HAL
open science

Contributions à l'automatisation raisonnée de différents processus du test logiciel

Lydie Du Bousquet

To cite this version:

Lydie Du Bousquet. Contributions à l'automatisation raisonnée de différents processus du test logiciel. Génie logiciel [cs.SE]. Université de Grenoble, 2010. tel-01005528

HAL Id: tel-01005528

https://theses.hal.science/tel-01005528

Submitted on 12 Jun 2014

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 de Grenoble

Mémoire présenté pour l'obtention d'une

Habilitation à Diriger des Recherches

Discipline : Informatique

présentée et soutenue publiquement

par

Lydie du Bousquet

le 3 novembre 2010

Contributions à l'automatisation raisonnée de différents processus du test logiciel

Composition du Jury :

Rapporteurs	Antonia Bertolino
	Lionel Briand
	Bruno Legeard
Examinateurs	Pascale LeGall
	Jean-Claude Fernandez
	Farid Ouabdesselam

Remerciements

Je remercie sincèrement toutes les personnes qui ont contribué de près ou de loin à l'aboutissement de cette habilitation, en particulier,

Antonia Bertolino, directrice de recherche au CNR (Italian National Research Council), Lionel Briand, professeur à l'université d'Oslo et Bruno Legeard, professeur à l'université de Franche-Comté pour avoir accepté de juger ce travail,

Pascale LeGall, professeur à l'université d'Evry-Val d'Essonne, Farid Ouabdesselam, professeur et président de l'Université J. Fourier pour avoir accepté de faire partie du jury et Jean-Claude Fernandez, professeur à l'Université J. Fourier pour l'avoir présidé,

Les membres de l'équipe VASCO et les personnes avec qui j'ai collaboré par ailleurs, sans qui les travaux présentés ici n'auraient pas pu prendre la dimension qu'ils ont pris,

Pierre-Claude Scholl, professeur à l'université J. Fourier, qui m'a fait découvrir l'informatique en DEUG, qui m'a conseillé plus d'une fois dans mon orientation, et sans qui je n'aurais probablement pas fait mon doctorat puis cette habilitation (« oui Pierre-Claude, onze c'est mieux que dix ! »),

Sophie, pour l'amitié qu'elle m'a accordée depuis ma thèse,

Ma Maman et les « hommes de ma vie » (Mon papa, mon mari et mes deux garçons), qui ont su me supporter et m'encourager à bon escient.

Résumé

Le test constitue aujourd'hui la principale activité de validation d'un logiciel. Dans un contexte où l'on cherche à réduire les coûts et augmenter la qualité, il est essentiel de proposer des solutions de test automatisées et de veiller à la productivité des ingénieurs de test. Les travaux présentés dans ce mémoire ont l'ambition de contribuer à ces objectifs.

Ces travaux se déclinent selon trois axes. Le premier axe concerne la génération de tests. L'originalité du travail se situe dans l'établissement de principes pour la production combinatoire de suites de test, à partir d'expressions abstraites. Ces principes ont été implantés dans un outil appelé Tobias.

Lorsque de grandes suites de test sont produites, il est important d'automatiser l'oracle des tests. Un deuxième axe vise à évaluer l'utililisabilité des assertions pour établir le verdict des tests pour des applications domotiques. Les résultats montrent que les assertions sont effectivement utilisables, mais que l'exécution de ces applications dans un environnement réel non complètement contrôlable ou observable peut conduire en l'émission de verdicts biaisés.

Ainsi, l'automatisation de la génération et de l'oracle permet de réduire le coût de la phase de test. Mais, pour réduire le coût du test, il est aussi important de considérer les facteurs internes au logiciel relatifs à la testabilité. De nombreuses métriques ont été proposées pour prédire le estimer la testabilité d'un système. Un troisième axe de recherche concerne l'évaluation la validation expérimentale de ces métriques. Les résultats des expérimentations démontrent que les métriques étudiées ne sont pas directement utilisables pour prédire le coût du test.

Table des matières

Avant propos 11

Chapitre 1 : Introduction 13

1 Contexte **13**

2 Les difficultés du test **14**

3 Contributions **15**

Chapitre 2 : Génération de tests 21

1 Introduction **21**

2 Motivations **22**

3 Un exemple d'application à tester **23**

4 Test combinatoire : du concept à l'outil **25**
4.1 Tobias-V1 25
4.2 Tobias-V2 27

5 Des stratégies de réduction de la suite de test **30**
5.1 Réduction aléatoire (équilibrée ou non) 30
5.2 Couverture des paires d'appels de méthodes 33
5.3 D'autres stratégies de réduction 37

6 Du problème de l'oracle **37**
6.1 Des assertions dans des schémas 38
6.2 De l'utilisation d'une spécification formelle pour calculer l'oracle 39

7 La validité des séquences **40**

8 Des travaux similaires **41**

9 Conclusion et perspectives **42**

Chapitre 3 : Des assertions comme oracle 45

1 Introduction **45**

2 Motivations **46**

3 Java Modeling Language pour le test **48**

4 Spécifier des services domotiques **49**
4.1 Les caractéristiques d'un système domotique 49
4.2 Des spécifications à plusieurs niveaux 50
4.3 Oracle pour les applications dynamiques 51
4.4 Limites de l'oracle dans le contexte domotique 53

5 La qualité des assertions comme oracle **54**

6 Pertinence des assertions comme oracle **56**

7 Conclusion et perspectives **59**

Chapitre 4 : Prédire la testabilité **61**

1 Introduction **61**

2 Motivations **63**

3 Validation de DIT dans le contexte objet **63**
3.1 De l'utilisation de DIT pour prédire la testabilité 64
3.2 Pour la validation de DIT 64
3.3 Expérimentation 65
3.4 Autres études 67

4 Validation de métriques de testabilité pour Lustre **68**
4.1 Le langage Lustre 68
4.2 La testabilité de programmes Lustre/Scade 69
4.3 Approche de validation 70

5 Conclusion et perspectives **71**

Chapitre 5 : Perspectives **75**

Annexe 1 : Analyse mutationnelle **77**

A Introduction **77**

B Principes de l'approche originale **77**

C Modèle de faute et opérateurs de mutation **79**

Annexe 2 : Trois articles **83**

Annexe 3 : Curriculum Vitae **129**

A Parcours professionnel **129**

B Encadrement d'étudiants **130**

C Participation et montage de projets **130**

D Service à la communauté académique et scientifique **131**

E Invitations **131**

F Enseignement **131**

G Publications **132**

Avant propos

Je dédie ce document à mon mari, mes enfants et mes parents.

« Et c'est la vérité ». Stephen King
(Troisième volume de la Tour Sombre)

Ma première expérience avec l'informatique remonte au jour où j'ai fait mon (seul) programme sur un ZX Spectrum+ que je possède toujours. J'ai consciencieusement écrit mes 10 lignes et je l'ai exécuté. Puis j'ai éteint l'ordinateur... et le programme a disparu : il fallait le sauvegarder sur une cassette audio (si si) et je ne l'avais pas fait. Après cette expérience, je n'aurais jamais imaginé que je ferais de l'informatique mon métier.

Ma seconde expérience s'est déroulée plusieurs années plus tard, en DEUG. L'idée véhiculée pendant ces deux années était qu'il fallait d'abord résoudre le problème de façon algorithmique, puis traduire la solution dans un programme. J'ai trouvé cette façon de résoudre les problème vraiment très intéresseante. Et c'est comme ça que j'ai poursuivi en Magistère d'informatique à l'ENS Lyon.

Lorsque Farid Ouabdesselam et Jean-Luc Richier m'ont proposé un stage de DEA sur la recherche d'interaction entre services téléphoniques, j'ai fait connaissance avec le monde du test. Finalement, je ne l'ai pas quitté. C'est un domaine passionnant car même si la problématique est toujours la même (trouver des erreurs), le contexte fait qu'il y a toujours quelque chose de nouveau à considérer.

A bien y réfléchir, je devais y être prédestinée. Aujourd'hui je me rends compte que j'ai suivi une voie tracée par mes parents. Mon père était ingénieur qualité dans le nucléaire. Il était « l'inspecteur des travaux finis », comme on disait alors. Ma mère avait pour mission de rechercher les défauts dans les écrans de télévision en sortie de production. Finalement, leurs métiers n'étaient pas si différents de celui que je fais. Juste une question de contexte en quelque sorte !

Depuis mon DEA jusqu'à cette HDR, j'ai eu la chance de naviguer entre le formel et l'empirique. Mais au-delà de ça, il demeure le principe qu'il faut toujours valider ce qui est entrepris. C'est ce qui m'a incité à faire participer l'équipe au concours de détection d'interaction entre services téléphoniques

en 1998. Et cela a orienté certains de mes travaux présentés plus tard dans ce document (chapitre 4).

Par ailleurs, on m'a appris depuis toujours si l'on propose des solutions, elles doivent adresser des problèmes réels. Et les solutions proposées doivent être faciles à mettre œuvre pour être acceptées. C'est ce qui a guidé mon travail (chapitres 2 et 3).

Bonne lecture !

Chapitre 1

Introduction

Question : « Quelles stratégies utiliser pour valider un logiciel ? »

Réponse : « Le test est l'une des stratégies les plus utilisées en milieu industriel. »

1. Contexte

Les systèmes informatiques sont de plus en plus présents dans notre vie quotidienne. Nombre d'entre-eux sont critiques : ils mettent en jeu des vies humaines ou des sommes importantes. On attend de ces systèmes qu'ils fonctionnent parfaitement. Or, la programmation est une activité qui est source d'erreurs. On peut se tromper car on n'a pas compris le problème ou parce que l'on a fait une faute en écrivant la solution. Les fautes dans le code entraînent de possibles défaillances [148]. Les conséquences d'une défaillance peuvent être dramatiques [122]. Selon une étude réalisée en juin 2002 par le NIST (*National Institute of Standards and Technology*), la facture annuelle des logiciels défectueux s'élève à environ 60 milliards de dollars pour l'économie américaine [222].

Pour réduire la facture, il faudrait produire des logiciels de meilleure qualité. Il est difficile de décrire exhaustivement ce qu'est un logiciel de *qualité*, car il existe nombreux critères : correction fonctionnelle, fiabilité, performance, ergonomie, ... Le type de logiciel et l'usage attendu déterminent lesquels doivent être considérés. Néanmoins, quels que soient les critères choisis, la qualité d'un logiciel se construit tout au long de son cycle de vie [108]. Ainsi, il existe de nombreuses méthodes de conception, des règles de développement, de maintenance, etc. que nous n'évoquerons pas ici [219, 124, 196, 139].

Quelle que soit la méthode de conception utilisée, il faut toujours s'assurer au *final* que le logiciel respecte les critères de qualité spécifiés. La phase du cycle de vie du logiciel qui est concernée par ce point est la phase de *Validation & Vérification* (V&V). Elle s'étale tout au long du développement. Les termes vérification et validation englobent toutes les activités permettant de s'assurer que le logiciel correspond bien à son cahier des charges d'une part et que le cahier des charges répond bien aux besoins de l'utilisateur. Plus spécifique-

ment, Boehm énonce que « la vérification s'assure que le produit est conforme à sa spécification ; la validation s'assure que le système implémenté correspond aux attentes du futur utilisateur » [26]. Plusieurs techniques peuvent être employées : revue de code [177], test [173], vérification par la preuve ou par model-checking [64]...

Parmi les méthodes de V&V, le test est la plus employée dans l'industrie du logiciel [20]. Elle est souvent plus facile à mettre en œuvre que les autres techniques de validation et elle apporte rapidement un minimum de confiance vis à vis de la qualité du produit. Tester un logiciel, c'est l'exécuter en contrôlant les entrées et observer les résultats, de façon à s'assurer que les résultats obtenus sont conformes aux attentes.

2. Les difficultés du test

Tester est une activité bien plus complexe qu'il peut paraître initialement. Quatre principales difficultés sont souvent identifiées.

Tout d'abord, comme le dit Edsger W. Dijkstra « *Program testing can be used to show the presence of bugs, but never to show their absence !* ». En effet, pour prouver l'absence d'erreur dans un programme, il faudrait le tester complètement ; c'est-à-dire exécuter toutes les entrées possibles dans tous les cas de figure. On parle alors de test *exhaustif*. En général, c'est impossible car l'espace des entrées, des états du programme et de son environnement sont trop grands pour pouvoir être explorés en un temps raisonnable.

La première difficulté du test consiste donc en la *sélection* d'un sous-ensemble de données d'entrée et/ou un nombre de situations limitées et se convaincre qu'elles sont représentatives des situations possibles, et par conséquent, qu'elles permettent de mettre en évidence les situations où les défaillances sont observables. Comme chaque système a des caractéristiques propres et des objectifs de validation spécifiques, il n'existe pas de solution universelle pour sélectionner un sous-ensemble de tests. Par exemple, pour évaluer la correction fonctionnelle, la non-régression ou la performance, on favorisera une situation plutôt qu'une autre. De même, les systèmes embarqués, les systèmes distribués, les systèmes non déterministes, etc. apportent un lot de contraintes qui leurs sont propres.

Lié au problème de la sélection, une seconde difficulté est qu'il faut être en mesure de décider quand s'*arrêter* de tester. On ne devrait s'arrêter de tester que lorsque toutes les fautes possibles ont été identifiées. Mais, on ne connaît pas le nombre de fautes a priori. Il n'est donc pas possible de garantir que la suite de test permet de découvrir effectivement toutes les fautes. Pour aider à ce choix, on peut évaluer la suite de test par rapport à des critères, dits « d'adéquation », qui spécifient les exigences attendues relatives à la suite de test.

La troisième difficulté est relative à la capacité du testeur à identifier les défaillances. Lorsque le système sous test est exécuté, il faut décider de la

correction des comportements observés. On parle du problème de l'*oracle*. Caractériser de façon précise le résultat attendu est difficile, car le système peut être complexe. De plus, le résultat attendu doit être exprimé par rapport aux différents critères qualité attendus : notamment correction fonctionnelle, performance, ergonomie, robustesse, ...

Enfin, comme le test est une activité coûteuse en temps. On cherche à trouver des solutions pour en *diminuer le coût*. Par exemple, on peut automatiser les différentes phases du test et/ou chercher à produire des systèmes faciles à tester.

