentons dans ce chapitre une étude de cas plus complète que l'exemple précédemment utilisé.

L'objectif de ce chapitre est d'évaluer notre approche ainsi que l'intégration des divers concepts présentés auparavant. Il s'agit aussi de présenter la manière dont nous avons utilisé le langage de schéma de tests. De cette manière, nous donnons des éléments méthodologiques et comparons deux façons de faire. La première consiste en la création de "petits" schémas permettant de bien maîtriser l'explosion combinatoire. La seconde s'appuie plus sur le principe combinatoire du langage pour générer de nombreuses séquences de tests. Ces deux approches ont leurs avantages et inconvénients comme nous le montrons dans ce chapitre. Néanmoins, nous n'avons pas pour objectif de dégager une méthodologie générale d'utilisation.

Une des techniques permettant d'évaluer la qualité d'une suite de tests est le test de mutation [DLS78, OH96]. Ce type de test a des contraintes fortes puisque pour obtenir des résultats ayant du sens, il est nécessaire de générer tous les mutants et de bien penser le type de mutation que l'on va effectuer. Toutefois, il n'existe pas de générateur de mutants pour VDM. Donc, plutôt que de générer des mutants, nous avons mené une expérience avec des étudiants de troisième cycle universitaire. Ces étudiants ont dû réaliser certaines des opérations de l'étude de cas en temps limité (trois heures). Ce faisant, nous avons récupéré leurs implantations qui nous ont servi de matière première pour l'expérimentation. Nous comparons ensuite le nombre d'implantations erronées détectées par les différentes suites de tests dont une a été écrite à la main et deux ont été générées à partir de schémas de tests.

Nous donnons, dans un premier temps, un cahier des charges informel et une spécification formelle VDM associée. Puis, nous présentons les différentes implantations à notre disposition. Nous exposons ensuite la manière dont ont été créés les tests. Enfin, nous étudions les résultats obtenus et les avantages des filtrages à la génération et à l'exécution.

8.1 Cahier des charges et spécification

Une classe d'étudiants est divisée en groupes de petite taille. Ces groupes doivent respecter les contraintes suivantes, comme illustré figure 8.1 :

1. un groupe est composé de 3 à 5 étudiants ; si la classe comporte moins de 5 étudiants, ils doivent appartenir au même groupe ;
2. le groupe le plus grand a, au plus, un étudiant de plus que le plus petit groupe ;
3. chaque étudiant appartient à un et un seul groupe.

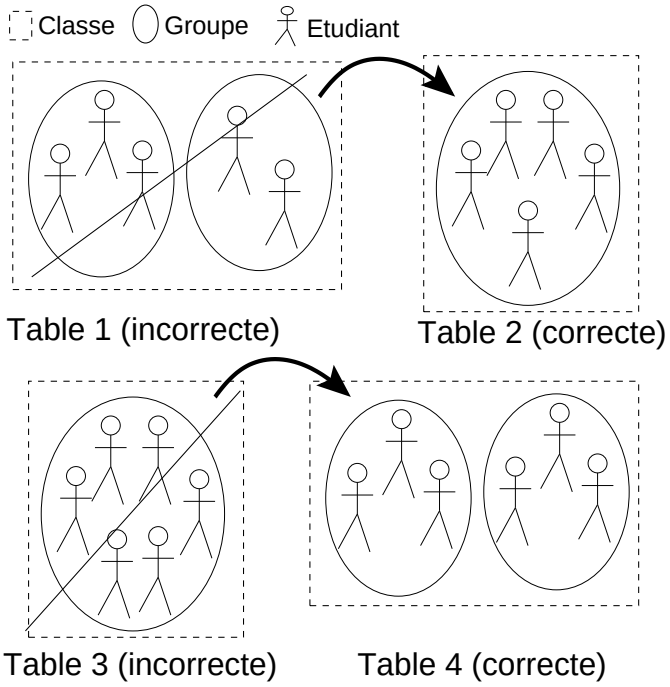


FIG. 8.1 – Quelques exemples de groupes d'étudiants corrects et incorrects

Si nous reprenons la figure 8.1, nous pouvons voir que la table 1 ne respecte pas la première contrainte. En effet, la classe est composée de 5 étudiants répartis en deux groupes. De plus, le second groupe n'est composé que de 2 étudiants. La bonne répartition est montrée table 2. La table 3, qui contient un groupe de 6 étudiants viole la première contrainte. La bonne répartition est montrée table 4. La classe est composée de deux groupes de 3 étudiants conformément aux contraintes 1 et 2.

Nous avons choisi de spécifier ces groupes en VDM de la manière suivante : les étudiants sont représentés par des séquences de caractères, les identificateurs de groupe sont des entiers

strictement positifs et le type `table` est défini comme une fonction (map) allant des étudiants vers les identificateurs de groupe. Le fait d'utiliser une fonction pour représenter la table, garantit structurellement qu'un étudiant appartient au plus à un groupe (troisième contrainte).

```
student = seq of char;
gr_id = nat1;
table = map student to gr_id;
```

Après avoir introduit ces types, nous pouvons définir les variables d'état. Il n'y a qu'une seule variable d'état `gr` (*ligne 2*) servant à stocker les assignations des étudiants à leur groupe. L'invariant d'état garantit que :

- la taille du groupe le plus grand (`size_max_gr`) est inférieure à la limite maximum (*ligne 4*) ;
- le groupe le plus grand a, au plus, un étudiant de plus que le groupe le plus petit (*ligne 6*) ;
- lorsqu'il y a plus d'un groupe, la taille du groupe le plus petit est supérieure à la taille limite (*lignes 8 et 9*).

On initialise la variable d'état comme une fonction vide (*ligne 11*). Les fonctions `size_max_gr` et `size_min_gr` sont données annexe B.1.

```
1  state groups of
2          gr : table
3  inv mk_groups(gr) ==
4          size_max_gr(gr) <= limit.max
5          and
6          (size_max_gr(gr) - size_min_gr(gr) <= 1)
7          and
8          if card rng gr > 1 then
9                  size_min_gr(gr) >= limit.min
10        else true
11 init G == G = mk_groups({|->})
12 end
```

Les tailles minimum et maximum (`limit.max` et `limit.min`) des groupes sont définies à l'aide de constantes (`values`). Dans cette spécification, nous utilisons les propriétés du type composé `lim` afin de renforcer la propriété de la constante, notamment à l'aide d'un invariant de type. Cet invariant permet de s'assurer que les valeurs que l'on donne comme taille minimum et maximum permettront, lorsqu'un groupe atteint la limite supérieure, de diviser ce groupe en deux groupes de taille plus petite et d'ajouter un nouvel étudiant.

Par exemple, une classe de 5 étudiants correspond à un seul groupe et une classe de 6 étudiants correspond à deux groupes. Si la limite minimum était fixée à 4 étudiants, il ne

serait pas possible de diviser une classe de 6 étudiants en deux groupes valides.

```
values limit : lim = mk_lim(3,5)
types
 lim :: min : nat
        max : nat1
 inv mk_lim(min,max) == ((max+1)div 2) >= min;
```

La définition des types et de l'état garantit les propriétés définies au début de cette section.

L'invariant portant sur `gr` étant particulièrement fort, la majorité des opérations définies ci-après entraîne des modifications dans l'affectation des étudiants aux groupes. L'intérêt de cette étude de cas est qu'elle possède des opérations d'ajout (de groupe ou d'étudiant) qui ont une spécification non-déterministe. Celles-ci laissent une grande liberté d'implantation et de modification de la table d'étudiants.

– `load_table` charge une table dans la variable `gr` ;

```
load_table:table ==> ()
load_table(t) == ...
pre size_max_gr(t) <= limite.max
        and
        (size_max_gr(t) - size_min_gr(t) <= 1)
        and
        if card rng t > 1 then
                size_min_gr(t) >= limit.min
        else true
post gr = t
```

La précondition de cette opération est très forte puisque, comme il s'agit d'affecter une nouvelle table à l'état, il faut assurer que celle passée en argument ne viole pas l'invariant.

– `swap_two_students` échange deux étudiants, chacun d'entre eux se retrouvant dans le groupe de l'autre ;

```
swap_two_students : student * student ==> ()
swap_two_students(e1,e2)== ...
pre e1 in set dom gr and e2 in set dom gr
post dom gr = dom gr~ and gr(e1) = gr~(e2) and gr(e2) = gr~(e1)
        and {e1,e2} <-: gr = {e1,e2} <-: gr~
```

La précondition spécifie que l'on ne peut échanger que deux étudiants effectivement présents dans la table. La postcondition établit que cette opération conserve les mêmes étudiants, que l'échange a bien eu lieu et que les autres étudiants ne changent pas de groupe.

– `add_group` augmente le nombre de groupes. Cette opération conserve les étudiants présents mais modifie leurs affectations dans les groupes ;

```
add_group: () ==> ()
add_group() == ...
pre card rng gr < max_nb_of_gr(gr)
post dom gr = dom gr~ and card rng gr = card rng gr~ + 1
```

La précondition spécifie que le nombre de groupes et le nombre d'étudiants sont suffisants pour pouvoir ajouter un nouveau groupe. La postcondition, non-déterministe, définit que l'ensemble des étudiants reste inchangé et que le nombre de groupes est augmenté de 1.

– `add_student` ajoute un nouvel étudiant à la classe ; cette opération peut modifier les affectations des étudiants si cela est nécessaire ;

```
add_student:student ==> gr_id
add_student(e) == ...
post dom gr = (dom gr~ union {e}) and RESULT = gr(e)
```

Cette opération peut être appelée à n'importe quel moment car il est toujours possible d'ajouter un étudiant, même s'il est déjà présent. Elle n'a donc pas de précondition. La postcondition spécifie que l'étudiant passé en paramètre est bien présent dans la table après l'opération et que le numéro du groupe renvoyé par l'opération correspond au numéro du groupe du nouvel étudiant.

8.2 Différentes implantations

Afin d'évaluer la qualité des tests qui ont été écrits à la main et générés à l'aide des schémas de tests, nous avons soumis en exercice à des étudiants de DESS Génie Informatique le développement de cette application. Les étudiants avaient à leur disposition le cahier des charges informel ainsi que les spécifications VDM des différentes opérations. Seule l'opération `load_table` leur était fournie avec son code. Ils devaient programmer et tester les trois autres opérations. Le temps qui leur était imparti était limité et les étudiants n'avaient qu'une connaissance restreinte du langage VDM. Implanter l'opération `add_student` correctement est relativement difficile, compte tenu des contraintes fortes de l'invariant. La spécification de l'opération laisse néanmoins de grandes libertés d'implantation. Nous avons ainsi obtenu *12 implantations* différentes des opérations `add_student`, `add_group` et `swap_two_students` qui avaient de grandes chances d'être erronées.

Nous avons de plus, deux implantations de référence élaborées par nos soins. L'opération, présentée ci-dessous, correspond à la première version que nous avons réalisée.

```
1 add_student:student ==> gr_id
2 add_student(e) ==
3 (if size_min_gr(gr) = limit.max then add_group();
4  let gid in set rng gr
5  be st card dom (gr :> {gid}) = size_min_gr(gr)
6        in (gr := gr ++ {e |-> gid}; return(gid)))
7 post dom gr = (dom gr~ union {e}) and RESULT = gr(e)
```

Notre première version, donnée ci-dessus, vérifie d'abord qu'il y a une place libre dans l'un des groupes. Si tous les groupes sont complets, l'opération `add_group` est appelée. Puis `add_student` affecte le nouvel étudiant à l'un des groupes qui a le plus petit nombre d'étudiants. Cette première implantation est erronée car si la table est vide au moment où nous appelons l'opération alors une erreur système est levée. En effet, l'instruction *ligne 6* cherche à sélectionner un numéro de groupe dans la table. Or, lorsque celle-ci est vide, cette instruction ne peut fonctionner. Une autre erreur se produit si nous avons une table comportant cinq étudiants : l'instruction *ligne 3* ajoute un nouveau groupe alors que l'étudiant n'a pas été ajouté, ce qui viole la précondition de l'opération `add_group`. Nous connaissions la seconde erreur avant d'écrire les schémas de tests mais la première erreur nous était inconnue.

La seconde version de référence est présentée ci-dessous. Elle a été réalisée après correction des erreurs détectées par les tests de la première version. Les tests, que nous avons créés, ne détectent aucune erreur dans cette version.

```
1 add_student(e) ==
2  (if gr = {|->} then (gr := {e |-> 1}; return(1))
3  else (
4  if e in set dom gr then return(gr(e))
5  elseif  size_min_gr(gr) = limit.max and (card rng gr = 1) then
6      (dcl buf : table := {|->};
7       dcl grnb : gr_id := 1;
8       for all et in set (dom gr union e) do
9            (buf := buf ++ {et |-> grnb};
10              if grnb = 1
11             then grnb := 2
12             else grnb := 1;);
13     gr := buf;
14     return(gr(e));
15     )
16   else (if size_min_gr(gr) = limit.max and (card rng gr > 1)
17         then add_group();
18         let gid in set rng gr
19         be st card dom (gr :> {gid}) = size_min_gr(gr)
```

```
20          in (gr := gr ++ {e |-> gid}; return(gid))))
21 )
22 post dom gr = (dom gr~ union {e}) and RESULT = gr(e)
```

Le code de l'opération fonctionne comme suit :

1. si la table d'étudiant `gr` est vide (*ligne 2*), l'étudiant passé en paramètre est mis dans le groupe 1 ;
2. si l'élément que l'on ajoute est déjà présent dans la table (*ligne 4*), on ne fait rien ;
3. nous allons ajouter l'étudiant à un groupe :
   - s'il y a un seul groupe ayant atteint sa taille maximale (*ligne 5*), auquel cas on construit une table temporaire `buf` en répartissant les étudiants du groupe existant et le nouvel étudiant dans deux groupes nommés 1 et 2 (*lignes 6 à 13*) puis on retourne le numéro de groupe du nouvel étudiant,
   - s'il y a plusieurs groupes et si tous ont atteint la taille maximale (*ligne 16*), on rajoute un nouveau groupe (*ligne 17*). On se retrouve alors dans le cas où la cardinalité du plus petit groupe est inférieure à la taille maximale (cas normal, ou résultant de l'ajout d'un nouveau groupe). Nous sélectionnons l'un des groupes ayant la cardinalité la plus faible et nous y ajoutons le nouvel étudiant (*lignes 18 à 20*).

Nous avons donc en tout 14 implantations à notre disposition : les 12 conçues par les étudiants, notre première version d'`add_student` et la version corrigée, toutes deux présentées ci-dessus.

8.3 Utilisation d'une suite de tests manuelle

8.3.1 Création des tests

Dans un premier temps, une suite de tests a été écrite à la main. Cette suite de tests vise à tester les principales opérations de l'application, à savoir `load_table`, `add_group`, `swap_two_students` et `add_student`. Elle comporte en tout *26 appels d'opérations* regroupés au sein d'une seule séquence de tests. Le choix des valeurs a été fait avec attention pour tester différentes configurations mais n'a pas été fait suivant une stratégie de test aux limites. Cette suite de tests a été définie de manière à garantir 100% de couverture[1] des opérations testées.

```
load_table(|->),
load_table("a"|->1),
load_table("a"|->1,"b"|->1),
load_table("a"|->1,"b"|->1,"c"|->1),
load_table("a"|->1,"b"|->1,"c"|->1,"d"|->1),
```

[1]La notion de couverture est expliquée section 8.4.3 de ce chapitre.

```
load_table("a"|->1,"b"|->1,"c"|->1,"d"|->1,"e"|->1),
load_table("a"|->1,"b"|->1,"c"|->1,"d"|->2,"e"|->2,"f"|->2),
load_table("a"|->1,"b"|->1,"c"|->1,"d"|->2,"e"|->2,"f"|->2,"g"|->2),
"cas erronnes (vérification de la précondition)",
load_table("a"|->1,"b"|->1,"c"|->1,"d"|->1,"e"|->1,"f"|->1,"g"|->1),
load_table("a"|->1,"b"|->1,"c"|->1,"d"|->1,"e"|->1,"f"|->2,"g"|->2),
load_table("a"|->1,"b"|->1,"c"|->1,"d"|->2,"e"|->2,"f"|->2,"g"|->2,
           "h"|->2),
"test de swap_two_students",
load_table("a"|->1,"b"|->1,"c"|->1,"d"|->2,"e"|->2,"f"|->2,"g"|->2),
swap_two_students("a","b"),
swap_two_students("a","d"),
swap_two_students("f","c"),
swap_two_students("a","a"),
"test de add_group() et add_student()",
load_table("a"|->1,"b"|->1,"c"|->1,"d"|->2,"e"|->2,"f"|->2,"g"|->2),
add_student("h"),
add_student("j"),
add_student("k"),
add_student("l"),
add_student("m"),
add_student("n"),
add_student("o"),
add_student("p"),
add_student("q")
```

8.3.2 Exécution des tests

L'exécution de la suite de tests manuelle a permis de détecter des erreurs dans 9 implantations de l'opération `add_student` parmi celles élaborées par les étudiants. Le potentiel de détection d'erreurs de cette suite de test nous conforte dans l'idée qu'elle était bien pensée. Malgré tout, cette suite de tests n'a pas réussi à détecter d'erreur dans 3 des implantations faites par les étudiants ni dans notre implantation initiale. Cette suite de tests manuelle a été écrite avant que nous ne découvrions l'erreur par relecture du code.

Aucune erreur n'a été détectée dans les autres opérations.

8.4 Utilisation des schémas de tests

Comme nous l'avons expliqué chapitre 4, section 4.4 le principe de fonctionnement des schémas de tests entraîne facilement un problème d'explosion combinatoire. L'utilisation des schémas de tests peut se faire de plusieurs façons. Il est ainsi possible, en augmentant le niveau de détail des schémas, de maîtriser l'explosion combinatoire. Cette approche a comme inconvénient de demander plus de travail à l'ingénieur. Une autre possibilité est, à l'inverse, d'utiliser la combinatoire pour décharger le plus possible l'ingénieur des tâches fastidieuses.

Nous présentons dans cette section ces deux approches et discutons de leurs avantages et inconvénients à partir des résultats obtenus à l'exécution.

La création des schémas de tests s'est faite en plusieurs étapes. A partir des spécifications informelles et formelles, nous avons identifié les propriétés à tester. En premier lieu, nous avons créé une suite de "tests simples" permettant de maîtriser l'explosion combinatoire en écrivant des schémas très simples. Ces tests visent à tester un unique appel d'opération après la configuration de l'état.

8.4.1 Suite de "tests simples" (*Suite 1*)

Une des facilités de test de cette application est qu'il est possible de charger directement une table d'étudiants, et donc un état, à l'aide d'un seul appel d'opération, contrairement au gestionnaire de processus, présenté chapitre 3, où il fallait plusieurs appels d'opération pour configurer l'état.

Nous avons donc créé plusieurs groupes [2] à partir du cahier des charges informel et en appliquant globalement une stratégie de test aux limites. Le prototype Tobias permettant d'ajouter facilement de nouvelles valeurs de tests, nous avons défini différentes tables correspondant à des comportements standards. Etant donné que nous étions en première phase de tests et qu'aucune propriété n'a été prouvée sur la spécification, nous avons aussi défini, pour chaque opération, des cas qui doivent normalement violer leur précondition. Ces cas permettent de nous assurer que les préconditions sont correctement spécifiées. Ils ont été créés à partir des spécifications informelles. Par exemple, la spécification dit que si la classe comporte moins de 5 étudiants, ils doivent appartenir au même groupe. Nous allons donc essayer de créer des tables avec 2 étudiants dans un groupe et 3 dans un autre.

Pour tous les schémas de tests, nous avons défini une instance `gr` sur laquelle sont appelées les opérations sous test.

Création des schémas

Chargement des tables. Nous voulons nous assurer que l'opération permettant de charger une table n'autorise pas le chargement d'une configuration invalide.

Le groupe `LTIGr` (voir annexe B.2.1) vise à essayer de mettre l'état dans une configuration non conforme à la spécification informelle. Ce faisant, nous essayons de charger des groupes qui sont :

- soit invalides :{`"e1"|->a`}, `a` doit être un entier or `a` n'est pas défini, donc cette table est invalide ;
- soit limites : {`|->`} ou encore {`"e1"|->1,...,"e20"|->43`} qui est une table qui contient 5 groupes de 5 étudiants

[2] Nous donnons en annexe B.2 l'ensemble des éléments composant les groupes qui ne sont que partiellement présentés ici.

afin de tester la conformité de l'implantation par rapport à la spécification VDM et la correction de la spécification VDM par rapport au cahier des charges informel. Ce type de test ne devrait plus avoir d'intérêt lorsque nous effectuerons des tests de non régression si la spécification formelle n'a pas été modifiée.

Groupe	Opération	Paramètre(s)	
LTIGr	load_table(t)	t	{|->}
			{"e1"|->a}
			...
			{"e1"|->1,...,"e20"|->43}

Le schéma de tests TestLoadTable nous permet de tester l'opération load_table en générant les *26 appels* spécifiés par le groupe LTIGr.

```
TestLoadTable = MTC!gr.LTIGr
```

Test de l'opération add_student**.** Nous définissons maintenant les différents tests pour l'opération add_student. Cette opération est la plus difficile à réaliser compte tenu des contraintes invariantes. En conséquence, il nous faut choisir avec attention les états dans lesquels cette opération sera testée. Nous commençons donc par créer un groupe permettant de charger différents états.

Le groupe LTVGr(voir annexe B.2.2) nous permet de charger 17 configurations de tables différentes comportant de 0 à 5 groupes d'étudiants. La taille et la configuration des différents groupes a été choisie de manière à être limite et correcte. Nous reprenons donc certaines des valeurs utilisées dans le groupe LTIGr. Par ailleurs, ce groupe nous resservira pour tester les autres opérations de l'application. C'est donc aussi en pensant aux autres tests que nous avons défini ces tables. Elles incluent donc des cas limites pour les autres opérations (plusieurs groupes de taille 3 ou 5, etc.).

Le choix des numéros de groupes est fait de manière à essayer de mettre en défaut l'implantation. Il est en effet possible qu'elle tienne compte de ces numéros pour effectuer son traitement, par exemple, en faisant le pari que tous les numéros de groupes sont consécutifs.

Groupe	Opération	Paramètre(s)	
LTVGr	load_table(t)	t	{|->}
			{"e1"|->1}
			...
			{"e1"|->1,...,"e20"|->43}

Puis nous définissons le groupe permettant de tester l'opération à proprement parler. Cette opération ne comporte pas de précondition, le test n'a donc pas besoin de chercher à la valider.

Le groupe `AddStudentGr` nous permet d'ajouter des étudiants. Globalement, lorsque l'on essaie d'ajouter un étudiant, soit il est déjà présent dans la table, soit il n'y est pas. Cela nous donne donc deux cas à tester (`"e1"` pour le cas de l'étudiant déjà présent et `"e55"` dans l'autre cas), sauf lorsque la table ne contient aucun étudiant.

Groupe	Opération	Paramètre(s)	
`AddStudentGr`	`add_student(s)`	`s`	`"e1"`
			`"e55"`

Ce nouveau schéma, donné ci-dessous, permet de générer *34 séquences* visant à tester l'opération `add_student`. Comme nous faisons la combinatoire des appels définis par les deux groupes utilisés, nous testons l'ajout d'un étudiant présent et d'un étudiant non présent avec différentes configurations de l'état.

```
TestAddStudent = MTC!gr.LTVGr ; MTC!gr.AddStudentGr
```

Test de l'opération `swap_two_students`. L'opération `swap_two_students` vise à échanger la place des deux étudiants passés en paramètre. Nous avons identifié, informellement, cinq cas différents pour le test de l'opération `swap_two_students`. Nous avons le cas où :

- nous essayons d'échanger un étudiant avec lui même,
- nous échangeons un étudiant avec un autre étudiant du même groupe,
- nous échangeons deux étudiants qui sont dans des groupes différents,
- nous échangeons un étudiant avec un étudiant qui n'existe pas et enfin,
- nous échangeons deux étudiants qui n'existent pas.

Nous avons déjà défini un groupe permettant de charger différentes configurations de l'état. Nous n'en créons pas de nouveau pour le test de cette opération car les différents cas énumérés ci-dessus peuvent se vérifier à partir des états existants. Il est à noter que la taille des groupes, contenant l'étudiant de départ et d'arrivée, peut avoir une influence sur le comportement de cette opération. Le mécanisme de génération combinatoire des tests va nous permettre de créer des séquences permettant de tester ces différentes configurations.

Les valeurs que nous avons choisies pour créer le groupe `Swap_two_studentsGr` nous permettent de tester les quatre cas définis plus haut en fonction de la table qui est chargée. Les propriétés qui ont guidé notre choix sont : deux valeurs identiques, deux valeurs différentes appartenant à un même groupe et, deux valeurs différentes appartenant à deux groupes.