Compte-tenu des différentes difficultés relatives au test de systèmes et de la diversité des applications, de nombreux travaux ont été menés sur le test. Ils sont parfois de natures très différentes. Ci-après, je présente les travaux que j'ai menés par rapport à ces différentes problématiques.

3. Contributions

Depuis ces dernières années, j'ai eu l'occasion d'aborder différents contextes applicatifs, ayant leurs lots d'objectifs propres. Ainsi, j'ai pu me poser la question de la génération des données de tests, de la caractérisation de l'oracle, de l'arrêt du test, de la qualité des tests et de la testabilité pour des systèmes embarqués sur carte à puce (projets RNTL COTE, ANR POSE et ANR TASCCC), pour les systèmes embarqués de l'avionique (projets ANR SIESTA), ou pour des services domotiques (projet PAI Sakura et projet UJF iPotest).

Ces systèmes ont été programmés en en Java ou en Lustre/Scade. Le point commun entre ces deux types de système est qu'ils ne sont pas purement fonctionnels : la réponse à une sollicitation donnée dépend de l'entrée courante et de l'état interne du système. Pour tester de tels système, il faut considérer des « *séquences de test* » (séquence d'appels de méthodes pour Java ou de vecteurs d'entrées pour Lustre). A la difficulté de trouver des valeurs, s'ajoute la difficulté de déterminer un enchaînement d'appels ou des vecteurs d'entrée. Evidemment, le choix des données influe sur les possibilités d'enchaîner les appels de méthodes et réciproquement, le choix de l'enchaînement restreint le domaine des valeurs à un point donné dans l'exécution.

Génération de séquences de test (chapitre 2)

Dans la suite, le chapitre 2 concerne la conception de séquences de tests pour les programmes objets. Il existe de nombreuses stratégies pour la génération automatique de séquences de tests, qui ont été proposées dans ce contexte [25, 109, 245, 133]. Elles se différencient souvent par la nature et la quantité d'informations nécessaires pour pouvoir appliquer la stratégie. Certaines ne s'appliquent que sur la base du code [25, 109, 245] ou de la seule interface des classes [25]. D'autres nécessitent la description d'une spécification ou d'un modèle [25, 133].

Mon travail sur ce sujet s'inscrit dans une trajectoire entamée au cours de mon doctorat [66], à savoir offrir des stratégies qui s'appuient sur un nombre limité d'informations (issues des phases de conception ou de validation). De façon plus précise, dans ce chapitre, je présente le travail effectué autour du test combinatoire. L'origine de ce travail provient de l'observation que, dans l'industrie, l'ingénieur de validation doit souvent produire des séquences de tests qui présentent de nombreuses similarités. Ainsi, lorsque l'on valide le système sous test à l'aide de séquences, l'ingénieur peut chercher à se convaincre que tout se passe bien, en considérant des variations sur une séquence.

J'ai pu observer que le travail de production de ces séquences similaires s'appuyait souvent sur du « copier-coller ». C'est un travail très fastidieux et sans réelle valeur ajoutée. En automatisant la génération de séquences similaires, on apporte une diminution notable de l'effort de réalisation des tests et on diminue les risques d'erreurs pendant le recopiage/modification des tests. Pour cela, le testeur est invité à décrire la suite de test qu'il souhaite à l'aide d'une expression régulière et des variations associées. C'est une façon de capturer les connaissances du testeur. La suite de test est ensuite construite par dépliage selon les dimensions choisies. Permettre d'exprimer une suite de test sous forme d'expression abstraite et fournir des outils pour la déplier est en une suite de test exécutable est ma contribution principale sur le thème de la génération de test. Lorsque cette approche a été proposée dans le cadre du projet ANR COTE (2000-2002), il n'existait pas, à ma connaissance, de solutions similaires dans la littérature.

L'approche est assez flexible, et peut donc être utilisée pour différents cas de figure. Elle a toutefois été essentiellement expérimentée pour construire des suites de test (unitaire ou système) de systèmes embarqués ou de services domotiques, pour lesquels elle est particulièrement adaptée. En effet, ce type d'application doit être robuste et donc doit être en mesure de réagir à des séquences d'entrées non attendues a priori. Les combinaisons d'appel de méthodes et de paramètres définies à l'aide d'expressions régulières permettent souvent d'explorer des scénarios non initialement envisagés par le testeur, et c'est l'une des forces de l'approche.

Ainsi, l'approche de test combinatoire proposée permet d'obtenir très rapidement de nombreux tests (à partir d'une définition très abstraite de la suite de test attendue). Le revers de la médaille est que l'on obtient vite trop de tests. On parle d'explosion combinatoire. Une partie de mon travail de recherche est donc d'explorer des solutions pour réduire la taille de la suite de test ou son exécution, tout en maintenant de bonnes propriétés.

Mes travaux autour de la *génération de tests* ont été menés en collaboration avec Yves Ledru et Catherine Oriat, dans le cadre des projets RNTL COTE (2000-2002), ANR POSE (2006-2007) et ANR TASCCC (2009-2012). Ils ont été au cœur des travaux que j'ai encadré pendant les doctorats de Pierre Bontron et

de Olivier Maury[1] [27, 167], des M2R de Taha Triki et Azzédine Amiar [224, 6], du mémoire d'ingénieur CNAM de Sébastien Ville, et du stage d'ingénieur de Elodie Rose. Ils ont été publiés dans [83, 156, 105, 153, 152, 117, 53, 154].

En parallèle du problème de génération des données, il faut s'intéresser à l'établissement du verdict de correction. En effet, une suite d'appels de méthode n'est un test que si les réactions du système sous test sont jugées. C'est la notion d'oracle. En toute généralité, l'oracle peut être manuel ou automatique. Evidemment, dans un contexte où l'on produit massivement des tests, l'oracle ne peut être qu'automatisé. L'approche sous-entend donc que le testeur soit en mesure de fournir un tel oracle, par exemple en exploitant une spécification exécutable. Il est par exemple possible d'exprimer l'oracle dans le programme de test sous la forme de d'assertions ou contrats [170]. L'utilisation d'assertions comme oracle du test a été expérimenté pour la validation de services domotiques. Les résultats sont présentés dans le chapitre 3 et résumés ci-après.

Des assertions comme oracle (l'exemple de JML pour spécifier des services domotiques) (chapitre 3)

Les langages de spécification basés sur les assertions permettent de décrire le comportement attendu du logiciel sous la forme d'annotations telles que des invariants, pré et post-conditions [170, 201]. Ces assertions peuvent être transformées en code exécutable et évaluées pendant l'exécution du programme. Un tel mécanisme peut être utilisé comme oracle pendant le test. Si une assertion est violée en cours d'exécution, une exception est levée, et un verdict d'échec peut être émis.

Les assertions sont un moyen de spécifier le système de plus en plus populaire. De plus en plus de langage comporte des instructions pour exprimer des assertions (Eiffel, Java, C, ...). Des langages complémentaires sont parfois disponibles, comme par exemple JML [151], JASS [14] ou Jawa [119] pour Java). Les assertions peuvent ne porter que sur une partie du système et peuvent être raffinées et/ou mises à jour au fur et à mesure du développement et de la maintenance en parallèle avec le code.

Une question qui se pose est de caractériser la nature des assertions nécessaires pour fournir un « bon » oracle pour les tests. Ceci a motivé deux études de cas issues du même contexte applicatif : les services domotiques. Les études de cas ont permis d'établir que les propriétés de sûreté étaient adaptées à la spécification de ces services et que ces dernières pouvaient être exprimées sous la forme d'assertion JML. De plus, nous avons pu constater que les assertions dérivées d'un processus de preuve et celles dérivées d'un processus de test ne sont pas forcément de même nature, même si elles visent à caractériser les mêmes comportements. Ces deux études de cas, les problématiques abordées et les résultats obtenus sont détaillés dans le chapitre 3.

---

1. J'ai seulement suivi le travail d'Olivier, qui a été encadré par Yves Ledru et Catherine Oriat.

Les travaux menés que j'ai menés sur ce thème ont pour cadre le projet UJF IPotest (collaboration avec l'équipe Adèle du LIG) et le projet Egide PHC-Sakura (2008-2009), en collaboration avec Masahide Nakamura (université de Kobe) et Ajitha Rajan (post-doc au LIG pendant IPotest). Ils ont été publiés dans [243, 244, 97, 86, 81, 197].

Analyse de métriques de testabilité (chapitre 4)

Puisque tester coûte cher, des stratégies ont été proposées dans la littérature pour en diminuer le coût. L'ensemble des travaux sur l'automatisation du test est un type de réponse que j'ai exploré et que je détaille dans les chapitres 2 et 3. Une autre vision des choses est de rendre le système plus facile à tester. On parle alors de testabilité [121, 181, 24].

Une façon d'estimer la testabilité d'un système logiciel consiste à utiliser des métriques. De très nombreuses métriques ont été proposées pour faire face aux nombreux types de systèmes et exigences relatives au test [228, 229, 46, 143, 24]. Il en résulte une certaine confusion : « Quelle(s) métrique(s) utiliser ? Que prédi(sen)t-elle(s) ? Comment interpréter les résultats ? ».

Les travaux que j'ai menés autour de la testabilité sont présentés dans le chapitre 4. Ils concernent la validation de métriques de testabilité proposés dans les contextes des programmes objets et des programmes synchrones (Lustre). Dans ces deux contextes, j'ai cherché à identifier les hypothèses associées à certaines métriques (souvent implicites) et à adapter les protocoles de validation empiriques proposés dans la littérature aux cas particulier des métriques validées.

Pour ce qui concerne les études menées pour la testabilité des programmes Java, nous nous sommes concentrés sur la validation de la métrique *Depth of Inheritance Tree* (DIT). L'étude de cette métrique est particulièrement intéressante car elle est suggérée comme étant une métrique prédictive de la testabilité [46, 24], mais les évaluations empiriques menées dans [38] ne corroborent pas cette suggestion. Les résultats de notre étude montrent que DIT est corrélé au nombre de méthodes à tester seulement dans le cas où une stratégie de test impose de re-tester toutes les méthodes héritées. Mais elle ne permet pas de prédire le nombre de tests à produire pour couvrir l'ensemble des branches des méthodes d'une classe [212, 214].

En ce qui concerne plus particulièrement Lustre, les métriques de testabilité étudiées sont au cœur du projet ANR SIESTA. Elles cherchent à identifier les faiblesses d'un système par une évaluation de son observabilité et de sa contrôlabilité (dans une démarche similaire aux travaux menés sur la testabilité des systèmes matériels). Grâce à ces métriques, il est possible d'identifier les parties difficiles à tester. Intuitivement, si une partie est difficile à observer et/ou à contrôler, alors il sera plus difficile de détecter des défaillances.

Pour valider cette intuition, j'ai cherché à identifier les parties pour lesquelles des fautes introduites intentionnellement sont difficiles à détecter, et je les ai comparé parties désignées comme difficiles à tester par les mesures.

L'introduction intentionnelle de faute a été effectuée au moyen d'une analyse mutationnelle [60, 182]. Cette approche, présentée en annexe, permet de produire, de façon systématique, des variantes d'un programme selon un modèle de faute prédéfini. A l'origine, l'analyse mutationnelle a été proposée pour déterminer la qualité d'une suite de test. Les travaux nécessaires à la mise en place de l'expérimentation (et notamment la création d'un outil de mutation adapté à Lustre) sont introduits dans ce chapitre 4. Les résultats de l'analyse montrent que l'intuition que l'on peut avoir quant a l'interprétation de ces métriques ne correspond pas aux observations faites sur la facilité de détecter des fautes [76].

Les travaux sur la validation de métriques objets ont été initiés dans le projet IMAG Contest. Ils ont donné lieu aux DEA de Boris Baldassari [12] et au doctorat de Muhammad Rabee Shaheen [210], que j'ai co-encadré. Il est fait partie des problématiques abordées dans le projet Egide PHC-Aurora (2009-2010). Mes travaux relatifs à la testabilité ont été publiés dans [12, 72, 75, 211, 212, 213, 214, 76].

Le problème de l'arrêt du test

Un dernier problème relatif au test que j'ai abordé dans mes travaux est *l'arrêt du test*. Dans l'idéal, on ne devrait s'arrêter de tester que lorsque *toutes* les fautes ont été détectées. Mais c omme on ne sait pas dire s'il reste ou non des fautes, d'autres stratégies ont été proposées [246].

On peut en particulier s'appuyer sur la notion de *critère d'adéquation*, qui caractérise les propriétés attendues de la suite de test par rapport à la structure du code ou de la spécification [25], la détection de fautes introduites volontairement [136, 60, 182], ou des hypothèses sur les défaillances observables [238, 25]. L'analyse mutationnelle appartient à la dernière catégorie de critères.

Mes travaux relatifs à l'arrêt du test ont essentiellement porté sur l'utilisation de l'analyse mutationnelle pour évaluer la qualité des données, mais aussi celle de l'oracle dans le contexte des programmes Lustre. Le principe de l'analyse mutationnelle est présenté en annexe. Ces travaux restent néanmoins anecdotiques [74, 82], c'est pourquoi ils ne font pas l'objet d'un chapitre particulier.

En résumé

Le fil directeur de mes travaux est la recherche ou l'évaluation de solutions pour l'automatisation du test. L'automatisation peut nécessiter de s'appuyer sur un nombre plus ou moins important d'informations, récoltées pendant les processus de développement et/ou de test. J'ai privilégié des approches offrant un compromis entre le degré d'automatisation souhaité et l'effort requis de la part du testeur. Dans la suite de ce document, je détaille ces travaux.

Chapitre 2

Génération de tests

Question : « Quels moyens offrir aux testeurs pour faciliter la génération de tests en l'absence de modèles ou spécification formelle ? »

Réponse : « Combiner l'exploitation des connaissances du testeur et l'exploration combinatoire. »

1. Introduction

On teste essentiellement pour *rechercher des défaillances* [173]. Et pour avoir la garantie de trouver toutes les défaillances, il faudrait tester tous les cas possibles dans toutes les situations possibles, c'est-à-dire tester *exhaustivement* le système. C'est souvent impossible car il y a trop de possibilités à envisager dans un temps raisonnable.