Groupe	Opération	Paramètre(s)			
`Swap_two_studentsGr`	`swap_two_student(s1,s2)`	`s1`	`"e1"`	`s2`	`"e1"`
					`"e3"`
					`"e20"`

Le schéma `TestSwapTwoStudents` génère les *51 séquences* permettant de tester l'échange de deux étudiants. En outre, les tests que nous générons devraient dans certains cas violer la précondition de l'opération d'échange. Ainsi, nous ne devons pouvoir exécuter l'opération que si nous échangeons des étudiants qui sont tous deux présents dans la table. Notons que, si le nombre de tests produits avait été trop important, nous aurions pu définir des groupes plus précis afin de créer un schéma de tests permettant d'évaluer la correction de la précondition et un autre schéma de tests permettant de tester la conformité de l'opération.

```
TestSwapTwoStudents = MTC!gr.LTVGr ; MTC!gr.Swap_two_studentsGr
```

Test de l'opération `add_group`. L'opération `add_group` est également une opération délicate. En effet, l'ajout d'un nouveau groupe n'est pas forcément trivial, car il nécessite souvent une nouvelle répartition des étudiants.

Il n'est, par contre, pas nécessaire ici de créer un groupe car l'opération n'a pas de paramètres. Nous continuons d'utiliser le groupe `LTVGr` car les configurations qu'il contient sont suffisamment limites pour poser des problèmes.

Nous avons défini le schéma `TestAddGroup`, générant *17 séquences*, permettant de tester l'ajout d'un groupe.

```
TestAddGroupe = MTC!gr.LTVGr ; MTC!gr.add_group()
```

Exécution des tests et résultats

L'exécution de l'ensemble des tests générés ici prend quelques secondes.

La suite de tests générée à partir du schéma `TestAddGroup` a permis de détecter une erreur dans l'opération `add_group` de l'une des implantations des étudiants. Cette erreur a été détectée car nous avons chargé une table contenant 15 étudiants répartis en 3 groupes. Une telle table peut aussi se décomposer en 5 groupes de 3 étudiants. L'implantation des étudiants n'a pas tenu compte de cette possibilité et a créé, en un seul appel d'opération, une table comportant 5 groupes. Elle a donc créé deux groupes supplémentaires ce qui viole la spécification d'`add_group` disant que l'on ne peut ajouter qu'un seul groupe à la fois.

L'opération qui pose le plus de difficultés est l'opération `add_student`. La première version, que nous avons écrite, comportait une erreur identifiée à la lecture du code et une erreur inconnue lors de l'élaboration des tests avec Tobias. Les erreurs détectées par le schéma `TestAddStudent` apparaissent, soit lorsqu'il n'y a aucun groupe, soit lorsque l'on essaie de rajouter un étudiant alors qu'il n'existe qu'un seul groupe et que celui-ci est plein. Si aucun groupe n'existe, il est impossible d'ajouter un étudiant au groupe le plus petit. Si l'on a un

seul groupe de 5 étudiants, on ne peut pas le diviser avant d'avoir ajouté l'étudiant, mais, si cet étudiant est déjà présent, il ne faut rien faire.

La seconde version de l'opération `add_student` n'a pas été créée du premier coup. Suite au test de la première version et aux deux erreurs que nous avons trouvées, nous avons corrigé une première fois le code de l'opération. Nous avons alors rejoué les tests générés à partir du schéma `TestAddStudent` sur cette nouvelle version et avons trouvé une nouvelle erreur ce qui nous a conduit à élaborer la seconde version présentée dans ce document et qui passe tous les tests générés.

Nous présentons section 8.4.3 les résultats de l'exécution de cette suite de tests sur les implantations des étudiants.

8.4.2 Suite de tests "force brutale" (*Suite 2*)

La seconde approche consiste à utiliser la capacité du langage de schéma de tests à produire de nombreuses séquences. Afin de ne pas tomber dans le piège de l'explosion combinatoire, nous devons éviter de décrire trop de valeurs de tests. Plutôt que de spécifier à la main tous les états initiaux, nous allons compter sur la combinatoire des appels d'opérations pour les atteindre. Nous espérons, de plus, atteindre des états auxquels nous n'avons pas pensé et ainsi nous permettre d'accroître notre niveau de confiance dans l'application.

Création du schéma

À cette fin, nous avons créé un groupe qui contient plusieurs opérations. Étant donné que nous souhaitons tester des appels d'opérations en séquence, nous avons défini différentes valeurs pour chaque paramètre. Il est à noter que la valeur "e10" du second paramètre de l'opération `swap_two_student` a été choisie pour correspondre à une valeur ajoutée par l'opération `add_student`.

<table>
<tr><th>Groupe</th><th>Opération</th><th colspan="5">Paramètre(s)</th></tr>
<tr><td rowspan="6">Melting</td><td rowspan="4">add_student(s)</td><td rowspan="4">s</td><td colspan="4">"e1"</td></tr>
<tr><td colspan="4">"e10"</td></tr>
<tr><td colspan="4">"e55"</td></tr>
<tr><td colspan="4">"e67"</td></tr>
<tr><td rowspan="2">swap_two_students(s1,s2)</td><td>s1</td><td>"e1"</td><td>s2</td><td>"e3"</td></tr>
<tr><td></td><td></td><td></td><td>"e10"</td></tr>
<tr><td>add_group()</td><td colspan="4"></td></tr>
</table>

Nous avons précédemment créé deux groupes `LTVGr` et `LTIGr` visant à tester les opérations de manière unitaire. Si nous voulons tester les opérations en séquence en évitant le problème de l'explosion combinatoire, il nous faut créer un groupe nous permettant de charger un

nombre plus réduit de tables. Nous choisissons donc quatre états initiaux correspondant à des états qui nous semblent importants car limites. Ainsi, nous avons choisi la table vide, une table avec *3 étudiants*, une table avec *24 étudiants* répartis en *5 groupes* (soit un groupe de quatre et quatre groupes de cinq) et une table comportant *20 étudiants* répartis dans *4 groupes* (soit quatre groupes de cinq étudiants).

Groupe	Opération	Paramètre(s)	
`LTGrosGr`	`load_table(t)`	`t`	`{|->}`
			`{"e1"|->1,"e2"|->1,"e3"|->1}`
			`{"e1"|->1,...,"e20"|->43}`
			`{"e1"|->1,...,"e30"|->43}`

Nous avons défini un schéma nous permettant, une fois que l'une des quatre tables définies dans `LTGrosGr` est chargée, de tester toutes les combinaisons des trois opérations `add_student`, `swap_two_student` et `add_group` du groupe `Melting`. Nous estimons ici que faire plus d'appels ne nous apportera pas plus d'informations et comptons sur la combinatoire pour générer des cas auxquels nous n'aurions pas pensé. Ce schéma permet de générer *1372 séquences*.

```
TestSeqEvent = MTC!gr.LTGrosGr ; MTC!gr.Melting^3..3
```

Exécution des tests et résultats

Le temps d'exécution de la suite de tests, générée à partir du schéma `TestSeqEvent`, est de 41 minutes et 5 secondes.

Elle a permis de détecter les mêmes erreurs que celles trouvées par l'ensemble des tests générés à partir des schémas de la suite 1.

8.4.3 Synthèse

Couverture des tests et détection d'erreurs

L'environnement VDMTools permet, de plus, d'évaluer la couverture de la spécification et du code des opérations par une suite de tests. Cette couverture est définie de la manière suivante :
chaque opération est composée d'expressions situées au niveau des pré- et postconditions et du code de l'opération. La couverture de code d'une opération correspond au nombre d'expressions qui sont évaluées à vrai ou exécutées par la suite de tests, divisé par le nombre total d'expressions définissant l'opération. La couverture totale correspond à la moyenne des couvertures des opérations composant le système sous test.

Nous présentons, table 8.1, un tableau récapitulatif des couvertures et du nombre d'exécutions de chaque opération par les différentes suites de tests, ainsi que le nombre d'opérations erronées détectées par les différentes suites de tests. Nous présentons dans cette table six fonctions (de `max-nb-of-gr` à `table2partition`) utilisées par les pré-et postconditions et dont la spécification est donnée annexe B.1. Nous les donnons ici afin de montrer leur fréquence d'appel, expliquant, entre autre, la lenteur d'exécution de certaines suites de tests.

	Suite de tests manuelle		Suite de tests 1[1]		Suite de tests 2[2]	
nombre de tests	-		128		1372	
Nombre d'implantations erronées trouvées pour les opérations :						
`add_student`	9		13		13	
`add_group`	0		1		1	
	couverture	#appels	couverture	#appels	couverture	#appels
load_table	100%	13	100%	120	100%	795
add_group	100%	2	100%	23	100%	612
add_student	100%	9	100%	30	100%	1383
swap_two_students	100%	4	100%	45	100%	672
max_nb_of_gr	100%	2	100%	23	100%	612
maximum	95%	166	95%	1415	95%	56811
minimum	95%	200	95%	1071	95%	36391
size_max_gr	94%	136	94%	1209	94%	33187
size_min_gr	94%	138	94%	893	94%	23392
table2partition	100%	274	100%	2106	100%	56579
Couverture totale	97,8%		97,8%		97,8%	

TAB. 8.1 – Couverture de code et détection d'erreurs des suites de tests

Les parties non couvertes de certaines fonctions correspondent à du code mort.

Comparaisons avec la suite de test manuelle. Nous pouvons constater que la couverture des opérations est la même avec les trois suites de tests mais les suites de tests générées par Tobias font un plus grand nombre d'appels à chaque fonction. La suite de tests définie à la main donne les mêmes résultats bien qu'elle ne trouve pas autant d'erreurs. Par ailleurs, nous pouvons constater ici les limites, bien connues, de la couverture de code : le fait d'exécuter une fois une instruction, ne garantit pas qu'elle sera correcte dans tous les cas d'exécution.

[1] 128 tests générés à partir des schémas `TestLoadTable`, `TestAddStudent`, `TestAddGroup` et `TestSwapTwoStudents`

[2] 1372 tests générés à partir du schéma `TestSeqEvent`

Nous constatons que les suites de Tobias permettent de détecter plus d'erreurs que la suite de tests manuelle. Parmi ces erreurs, une était connue au moment d'écrire les schémas de tests, mais deux autres (une pour `add_student` et l'erreur dans `add_group`) n'étaient pas connues. Le caractère systématique de la génération permet d'avoir une suite plus complète.

Comparaisons des suites de tests 1 et 2. Nous pouvons voir que ces deux suites de tests permettent d'obtenir le même taux de couverture et qu'elles détectent les mêmes erreurs. La différence se fait donc sur trois niveaux.

1. Le temps de création : la *suite 1* nécessite de définir un nombre plus important de valeurs pour l'état initial, contrairement à la *suite 2* qui ne requiert la définition que d'un nombre réduit de valeurs. En conséquence, la première demande plus d'efforts d'écriture à l'ingénieur.
2. Le temps d'exécution : ici les choses sont inversées. En effet, la *suite 2* met environ 41 minutes à s'exécuter alors que la première ne met que quelques dizaines de secondes. Ce temps d'exécution est dû, tant au nombre de tests, qu'au fait que le code des fonctions, qui interviennent dans la définition des pré et postconditions, n'est pas optimisé comme nous pouvons le constater en regardant le nombre des appels aux diverses fonctions : la première suite de tests entraîne 6717 appels de fonctions, alors que la seconde en totalise 206572, ce qui fait 30 fois plus d'appels de fonctions.
3. Le temps d'analyse des résultats : si l'exécution de 1372 tests prend du temps, le temps passé à les dépouiller est lui considérable car nous ne disposons pas d'un outil d'analyse et de présentation synthétique des résultats. Ce que nous gagnons à la création risque donc d'être perdu au moment de l'analyse des résultats, même si l'utilisation des techniques de filtrage, présentées chapitres 7 et 6, permet de réduire ce coût comme nous le montrons section 8.5.