Une partie du travail du testeur consiste donc à *sélectionner* des données susceptibles de mettre en évidence des défaillances, voire même le plus de défaillances possibles. C'est un travail difficile car rien ne garantit que toutes les défaillances pourront être trouvées. Pour aider la génération des données de test, différentes stratégies ont été proposées. Elles s'appuient sur le code, la spécification ou les deux [246]. La plupart du temps, une stratégie s'exprime sous la forme d'un critère [32] qui concerne [246] :

– la *structure* de code ou de la spécification,
par exemple, les branches, les conditions ou les chemins du graphe de contrôle [173], ou les transitions d'un automate [25],

– la *recherche de fautes*,
on évalue la capacité d'une suite de test à détecter des fautes introduites artificiellement le plus souvent par injection de fautes [136] ou analyse de mutation [60, 182], ou

– des *hypothèses sur les erreurs observables*
ces hypothèses conduisent à créer des partitions du domaine d'entrée, comme par exemple, catégorie et partition [238] ou l'analyse aux limites [25].

Depuis une trentaine d'années, des travaux ont été menés pour l'automatisation de la génération des tests à partir du code ou de spécifications

[109, 245, 133]. En ce qui concerne la génération de données à partir du code, des approches consistent souvent en (1) l'analyse du programme pour en sélectionner des chemins d'exécution, (2) la détermination d'un ensemble de contraintes pour activer un chemin choisi et (3) l'utilisation de ces contraintes pour sélectionner des données activant le dit chemin [109, 245]. Par exemple, on peut citer les outils Euclide [125] ou PathCrawler [240] pour la génération de tests de programmes C.

Les travaux relatifs à l'automatisation de la génération à partir de modèles sont assez nombreux [56, 133]. Par exemple, les travaux [159, 134, 30, 138] sont relatifs à l'automatisation de tests à partir de spécifications Z, B ou VDM. Les travaux [37, 36, 140, 49, 234, 22, 29] sont, eux, centrés sur des modèles à base d'automates, alors que les travaux [135, 205, 34] s'intéressent plus spécifiquement à des spécifications exprimées par plusieurs vues UML.

2. Motivations

Les travaux exposés dans la suite de ce chapitre sont motivés par les deux constations suivantes.

D'une part, les outils de générations automatiques ne sont pas encore utilisés systématiquement dans l'industrie [178, 123, 127, 41]. Plusieurs raisons sont évoquées. Les générateurs à partir du code sont encore limités [240, 125]. De plus, ils ne calculent pas l'oracle. Ceux à partir de modèles nécessitent un/des modèles que beaucoup d'entreprises ne sont pas encore en mesure de fournir [178]. Ainsi, la génération des tests est encore principalement basée sur *l'expérience* du testeur [123].

D'autre part, les outils de génération automatique de tests ont souvent pour objectif de sélectionner un nombre *restreint* de tests satisfaisant un critère choisi. Or, dans un cas de figure où le coût de génération et d'exécution des tests est *marginal*, un ensemble *restreint* de tests peut apparaître insatisfaisant. C'est ce que j'ai pu observer à plusieurs reprises auprès de certains industriels [84]. La multiplication des tests est plutôt de nature à *augmenter le niveau de confiance* vis-à-vis du système sous test.

Ainsi, depuis 2000, une partie de mon travail de recherche consiste à explorer des solutions pour produire *vite* de *nombreux* tests exécutables, afin de se faire rapidement une idée du niveau de confiance que l'on peut accorder au système sous test. La solution proposée consiste à s'appuyer sur le savoir faire du testeur qui est invité à décrire la suite de test souhaitée sous la forme de *schéma*, comprenant une expression régulière décrivant les scénarios souhaités et l'espace des variations associées. La génération de la suite de test se caractérise par une exploration systématique de l'espace des tests défini par cette description abstraite.

Cette approche permet de libérer le testeur des tâches cléricales (parmi lesquelles, l'écriture du code des tests) pour se concentrer sur des tâches créatives. L'approche est particulièrement adaptée au test (unitaire ou système) d'appli-

cations pour lesquels il est possible d'automatiser *l'exécution* les tests : si de grandes suites sont produites, le testeur ne pourra pas les exécuter manuellement. Les tests doivent par ailleurs se présenter sous la forme d'un ensemble de sollicitations, éventuellement ordonné (type *séquences* d'appels de méthode).

L'approche est assez facile à mettre en œuvre car elle ne nécessite pas d'autres apports (artefacts) que la connaissance du testeur pour la *production des schémas*. Elle doit toutefois être couplée avec un oracle *automatique* pour prétendre à produire des « tests » (et pas simplement des séquences d'appels de méthodes). De plus, le couplage de Tobias à une forme de spécification exécutable permet d'identifier des séquences de test qui ne serait pas « valides », i.e. non conformes à l'usage attendu du système.

Dans la suite de ce chapitre, je détaille la solution proposée, les limites et les stratégies proposées pour les surmonter. D'abord, je décris un exemple qui sera utilisé dans la suite (section 3). Puis j'introduis les concepts de test combinatoire et l'outil Tobias (section 4). Ensuite, j'aborde les problèmes de la réduction des suites de tests (section 5, du calcul de l'oracle (section 6) Et de la validité des séquences (section 7). Enfin, je compare ces travaux avec l'existant (section 8) et je dresse quelques perspectives sur le thème de la génération de tests.

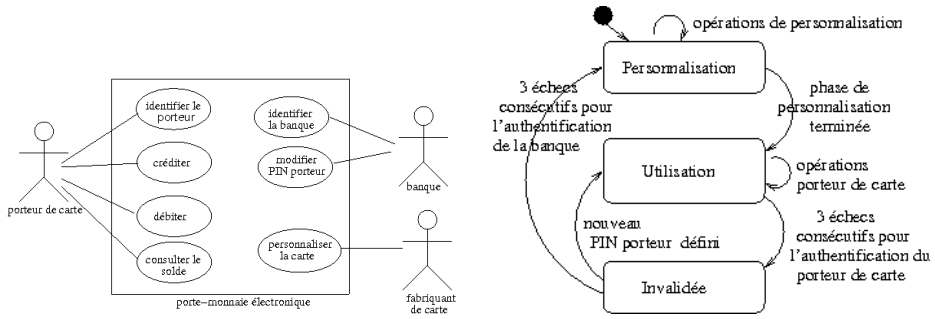


Figure 1. *Un porte-monnaie électronique*

3. Un exemple d'application à tester

Pour illustrer les stratégies de génération mises en œuvre, nous allons utiliser un exemple de porte-monnaie électronique, spécifié dans le cadre du projet RNTL POSE [53]. Cette application est destinée à être embarquée sur une carte à puce (cf. Fig. 1).

Ce porte-monnaie contient deux codes PIN, permettant d'identifier respectivement la banque et le porteur de la carte (le client). Le cycle de vie de la carte est composé de trois modes : *personnalisation*, *utilisation*, et *invalidée*. La phase de *personnalisation* permet de fixer la valeur des deux codes PIN. Cette opération s'effectue sur un terminal spécifique, dit administratif (propriété

du fabriquant de la carte). Dès que les deux codes PIN sont fixés (par les méthodes `setBpc` et `setHpc`), la carte passe en mode *utilisation*. Lorsque le porteur de carte échoue à trois authentifications consécutives, la carte est *invalidée*. La banque doit alors s'authentifier (par la méthode `authBank`) pour pouvoir ensuite modifier le code PIN du porteur. Si la banque échoue pendant son identification après trois essais, la carte repasse en mode *personnalisation*. C'est le constructeur (via son terminal administratif) qui devra refaire la procédure de personnalisation.

Lorsque la carte est en mode *utilisation*, le porteur de carte peut consulter le solde, créditer ou débiter son porte-monnaie (méthodes `getBalance`, `credit`, et `debit`). Le crédit n'est autorisé que si le porteur s'est authentifié au préalable (méthode `checkPin`). L'authentification n'est pas nécessaire pour le débit et la consultation du solde. Le solde du porte-monnaie est exprimé en centimes et est borné (par exemple à 6000 centimes). Il n'est pas possible d'effectuer un paiement si le solde du porte-monnaie est insuffisant.

On appelle *session*, une séquence d'interactions entre un terminal et la carte. Avant d'effectuer l'une des opérations ci-dessus, il faut démarrer une session (avec `beginSession`). La fermeture d'une session (par `endSession`) annule toutes les authentifications précédemment réalisées sur la carte.

On considère trois types de terminaux : administratif (`ADMIN`) , bancaire (`BANK`), et pda (`PDA`) . Tous les terminaux (`BANK`, `ADMIN` ou `PDA`) peuvent démarrer et fermer une session. Il n'est possible de définir le code PIN de la banque que sur un terminal administratif en phase de personnalisation. Le code PIN du porteur peut être défini sur un terminal administratif en phase de personnalisation, ou sur un terminal bancaire lorsque la carte est invalidée et la banque authentifiée. La vérification du code PIN du porteur n'est possible qu'à partir d'un terminal bancaire en phase d'utilisation. Le crédit ne peut s'effectuer que sur un terminal bancaire. Le débit ou la consultation peuvent s'effectuer depuis un terminal bancaire ou un pda.

Cette application a été implantée par la classe Java `Bankcardkernel`. Les méthodes de cette classe sont :

- `beginSession(Terminal su)`
- `endSession()`
- `setBpc(int pin)`
- `setHpc(short pin)`
- `authBank(int pin)`
- `checkPin(short pin)`
- `credit(short value)`
- `debit(short value)`
- `getBalance()`

Ces méthodes renvoient `void` à l'exception de `getBalance` qui retourne un entier de type `short`. La classe `Terminal.java` contient les 4 constantes sta-

tiques : `ADMIN`, `BANK`, `PDA` et `NONE`. Dans la suite, pour simplifier l'écriture des exemples, on écrira « `ADMIN` » plutôt que « `Terminal.ADMIN` ».

L'exemple ci-dessus a été proposé comme exemple de travail dans le cadre du projet RNTL POSE [53]. Il était considéré comme représentatif des applications embarquées sur carte à puce par Gemalto : les différentes méthodes peuvent être appelées à tout moment, mais leur fonctionnement dépend de l'état de la carte.

4. Test combinatoire : du concept à l'outil

Dans ce qui suit, une *suite de test* est un ensemble (ordonné ou non) de cas de tests. Un *cas de test* est une séquence d'appels de méthodes. Chaque appel de méthode a des paramètres instanciés et s'adresse à un objet/système particulier.

Au cours de mon post-doc, j'ai pu constater chez un industriel que pour générer *manuellement* une suite de tests, on pouvait procéder ainsi. Tout d'abord, une première séquence d'appel de méthodes correspondant à un comportement typique (par exemple correspondant à un cas d'utilisation) est identifiée et encodée dans un test. Puis cette séquence est reproduite en introduisant des « variations » par « copié-collé » et ajustement(s). Ce processus est ensuite répété pour chaque séquence jugée pertinente. Cette méthode de génération est longue et présente des grands risques d'erreurs.

Offrir un outil pour effectuer automatiquement la tâche d'introduction des variations va permettre à l'utilisateur de se concentrer sur la recherche de séquences pertinentes. Cela permet aussi de limiter les erreurs ou les oublis dans la construction des variations et apporte un caractère systématique à la génération.

L'idée d'un tel outil est la suivante. L'utilisateur spécifie la suite de test qu'il souhaite par (1) un « schéma » de test (sorte d'expression régulière) et (2) des éléments de variations de ce schéma. L'outil prend en charge le « dépliage » de l'expression par rapport aux éléments de variations de telle sorte que toutes les combinaisons de variations sont produites. Cette vision des choses permet au testeur d'élever le niveau d'abstraction lors de la conception de suite de test et de prendre du recul sur son activité.

4.1. *Tobias-V1*

La construction d'un tel outil a été entreprise dans l'équipe VASCO au cours du projet RNTL COTE (2000-2002). Une première version de l'outil, Tobias-V1, a été disponible en 2002 (cf. Fig. 2). Tobias-V1 est basé sur deux types de constructions : les schémas et les ensembles de méthodes instanciées. Les éléments de variation possibles portent sur la description des ensembles de méthodes instanciées et le nombre de fois où un ensemble peut être répété. Le

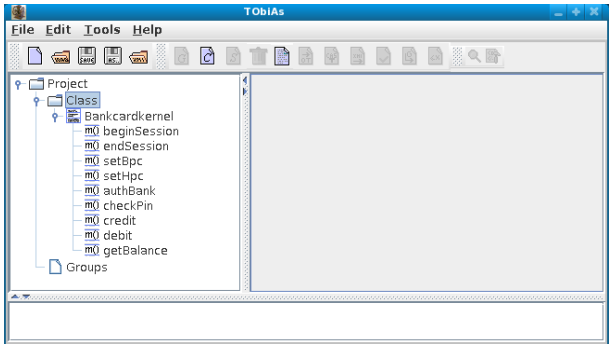


Figure 2. *L'interface de Tobias-V1*

choix des types de constructions et des espaces de variation et l'outil qui en résulte sont le résultat des thèses de O. Maury [167] et P. Bontron [27].

Illustrons l'utilisation de Tobias-V1 sur l'exemple du porte-monnaie. On souhaite créer des variations autour du scénario suivant : personnalisation de la carte, ouverture d'une session PDA, authentification de l'utilisateur (avec un code correct ou incorrect), suivi de trois opérations de crédit ou de débit et d'une consultation du solde du porte-monnaie. Cette suite permet d'explorer, de façon systématique, différents comportements relatifs à l'utilisation du porte-monnaie par le porteur de carte. Pour décrire une telle suite de test dans Tobias-V1, on peut spécifier le schéma et les deux groupes comme ci-dessous :

```
Schema  TS = {
     "PM.beginSession(ADMIN); PM.setBpc(1234); PM.setHpc(7187);
      PM.endSession(); PM.beginSession(PDA); checkP_gr ;
      UserAction_Gr^3..3;  PM.getBalance(); PM.endSession()"
  }
Group CheckP_gr = {
     void PM.checkPin(pin[1234,7187])
  }
Group UserAction = {
     void PM.credit(value[0,1000, 1500, 3000]),
     void PM.debit(value[0,1000, 2000, 3000])
  }
```

Pour bien différencier les séquences d'appels et les ensembles de méthodes, on représente les séquences entre guillemets. Ici, on a choisi 2 valeurs possibles pour le code PIN du porteur de carte et 8 valeurs possibles pour les opérations de débit et de crédit. Une fois le schéma déplié, on obtient une suite de 1024 tests (2*8*8*8).