Analyse des tests *inconclusifs* :

Si les tests générés à partir du schéma `TestSeqEvent` n'ont pas permis de révéler plus d'erreurs que les précédents, ils illustrent néanmoins un point intéressant abordé chapitre 2. Nous pouvons constater ici qu'une même séquence de test peut, pour une implantation donnée, conduire à un verdict *inconclusif* et, pour une autre implantation, à un verdict *pass*. La séquence qui permet cette observation est la suivante :

```
load_table("e1"|->1,"e2"|->1,"e3"|->1,"e4"|->1,"e5"|->1,
           "e6"|->2,"e7"|->2,"e8"|->2,"e9"|->2,"e10"|->2,
           "e11"|->11,"e12"|->11,"e13"|->11,"e14"|->11,"e15"|->11,
           "e36"|->43,"e27"|->43,"e28"|->43,"e29"|->43,"e20"|->43);
add_student("e10");
add_group();
add_group()
```

Dans cette séquence de tests, nous essayons d'ajouter un étudiant déjà présent dans une table comportant quatre groupes de cinq étudiants. Le verdict *inconclusif* résulte, en fait, d'une particularité de l'implantation d'`add_student`. En effet, cette implantation ne tient pas compte du fait que l'étudiant est déjà présent dans la table. Elle essaie de l'ajouter et donc commence par ajouter un nouveau groupe. Dès lors, nous nous retrouvons avec cinq groupes de quatre étudiants. Il n'est donc plus possible d'ajouter deux nouveaux groupes car lorsque nous avons vingt étudiants, répartis en sept groupes, nous obtenons 2,85 étudiants par groupe ce qui violerait l'invariant. Le second appel à l'opération `add_group` viole sa précondition entraînant un verdict *inconclusif*. Il nous faut donc nous interroger afin de déterminer si ce comportement est ou non acceptable du point de vue de la spécification informelle. Dans ce cas particulier, nous pouvons considérer que le verdict *inconclusif* de cette séquence de tests met en évidence le fait que l'opération `add_student` est sous-spécifiée et peut, dans certaines circonstances, modifier le nombre de groupes.

8.5 Filtrage

8.5.1 Filtrage à la génération

Création des contraintes

Nous avons défini chapitre 6 un mécanisme de filtrage à la génération. Ce système permet d'exprimer des contraintes sous la forme d'opérations VDM retournant un booléen. Ces opérations renvoient la valeur vrai lorsque la contrainte est satisfaite et faux dans le cas contraire. Pour chaque test généré, nous vérifions si la contrainte est satisfaite. Si tel est le cas, le test est conservé sinon il est supprimé.

Le schéma `TestSeqEvent` génère *1372 séquences* de test. Parmi toutes ces séquences, un certain nombre nous semblent inutiles comme par exemple :

seq 1 = `load_table({"e1"|->1, "e2"|->1, "e3"|->1}); add_student("e1");`
`add_student("e1"); add_student("e1")`
seq 2 = `load_table({"e1"|->1, "e2"|->1, "e3"|->1});`
`swap_two_student("e1","e10") ...`

La première de ces deux séquences ne nous intéresse pas car ajouter trois fois le même étudiant est, selon nous, équivalent à l'ajout d'un étudiant déjà présent dans la table. Nous faisons donc l'hypothèse de régularité suivante : ajouter un étudiant déjà présent est équivalent au fait d'ajouter plusieurs fois le même étudiant.

La deuxième séquence va nécessairement conduire à un verdict *inconclusif* puisqu'elle viole la précondition de l'opération `TestSwapTwoStudents`.

Nous avons défini une opération VDM renvoyant un booléen qui nous permet d'éliminer une grande partie de ces tests. Pour ce faire, nous créons des variables et les lions aux

paramètres des opérations entrant en jeu dans ces tests. Ainsi, :

- `tableIn` est la variable reliée au paramètre de `load_table`,
- `studentAdd` est la variable reliée au paramètre d'`add_student`,
- `studentDep` et `studentArr` sont les variables liées respectivement au premier et second paramètre de l'opération `swap_two_students`.

Ces variables représentent des séquences d'éléments. Ainsi `tableIn` est une séquence de tables. Par exemple, si nous reprenons les deux séquences présentées ci-dessus, nous aurons :

1. pour *seq 1* :
   - `tableIn = [{"e1"|->1, "e2"|->1, "e3"|->1}]`
   - `studentAdd = ["e1", "e1", "e1"]`
   - `studentDep = []`
   - `studentArr = []`
2. pour *seq 2* :
   - `tableIn = [{"e1"|->1, "e2"|->1, "e3"|->1}]`
   - `studentAdd = []`
   - `studentDep = ["e1"]`
   - `studentArr = ["e10"]`

   L'opération est la suivante :

```
contrainte1: () ==> bool
contrainte1 == (
dcl setIn : set of student:={};
for all t in set elems(tableIn) do
      (setIn := setIn union dom t;);
return (
   card(elems(studentAdd)) = len(studentAdd) and
   elems(studentArr^studentDep) subset (setIn union elems(studentAdd))
   ))
```

La variable `setIn` permet de récupérer tous les étudiants présents dans les différents groupes sous la forme d'un ensemble afin de pouvoir exprimer nos contraintes.

- `card(elems(studentAdd)) = len(studentAdd)` garantit que les étudiants ajoutés sont tous différents : `elems` prend une séquence d'éléments en entrée et renvoie l'ensemble contenant tous les éléments contenus dans la séquence (si un élément est présent plusieurs fois dans la séquence, il ne sera présent qu'une seule fois dans l'ensemble) ; `len` renvoie la longueur de la séquence. Cela nous permet d'éliminer les cas où l'on essaie d'ajouter deux fois le même étudiant. Les tests correspondant à *seq 1* sont ainsi supprimés.

- `elems(studentArr^studentDep) subset (setIn union elems(studentAdd))` permet de supprimer des cas *inconclusifs* puisqu'il s'agit d'enlever toutes les tentatives de permutation lorsque l'un des deux étudiants n'est jamais présent dans la table au cours du test. En effet, nous concaténons les séquences `studentArr` et `studentDep`,

construisons l'ensemble des étudiants qui sont présents dans la table au cours du test `(setIn union elems(studentAdd))` et nous vérifions que l'ensemble des étudiants que l'on va permuter
`elems(studentArr^studentDep)` est bien un sous-ensemble de l'ensemble des étudiants présents dans la table.

Comme nous ne prenons pas en compte l'ordre des appels dans la contrainte, les séquences de test de la forme

seq 3 = `load_table({"e1"|->1, "e2"|->1, "e3"|->1});`
`swap_two_students("e1","e10"); add_student("e10") ...`

ne sont pas rejetées alors que la spécification de `swap_two_students` est violée.

Nous avons montré au chapitre 6 que de telles séquences pouvaient être prises en compte en extrayant une information plus détaillée, mais cela se fait au détriment de l'expression de la contrainte. Pour ce faire, les variables que nous avons utilisées doivent être définies sous la forme de fonctions ayant pour domaine l'indice d'appel et pour codomaine la valeur de l'appel, et non plus sous la forme de séquences. Nous aurons alors :

1. pour *seq 1* :
   - `tableIn = {1|->{"e1"|->1, "e2"|->1, "e3"|->1}}`
   - `studentAdd = {2|->"e1", 3|->"e1", 4|->"e1"}`
   - `studentDep = {}`
   - `studentArr = {}`
2. pour *seq 2* :
   - `tableIn = {1|->{"e1"|->1, "e2"|->1, "e3"|->1}}`
   - `studentAdd = {}`
   - `studentDep = {2|->"e1"}`
   - `studentArr = {2|->"e10"}`
3. pour *seq 3* :
   - `tableIn = {1|->{"e1"|->1, "e2"|->1, "e3"|->1}}`
   - `studentAdd = {3|->"e10"}`
   - `studentDep = {2|->"e1"}`
   - `studentArr = {2|->"e10"}`

Si nous écrivons la contrainte :

```
1  contrainte2: () ==> bool
2  contrainte2 == (
3   return (
4    (forall i in set dom studentArr &
5      exists j in set 1,...,i-1 &
6        (j in set dom studentAdd
7         and studentAdd(j) = studentArr(i)
8        ) or (
9         j in set dom tableIn
10         and studentArr(i) in set dom tableIn(j)
```

```
11        )
12    ) and (
13    forall i in set dom studentDep &
14     exists j in set 1,...,i-1 &
15       (j in set dom studentAdd
16        and studentAdd(j) = studentDep(i)
17       ) or (
18        j in set dom tableIn
19        and studentDep(i) in set dom tableIn(j)
20       )
21   )
22  )
```

Nous voyons que `contrainte2` est beaucoup plus complexe que `contrainte1`. Pour chaque étudiant d'arrivée de l'opération `swap_two_students` (ligne 4), nous vérifions qu'il existe un indice antérieur (ligne 5) tel que, l'étudiant d'indice `i` dans `studentArr` ait été précédemment ajouté (lignes 6 et 7) par l'opération `add_student` ou qu'il ait été chargé dans une table (lignes 7 et 8) par l'opération `load_table`. Les lignes 13 à 19 font la même chose avec l'étudiant de départ de l'opération `swap_two_students`.

Exécution des contraintes

Nous avons généré les tests à partir du schéma `TestSeqEvent`. Le temps d'évaluation de la contrainte VDM définie sur ce schéma est de *1 minutes et 11 secondes*, elle permet d'éliminer *574 séquences* de tests ce qui nous donne une nouvelle suite de tests appelée *suite 3*.

Lorsque l'on étudie les erreurs levées par cette suite de tests (voir table 8.2), nous pouvons constater que ce sont les mêmes que celles levées par la suite initiale. Le filtrage n'a donc pas diminué la qualité de la suite de tests nous confortant ainsi dans l'idée que nous avons fait de bonnes hypothèses de test.

8.5.2 Filtrage à l'exécution

Tous les tests ont été exécutés à l'aide du pilote de tests optimisé. Son utilisation a permis d'éviter l'exécution de nombreux tests facilitant ainsi leur dépouillement et améliorant le temps d'exécution. Néanmoins, dans certains cas, les résultats sont moins significatifs que sur l'étude de cas du gestionnaire de processus.

Nous présentons dans la table 8.3 les résultats obtenus sur l'exécution des tests générés à partir du schéma `TestSeqEvent` avec et sans filtrage à la génération. Ici, le bénéfice de l'optimisation à l'exécution est moins flagrant que précédemment. Le temps d'exécution de certaines séquences étant relativement long, le temps que l'on gagne en évitant d'en jouer d'autres, souvent moins longues à s'exécuter, fait que le gain de temps devient négligeable

[1] 798 tests générés à partir du schéma `TestSeqEvent` filtré

	Suite de tests 3 [1]	
Nombre d'implantations erronées trouvées pour les opérations :		
`add_student`	13	
`add_group`	1	
	couverture	#appels
load-table	100%	594
add-group	100%	486
add-student	100%	887
swap-two-students	100%	591
max-nb-of-gr	100%	486
maximum	95%	44757
minimum	95%	27695
size-max-gr	94%	25460
size-min-gr	94%	17532
table2partition	100%	42992
Couverture totale	97,8%	

TAB. 8.2 – Couverture de code et détection d'erreurs de la *suite 3*

par rapport au temps global d'exécution. En revanche, le fait d'avoir filtré les tests à la génération a permis de réduire le temps global d'exécution de la suite de tests (temps de génération et d'exécution).

Suite de tests	Nb de Tests	Optimisation à l'exécution	Temps total d'exécution	Nb de tests non joués
`TestSeqEvent`	1372	aucune	$\sim$ 41 min 5 s	0
		complète	$\sim$ 35 min 30s	576
`TestSeqEvent` filtré	798	aucune	$\sim$ 31 min	0
		complète	$\sim$ 27 min	203

TAB. 8.3 – Temps d'exécution et nombre de tests éliminés à l'aide des divers pilotes de test.

8.6 Conclusions

Les expérimentations, que nous avons menées autour de l'étude de cas des étudiants, montrent clairement les avantages de l'utilisation de Tobias par rapport à une écriture manuelle des tests. Nous avons à notre disposition 14 implantations différentes. La suite de

tests écrite à la main a montré sa capacité à trouver des erreurs puisqu'elle a détecté *9 implantations* erronées de l'opération `add_student` sur les *14 implantations*. Les tests générés avec Tobias ont, quant à eux, permis de trouver *13 implantations* erronées de cette même opération ainsi qu'une version erronée de l'opération `add_group`. Le caractère systématique de cette approche facilite grandement le travail de l'ingénieur en lui assurant que tous les cas seront couverts. Ce faisant, il ne lui reste plus qu'à se consacrer aux aspects créatifs que sont le choix des valeurs et la création des schémas. L'outil, à travers son interface, se veut facile d'utilisation pour permettre à l'utilisateur de créer et de rectifier rapidement le contenu des groupes et des schémas.