L'outil Tobias-V1 a été utilisé avec succès dans le cadre de plusieurs études de cas, parmi lesquelles le test d'une petite application bancaire spécifiée et construite par Gemplus [80] (500 lignes de Java), et un moteur de fusion de

modalité pour une IHM (900 lignes de Java) [105]. Les deux suites de tests comportaient respectivement 1100 et 6000 cas de tests et ont chacune révélé plusieurs erreurs. Grace à son interface robuste et simple, l'outil a pu être utilisé par d'autres équipes que la nôtre.

L'approche proposée (i.e. dépliage combinatoire d'une expression) permet de produire rapidement de grandes suites de test. L'un des points forts de l'approche est l'introduction de la notion de test abstrait. Cela permet d'adapter la traduction de la suite dans la technologie cible adaptée au contexte. La rapidité d'obtention de la suite est un autre point fort de l'approche : une grande suite de test peut être définie par une expression de haut niveau. La correction ou l'évolution de la suite est effectuée par la modification de cette expression, ce qui facilite la maintenance de la suite de test.

L'approche combinatoire permet de produire de nombreux cas de test à partir d'expressions simples. L'obtention de très nombreux cas de test peut être souhaité ou non. Dans le premier cas, il faut s'assurer que l'outil permet de le faire. De ce point de vue là, Tobias-V1 était limité, car il ne permettait pas de générer des suites de test comprenant plus de 40 000 cas de test. Par ailleurs, si l'utilisateur souhaite ne conserver qu'un sous-ensemble de cas de test, il est important qu'il dispose de mécanismes permettant de *réduire la taille* d'une suite de test s'il le souhaite. L'architecture de Tobias-V1 était fort peu adaptée aux évolutions nécessaires. C'est pourquoi, Tobias a été complètement reconstruit en 2006.

4.2. *Tobias-V2*

La phase de reconstruction s'est effectuée en deux temps. Tout d'abord, les concepts clefs de Tobias ont été formalisés en Z. Cette spécification (200 lignes) a été conçue et utilisée comme un prototype (Tobias-Z) à l'aide de l'animateur Jaza [153, 154].

La possibilité d'exécuter la spécification nous a permis d'assurer une forme de non-régression. Ainsi, nous avons exprimé les schémas précédemment définis dans Tobias-V1 dans Jaza et les avons dépliés avec Tobias-Z. Nous avons comparé les suites de tests produites par Tobias-V1 et Tobias-Z et avons constaté qu'elles étaient identiques. On a ainsi pu montrer que la spécification décrivait bien les fonctions existantes de Tobias-V1.

La réflexion autour de la spécification a permis d'identifier des bons principes de Tobias-V1 et a résulté, entre autres, en l'unification des notions de groupes et de schémas. Cette analyse a été le fondement de la conception de la seconde version de Tobias (Tobias-V2).

Dans Tobias-V2, un groupe est un ensemble d'objets, de valeurs, de méthodes, de paramètres ou d'instructions. La définition d'un groupe peut être faite en extension, à l'aide d'opérateur binaire (union, intersection, différence)

ou par la concaténation séquentielle ou parallèle des éléments de deux groupes, à l'aide d'une itération ou d'une corégion[1].

Ainsi, dans Tobias-V2, il est désormais possible de décrire les suites de test avec plus de flexibilité. Pour illustrer ce point, reprenons l'exemple de notre porte-monnaie. Le schéma TS précédent peut être ré-écrit en Tobias-V2 de la façon suivante :

```
Group TS         = { "Perso ; Terminal t = PDA; AuthUser;
                     UserAction_Gr^3..3" ; PM.getBalance()" }
Group Perso      = { "PM.beginSession(ADMIN); PM.setBpc(1234);
                     PM.setHpc(7187); PM.endSession();" }
Group AuthUser   = { "PM.beginSession(t); checkP_gr ;" }
Group CheckP_gr  = { PM.checkPin(pin_val) }
Group UserAction = { PM.credit(som_val),PM.debit(som_val) }
Group pin_val = { 1234, 7187 }
Group som_val = { 0, 1000, 2000, 3000 }
```

Comme on peut le constater sur cet exemple, il est désormais possible de créer des groupes de valeurs réutilisables (`pin_val` et `som_val`), de créer des groupes dénotant des sous-séquences d'appels de méthode (`Perso`) et de combiner l'utilisation d'instructions et de groupes pour définir des séquences d'appels de méthode (`Perso` et `TS`). D'un point de vue technique, chaque groupe possède une propriété « unfold » (non représenté ici). Lorsque cette propriété est vraie, le groupe est considéré comme la définition d'une suite de test et déplié comme telle.

D'un point de vue architectural, Tobias-V2 est constitué d'un moteur de dépliage et d'une interface. Ce moteur de dépliage prend en entrée un fichier XML et produit un fichier XML de sortie pour chaque groupe à déplier. Ces fichiers de sorties représentent des suites de tests abstraites. On utilise des traducteurs XSLT pour obtenir des suites de tests exécutables dans le langage cible de son choix (Java, Java pour Junit, CPPunit, etc.). L'interface se présente comme un plug-in Eclipse. Elle permet à l'utilisateur de définir le fichier d'entrée (i.e. les groupes) et de lancer les dépliages. Elle est présentée Fig. 3. L'interface fait aussi le lien entre les suites de tests abstraites et la traduction en suites de test exécutables.

Depuis sa création, Tobias-V2 a été utilisé pour plusieurs études de cas de différentes natures : programmes Java classiques [86, 244], services iPOJO [97], applications embarquées décrites en systemC [117], site web, génération de contenus pour une base de données... A ce jour, toutes les suites de tests souhaitées ont pu être décrites à l'aide des constructions disponibles. Ceci nous conforte dans le fait que l'identification des concepts clefs a été bien faite. Le découplage entre tests abstraits et tests concrets a permis de s'adapter à différentes technologies.

1. La corégion permet de produire les permutations possibles calculées à partir d'un groupe.

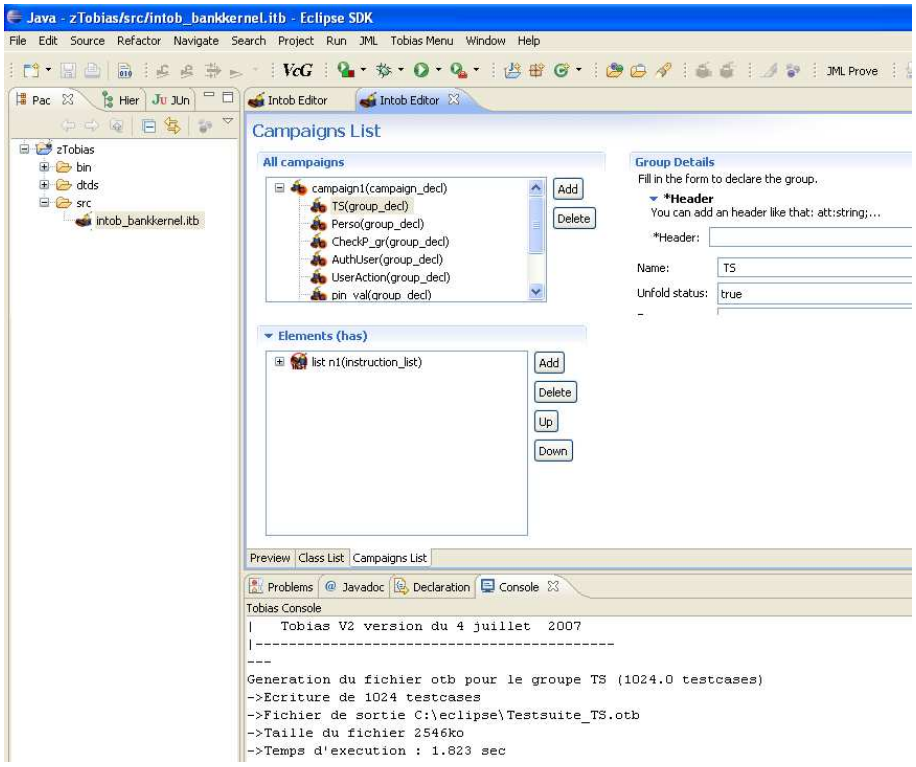


Figure 3. *L'interface Eclipse de Tobias*

D'un point de vue méthodologique, l'utilisation d'un schéma de test facilite la mise au point et la maintenance de la suite de test : les corrections se limitent au schéma et se répercutent dans les tests exécutables en quelques instants. Par exemple, si l'on souhaite modifier la suite `TS` précédente pour exécuter les opérations utilisateur sur les terminaux `BANK`, il suffit de remplacer « `PDA` » par « `BANK` » dans le groupe `TS`. Cette seule modification permettra de mettre à jour les 1024 cas de test.

Comme je l'ai dit plus tôt, la génération systématique de toutes les combinaisons d'appels de méthodes et de leurs paramètres est une des forces de Tobias. C'est aussi une de ces faiblesses, puisque très rapidement, on aboutit à une explosion combinatoire. Il faut offrir à l'utilisateur différents moyens pour sélectionner un sous-ensemble de tests.

Pour réduire la taille de la suite de test, nous avons caractérisé deux types de méthodes grâce au travail de spécification formelle de Tobias. La première méthode, appelée *filtrage*, consiste à choisir (ou non) un cas de test par rapport à un prédicat [156] (voir Annexe 2). On applique ainsi le prédicat pour chaque cas de test de la suite en cours de dépliage. Si un cas de test satisfait le prédicat, il figure dans la suite réduite. Il n'y figure pas sinon.

La seconde méthode de réduction consiste à sélectionner un sous-ensemble de cas de test de telle sorte que le sous-ensemble satisfasse une propriété. Un *sélecteur* peut par exemple, être défini pour extraire un sous-ensemble de cas de test garantissant l'obtention d'un niveau de couverture.

L'architecture ouverte de Tobias-V2 permet de définir des sélecteurs et des filtres de son choix sous la forme de programmes Java [152]. La suite de cette partie décrit différentes stratégies de réduction, qui ont été ou qui peuvent être implantées par des filtres ou des sélecteurs.

5. Des stratégies de réduction de la suite de test

Les stratégies de réduction consistent à sélectionner un sous-ensemble de cas de test à partir d'un ensemble déterminé. Différents types de critères peuvent être étudiés. Dans tous le cas, le problème réside ici dans le fait qu'en sélectionnant un sous-ensemble de cas de test, on risque de ne pas mettre en évidence des défaillances qui aurait pu l'être avec les cas de tests non sélectionnés. Plus les fautes sont difficiles à détecter, plus ce risque est grand.

La stratégie la plus élémentaire pour sélectionner un sous-ensemble de cas de test consiste en une sélection aléatoire [173]. Cette stratégie a souvent donné de très bons résultats en comparaison avec des méthodes sensées être plus évoluées [106, 129, 10]. C'est pourquoi, les techniques de réductions doivent toujours être évaluées par rapport à une stratégie aléatoire [9, 8].

Dans la suite, on présente une implémentation de stratégie aléatoire élémentaire et une autre guidée. Puis une stratégie basée sur la couverture de paires d'appel de méthode et d'autres stratégies envisagées.

5.1. *Réduction aléatoire (équilibrée ou non)*

Une stratégie élémentaire de réduction consiste en une sélection aléatoire d'un sous-ensemble de cas de test de la suite originale. Cette stratégie a été implantée par un « sélecteur ». L'algorithme `randomSelect` (Fig. 4) prend en entrée la suite de test originale `FullTS` et `size`, la taille de la suite attendue. Le résultat de l'algorithme est une suite de test de taille réduire (`RedTS`). Cette dernière est construite en sélectionnant un cas de test au hasard dans la suite originale autant de fois que nécessaire, en veillant à ne pas choisir deux fois le même.

Une version améliorée de l'algorithme `randomSelect` permet de ne pas déplier la suite originale. L'algorithme `randomSelectBis` prend en entrée la taille de la suite originale et la taille de la suite réduite. Il retourne un ensemble d'entiers correspondant aux numéros des cas de tests devant être sélectionnés. Tobias ne déplie alors que ces cas de test. On exploite ici le fait que Tobias déplie les groupes de façon déterministe. De ce fait, les suites de tests produites sont ordonnées.

```
RedTS ← randomSelect(FullTS, size)
begin
   if (size > card(FullTS)) then
      return FullTS
   end if
   RedTS = ∅
   while (size ≥ 0) do
      rand = random(0, card(FullTS))
      RedTS = RedTS ∪ {FullTS[rand]}
      FullTS = FullTS \ {FullTS[rand]}
      size = size - 1
   done
   return RedTS
end
```

Figure 4. *Un algorithme de réduction aléatoire de suite de test*

Une telle stratégie présente les avantages et les inconvénients d'une approche aléatoire. Elle est simple à mettre en œuvre mais ne garantit pas une répartition intéressante des cas de tests sélectionnés. Soit l'exemple suivant :

```
TS2      = { "GrPerso; GrCheckP; GrModify;" }
GrPerso  = { "PM.beginSession(ADMIN); PM.setBpc(1234); PM.setHpc(7187);
              PM.endSession();"}
GrCheckP = { "PM.beginSession(PDA); PM.checkPin(7187);" }
GrModify = {  PM.debit(GrValDeb)^3, PM.credit(1000)^3, PM.getBalance() }
GrValDeb = { 0,-1, 1, 1000, 32768, 3000 }
```

La suite définie par `TS2` comprend 218 cas de test. Si l'on observe les cas de tests, on constate que l'on peut les répartir dans trois sous-ensembles selon la suite des noms des méthodes.

```
a. Prefix; debit; debit; debit
b. Prefix; credit; credit; credit
c. Prefix; getBalance
```

Ces ensembles correspondent à une abstraction des comportements de l'utilisateur (les paramètres des méthodes sont abstraits). Il y a 216 cas dans le premier sous-ensemble contre un seul pour chacun des deux autres. En utilisant une stratégie purement aléatoire, il y a une faible probabilité qu'un des cas de test issu de (b) ou (c) soit choisi.

Pour remédier à ce problème, nous avons proposé une stratégie de sélection aléatoire *équilibrée* qui, pour une taille de suite réduite fixée `size`, et une relation d'abstraction φ, impose que la suite résultat possède un représentant au moins dans chaque sous-ensemble défini par φ (si `size` est supérieur au nombre de sous-ensembles définis par φ).

Soit l'algorithme `directedRandomSelect` retournant la suite de test réduite `RedTS` (Fig. 5).

Cet algorithme prend trois paramètres en entrée : `FullTS` la suite de test originale, `size` la taille de la suite réduite et φ, une fonction d'abstraction permettant la répartition des cas de test dans des classes d'équivalences. L'algorithme `RedTS` calcule d'abord une partition des cas de test par rapport à la fonction d'abstraction φ. Ensuite, des cas de test sont tirés dans chaque partition proportionnellement à la taille de la partition ; au moins une si `size` le permet. Lorsque le tirage dans chaque partition est effectué, il est possible que la suite réduite `RedTS` n'atteigne par la taille `size` souhaitée, du fait des erreurs d'arrondi. La suite est alors complétée par un tirage aléatoire parmi les tests non encore sélectionnés.

L'algorithme `directedRandomSelect` garantit que la suite finale comportera un représentant de chacune de ces classes d'équivalences, si la taille de la suite le permet. Les relations d'abstraction sont laissées à la discrétion du testeur. La principale relation d'abstraction que nous avons utilisée est cette décrite par l'exemple ci-dessus. Les valeurs de paramètres sont abstraites et seuls les noms de méthodes sont conservés. Les algorithmes de sélection aléatoire basique et équilibrée sont détaillés et illustrés dans [53].

```
RedTS ← directedRandomSelect(FullTS, size, φ)
begin
   if (size > card(FullTS)) then
      return FullTS
   end if
   RedTS = ∅
   PartTS = φ(FullTS)
   nb = size div card(PartTS)
   rem = size div card(PartTS)
   foreach (Part ∈ PartTS) do
      RedTS = RedTS ∪ randomSelect(Part,nb)
      if nb > card(Part) then
         rem = rem + (nb - card(Part))
      end if
   done
   if rem > 0 then
      Red2 = randomSelect(FullTS\RedTS,rem)
      RedTS = RedTS ∪ Red2
   end if
   return RedTS
end
```

Figure 5. *Un algorithme de réduction aléatoire équilibrée de suite de test*

Les suites de tests produites par l'algorithme `directedRandomSelect` ont été comparées aux suites de test produites aléatoirement. Lors d'une première évaluation, ont été évaluées la couverture du code et le score de mutation

obtenu pour des suites produites par les deux algorithmes. Pour l'exemple de BankcardKernel, 6 schémas, dont `TS2`, ont été réduits à 3, 5, 10, 20 et 30% de la taille de la suite originale définie par le schéma avec `directedRandomSelect` et `randomSelect`. La relation d'abstraction utilisée est celle présentée précédemment (abstraction des valeurs de paramètres). Les résultats obtenus montrent que les suites réduites avec `directedRandomSelect` permettent d'obtenir au moins le même résultat que les suites produites aléatoirement en termes de taux de couverture d'instruction et de détection de fautes. Les suites produites par `directedRandomSelect` donnent de meilleurs résultats dans le cas où les fautes ne sont détectables que par l'exécution des cas de test issus des classes d'abstraction ayant peu de séquences de test [53].

5.2. *Couverture des paires d'appels de méthodes*

Depuis le début de ce chapitre, nous nous sommes placés dans un cadre où les cas de tests sont des appels de séquence. Tobias cherche à générer toutes les combinaisons de tous les appels de méthodes (définies par un schéma et ses variations). Cela correspond à une stratégie de couverture de tous les chemins satisfaisant le schéma.

Une question est de savoir s'il est nécessaire de couvrir toutes les combinaisons d'appel de méthodes ou si l'on peut se contenter de couvrir un *sous-ensemble* de ces combinaisons. Dit autrement, les défaillances mises en évidence par un cas de test sont elles dues à la *séquence d'appels dans sa totalité* ou à un sous-ensemble particulier d'appels dans la séquence ?

Dans la littérature, une approche de réduction d'une suite de test combinatoire a été proposée pour le test de méthodes/fonctions ayant plusieurs paramètres. Cela consiste à sélectionner un ensemble de valeurs qui assure une couverture par paires des valeurs de paramètres [126, 180]. Cette stratégie garantit que chaque combinaison de deux valeurs de deux paramètres est couverte, sans assurer la combinaison de tous les paramètres. La stratégie s'applique également pour les triplets, quadruplets et autres n-uplets. L'application d'un critère de couverture par paires permet de réduire considérablement la taille d'une suite de test. Par exemple, soit une méthode de 4 paramètres, chacun devant être testé avec 4 valeurs différentes. La combinatoire totale des valeurs correspond à 4^4=256 combinaisons. Mais il suffit de 16 n-uplets (bien choisis) pour assurer la couverture par paires de paramètres (cf. Fig. 6).

Le critère de couverture par paires vise à garantir l'identification des défaillances dues à des combinaisons spécifiques de valeurs de paramètres ou de configurations [239]. Cela repose sur le constat que la plupart des défaillances sont dues à une combinaison restreinte de valeurs de paramètres. Il est donc inutile de tester l'ensemble des combinaisons [180].

Le problème du calcul d'un ensemble minimum couvrant ces combinaisons est NP-complet. Toutefois, plusieurs outils proposent des heuristiques comme AETG [50], ou ACTS [221, 160]. Pour calculer une suite de test assu-

#	ρ_1	ρ_2	ρ_3	ρ_4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	1	4	4	4
5	2	1	2	3
6	2	2	1	4
7	2	3	4	1
8	2	4	3	2
9	3	1	3	4
10	3	2	4	3
11	3	3	1	2
12	3	4	2	1
13	4	1	4	2
14	4	2	3	1
15	4	3	2	4
16	4	4	1	3

Figure 6. *Exemple de couverture par paires pour 4 facteurs de 4 valeurs*

rant une couverture par paires, la démarche classiquement utilisée s'appuie sur la notion de matrice de couverture [126, 239, 221, 160, 180]. Soit

– $F = \{\rho_1, \rho_2, \ldots \rho_k\}$, un ensemble de k facteurs indépendants, correspondant aux paramètres d'une méthode ou aux options déterminant une configuration d'un logiciel.

– V_i, l'ensemble des valeurs possibles du facteur ρ_i,

– $n_i = |V_i|$, le nombre de valeurs possibles pour le facteur i,

– T, l'ensemble des cas de tests ou des configurations possibles ($|T| = \prod_{i=1}^{k} n_i$).

Soit R, les sous-ensembles de facteurs devant être couverts. R représente les *exigences de couverture* [232] ($R \subset 2^F$, où 2^F représente l'ensemble des parties de F). Lorsque R contient tous les sous-ensembles d'une taille $\tau \leq k$ et rien d'autre (R_τ), on parle de τ-*way testing*. On parle de *combinatoire par paire* lorsque $\tau = 2$ (R_2).

Une matrice de couverture (CA pour *Covering Arrays* peut s'exprimer de la façon suivante[2] : $CA(N, k, F, R)$ est une matrice de taille $N \times k$ ayant la propriété que pour un sous-ensemble de facteurs donné de R ($\{\rho_{i_1}, \rho_{i_2}, \ldots, \rho_{i_\tau}\} \in R$), chaque combinaison d'instanciation possible des facteurs $(a_{i_1}, a_{i_2}, \ldots, a_{i_\tau}) \in V_{i_1} \times V_{i_2} \times \ldots \times V_{i_\tau}$ est présent dans la sous-matrice $N \times \tau$ correspondante.

2. Dans les formalisations proposées dans la littérature, N représente le nombre de tests représentés dans la matrice. Dans le cas où l'on vérifie que R est couverte, N est connu. Dans le cas où l'on cherche à générer une suite de test, N est déterminé par l'algorithme calculant la suite de test.

Ainsi, lorsque l'on considère R_2, toute paire de colonnes doit contenir l'ensemble des combinaisons des deux facteurs correspondants (comme c'est le cas Fig. 6).

L'application de la notion de couverture par paires pour les appels de méthodes n'avait pas encore été proposée à ma connaissance. Dans ce qui suit, considérons un cas de test comme une séquence de facteurs. Chaque facteur ρ_i représente les appels de méthodes possibles à la $i$ème position i dans le cas de test. Les valeurs associées à un facteur sont les instanciations possibles des appels de méthode.

```
TS3      = { "GrPerso; GrCheckP; GrModify; GrModify; GrModify;" }
GrPerso  = { "PM.beginSession(ADMIN); PM.setBpc(1234); PM.setHpc(7187);
               PM.endSession();"}
GrCheckP = { "PM.beginSession(PDA); PM.checkPin(7187);" }
GrModify = { PM.credit(GrVal), PM.debit(GrVal), PM.getBalance() }
GrVal    = { 1000, 3000 }
```

Considérons la suite de test `TS3`. Elle est définie par 9 facteurs ($\{\rho_1, \rho_2, ..., \rho_9\} = F'$) car chaque cas de tests comporte 9 appels de méthodes). Les 6 premiers facteurs ont une seule valeur possible. Par exemple, on a
V_1 ={ `PM.begin-Session(ADMIN)` } et V_6 ={ `PM.checkPin(7187)` }. Les 3 derniers facteurs ont le même espace de valeur : $V_7 = V_8 = V_9$ ={ `PM.credit(1000), PM.credit(3000), PM.debit(1000), PM.debit(3000), PM.getBalance()`}.

Le schéma `TS3` définit une suite comportant $5 * 5 * 5 = 125$ cas de test au total. Les facteurs ont été soumis à l'outil ACTS [221, 160], qui en réponse a généré une matrice de couverture CA=(31,9,F',R_2). Elle a été transformée en la suite de test donnée Fig. 7.

La stratégie de réduction des suites de test basée sur les paires de méthodes est en cours expérimentation. L'idée consiste à produire des schémas de test de type
`TS`^{n} `: "appelAunContructeur ; appelDeMethodeInstancié`^{n}`"`, et de les déplier de façon combinatoire et en appliquant le critère de couverture des paires d'appels de méthode. En appliquant le critère de couverture sur les paires d'appels de méthode, il est possible d'allonger considérablement la longueur des cas de test tout en conservant un nombre de d'appels de méthode comparable. Or ce n'est pas possible avec le dépliage combinatoire qui ne permet de déplier que des schémas `TS`^{n} avec un n limité.

Pour évaluer la pertinence du dépliage par paires, j'ai mené une expérimentation qui consiste à comparer la capacité des suites `TS`^{n} dépliées par paires à détecter des fautes en comparaison avec des suites de même taille produites de par sélection aléatoire des cas de test dans le schéma. Les fautes sont introduites par une analyse mutationnelle (cf. annexe 1).

A ce jour, l'expérimentation porte sur dix classes Java. Elles ont comme caractéristique d'avoir un état (représenté par un ses attributs). Des fautes

```
 1 init ; PM.credit(1000); PM.credit(1000); PM.credit(1000);
 2 init ; PM.credit(1000); PM.credit(3000); PM.credit(3000);
 3 init ; PM.credit(1000); PM.debit(1000) ; PM.debit(1000) ;
 4 init ; PM.credit(1000); PM.debit(3000) ; PM.debit(3000) ;
 5 init ; PM.credit(1000); PM.getBalance(); PM.getBalance();
 6 init ; PM.credit(3000); PM.credit(1000); PM.credit(3000);
 7 init ; PM.credit(3000); PM.credit(3000); PM.credit(1000);
 8 init ; PM.credit(3000); PM.debit(1000) ; PM.debit(3000) ;
 9 init ; PM.credit(3000); PM.debit(3000) ; PM.debit(1000) ;
10 init ; PM.credit(3000); PM.getBalance(); PM.getBalance();
11 init ; PM.debit(3000) ; PM.credit(1000); PM.debit(3000) ;
12 init ; PM.debit(1000) ; PM.credit(3000); PM.debit(3000) ;
13 init ; PM.debit(1000) ; PM.debit(1000) ; PM.credit(1000);
14 init ; PM.debit(1000) ; PM.debit(3000) ; PM.credit(3000);
15 init ; PM.debit(1000) ; PM.getBalance(); PM.getBalance();
16 init ; PM.debit(3000) ; PM.credit(1000); PM.debit(3000) ;
17 init ; PM.debit(3000) ; PM.credit(3000); PM.debit(1000) ;
18 init ; PM.debit(3000) ; PM.debit(1000) ; PM.credit(3000);
19 init ; PM.debit(3000) ; PM.debit(3000) ; PM.credit(1000);
20 init ; PM.debit(3000) ; PM.getBalance(); PM.getBalance();
21 init ; PM.getBalance(); PM.credit(1000); PM.getBalance();
22 init ; PM.getBalance(); PM.credit(3000); PM.credit(1000);
23 init ; PM.getBalance(); PM.debit(1000) ; PM.credit(3000);
24 init ; PM.getBalance(); PM.debit(3000) ; PM.debit(1000) ;
25 init ; PM.getBalance(); PM.getBalance(); PM.debit(3000) ;
26 init ; PM.getBalance(); PM.getBalance(); PM.credit(1000);
27 init ; PM.getBalance(); PM.getBalance(); PM.credit(3000);
28 init ; PM.getBalance(); PM.getBalance(); PM.debit(1000) ;
29 init ; PM.getBalance(); PM.credit(3000); PM.getBalance();
30 init ; PM.getBalance(); PM.debit(1000) ; PM.getBalance();
31 init ; PM.getBalance(); PM.debit(3000) ; PM.getBalance();
```

où `init` correspond à la sous-séquence : `PM.beginSession(ADMIN) ; PM.setBpc(1234) ; PM.setHpc(7187) ; PM.endSession() ; PM.beginSession(PDA) ; PM.checkPin(7187) ;`

Figure 7. *Une séquence de test réduite produite à partir de* `TS3`

sont introduites avec MuClipse [218]. Les schémas de tests *TS* ont été construits semi-automatiquement : les méthodes ont été extraites automatiquement, les valeurs des paramètres ont été déterminées manuellement. Différentes suites de tests TS^n ont été considérées avec n variant de 3 à 6 (au moins). Les premiers résultats montrent qu'à nombre de cas de test comparable (suite aléatoire et suite dépliée par paires), les suites dépliées par paires tuent plus de mutants et de façon plus systématique. Toutefois, l'écart est très faible entre les 2 modes de génération. Ceci peut s'expliquer par le fait que, d'une façon générale, les suites produites aléatoirement couvrent une forte proportion des paires [10].