L'utilisation du filtrage à la génération a permis de réduire de manière significative le nombre de tests sans diminuer la qualité de la suite de tests. Le temps de filtrage s'est par ailleurs révélé assez faible. En revanche, l'utilisation des pilotes de tests permet des gains de temps supplémentaires. Sur cette étude de cas, le gain de temps à l'exécution est nettement moins intéressant que sur l'étude de cas du gestionnaire de processus (chapitre 7, section 7.4). Il semblerait qu'il y ait un noyau dur de tests utiles qui prennent beaucoup de temps à s'exécuter. En effet, alors que le filtrage à l'aide de contraintes réduit de presque 50% le nombre de tests, le temps d'exécution, lui, ne diminue que d'environ 25%. En revanche, le fait de limiter le nombre de tests joués facilite l'analyse des erreurs.

À l'issue de ces tests, nous pouvons observer que, compte tenu des éléments mis à notre disposition, les tests de la suite 1 permettent d'obtenir les mêmes résultats que les tests des suites 2 et 3. Ceci nous montre qu'il est possible d'utiliser l'outil de différentes façons afin d'en tirer le meilleur parti.

1. L'écriture de la suite de tests 1 a demandé plus d'efforts car il a fallu définir un plus grand nombre de valeurs pour les diverses opérations. Le temps passé à écrire cette suite s'avère supérieur au temps passé à écrire la suite manuelle au regard du nombre de lignes qu'il a fallu écrire : *26 lignes* pour la suite manuelle contre *52 lignes* pour la suite 1. Néanmoins, l'écriture de schémas relativement simples a permis de ne générer qu'un nombre assez restreint de tests et permet donc de maîtriser complètement l'explosion combinatoire.
2. L'élaboration de la *suite 2* a été plus rapide car nous avons compté sur la combinatoire de l'outil pour atteindre certains états sensibles. En termes de nombre de lignes, elle correspond à l'écriture de *12 lignes* dans Tobias contre *26 lignes* pour la suite manuelle et *52 lignes* pour la *suite 1*. Le nombre de tests générés est bien plus important et leur exécution beaucoup plus longue. Néanmoins, nous augmentons notre confiance dans le code en exécutant un plus grand nombre de fois les opérations et fonctions du système sous test.
3. La sélection des tests à la génération a permis de réduire le nombre de tests nous donnant la *suite 3*. Nous avons pu constater un gain de temps à l'exécution par rapport à la *suite 2* mais il faut néanmoins tenir compte des temps d'écriture de la contrainte et d'exécution de celle-ci et du risque d'erreur dans l'écriture des contraintes.

Chapitre 9

Conclusions et perspectives

Conclusions

Dans cette thèse, nous avons abordé le problème de la génération de tests combinatoire pour le test de conformité. Nous avons rappelé les limites des approches combinatoires. Néanmoins, nous pensons que si nous pouvons maîtriser les problèmes liés à ces techniques, nous pouvons offrir à l'utilisateur un moyen simple et efficace pour produire ses tests. Cette thèse a proposé plusieurs techniques pour maîtriser ces problèmes et les a mises en œuvre sur diverses études de cas.

Langage de synthèse de tests

Nous avons défini un langage de synthèse de tests (chapitre 4) permettant de décrire de manière synthétique un ensemble de tests sous la forme d'un schéma de tests. Les schémas de tests sont dépliés de manière combinatoire et permettent de produire des tests abstraits. Ces tests abstraits peuvent ensuite être traduits en tests exécutables. Dans cette thèse, nous avons ainsi produit des tests exécutables pour les plates-formes Java et VDM.

Le langage de synthèse de tests a été mis en œuvre dans l'outil Tobias. Cet outil nous a permis de générer des tests pour diverses études de cas.

Les deux principales études de cas (chapitres 5 et 8), que nous présentons dans cette thèse, montrent que l'approche combinatoire par schémas de tests permet de trouver des erreurs. Nous n'avons cependant pas pu comparer la capacité de détection d'erreur de notre approche avec des outils générant automatiquement des séquences de test à partir de la spécification. Néanmoins, l'étude de cas Gemplus a montré que certaines erreurs sont hors de la portée de la spécification : elles n'auraient pas été détectées par de tels outils, alors que Tobias a permis de les mettre en évidence.

Une troisième étude de cas [DCdBBL05] présente le test de l'application multimodale ICARE à l'aide de l'outil Tobias. Le test de cette application Java a été réalisé par une équipe de recherche extérieure au laboratoire LSR. L'ingénieur en charge du test a écrit 19 schémas en une semaine qui ont permis de générer 6000 cas de tests. Cette étude a permis

de montrer que notre approche avait un sens : simplicité de prise en main et efficacité pour trouver des erreurs.

Tobias base sa génération de tests sur la compétence de l'ingénieur de test, qui tire de son expérience et de la connaissance qu'il a du programme à tester les séquences de test les plus pertinentes. Les schémas de tests ne sont là que pour amplifier sa créativité, en produisant des tests auxquels il n'aurait pas pensé ou en l'incitant à en produire un nombre plus important. De plus, Tobias facilite la gestion des tests, en permettant de revenir simplement sur les valeurs choisies sans risquer de commettre des erreurs (oubli de certains cas de tests, valeurs mal reportées...) et en structurant les tests.

Nous avons montré, à travers ces études de cas, qu'il était possible, dès la définition des schémas de tests, de contrôler l'explosion combinatoire du nombre de tests.

Cependant, exercer un tel contrôle au moment de l'écriture des schémas de tests peut s'avérer très coûteux en temps, diminuant l'intérêt d'un tel système. Aussi nous proposons des méthodes complémentaires visant à encore mieux contrôler le phénomène de l'explosion combinatoire. Les méthodes que nous proposons s'attaquent à ce problème suivant deux angles. En premier lieu, nous proposons différentes méthodes permettant de réduire le nombre de tests produits (filtrage à la génération). Ensuite, nous proposons de réduire le temps d'exécution des suites de tests (filtrage à l'exécution).

Filtrage à la génération

Nous avons étudié dans cette thèse trois axes de filtrage à la génération :

1. sélection d'un sous-ensemble de tests à l'aide de contraintes exprimées sur les schémas de tests ;
2. définition de nouveaux itérateurs au niveau du langage de synthèse de tests permettant d'effectuer des itérations ordonnées ;
3. utilisation de la programmation logique avec contraintes afin d'exploiter la spécification lorsqu'elle est disponible.

Sur les trois pistes explorées, seules les pistes 1 et 3 ont été mises en œuvre dans l'outil Tobias. Nous avons pu constater les bénéfices apportés par ces deux approches, bien que l'utilisation de contraintes sur les schémas ne soit pas sans risques. En effet, l'utilisateur peut très bien éliminer des tests utiles contrairement à la troisième approche qui se base sur la spécification et n'élimine que les tests qui la violent.

Filtrage à l'exécution

Nous avons ensuite défini des pilotes de test permettant de réduire le nombre de tests exécutés. Nous avons montré que l'utilisation de ces pilotes permettait de réduire, sans risques, le temps d'exécution lorsque les applications ont un comportement déterministe. De plus, leur utilisation permet de réduire, parfois de manière très significative, le temps d'exécution des suites de tests.

Perspectives

Nous pensons qu'il serait intéressant de poursuivre ce travail suivant trois axes. D'une part, nous pourrions combler les faiblesses du langage que nous proposons dans ce document. D'autre part, nous pourrions offrir à l'utilisateur une autre manière de maîtriser l'explosion combinatoire ce qui complèterait les techniques déjà proposées. Enfin, l'utilisation de notre outil pour tester des logiciels de plus grande taille permettrait d'évaluer concrètement si les objectifs que nous nous étions fixés sont atteints, à savoir, fournir à l'utilisateur un canevas logiciel simple et flexible permettant d'amplifier sa créativité, de structurer ses tests et disposant d'un ensemble de techniques et outils permettant de maîtriser l'explosion combinatoire du nombre de tests.

Évolution du langage

Les différentes études de cas qui ont été effectuées ont permis d'identifier des pistes pour l'amélioration du langage de synthèse de tests.

Introduction de variables

Nous n'avons pas de notion de variable. Nous savons que ceci est un défaut du langage de synthèse de tests. En effet, il est important de pouvoir réutiliser dans d'autres appels, les valeurs retournées par des méthodes précédemment exécutées. Il est aujourd'hui possible de contourner le problème en définissant, spécifiquement pour le test, des méthodes "set" et "get". L'appel de méthode, dont on souhaite réutiliser la valeur de retour, est alors passé en paramètre de la méthode "set". Puis, nous utilisons la méthode "get" correspondante comme paramètre de la méthode qui utilise la valeur de retour stockée. Il est évident que ce n'est pas la bonne solution pour ce problème. Il est donc important de bien réfléchir à la manière d'intégrer au langage de synthèse de tests les notions de variable et d'affectation. Ainsi, nous devons nous demander à quel niveau nous devons définir les variables. Nous envisageons principalement deux possibilités :

1. les variables sont définies à l'intérieur des groupes, uniquement pour les méthodes qui nous intéressent. Par exemple, soit le groupe A composé des méthodes `v :=foo` et `foo2`; soit la variable v la valeur de retour de la méthode `foo`; soit le groupe B composé des méthodes `bar(v)` et `bar2` avec `bar` qui fait appel à la variable v. Nous pouvons ensuite définir le schéma $S1 = A; B$ qui permettra de générer, entre autres, la séquence `v :=foo() ; bar(v)`.
2. nous considérons les groupes comme des sortes de méta-méthodes. Ceux-ci ont alors des paramètres et peuvent retourner des valeurs. Nous définissons alors les variables à l'intérieur des schémas. Par exemple, A est un groupe retournant une valeur et B est un groupe avec un paramètre, utilisé par les méthodes définies dans B. Nous pouvons alors écrire le schéma suivant $S2 = v := A; B(v)$.

Deux problèmes se présentent à nous :

– lorsque nous allons faire la combinatoire des appels de méthode de ces groupes, nous pouvons générer une séquence faisant appel à `foo2` suivi d'un appel à `bar` qui utilise alors une variable non définie. Ce type de séquence ne doit pas être généré car elle n'a pas de sens.
– par ailleurs, nous devons nous assurer que les séquences sont correctement typées (le type de la valeur de retour est le même que celui du paramètre, en tenant compte des éventuelles relations d'héritage etc.).

Ces problèmes peuvent être résolus en exploitant le caractère séquentiel des cas de test générés. Il est en effet facile d'identifier quelles variables sont utilisées et si elles ont été correctement typées et initialisées.

Réutilisation de groupes ou de schémas

Le langage de synthèse de tests manque de flexibilité au niveau structurel. En effet, nous ne pouvons ni composer des groupes entre eux ni même des schémas.

– Si nous avons déjà défini, dans un groupe, un ensemble de méthodes avec leurs valeurs de paramètres et que nous souhaitons définir un nouveau groupe reprenant certaines de ces méthodes, l'utilisateur va devoir les réécrire. En offrant la possibilité d'agréger les groupes entre eux, l'utilisateur peut définir dans un premier temps de groupes plus petits ne contenant qu'une ou deux méthodes par exemple. Puis, il peut réutiliser ces groupes dans des groupes plus importants. Cette notion de composition de groupes permet à l'utilisateur de mettre plus facilement en œuvre une méthodologie "bottom up" de conception de schémas. Il peut ainsi plus facilement partir de séquences de test typiques pour définir des groupes simples, qu'il peut ensuite agréger pour créer des schémas générant plus de tests.
– Le fait de pouvoir composer des schémas devient important si l'on souhaite pouvoir définir des préfixes permettant de positionner le système dans un état donné, ou des suffixes permettant de remettre le système dans un état stable permettant d'enchaîner un autre test. L'utilisateur ne peut pas, aujourd'hui, définir un schéma générant quelques séquences permettant de positionner le système dans différents états, puis composer ce schéma avec d'autres produisant des tests qui partent de ces états. Il doit définir un suffixe dans chaque schéma, ce qui est un perte de temps et de flexibilité : une modification du suffixe nécessite de modifier tous les schémas au lieu d'un seul.

Nous voyons ici qu'il est important de permettre plus de flexibilité au niveau structurel.

Mise en œuvre d'un nouvel algorithme de génération

Nous avons proposé dans ce document différentes manières de maîtriser l'explosion combinatoire. Il en reste néanmoins une qui nous semble intéressante à mettre en œuvre et à évaluer : l'utilisation d'algorithmes effectuant une combinatoire partielle des arguments. Ces algorithmes, mis en œuvre dans l'outil AETG section 2.3.2, permettent de ne pas faire toutes les combinaisons d'arguments mais toutes les combinaisons de n arguments. Il est ainsi possible de produire toutes les combinaisons deux à deux ou trois à trois. Ceci permet, lorsque

les méthodes ont beaucoup d'arguments, de réduire significativement le nombre d'instanciations produites. Il est important de noter qu'AETG produit des cas de tests comportant un seul appel de méthode alors que Tobias lui produit des séquences d'appels. Il serait donc intéressant de voir comment nous pouvons combiner les principes d'AETG avec ceux de Tobias. En effet, l'hypothèse (forte) faite par AETG est qu'une faute dans un programme résulte d'une combinaison de quelques paramètres. Il n'est donc pas nécessaire d'effectuer la combinatoire de toutes les valeurs de paramètre pour trouver toutes les fautes.