5.3. *D'autres stratégies de réduction*

D'autres techniques de réduction de suites de tests peuvent être envisagées. Elles s'appuient sur le choix d'un sous-ensemble de cas de test, en garantissant l'obtention d'un niveau de couverture [131]. On peut ainsi chercher à garantir une couverture *structurelle* (instructions, branches, ...) ou un score de mutation.

Ces stratégies de réduction nécessitent de calculer et d'exécuter la suite originale avant d'effectuer la réduction. Ce n'est pas toujours possible si la suite est très grande ou si l'exécution de la suite originale sur la cible à tester est trop chère. Toutefois, une telle approche pourrait être envisagée pour la validation d'applications embarquées sur carte. La couverture de la suite originale pourrait être évaluée sur un programme instrumenté hors carte, sur un simulateur (alors que l'exécution de la suite est moins coûteuse). Puis, la suite *réduite* pourrait être exécutée sur l'application cible (avec un banc de test). L'applicabilité de cette approche est déterminée par les points suivants : (1) taille de la suite initiale et son temps d'exécution sur le simulateur, (2) taille de la suite réduite et son temps d'exécution sur le banc. L'approche n'aura de sens que si (1) le temps de production et d'exécution de la suite réduite est inférieur (ou comparable) à celui d'une autre stratégie et (2) si les cas de tests sélectionnés sont de qualité comparable (fautes détectées) à des tests aléatoires [8].

6. Du problème de l'oracle

Être capable d'aider à la génération des données est une chose. Juger de la correction du système sous test à l'exécution des tests en est une autre. Dans la littérature, ce dernier point est appelé le « problème de l'oracle ». Dans la mythologie grecque, l'oracle est la réponse donnée par un dieu que l'on a consulté à une question personnelle, concernant généralement le futur. Dans le contexte du test logiciel, on appelle « oracle » une procédure manuelle ou automatique qui permet de fournir un verdict lors de l'exécution de test.

L'oracle est souvent une procédure manuelle [123, 127, 178]. En soit, ceci n'est pas étonnant. La production d'un oracle automatique nécessite un effort important de conception et de réalisation. Cet effort a un sens lorsqu'au final cela coûte moins cher que l'utilisation d'une procédure manuelle [194], ce qui peut être le cas lorsque les tests sont exécutés à plusieurs reprises. Ainsi, Odile Laurent, lors de la conférence ICST 2010 souligne que, chez Airbus, le travail d'analyse de trace pour y détecter des défaillances est manuel, même si une spécification formelle est disponible [107].

La principale difficulté relative à l'oracle est la capacité à exprimer de façon correcte et suffisamment précise le résultat attendu. Dans une perspective où l'on cherche à produire des tests de façon massive, on ne peut pas s'attendre à ce que les résultats de l'exécution des milliers de tests puissent être examinés manuellement. Il est donc important d'examiner des stratégies d'automatisation de l'oracle. Une stratégie consiste en l'utilisation d'assertions dans les

schémas. Une autre consiste à déduire l'oracle d'une spécification lorsque celle-ci est disponible. Cette spécification peut se présenter de différentes façons : insérée dans le code, par exemple par des contrats, ou sous la forme d'un artéfact indépendant du code.

6.1. *Des assertions dans des schémas*

Le mécanisme d'assertion permet au programmeur de vérifier dynamiquement des conditions. En Java, les instructions de type « assert » permettent d'exprimer ces conditions [195]. Ainsi, si l'on souhaite s'assurer que la condition C est vraie à un endroit particulier du programme, on y insère l'instruction « `assert(`C`)` ». Si, à l'exécution, la condition n'est pas vérifiée, une exception est levée.

Puisque dans notre cas, les tests sont aussi des programmes, il est possible d'utiliser le principe d'assertion. D'ailleurs, plusieurs variantes d'assertions sont proposées au sein de JUnit. Par exemple, « `assertEquals(`a`,`b`)` » permet de comparer l'égalité de deux variables et « `assertNotNull(`o`)` » permet de s'assurer qu'un objet est non nul. Si une telle assertion est violée à l'exécution, une exception est levée et JUnit émet un verdict d'échec pour le cas de test.

Tobias étant un outil très souple, il est possible de décrire explicitement des instructions au sein d'un schéma. Cette possibilité permet, entre autres, d'insérer des assertions Java au cœur d'un schéma. Par exemple, `TS4` défini ci-dessous, est une extension de `TS3` qui permet de s'assurer qu'à la fin de chaque cas de test, le solde de la carte est positif ou nul[3].

```
TS4      = { "TS3; assertTrue(PM.getBalance()>=0) ;" }
TS3      = { "GrPerso; GrCheckP; GrModify; GrModify; GrModify;" }
GrPerso  = { "PM.beginSession(ADMIN); PM.setBpc(1234); PM.setHpc(7187);
               PM.endSession();"}
GrCheckP = { "PM.beginSession(PDA); PM.checkPin(7187);" }
GrModify = { PM.credit(GrVal), PM.debit(GrVal), PM.getBalance() }
GrVal    = { 1000, 3000 }
```

L'utilisation d'assertions dans un schéma est parfois complexe. En effet, un schéma peut regrouper des cas de test qui impliquent respectivement l'observation d'une même condition à vrai ou à faux. Si l'on veut pouvoir exprimer des oracles détaillés, il faut prendre en compte le comportement spécifique de chaque cas de test et récolter les informations nécessaires à l'établissement du verdict aux travers de variables intermédiaires. Cette solution n'est pas complètement satisfaisante dans un contexte où l'on cherche à faciliter le test au maximum. J'ai donc étudié la possibilité d'utiliser une spécification exécutable comme oracle du test. La section suivante décrit le principe général d'utilisation d'une spécification pour calculer l'oracle. Le chapitre 3 décrit

3. La méthode `getBalance()` retourne un entier de type `short`.

plus précisément l'exploitation de spécifications sous forme d'assertions (JML) comme oracle de test.

6.2. *De l'utilisation d'une spécification formelle pour calculer l'oracle*

Lorsque l'on dispose d'une spécification formelle, il est en principe possible de l'utiliser pour calculer l'oracle des tests. Jusqu'à présent, j'ai étudié deux types de spécifications. L'une se présente sous la forme d'assertions exécutables insérées dans le code. L'autre se présente sous la forme d'un artéfact indépendant du code.

Pour calculer l'oracle à partir d'assertions exécutables insérées dans le programme (invariants, pré et post-conditions), on observe l'exécution du programme. Si des données mettent en évidence une violation des assertions, une exception est levée et un verdict d'échec est posé. Un verdict de réussite est posé sinon.

L'utilisation d'une spécification exprimée sous forme d'assertions est particulièrement bien adaptée à Tobias. En effet, la détermination de la suite des séquences d'appels de méthodes est générée dans Tobias. La spécification est exprimée de façon indépendante. Pendant ces dernières années, j'ai mené un travail important quant à l'évaluation de l'utilisabilité des assertions dans un contexte domotique. Ce travail est détaillé au chapitre suivant.

Lorsque la spécification se présente sous la forme d'un artéfact indépendant du code, les séquences de tests doivent être « jouées » sur la spécification. Selon la nature de la spécification, « jouer » signifie « simuler » ou « exécuter ». Les réponses aux simulations observées ou calculées à partir de la spécification sont alors utilisées comme oracle, et comparées aux réponses du système sous test à l'exécution.

La difficulté ici réside principalement dans le fait qu'une telle spécification peut se présenter sous une forme plus abstraite que l'implantation finale. Dans ce cas, les tests calculés au niveau de la spécification ne seront pas directement exécutables sur l'implantation. Une étape de « concrétisation » des tests est alors nécessaire.

Tobias n'a pas été initialement prévu pour calculer l'oracle à partir d'une spécification. Dans le cas d'une spécification indépendante, Tobias doit être couplé avec un autre outil pour permettre le calcul des réponses attendues. Ceci a partiellement été étudié dans le cadre du projet COTE, où Tobias a été couplé avec UMLAUT/TGV [27, 167]. Elle est actuellement en cours d'étude dans le projet TASCCC, où Tobias est couplé avec l'outil TestDesigner de SmartTesting. Dans ce cas précis, la spécification se présente sous la forme d'un modèle UML/OCL, plus abstrait que l'implantation finale. Les schémas de tests sont exprimés au niveau d'abstraction que le modèle UML. L'animation des séquences produites par Tobias est prise en charge par TestDesigner pour le calcul de l'oracle, le filtrage des séquences invalides et la concrétisation des

séquences. L'évaluation de cette approche est une perspective à très court terme de mon travail.

7. La validité des séquences

La plupart des systèmes sous test ont un comportement contraint. Ils n'acceptent pas toutes les entrées possibles dans toutes les situations possibles. L'ensemble des comportements qui doivent être acceptés par l'application sous test est décrit dans le cahier des charges et/ou dans la spécification du système.

Lorsque le testeur décrit ses schémas de test, il est de sa responsabilité de s'assurer que les cas de tests produits sont « valides », c'est-à-dire qu'ils correspondent aux comportements qui doivent être testés[4]. Plus le schéma sera général, plus la suite de test sera de grande taille et plus le risque de produire des cas de tests invalides est grand.

La difficulté ici est que l'exécution de cas de tests invalides peut entraîner la production d'un verdict incorrect. En effet, si une défaillance est produite par un cas de test invalide, le système sous test n'est pas (forcément) incorrect puisqu'il a été sollicité en dehors du comportement qui a été spécifié. On parle de « faux négatif » lorsque les tests concluent à une défaillance alors qu'il n'y en a pas.

Pour éviter les faux négatifs, il faut donc s'assurer que le cas de tests qui l'a mis en évidence une défaillance est « valide ». Tobias n'est pas conçu pour effectuer une telle vérification. La phase de génération des tests avec Tobias peut toutefois être couplée avec d'autres outils permettant d'éliminer les faux négatifs.

Ainsi, lorsque l'on dispose d'une spécification formelle, il est envisageable de filtrer les séquences proposées sur la base de la spécification, pour ne conserver que celles qui sont conformes à la spécification. Jusqu'à présent, nous avons essentiellement étudié des systèmes spécifiés par des assertions exécutables. La détection des séquences invalides se fait alors à l'exécution de la suite de test. Les cas où le système sous test est sollicité en dehors de ses pré-conditions sont identifiés comme un cas particulier d'exception. A l'exécution d'un cas de test, le verdict est calculé ainsi. On obtient `PASS` en l'absence d'exception levée. On obtient `INCONCLUSIF`, lorsqu'une exception relative à la satisfaction d'une pré-condition du système est levée. On obtient `FAIL` lorsqu'un autre type d'exception est levé. La façon de calculer le verdict avec des assertions est détaillé dans le chapitre 2.

---

4. Le test de robustesse s'intéresse à la génération de tests dehors de la spécification « fonctionnelle ». L'ensemble des comportements robustes qui sont supportés par la spécification doit aussi être spécifié. Il détermine les limites des propriétés de robustesses. Des comportements autres que ceux décrits par les spécifications fonctionnelles et de robustesse sont alors considérés comme invalides.

Une autre façon de gérer les cas de tests invalides consiste à utiliser la spécification pour filtrer les cas de test, pour ne conserver que les valides. Cette possibilité est en cours d'étude dans le projet TASCCC, où la spécification est décrite en UML et OCL. La détermination de la validé de la séquence est opérée par la collaboration de Tobias et de l'outil Test Designer. Le principal défi ici est d'ordre opérationnel : il faut s'assurer que la connexion des outils permet le filtrage des cas de test en un temps raisonnable.

8. Des travaux similaires

En ce qui concerne le test combinatoire, on peut identifier deux types de travaux. Tout d'abord, de nombreux travaux s'intéressent à la production d'algorithmes pour optimiser la génération de données satisfaisant la couverture des paires ou n-uplets (pour les configurations ou les paramètres de méthodes) [180]. Ceci se situe en dehors du périmètre de mes travaux. Je suis utilisatrice et non contributrice. C'est pourquoi, ces travaux ne sont pas décrits ici.

Un autre type de travaux concerne la génération de cas de test. De ce point de vue, Tobias figure dans les précurseurs, parmi JML-JUnit et Korat. JML-JUnit calcule des suites de test dont chaque cas test comporte un préfixe (unique) suivi d'un seul appel de méthode [45]. Le calcul des instanciations possibles pour le dernier appel de méthode est combinatoire. La combinatoire porte sur le choix des méthodes et des valeurs paramètres.

Korat est un outil de génération combinatoire d'objets Java. Par exemple, pour tester une méthode qui enlève un élément d'un arbre binaire, il est important d'être en mesure d'exhiber des arbres binaires de structures différentes. Korat aborde ce problème. L'outil s'appuie sur un prédicat qui définit les propriétés que doit satisfaire l'objet sous test. Korat produit de façon exhaustive tous les objets (non isomorphes) satisfaisant le prédicat. La génération est limitée par la taille des objets produits. Le prédicat est calculé à partir de la spécification associée à la classe sous test, notamment invariants, pré et post-conditions.

L'originalité de l'approche (et de Tobias) est de permettre à l'utilisateur de définir la suite de test attendue à partir d'une description abstraite (le schéma). Cette idée a depuis inspiré d'autres travaux.

– Ainsi, un plug-in de test combinatoire a été introduit dans l'environnement Overture pour VDM [204]. Ce plug-in permet de décrire des scénarios sous la forme d'expressions régulières (proches des schémas de Tobias). La suite de test est ensuite dépliée et peut être filtrée sur la base de la spécification VDM pour éliminer les tests non pertinents (menant à un verdict `Fail` ou `Inconclusif`) et calculer l'oracle.