Les algorithmes d'AETG peuvent s'appliquer à plusieurs niveaux dans notre outil : nous pouvons l'appliquer localement lors du dépliage des groupes en évitant pour chacune des méthodes de faire la combinatoire complète de toutes les valeurs spécifiées ; nous pouvons aussi l'appliquer en considérant qu'une séquence peut être vue comme une seule méthode, par exemple, la séquence suivante `foo(a,b,c) ;bar(d,e)` serait vue comme un appel de méthode comportant cinq paramètres `M(a, b, c, d, e)` sur lequel il est facile d'utiliser AETG. Il faut aussi étudier dans quelle mesure l'hypothèse d'AETG peut être effectuée quand les paramètres apparaissent dans plusieurs appels en séquence.

Nouvelles études de cas

Les études de cas présentées dans cette thèse avaient toute une taille assez réduite en terme de nombre de lignes de code et de nombre de méthodes à tester. Deux axes sont à suivre ici.

- D'une part, il serait intéressant, une fois les modifications du langage de synthèse de tests faites, d'utiliser notre outil sur différents projets de tailles plus importantes. En effet, nous devons vérifier si le passage en vraie grandeur va rendre l'explosion combinatoire plus difficile à maîtriser ou si cela va finalement plus dépendre des applications testées que de leur taille.
- D'autre part, il serait intéressant d'utiliser, sur un même projet, différents outils de génération de tests, dont Tobias afin de pouvoir comparer les forces et faiblesses de notre approche par rapport aux approches plus automatiques (test aléatoire, génération de tests à partir de spécifications ou à partir du code).

Les travaux présentés dans cette thèse définissent une méthode de synthèse de tests basée sur un langage de synthèse de tests simple et sur des techniques permettant de contrôler l'explosion combinatoire. Les propositions, qui ont été faites, ont été mises en œuvre dans l'outil Tobias. Les études de cas réalisées avec cet outil ont permis de montrer l'utilité de notre approche. Elles ont, de plus, ouvert de nouvelles pistes de recherche visant à améliorer notre outil.

Annexe A

Tobias

Tobias est un outil qui sert d'interface pour écrire des schémas de test et qui génère des objectifs de test ou des séquences de tests correspondant à ces schémas de test. Tobias prend en compte le diagramme de classe d'une spécification UML pour récupérer les classes et les méthodes asociées à partir desquelles peuvent être écrits les schémas de test.

Tobias est un outil que nous avons développé en Java. Nous avons essayé de mettre l'accent sur la modularité du logiciel afin de permettre d'ajouter facilement des extensions à l'outil et de permettre une programmation plus propre. C'est la raison pour laquelle nous avons défini quatre packages A.1.

A.1 Données quantitatives

Tobias représente 14450 lignes de code réparties en 77 classes. Chaque package correspond à une partie bien précise des fonctionnalités offertes par Tobias. La classe IHM permet de gérer l'interface entre l'utilisateur et Tobias. Cette classe est très grosse (1660 lignes de code) car elle doit permettre l'accès à l'ensemble des fonctionnalités de Tobias ainsi que l'affichage de Tobias, elle se trouve liée à tous les packages de Tobias que nous décrivons ici.

- **TTYPE** (18 classes) est le package qui définit tous les types de base spécifiques à Tobias et aux schémas de test. Cela comprend des représentations des instances, des classes, des méthodes, des paramètres des méthodes, des groupes, des campagnes de test ainsi que des schémas de test et objectifs de test. La hiérarchisation des différents types est représentée dans la figure A.2 par un diagramme de classes UML.
- **CORE** (11 classes) rassemble les sous-programmes qui permettent les différentes transformations comme la génération des séquences de test à partir des schémas de test. Il gère les routines de sauvegarde et chargement en relation avec le module PAJAX, ou la transformation des séquences d'appel vers les formats JML, aldébaran et VDM.
- **INTERACTION** (27 classes) constitue le package associé à l'interface de Tobias, il comporte toutes les fenêtres et panels de Tobias. C'est dans ce package, entre autres, qu'est gérée l'arborescence des campagnes de test, groupes, schémas de test. Il fournit

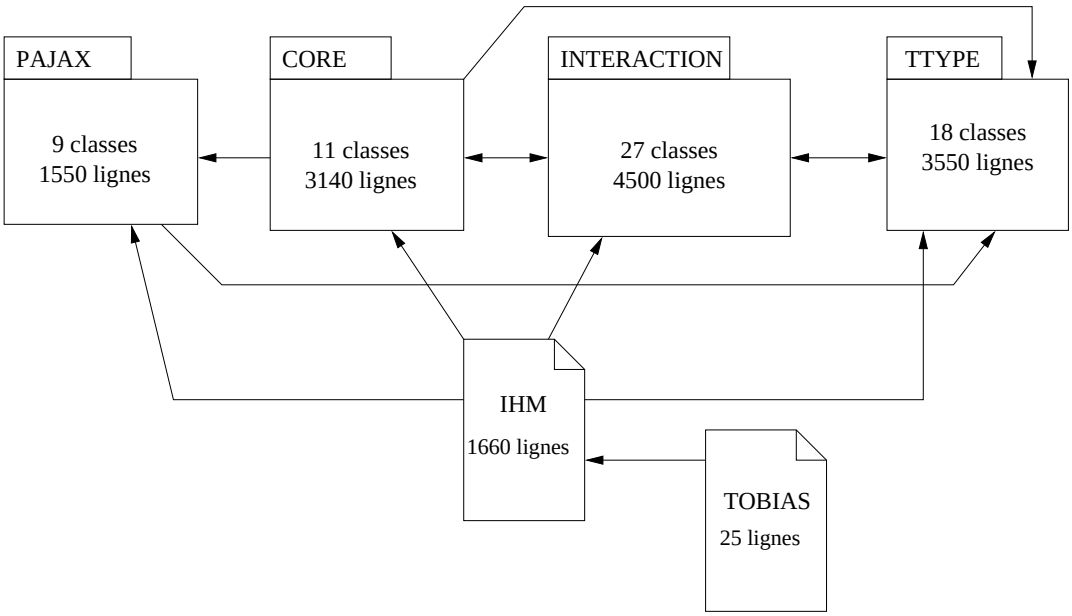


FIG. A.1 – architecture de Tobias sous forme de packages

aussi l'interface de toutes les commandes possibles dans Tobias, que ce soit pour gérer les sauvegardes et chargements, ou bien la création des groupes par exemple.

– **PAJAX** (9 classes), pour **PArseur JAva Xml**, est un module qui a été intégré à Tobias pour permettre la lecture ainsi que l'écriture de fichiers XML. A la base, PAJAX est indépendant de Tobias. Nous avons introduit une classe qui permet de traiter les informations extraites par PAJAX pour les mettre au format géré par Tobias. Une autre classe permet la transformation dans l'autre sens. Pour la sauvegarde de l'état interne d'un projet dans Tobias nous avons choisi un format XML afin de réutiliser le module PAJAX qui est à notre disposition.

A.2 Création de schémas de test dans Tobias

Tobias permet de structurer les schémas de test produits pour une application donnée sous la forme de campagnes de test. Pour chaque campagne de test nous pouvons définir une configuration différente de l'application en terme d'instances à déployer.

Nous définissons des **groupes**, soit de manière globale pour l'application, soit de manière localisée dans une campagne de test. Dans un groupe nous précisons les méthodes qui vont en faire partie et pour chaque paramètre de chaque méthode nous définissons la liste des valeurs qu'il peut prendre.

L'interface de Tobias permet d'exprimer les abstractions sur les méthodes à partir des groupes et sur les paramètres dans chaque campagne de test que le testeur peut souhaiter définir. A partir de ces informations sur les groupes, le testeur peut écrire dans Tobias des

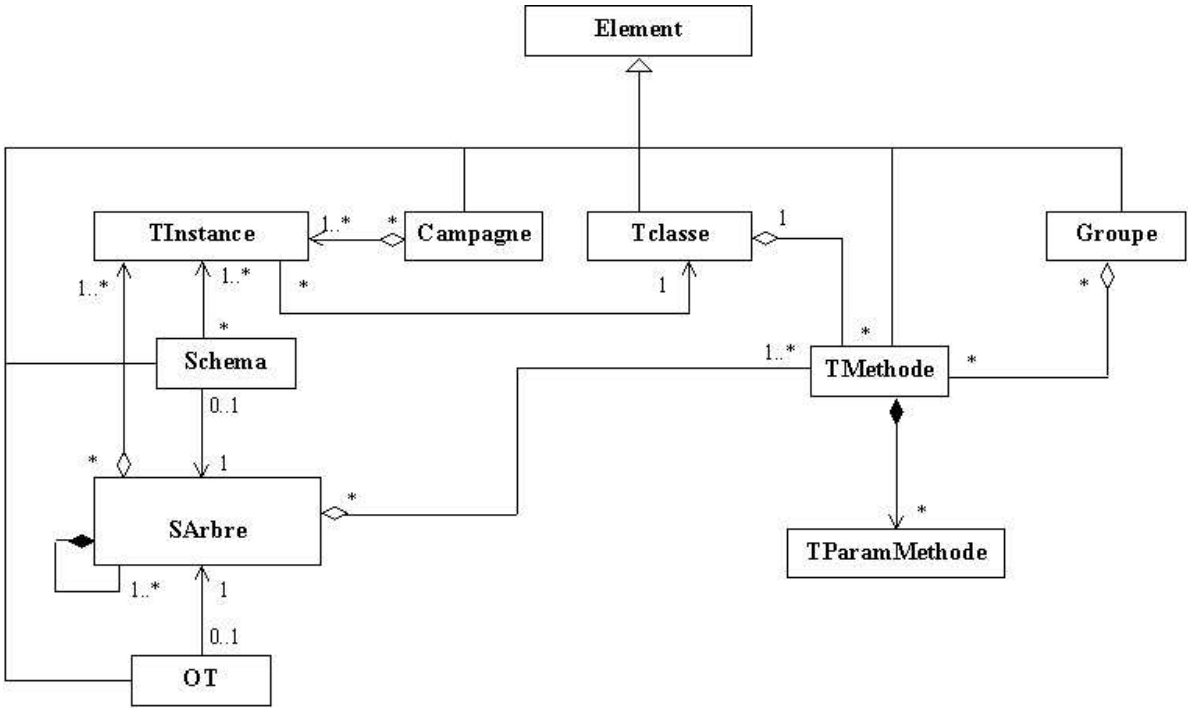


FIG. A.2 – modèle UML de la hiérarchisation des types dans Tobias

schémas de test correspondant à la grammaire définie dans le tableau 4.1 section 4.2 avec **Schéma** comme axiome de départ.

Annexe B

Étude de cas VDM : compléments

B.1 Spécifications

B.1.1 size_min_gr

Cette fonction renvoie la taille du groupe le plus petit. La fonction `minimum` prend en paramètre un ensemble d'entiers et renvoie le plus petit entier de cet ensemble.

```
1   minimum : set of int +> int
    minimum(e) ==
    let x in set e in
            if e = {x} then x
5           else let min = minimum(e\{x}) in
                    if x < min then x else min
    pre e <> {}
    ;

10  size_min_gr : table +> nat
    size_min_gr(t) ==
    let sizes_set = {card s | s in set table2partition(t)} in
            if sizes_set <> {} then minimum(sizes_set) else 0
    post forall i in set rng t & card dom (t :> {i}) >= RESULT
15          and
         (t <> {|->} => exists i in set rng t & card dom (t:> {i}) = RESULT)
            and
         (t = {|->} => RESULT = 0)
    ;
```

B.1.2 size_max_gr

Cette fonction renvoie la taille du groupe le plus grand. La fonction `maximum` prend en paramètre un ensemble d'entiers et renvoie le plus grand entier de cet ensemble.

```
1   maximum : set of int +> int
    maximum(e) ==
    let x in set e in
```

```
        if e = {x} then x
        else let max = maximum(e\{x}) in
                if x > max then x else max
pre e <> {}
;

size_max_gr : table +> nat
size_max_gr(t) ==
let sizes_set = {card s | s in set table2partition(t)} in
        if sizes_set <> {} then maximum(sizes_set) else 0
post forall i in set rng t & card dom (t :> {i}) <= RESULT
        and
     (t <> {|->} => exists i in set rng t & card dom (t:> {i}) = RESULT)
        and
     (t = {|->} => RESULT = 0)
;
```

B.1.3 max_nb_of_gr

Cette fonction renvoie le nombre maximum de groupes qu'il est possible d'avoir à un moment donné, en fonction de la taille minimale exprimée par la constante `limit.min`.

```
max_nb_of_gr : table +> nat
max_nb_of_gr(t) ==
 (card dom t) div limit.min
;
```

B.1.4 table2partition

Cette fonction prend en paramètre une table d'étudiants et renvoie un ensemble d'ensembles d'étudiants où tous les étudiants appartenant à un même ensemble étaient dans le même groupe.

```
table2partition : table +> partition
table2partition(t) ==

{dom (t :> {gid}) | gid in set rng t}
post dom t = dunion RESULT
        and
        forall s in set RESULT & (s <> {} and let e in set s in s = dom (t :> {t(e)}) )
;
```

B.2 Groupes pour Tobias

B.2.1 Groupe *LTIGr*

$LTIGr = \{load_table(t)\}$ avec $t = \{$

1. {|->},

2. {"e1"|->a},
3. {"e1"|->0},
4. {"e1"|->1},
5. {"e1"|->1246312},
6. {"e1"|->1, "e1"|->1},
7. {"e1"|->1, "e2"|->1},
8. {"e1"|->1, "e2"|->1, "e3"|->1},
9. {"e1"|->1, "e2"|->2, "e3"|->1},
10. {"e1"|->44, "e2"|->25, "e3"|->2},
11. {"e1"|->1, "e2"|->1, "e3"|->1, "e4"|->1},
12. {"e1"|->1, "e2"|->1, "e4"|->1, "e3"|->1, "e5"|->1},
13. {"e1"|->1, "e2"|->1, "e4"|->1, "e3"|->1, "e5"|->2},
14. {"e1"|->1, "e2"|->1, "e4"|->1, "e3"|->1, "e5"|->1, "e6"|->2},
15. {"e1"|->1, "e2"|->1, "e3"|->1, "e6"|->2, "e7"|->2, "e8"|->2},
16. {"e1"|->1, "e2"|->1, "e4"|->1, "e7"|->1, "e3"|->2, "e5"|->2,
    "e6"|->2, "e8"|->2},
17. {"e1"|->1, "e2"|->1, "e4"|->1, "e7"|->1, "e8"|->1, "e3"|->2,
    "e5"|->2, "e6"|->2, "e9"|->2},
18. {"e1"|->1, "e2"|->1, "e4"|->1, "e7"|->1, "e8"|->2, "e3"|->2,
    "e5"|->2, "e6"|->2, "e9"|->2},
19. {"e1"|->1, "e2"|->1, "e3"|->3, "e4"|->1, "e6"|->2, "e7"|->2,
    "e8"|->2, "e9"|->3, "e0"|->3},
20. {"e1"|->1, "e2"|->1, "e4"|->1, "e7"|->1, "e8"|->1, "e3"|->2,
    "e5"|->2, "e6"|->2, "e9"|->2, "e10"|->2},
21. {"e1"|->1, "e2"|->1, "e4"|->1, "e7"|->2, "e8"|->2, "e3"|->2,
    "e5"|->3, "e6"|->3, "e9"|->3, "e10"|->3},
22. {"e1"|->1, "e2"|->1, "e4"|->1, "e7"|->1, "e8"|->2, "e3"|->2,
    "e5"|->2, "e6"|->2, "e9"|->3, "e10"|->3, "e11"|->3},
23. {"e1"|->1, "e2"|->1, "e4"|->1, "e7"|->3, "e8"|->2, "e3"|->2,
    "e5"|->2, "e6"|->3, "e9"|->3, "e10"|->3, "e11"|->3},
24. {"e1"|->1, "e2"|->1, "e3"|->1, "e4"|->1, "e5"|->1, "e6"|->2,
    "e7"|->2, "e8"|->2, "e9"|->2, "e10"|->2, "e11"|->11, "e12"|->11,
    "e13"|->11, "e14"|->11, "e15"|->11},
25. {"e1"|->1, "e2"|->1, "e3"|->1, "e4"|->1, "e5"|->1, "e6"|->2,
    "e7"|->2, "e8"|->2, "e9"|->2, "e10"|->2, "e11"|->11, "e12"|->11,
    "e13"|->11, "e14"|->11, "e15"|->11, "e16"|->12, "e17"|->12, "e18"|->12,
    "e19"|->12, "e36"|->43, "e27"|->43, "e28"|->43, "e29"|->43, "e20"|->43},

26. {"e1"|->1, "e2"|->1, "e3"|->1, "e4"|->1, "e5"|->1, "e6"|->2,
"e7"|->2, "e8"|->2, "e9"|->2, "e10"|->2, "e11"|->11, "e12"|->11,
"e13"|->11, "e14"|->11, "e15"|->11, "e36"|->43, "e27"|->43, "e28"|->43,
"e29"|->43, "e20"|->43}}

B.2.2 Groupe *LTVGr*

$LTVGr = \{load_table(t)\}$ avec $t = \{$

1. {|->},
2. {"e1"|->1},
3. {"e1"|->1, "e2"|->1},
4. {"e1"|->1, "e2"|->1, "e3"|->1},
5. {"e1"|->5, "e2"|->5, "e3"|->5},
6. {"e1"|->1, "e2"|->1, "e3"|->1, "e4"|->1},
7. {"e1"|->1, "e2"|->1, "e3"|->1, "e4"|->1, "e5"|->1},
8. {"e1"|->1, "e2"|->1, "e3"|->1, "e6"|->2, "e7"|->2, "e8"|->2},
9. {"e1"|->1, "e2"|->1, "e4"|->1, "e7"|->1, "e3"|->2, "e5"|->2,
"e6"|->2, "e8"|->2},
10. {"e1"|->1, "e2"|->1, "e4"|->1, "e7"|->1, "e8"|->1, "e3"|->2,
"e5"|->2, "e6"|->2, "e9"|->2},
11. {"e1"|->1, "e2"|->1, "e4"|->1, "e7"|->1, "e8"|->2, "e3"|->2,
"e5"|->2, "e6"|->2, "e9"|->2},
12. {"e1"|->1, "e2"|->1, "e3"|->3, "e4"|->1, "e6"|->2, "e7"|->2,
"e8"|->2, "e9"|->3, "e0"|->3},
13. {"e1"|->1, "e2"|->1, "e4"|->1, "e7"|->1, "e8"|->1, "e3"|->2,
"e5"|->2, "e6"|->2, "e9"|->2, "e10"|->2},
14. {"e1"|->1, "e2"|->1, "e4"|->1, "e7"|->1, "e8"|->2, "e3"|->2,
"e5"|->2, "e6"|->3, "e9"|->3, "e10"|->3, "e11"|->3},
15. {"e1"|->1, "e2"|->1, "e3"|->1, "e4"|->1, "e5"|->1, "e6"|->2,
"e7"|->2, "e8"|->2, "e9"|->2, "e10"|->2, "e11"|->11, "e12"|->11,
"e13"|->11, "e14"|->11, "e15"|->11},
16. {"e1"|->1, "e2"|->1, "e3"|->1, "e4"|->1, "e5"|->1, "e6"|->2,
"e7"|->2, "e8"|->2, "e9"|->2, "e10"|->2, "e11"|->11, "e12"|->11,
"e13"|->11, "e14"|->11, "e15"|->11, "e16"|->12, "e17"|->12, "e18"|->12,
"e19"|->12, "e36"|->43, "e27"|->43, "e28"|->43, "e29"|->43, "e20"|->43},
17. {"e1"|->1, "e2"|->1, "e3"|->1, "e4"|->1, "e5"|->1, "e6"|->2,
"e7"|->2, "e8"|->2, "e9"|->2, "e10"|->2, "e11"|->11, "e12"|->11,
"e13"|->11, "e14"|->11, "e15"|->11, "e36"|->43, "e27"|->43, "e28"|->43,
"e29"|->43, "e20"|->43}}

B.2.3 Groupe *LTGrosG*

$LTGrosGr = \{load_table(t)\}$ avec $t = \{$

1. {|->},
2. {"e1"|->1, "e2"|->1, "e3"|->1},
3. {"e1"|->1, "e2"|->1, "e3"|->1, "e4"|->1, "e5"|->1, "e6"|->2,
   "e7"|->2, "e8"|->2, "e9"|->2, "e10"|->2, "e11"|->11, "e12"|->11,
   "e13"|->11, "e14"|->11, "e15"|->11, "e16"|->12, "e17"|->12, "e18"|->12,
   "e19"|->12, "e36"|->43, "e27"|->43, "e28"|->43, "e29"|->43, "e20"|->43},
4. {"e1"|->1, "e2"|->1, "e3"|->1, "e4"|->1, "e5"|->1, "e6"|->2,
   "e7"|->2, "e8"|->2, "e9"|->2, "e10"|->2, "e11"|->11, "e12"|->11,
   "e13"|->11, "e14"|->11, "e15"|->11, "e36"|->43, "e27"|->43, "e28"|->43,
   "e29"|->43, "e20"|->43}}

Bibliographie

[Abr96] J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[BCC+05] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3) :212–232, juin 2005.

[BDLU05] F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. JML-Testing-Tools : a symbolic animator for JML specifications using CLP. In N. Halbwachs and L. Zuck, editors, *Procs of the 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, Tool session (TACAS'05)*, volume 3440 of *LNCS*, pages 551–556, Edinburgh, UK, Avril 2005. Springer-Verlag.

[Beg02] S. Beghdadi. *Exploitation d'objectifs de test par ajout d'un langage de contrainte*. DEA d'Informatique : Systèmes et Communications - Université Joseph Fourier, Juin 2002.

[Bei90] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1990.

[Ber91] G. Bernot. Testing against formal specifications : A theoretical view. In Samson Abramsky and T. S. E. Maibaum, editors, *TAPSOFT, Vol.2*, volume 494 of *Lecture Notes in Computer Science*, pages 99–119. Springer, 1991.

[BGM91] Gilles Bernot, Marie Claude Gaudel, and Bruno Marre. Software testing based on formal specifications : a theory and a tool. *Software Engineering Journal*, 6(6) :387–405, 1991.

[BGMR03] J. Bézivin, S. Gérard, P.-A. Muller, and L. Rioux. Mda components : Challenges and opportunities. In *Metamodelling for MDA, York*, York, England, November 2003.

[Bho00] Abhay Bhorkar. A run-time assertion checker for Java using JML. Technical Report 00-08, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011, 2000.

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

[BKKP02] Igor B. Bourdonov, Alexander Kossatchev, Victor V. Kuliamin, and Alexandre Petrenko. Unitesk test suite architecture. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*, pages 77–88. Springer-Verlag, 2002.

[BKM02] C. Boyapati, S. Khurshid, and D. Marinov. Korat : Automated testing based on java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis*, Rome, 22–24 2002. IEEE.

[BL03] F. Bouquet and B. Legeard. Reification of executable test scripts in formal specification-based test generation : The java card transaction mechanism case study. In *Proc. of FME'03, Formal Method Europe*, volume 2805 of *LNCS*, pages 778–795, Pisa, Italy, September 2003.

[BLP02] F. Bouquet, B. Legeard, and F. Peureux. CLPS-B - A Constraint Solver for B. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS2002*, volume LNCS 2280, pages 188–204, Grenoble, France, Avril 2002. Springer-Verlag.

[BMdB+01a] P. Bontron, O. Maury, L. du Bousquet, Y. Ledru, C. Oriat, and M.-L. Potet. TOBIAS : un environnement pour la création d'objectifs de tests à partir de schémas de tests. In J.C. Rault, editor, *ICSSEA 2001*, Décembre 2001.

[BMdB+01b] P. Bontron, O. Maury, L. du Bousquet, Y. Ledru, C. Oriat, and M.-L. Potet. COTE document O-TeLa. Rapport sous-projet 3 - activité 3.1, Projet COTE, Juin 2001.

[Bon05] P. Bontron. *Les sch´emas de test : une abstraction pour la g´en´eration de tests de conformit´e et pour la mesure de couverture.* PhD thesis, Université Joseph Fourier - Grenoble 1, Grenoble, France, 2005.

[BRL03] Lilian Burdy, Antoine Requet, and Jean Louis Lanet. Java applet correctness : A developer-oriented approach. In *FME*, pages 422–439, 2003.