– Trusted-Labs propose un environnement de test `TL-CAT` incluant le principe de définition d'un scénario qui est déplié selon un ensemble de variations[5].

– JSynoPSys, un outil de test pour B, intègre une notion de scénarios qui peuvent être dépliés à la façon de Tobias. Les scénarios peuvent représenter une séquence incomplète et l'outil s'appuie sur le modèle B pour déterminer les actions ou les valeurs manquantes, en vue d'atteindre un objectif fixé [55].

L'approche combinatoire a aussi été utilisée pour décrire l'oracle des tests pour des logiciels réactifs synchrones. L'idée est ici de faciliter l'analyse des traces pour les tests menés par Airbus, qui est habituellement un travail manuel au sein de cette entreprise. G. Durieu *et al.* ont observé que l'analyse devait être menée pour des propriétés similaires avec des variations de paramétrage. C'est pourquoi, pour aider les testeurs dans l'analyse de traces, l'outil LETO a été défini [107]. Il permet d'exprimer les différents objectifs de tests sous la forme de *schémas de test génériques* et de variations, appelées *schémas d'instanciation*.

L'ensemble de ces travaux similaires me confirme que la direction de recherche choisie il y a 10 ans était raisonnable et qu'elle doit être poursuivie. Les différentes perspectives qui s'ouvrent sont décrites ci-après.

9. Conclusion et perspectives

Pour répondre au problème de la sélection des données de test, j'ai exploré une approche combinatoire depuis plus de 10 ans [84]. Le principe vise à capturer les connaissances du testeur aux travers d'expression de très haut niveau (appelés schéma), qui sont ensuite dépliées de façon automatique pour produire des suites de test exécutables. L'humain est déchargé des tâches répétitives et sans valeurs ajoutées, et peut ainsi se concentrer sur la sélection des scénarios. Cette approche a été outillée et utilisée au cours de différentes études de cas, issus, entre autres de collaborations et de projets.

Ces travaux ont été menés en collaboration avec Yves Ledru et Catherine Oriat, dans le cadre des projets RNTL COTE (2000-2002), ANR POSE (2006-2007) et ANR TASCCC (2009-2012). Ils ont été au cœur des travaux de doctorats de Pierre Bontron que j'ai co-encadré et de Olivier Maury que j'ai suivi [27, 167], des travaux de M2R de Taha Triky et Azzédine Amiar [224, 6], du mémoire d'ingénieur CNAM de Sébastien Ville, du stage d'ingénieur de Elodie Rose. Ils ont été publiés dans [83, 156, 105, 153, 152, 117, 53, 154]. Ils ont inspiré des approches et outils similaires [204, 55]

Ma principale perspective relative à la *génération des tests* est relative à la conception des schémas de test. A l'heure actuelle, la capture de la connaissance du testeur se fait grâce à une action volontaire. Le testeur doit exprimer ses connaissances par un ou plusieurs schémas. Le défi consiste à exploiter d'autres sources d'information pour produire des schémas de façon plus automatique, et qui pourront être utilisés en complément de ceux fournis par le testeur.

5. http ://www.trusted-labs.com/spip.php ?rubrique48

J'ai identifié deux stratégies deux types de stratégies relatives à la production de schémas. La première consiste à extraire un schéma à partir d'une suite de test. La seconde consiste à identifier des schémas récurrents (« schémas d'ordre supérieur » ou de « méta-schémas ») et de les instancier en fonction du contexte. Ci-après, je détaille ces deux stratégies.

Lors du projet RNTL COTE, Gemplus nous a fourni une application bancaire et une suite de test conçue manuellement. Cette suite était composée de 40 tests. Nous l'avons analysée et nous l'avons reformulée à l'aide de 5 schémas Tobias. Une fois dépliée, la nouvelle suite de test comportait 1900 cas de test. Cette expérience nous laisse à penser qu'il serait possible de procéder d'une façon similaire et d'extraire automatiquement des schémas par inférence ou apprentissage. Cette stratégie pourrait s'inspirer des nombreux travaux sur l'inférence grammaticale [102]. Cette idée est à l'état d'ébauche : nous n'avons pas commencé d'études sérieuses en ce sens. Les verrous se situent dans la façon dont on va généraliser les cas de tests : on pourrait obtenir assez vite des schémas qui produisent trop de tests (explosion combinatoire).

Pour aider à la construction de schémas, une seconde stratégie consiste à identifier des schémas récurrents et de les exprimer sous forme de schémas génériques. On pourrait parler de méta-schémas ou de schémas d'ordre supérieur. Jusqu'à présent, nous avons identifié deux schémas génériques issus de notre expérience et de [15].

Le premier schéma générique (G_e) est le plus élémentaire. Il consiste à créer l'objet sous test puis à appeler chacune des méthodes de façon combinatoire [15]. La définition de ce schéma générique s'appuie sur deux groupes : C correspond à l'ensemble des appels aux méthodes constructeur et M correspond aux appels des autres méthodes. Ce schéma générique est paramétré par le nombre d'appels de méthodes consécutifs que l'on souhaite observer[6] : G_e(i) = "C ; $M^{i..i}$"

Un second schéma d'ordre supérieur identifié consiste en une variation du schéma ci-dessus en exploitant la stratégie def-use [15]. L'idée en est la suivante. Pour chaque attribut A d'une classe, on identifie deux groupes : les méthodes qui modifient le dit attribut (Set_A), et celles qui le consultent (Get_A). On définit alors un schéma générique par attribut A. Il consiste en l'appel du constructeur, puis une répétition d'appel de méthodes qui modifient et consultent la valeur de l'attribut donné.
$G_{def-use}$(i,A) = "C ; (Set_A ; Get_A)$^{i..i}$"

La construction automatique et l'évaluation de ces schémas a été entreprise par A. Amiar dans le cadre de son M2R [6]. Pour cela, un prototype d'analyseur de code a été construit. Il détecte les méthodes qui consultent et celles qui modifient les attributs et, calcule les groupes et applique les schémas d'ordre

6. Le schéma choisi G_e(i) = "C ; M^i" a été préféré à sa variante G_e(i,j) = "C ; $M^{i..j}$" car les cas de tests produits pour $M^{i..j-1}$ sont couverts par les cas de tests produits pour M^j.

supérieur. Le fichier d'entrée pour Tobias ainsi généré doit être complété par l'utilisateur pour le choix des données pour l'instanciation des valeurs.

Une fois que les schémas d'ordre supérieur sont identifiés, la difficulté se situe au niveau de leur instanciation, et notamment pour le choix des valeurs de paramètres. Ce problème ne se posait pas jusqu'à présent car le choix était de la responsabilité du testeur. Dans une démarche où l'on cherche à automatiser la production de schémas, il faut aussi proposer des valeurs. Ce problème est abordé de différentes façons dans la littérature [109] : choix aléatoire, choix contraint par des classes d'équivalence prédéfinies ou sur la base d'une analyse de la structure du code ou du modèle [132], etc. Dans la cadre du projet TASCCC, nous pourrons expérimenter le choix des données en liaison avec l'outil TestDesigner de Smartesting. L'idée est d'utiliser les valeurs identifiées par TestDesigner pour créer des séquences valides par rapport la spécification.

Chapitre 3

Des assertions comme oracle :

L'exemple de JML pour spécifier des services domotiques

Question : « JML peut-il être utilisé pour l'expression des spécifications de systèmes domotiques et servir d'oracle ou de moniteur dans des environnements techniques complexes ? »

Réponse : « J'ai mené deux études de cas dans le domaine de la domotique pour apporter un début de réponse. »

1. Introduction

Comme je l'ai dit précédemment, une séquence d'appels de méthodes n'est un test que si l'on dispose d'un moyen de juger les réponses du système sollicité. On parle d'oracle du test. En toute généralité, l'oracle peut être manuel ou automatique. Mais, dans la mesure où l'on s'intéresse à la production massive des tests, il est nécessaire de disposer d'une procédure *automatique* de jugement de l'exécution des tests.

Lorsque l'oracle doit être automatisé, il faut être capable de caractériser le résultat attendu [236, 201, 199, 225, 208]. En théorie, un oracle devrait être *complet* et *fiable* [208]. C'est-à-dire qu'il devrait caractériser les sorties attendues *quelques soient* les entrées proposées, et ce sans erreur.

La difficulté est qu'il est rarement possible de fournir une telle description. Le nombre de comportements à caractériser peut être trop important [199]. Dans d'autres cas, on peut être capable d'identifier les résultats improbables sans être en mesure de dire précisément le résultat attendu [236]. Par exemple, pour un programme de calcul scientifique, si un résultat ne correspond pas à l'ordre de grandeur attendu, il est certainement faux (e.g. 100 au lieu de 10 000) ; par contre, il peut être difficile de dire si le résultat correct est 11 234,5 ou 11 235,4. E. Weyuker nomme cette situation « oracle partiel » [236].

Selon [225], une autre difficulté provient du fait que le résultat n'est pas toujours suffisamment « *observable* » pour permettre d'identifier une erreur. Une raison possible est que le résultat n'est peut-être pas disponible assez

longtemps pour être capturé ou masqué par des phénomènes transitoires. Cela peut aussi être dû au fait que le système sous test est plongé dans un environnement plus complexe et/ou que les « points d'observation » sont réduits, comme par exemple pour des systèmes distribués [43] ou embarqués.

Diverses solutions ont été proposées dans la littérature pour produire un oracle automatique. Par exemple, une partie du système peut être implémentée par une autre équipe et/ou une autre méthode de développement (pseudo-oracle) [57]. Si plusieurs versions du système sont disponibles, les versions antérieures à la version courante peuvent être utilisées pour établir la non régression [202]. Les comportements du système sous test peuvent aussi être appris par analyse de cas particuliers puis extrapolés sous la surveillance du testeur [111, 209, 217, 208]. Enfin, il peut-être possible d'exploiter une spécification formelle du système, lorsqu'elle est disponible [56, 133, 199]. La spécification peut, par exemple, se présenter sous la forme d'un modèle B ou UML [135, 138, 205], d'une spécification Z [133], de systèmes de transitions étiquetés [36, 140], d'assertions [48, 52, 201], de propriétés [93, 190], etc.

Dans tous les cas, et pour les raisons décrites précédemment, lorsqu'un oracle est fourni, il est a priori incomplet et contient possiblement des fautes (au même titre que tout programme). Ainsi, la question de sa qualité se pose selon deux critères : (1) la description est-elle correcte ? Et (2) est-elle suffisante pour identifier les défaillances ?

2. Motivations

En ce qui concerne l'expression de l'oracle, j'ai étudié en particulier une solution reposant sur l'utilisation d'*assertions* [48, 52]. Une assertion est une contrainte formelle sur le comportement du programme, exprimée par des annotations insérées dans le code source [201]. Ces annotations (invariants, pré et post-conditions) sont transformées en code exécutable et évaluées pendant l'exécution du programme. Lorsqu'une assertion est violée, l'environnement du programme est alerté, par exemple à l'aide d'exceptions. Ainsi, pour tester un programme spécifié par des assertions, il suffit de le solliciter avec des données, le verdict du test sera calculé par les assertions. Un programme contenant des assertions est parfois appelé « programme *auto-testable* » (ou *self-checking*) [52].

L'utilisation des assertions a été popularisé par le langage Eiffel, qui contient des constructions natives permettant de les exprimer [170]. De nombreux langages d'assertion ont été proposés pour Ada [161], C/C++ [207, 223], Java [150, 119, 14] ... Sur le principe, utiliser des assertions pour décrire l'oracle est intéressant. La spécification peut être partiellement décrite, et raffinée au fur et à mesure des besoins.

Pour cette raison, j'ai souhaité évaluer l'applicabilité des assertions comme oracle de test. En effet, les assertions peuvent ne pas être appropriées au type de spécifications que l'on choisit d'exprimer. Des limitations peuvent apparaître

du fait de la nature du langage d'assertions utilisé (il peut ne pas offrir assez de constructions pour exprimer la spécification choisie) ou à cause de son mode d'évaluation. Par exemple, les invariants ne sont en général pas évalués continuellement pendant l'exécution, mais à certains moments précis. Pour faire une telle évaluation, je me suis placée dans un contexte précis, celui de la validation de services domotiques.

L'étude d'applications domotiques est intéressante pour plusieurs raisons. Tout d'abord, les services domotiques peuvent être considérés comme des applications critiques car certains peuvent influer sur la vie des personnes ou sur des biens [235]. De plus, un service peut-être déployé en présence d'autres, ce qui peut entraîner des interactions inattendues, voire dangereuses [198, 99]. Or, les services peuvent être déployés dans de multiples configurations : l'architecture des bâtiments et leurs contenus diffèrent d'un site à l'autre). Pire encore, la configuration d'un lieu est amené à changer au cours du temps, ce qui nécessite de produire une solution s'adaptant au changement de façon dynamique [31].

Depuis 2005, j'ai mené un travail de spécification en JML de deux applications domotiques. Ceci a permis d'identifier les difficultés relatives à l'expression de l'oracle par des assertions pour le test de services domotiques. Le premier système spécifié a été développé par les universités de Nara et Kobe, pour offrir des services domotiques dans un contexte où l'on utilise des appareils non connectés au réseau informatique (*legacy appliances*) [174]. La version I1 de ce système est déployée dans le show-room de Kobe, avec de vrais équipements. La version I2 du système est utilisée comme simulateur en France. Le second système spécifié a été développé par l'équipe ADELE du LIG, pour expérimenter les problématiques de (re)configurations dynamiques [112].

Les questions qui ont été abordées au travers de ces études de cas sont :

– la langage d'assertions est-il adapté pour décrire la spécification d'un système domotique ? En particulier, les assertions peuvent-elles être observables et caractériser les reconfigurations dynamiques ?

– Comment établir la qualité de l'oracle décrit par les assertions ?

– Enfin, toute assertion est-elle pertinente comme oracle du test ?