[BY98] K. Burr and W. Young. Combinatorial test techniques : Table-based automation, test generation and code coverage. In *Intl. Conf. on Software Testing Analysis & Review*, San Diego, 1998.

[BY01] Luciano Baresi and Michal Young. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., Aout 2001. http ://www.cs.uoregon.edu/ michal/pubs/oracles.html.

[CDFP97] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG System : An Approach to Testing Based on Combinatorial Design. In *IEEE Transactions On Software Engineering*, volume 7-23, pages 437–444, Juillet 1997.

[Cha94] O. Charles. *Application des hypothèses de test à une définition de la couverture.* PhD thesis, Université Henri Poincaré - Nancy 1, Nancy, France, 1994.

[CL02a] Y. Cheon and G. T. Leavens. A Runtime Assertion Checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *International Conference on Software Engineering Research and Practice (SERP '02)*, pages 322–328, Las Vegas, Nevada, Juin 2002. CSREA Press.

[CL02b] Y. Cheon and G.T. Leavens. A simple and practical approach to unit testing : The JML and JUnit way. In B. Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming, 16th European Conference, Malaga, Spain, Proceedings*, LNCS 2474, pages 231–255. Springer, 2002.

[CLRZ99] Ana R. Cavalli, David Lee, Christian Rinderknecht, and Fatiha Zaïdi. Hit-or-Jump : An algorithm for embedded testing with applications to IN services. In Jianping Wu, Samuel T. Chanson, and Qiang Gao, editors, *FORTE*, volume 156 of *IFIP Conference Proceedings*, pages 41–56. Kluwer, 1999.

[dBMJ01] L. du Bousquet, H. Martin, and J.-M. Jézéquel. Conformance Testing from UML specifications, Experience Report. In Gesellschaft für Informatik, editor, *p-UML workshop, Lecture Notes in Informatics*, volume P-7, pages 43–56, Toronto, 2001.

[dBORZ99] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Lutess : a specification-driven testing environment for synchronous software. In *21st International Conference on Software Engineering*. ACM, Mai 1999.

[DCdBBL05] S. Dupuy-Chessa, L. du Bousquet, J. Bouchet, and Y. Ledru. Test of the ICARE platform fusion mechanism. In *12th International Workshop on Design, Specification and Verification of Interactive Systems (DSVIS'05)*, Juillet 2005.

[DF93] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications . In *FME'93 : Industrial-Strength Formal Methods*. LNCS 760, Springer-Verlag : 268-284, Avril 1993.

[DLS78] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection : Help for the practicing programmer. *Computer, 11*, pages 34–41, 1978.

[FL98] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. ISBN 0-521-62348-0.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *PLDI*, pages 234–245, 2002.

[GA96] P. Le Gall and A. Arnould. Formal specifications and test : Correctness and oracle. In M. Haveraaen, O. Owe, and O-J. Dahl, editors, *Recent Trends in Data Type Specification*, LNCS 1130, pages 342–358, Oslo, 1996.

[Gau95] M.-C. Gaudel. Testing can be formal, too. In *Proc. of the Sixth International Joint Conference CAAP/FASE : TAPSOFT'95-Theory and Practice of Software Development*, pages 82–96, Aarhus, Denmark, 1995.

[GBR98] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. In *Proceedings of ACM SIGSOFT international symposium on Software testing and analysis*, pages 53–62. ACM Press, 1998.

[GHG+93] J. V. Guttag, J. J. Horning, S. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch : Langages and Tools for Formal Specification*. Springer-Verlag, New-York, 1993.

[Got00] A. Gotlieb. *Génération automatique de cas de test structurel avec la programmation logique par contraintes*. PhD thesis, Université de Nice-Sophia Antipolis, Nice-Sophia Antipolis, France, 2000.

[Gro00] The VDM Tool Group. VDM-SL Toolbox User Manual. Technical report, IFAD, Octobre 2000. ftp ://ftp.ifad.dk/pub/vdmtools/doc/userman_letter.pdf.

[HHS03] Robert M. Hierons, Mark Harman, and Harbhajan Singh. Automatically generating information from a z specification to support the classification tree method. In *ZB*, pages 388–407, 2003.

[HJGP99] Wai Ming Ho, Jean-Marc Jézéquel, Alain Le Guennec, and Francois Pennaneac'h. UMLAUT : An extendible UML transformation framework. In *Automated Software Engineering*, pages 275–278, 1999.

[ISO91] ISO. *Information Technology, Open System Interconnection, Conformance Testing Methodology and Frameworks*. International Standard IS-9646, ISO, 1991. CCITT X.290–X.294.

[ISO95] ISO. *Formal Method in Conformance Testing*. ISO, 1995.

[ISO96] ISO. *Information Technology — Programming Languages, their environments and system software interfaces — Vienna Development Method-Specification Language Part 1 : Base language*, 1996.

[Jac00] D. Jackson. Automating first-order relational logic. In *Proc. 8th ACM SIGSOFT Symposium on the Foudations of Software Engineering*, San Diego, CA, Novembre 2000.

[JCo] JCoverage. http ://jcoverage.com/.

[JM99] T. Jéron and P. Morel. Test Generation Derived from Model-checking. In *Computer Aided Verification (CAV)*. LNCS 1633, Springer-Verlag, 1999.

[Jon90] C. B. Jones. *Systematic Software Development Using VDM (Second Edition)*. Prentice-Hall, London, 1990. C.A.R. Hoare editor.

[JSS00] D. Jackson, I. Schechter, and I. Shlyakhter. ALCOA : The Alloy constraint analyser. In *Proc. 22th International Conference on Software Engineering*, Limerick, Ireland, Juin 2000.

[JSS01] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *Proc. 9th ACM SIGSOFT Symposium on the Foudations of Software Engineering*, Vienna, Australia, Septembre 2001.

[JUn] JUnit. http ://www.junit.org.

[Kes03] M. Kessis. *Test de conformité des programmes JAVA*. DEA en Systèmes d'Informations - Université Joseph Fourier, Juin 2003.

[KKP02] A.A. Koptelov, V.V. Kuliamin, and A.K. Petrenko. Vdm++TesK : Testing of VDM++ programs. In *3rd VDM Workshop (at FME2002)*, Juillet 2002.

[Kul04] V.V. Kuliamin. Multi-paradigm Models as Source for Automated Test Construction. In *Workshop "Model Based Testing" de ETAPS 04*, Mars 2004.

[LBR99] G.T. Leavens, A.L. Baker, and C. Ruby. JML : A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.

[LBR02] G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML : A behavioral interface specification language for Java. Technical Report 98-06q, Iowa State University, Department of Computer Science, Juin 2002.

[LdBMB04] Yves Ledru, Lydie du Bousquet, Olivier Maury, and Pierre Bontron. Filtering tobias combinatorial test suites. In *FASE*, pages 281–294, 2004.

[Led91] Guy Leduc. Conformance Relation, Associated Equivalence, and New Canonical Tester in LOTOS. In *Proceedings of the IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XI*, pages 249–264. North-Holland, 1991.

[LG01] A. Le Guennec. *Génie Logiciel et Méthodes Formelles avec UML : Spécification, Validation et Génération de tests*. PhD thesis, Université de Rennes 1, Rennes, France, 2001.

[LP02] B. Legeard and F. Peureux. B-Testing-Tools : génération de tests aux limites à partir de spécifications B. *TSI, Techniques et Sciences Informatiques, Hermès-Lavoisier*, 21(9) :1189–1218, 2002.

[LPU02] B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from Z and B. In L.-H. Eriksson and P.A. Lindsay, editors, *Formal Methods – Getting IT Right*, LNCS 2391, pages 21–40. Springer, 2002.

[MA00] Bruno Marre and Agnes Arnould. Test Sequences Generation from LUSTRE Descriptions : GATEL. In *ASE '00 : Proceedings of the The Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*, page 229, Washington, DC, USA, 2000. IEEE Computer Society.

[Mar95] Bruno Marre. LOFT : A Tool for Assisting Selection of Test Data Sets from Algebraic Specifications. In *TAPSOFT*, pages 799–800, 1995.

[Mar01] H. Martin. *Une méthodologie de génération automatique de suites de tests pour applets Java-card*. PhD thesis, Université de Lille 1, Lille, France, 2001.

[Mey92a] B. Meyer. Applying "Design by Contract". In *Computer Vol. 25, No. 10*, pages 40–51. IEEE, Octobre 1992.

[Mey92b] B. Meyer. *Eiffel : The language*. Prentice-Hall, New-York, 1992. Object-Oriented Series.

[MK01] D. Marinov and S. Khurshid. TestEra : A novel framework for automated testing of Java programs. In *Proc. 16th IEEE International Conference on Automated Software Engineering (ASE)*, San Diego, CA, Novembre 2001.

[MLdB03] O. Maury, Y. Ledru, and L. du Bousquet. Intégration de TOBIAS et UCASTING pour la génération de tests. In *16th International Conference Software and Systems and their applications-ICSSEA*, Décembre 2003.

[Nta88] S. C. Ntafos. A comparison of some structural testing strategies. *IEEE Trans. Softw. Eng.*, 14(6) :868–874, 1988.

[Nta98] S. Ntafos. On random and partition testing. In *Proceedings of ACM SIGSOFT international symposium on Software testing and analysis*, pages 42–48. ACM Press, 1998.

[OH96] A. J. Offutt and Hayes J. H. A semantic model of program faults. In Steven J. Zeil, editor, *International Symposium on Software Testing and Analysis (ISSTA)*, pages 195–200, San Diego, Californy, USA, Janvier 1996.

[OMG] OMG. MDA : http ://www.omg.org/mda/.

[OMG02] OMG. Uml 2.0 testing profile specification. Technical report, Object Management Group, 2002.

[Ori02] Catherine Oriat. Jartege : a tool for random generation of unit tests for java classes. Rapport de recherche RR1069 LSR 19, LSR-IMAG, 2002.

[Peu02] F. Peureux. *Génération de tests aux limites à partir de spécifications B en Programmation Logique avec Contraintes ensemblistes*. PhD thesis, Université de Franche-Comté, Besançon, France, 2002.

[Pha94] M. Phalippou. *Relations d'implantations et hypothèses de test sur des automates à entrées et sorties*. PhD thesis, Université de Bordeaux 1, Bordeaux, France, 1994.

[PJH+01] S Pickin, C Jard, T Heuillard, J-M Jézéquel, and P Desfray. A uml-integrated test description language for component testing. In Andy Evans, Robert France, Ana Moreira, and Bernhard Rumpe, editors, *Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists. Workshop of the pUML-Group held together with the UML 2001*, volume P-7 of *LNI*, pages 208–223, Toronto, Canada, October 2001. German Informatics Society.

[PLT00] L. Py, B. Legeard, and B. Tatibouet. Évaluation de spécifications formelles en programmation logique avec contraintes ensemblistes – Application à l'animation de spécification B. In *AFADL'2000*, pages 21–35, Grenoble, 2000.

[Spi92] J.M. Spivey. *The Z Notation - A Reference Manual*. Prentice Hall International - London, 1992.

[TB99] J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99 : 7^{th} European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, Novembre 8–12, 1999. EuroStar Conferences, Galway, Ireland.

[Tre92] J. Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[Tre96] J. Tretmans. Test Generation with Inputs, Outputs, and Quiescence. In T. Margaria and B. Steffen, editors, *Second Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer-Verlag, 1996.

[Tre00] J. Tretmans. Specification Based Testing with Formal Methods : From Theory via Tools to Applications. In A. Fantechi, editor, *FORTE / PSTV 2000 Tutorial Notes*, Pisa, Italy, Octobre 10 2000. Transparencies.

[UML99] UML. UML Revision Task Force. OMG Unified Modeling Language Specification version 1.3. Technical Report ad/99-06-08, Object Management Group, 1999. http ://www.omg.org/uml.

[Van98] L. Van Aertryck. *Une méthode et un outil pour l'aide à la génération de jeux de tests de logiciels*. PhD thesis, Université de Rennes I, Rennes, France, 1998.

[VBL97] L. Van Aertryck, M. Benveniste, and D. Le Métayer. CASTING : A formally based software test generation method. In *The 1st Int. Conf. on Formal Engineering Methods, IEEE, ICFEM'97*, Hiroshima, 1997.

[VJ03] L. Van Aertryck and T. Jensen. UML-CASTING : Test synthesis from UML models. In *Proc of AFADL'2003, Approches Formelles dans l'Assistance au Développement de Logiciels*, Rennes, 2003.