Dans la suite de ce chapitre, je présente tout d'abord JML et la façon dont il peut être utilisé comme oracle (section 3). Puis, je décris caractérise la spécification de services domotiques (section 4). J'aborde le problème de la qualité des assertions (section 5) et de leur pertinence comme oracle (section 6). Enfin, je conclus et je dresse des perspectives (section 7).

La spécification et la validation de services domotiques a été entreprise dans le cadre du projet Egide PHC-Sakura (collaboration avec l'université de Kobe) et le projet UJF IPotest (collaboration avec l'équipe ADELE du LIG). Les résultats ont été publiés dans [243, 244, 97, 86, 81, 197].

3. Java Modeling Language pour le test

Java Modeling Language (JML) est un langage conçu pour la spécification de programmes Java [150, 151, 142]. JML permet de spécifier les classes et leurs méthodes par l'expression de propriétés sur le principe de l'approche « Design By Contract » [170]. Le cœur du langage JML est basé sur Java, avec quelques mots clefs et constructions logiques supplémentaires [142, 151]. Les annotations sont décrites au sein de commentaires (entre /*@ et @*/ ou sur une (fin de) ligne commençant par //@).

Les annotations JML reposent sur trois types d'assertions : les invariants, les pré-conditions et les post-conditions. Les invariants sont des propriétés qui doivent être vraies dans tous les états visibles de la classe. Un état visible correspond à l'état initial ou à l'état final de l'invocation d'une méthode. Une pré-condition d'une méthode (clause `requires`) indique que l'assertion doit être satisfaite avant l'exécution de la méthode, faute de quoi on ne peut garantir le résultat attendu après exécution de la méthode. Une post-condition d'une méthode (clause `ensures`) indique que l'assertion doit être satisfaite après l'exécution de la méthode (si les pré-conditions ont été respectées).

Le langage JML comporte des mots clefs spécifiques pour exprimer les assertions. Par exemple, `\old(x)` correspond à la valeur qu'a `x` dans l'état initial de l'appel de la méthode. La clause `assignable` décrit les attributs qui peuvent être modifiés au cours de l'exécution de la méthode ; la clause `signals` permet de spécifier les exceptions attendues. Parmi les mots clefs de JML, on trouve aussi `\result` qui dénote la valeur retournée par une méthode (uniquement utilisable dans une post-condition) ; `\forall` et `\exists` correspondant aux quantificateurs universel et existentiel.

Les annotations JML insérées dans le fichier source peuvent être compilées avec le compilateur JML `jmlc`. L'outil `jmlc` transforme les annotations JML en des assertions embarquées dans le code Java compilé [44]. Les assertions JML sont alors évaluées à l'exécution et comparées au comportement du programme. Si une assertion n'est pas vérifiée, une exception est générée, indiquant quel type d'assertion a été violé (invariant, pré ou post-condition).

Les assertions JML peuvent être utilisées par d'autres outils. Par exemple, elles peuvent être utilisées par des outils de preuve formelle de programmes tels que Loop [226], Krakatoa [171] ou Jack [40]. L'outil ESC/Java permet d'identifier des erreurs à l'aide d'une analyse statique [120]. Enfin, le code `jmlc` peut être utilisé comme oracle lors d'un processus de test, par exemple avec JUnit [144].

Au cours du test d'un programme compilé avec `jmlc`, les invariants, pre et post-conditions sont évalués au début et à la fin de l'exécution de chaque méthode. Lorsqu'une opération est exécutée, trois cas peuvent se produire :

– Toutes les vérifications se sont déroulées correctement : le comportement de l'opération satisfait sa spécification pour les entrées choisies, dans l'état initial considéré. Le test retourne le verdict PASS.

– Une vérification intermédiaire ou finale échoue. Cela traduit une inconsistance entre le comportement de l'opération et sa spécification. Autrement dit, l'implantation n'est pas conforme à sa spécification et le test retourne le verdict FAIL.

– Une vérification initiale (pre-condition) échoue : dans ce cas, exécuter le test jusqu'au bout n'apportera pas d'informations supplémentaires car le comportement observé est en dehors du comportement spécifié. Le test retourne le verdict INCONCLUSIVE.
Par exemple, $\sqrt{x}$ a comme pré-condition que x doit être positif. Ainsi, un test de cette méthode avec la valeur -1 fournira un verdict INCONCLUSIVE.

4. Spécifier des services domotiques

Dans cette section, je commence par décrire les caractéristiques d'un tel système (section 4.1). Puis, je décris le travail effectué autour de la spécification de services domotiques. L'objectif était de déterminer si un langage de spécification basé sur des assertions permettait de décrire les comportements attendus du système, en vue de disposer d'un oracle pendant le test. Les deux études de cas montrent qu'un tel langage est adapté pour décrire les propriétés fonctionnelles attendues (section 4.2), et plus particulièrement dans un contexte dynamique (section 4.3). Mais que la nature des applications domotiques dresse certaines limites quant à la pertinence de l'oracle (section 4.4).

4.1. *Les caractéristiques d'un système domotique*

La domotique regroupe l'ensemble des techniques et technologies permettant de superviser, d'automatiser, de programmer et de coordonner les tâches de confort, de sécurité, de maintenance et plus généralement de services dans l'habitat. Aujourd'hui, il existe peu de solutions disponibles pour le grand public [176, 11, 230]. Ce qui existe est souvent monolithique et mono-fonction (gestion d'alarme, centrale de chauffage,...).

De nombreux travaux cherchent à adapter une approche orientée service pour offrir des solutions plus diverses, adaptables et coopérantes [13, 4, 31, 175]. Les applications visées sont essentiellement la sécurité des biens et des personnes, la maîtrise de l'énergie, l'hospitalisation à domicile et les loisirs.

Dans la suite, nous considérons qu'un système domotique propose différents *services intégrés*. Chacun de ces services pilote différents *appareils* ou *dispositifs* en vue d'effectuer une tâche particulière. Par exemple, le service intégré « Home cinéma » , défini par Nakamura *et al.* dans [175], coordonne le fonctionnement de la télévision, du lecteur de DVD, des enceintes, des lumières et des rideaux, afin d'offrir une ambiance « salle de cinéma » à la suite d'une seule requête.

Un système domotique est plongé dans un *environnement*. Les actions des services intégrés permettent de modifier l'état de certains appareils, et par voie de conséquence l'état de cet environnement. Ainsi, lorsque le service « Home-cinéma » est activé, la luminosité et le volume sonore de la pièce sont modifiés. D'autres éléments de l'environnement peuvent évoluer indépendamment du système, comme l'éclairage naturel de la pièce par le soleil. L'évolution de l'environnement peut être capturée à l'aide de capteurs, qui sont un type particulier d'appareils.

Cette vision des services domotiques est assez générique. Elle correspond aux deux études de cas étudiées, mais aussi à d'autres systèmes, tels que [241] ou [231].

4.2. *Des spécifications à plusieurs niveaux*

Pour spécifier des services dans ce contexte, Nakamura *et al.* ont identifié trois niveaux de description [174, 158]. Les « *propriétés locales* » visent à décrire les instructions relatives à la bonne utilisation d'un appareil en particulier. Par exemple, pour un lecteur de DVD, il est recommandé d'arrêter la lecture avant de procéder à l'éjection du DVD. Les « *propriétés globales* » décrivent les propriétés attendues au niveau d'un service intégré impliquant des appareils différents. Par exemple, le service « Home cinéma » doit ajuster la lumière au niveau décrit dans la spécification du service. Enfin, les « *propriétés issues de l'environnement* » sont des contraintes imposées au niveau de la maison et son environnement dans leur ensemble. Par exemple, dans certaines copropriétés, les usagers doivent prendre soin de ne pas faire de bruit après 22h.

Pour les systèmes domotiques de Nara/Kobe et de ADELE, nous avons exprimé les spécifications sous la forme d'invariants, de pré et de post-conditions, au niveau des appareils, des services et du système global (représentant la maison). Dans cette section, je me concentre sur la description des assertions décrites pour le système japonais.

Deux versions ont été considérées pour le système japonais. L'implémentation I1 est l'implantation originale utilisée dans les show-rooms des universités de Nara et Kobe, pour contrôler de vrais équipements (cf. Fig. 8). Seuls quelques appareils et services ont été spécifiés, essentiellement pour valider l'utilisation embarquée de JML dans le cadre d'un fonctionnement avec les appareils réels [86]. L'implantation I2 est une abstraction de I1, utilisée comme un simulateur. Elle est composée de 14 appareils *simulés* et 7 services intégrés. Elle a été spécifiée à l'aide de 209 annotations JML (17 pré-conditions, 150 post-conditions, et 42 invariants).

L'étude des annotations de I2 a montré que les invariants portaient essentiellement sur la spécification de l'état des appareils ou des services. Les pré-conditions spécifient essentiellement les restrictions d'utilisation des appareils. Les post-conditions spécifient l'évolution de l'état des services et des appareils en fonction des sollicitations. D'une façon générale, la plupart de ces

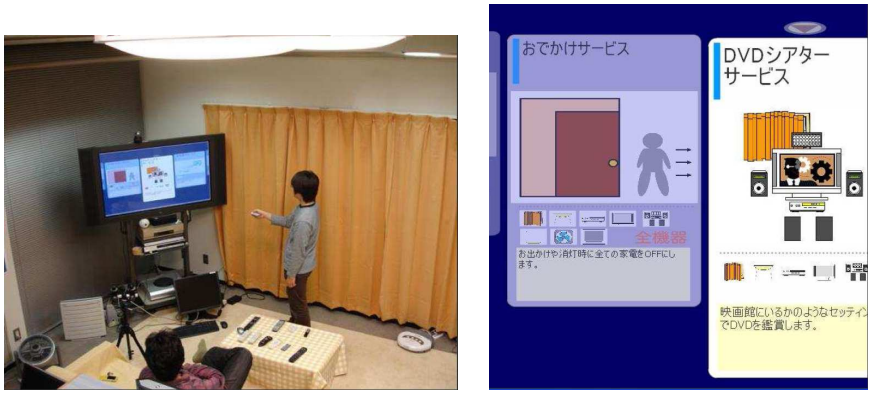


Figure 8. *Le show-room et l'interface utilisateur (japonaise) du système domotique développé par les universités de Kobe et Nara*

propriétés sont assez simples, comme on peut le constater sur l'extrait du code associé à la classe `TV.java`, donné ci-dessous.

```
public   class TV extends Appliance{
   [...]
   //Prop : la télévision a plusieurs états qui doivent être cohérents
   //@ public invariant (powerState.equals("ON") ==>
   //@       (internalState.equals("ON")||internalState.equals("OFF")));
   //@ public invariant inputMode.equals("TV")||inputMode.equals("DVD");
   //@ public invariant outputMode.equals("TV") ||
   //@                                   outputMode.equals("SPEAKER");
```

La raison pour laquelle les assertions sont simples s'explique par le fait que les services intégrés sont eux-mêmes très simples dans les implantations I1 et I2 (cf. Fig. 9). En effet, au moment de leurs activations, ces services exécutent une petite suite d'actions et s'arrêtent. Ils n'ont pas vocation à surveiller et à faire évoluer l'environnement sur une longue période. De ce point de vue, ils peuvent être considérés comme « instantanés ».

La spécification d'un tel service intégré est assez simple. Lorsque le service doit modifier l'état d'un appareil, une post-condition doit décrire l'état attendu du dit appareil une fois l'exécution du service terminée. Si l'état de l'appareil ne peut être observé directement, la post-condition s'assure que les *actions nécessaires* à l'obtention du résultat ont bien été effectuées. Par exemple, on s'assure que l'action d'allumer la télévision a bien été exécutée.

4.3. *Oracle pour les applications dynamiques*

Le système de Nara/Kobe est un système assez statique : il fonctionne tant que rien n'est modifié. Pourtant, le contenu d'une pièce évolue au cours du

```
Public   DVD TheaterService {
         DigitalTV     tv = new DigitalTV();
         DVDPlayer  dvd = new DVDPlayer();
         SoundSystem  sound =new SoundSystem();
         Light   light = new Light();
         Curtain  curtain = new   Curtain();

         tv.on();                            /* Turn on TV */
         tv.setVisualInput('DVD');

         dvd.on();                           /* Turn on the DVD Player */
         dvd.setSoundOutput('5.1');

         sound.on();                         /* Turn on the Sound System */
         sound.setInputSource('DVD');
         sound.setVolumeLevel(25);

         curtain.closeCurtain();             /*  Close curtain  */

         light.setBrightnessLevel(1);        /*  Minimize brightness  */

         tv.playTv();                        /*  Play TV   */
         dvd.playDvd();                      /*  Play DVD  */
}
```

Figure 9. *Code pour le service Home-Cinéma (implantation I2)*

temps. Le rythme de son évolution est variable. Les gros appareils (du type électroménager) peuvent être changés à un rythme allant de 2 à 20 ans. Par contre, les appareils plus petits peuvent être déplacés ou changés à un rythme beaucoup plus rapide. Ainsi, la plupart du temps, un téléphone portable suit sont propriétaire : d'un instant à l'autre, il est présent ou non dans une pièce.

Un système domotique doit supporter les différentes reconfigurations de son environnement de façon dynamique. Il doit s'adapter au changement des appareils, tels que la télévision, sans avoir à être reprogrammé. Il doit supporter l'apparition et la disparition d'appareils tels que des téléphones, sans avoir à être redémarré.

Prenons l'exemple du service « EcoHeaterMon ». Celui-ci allume ou éteint le nombre d'appareils de chauffage en fonction du nombre de chauffages présents dans la pièce, de la température souhaitée et de la température courante. Plus l'écart de température est important, plus le service active de chauffages. Au contraire, si l'écart de température est faible, le service n'allume que quelques chauffages parmi ceux disponibles. Un tel service fonctionne sur la durée (il n'est pas « instantané » comme pouvait l'être « Home-cinéma »). A ce titre, pendant son exécution, le nombre de chauffages peut évoluer. On s'attend à ce que le service adapte son comportement à une nouvelle configuration sans avoir à être éteint puis relancé. Ainsi, si la différence de température est importante et qu'un nouveau chauffage (mobile) est ajouté à la pièce, alors le service doit démarrer le chauffage au plus vite[1].

1. Dans notre implémentation, le service ajuste le nombre de chauffages allumés par rapport à la différence de température toutes les 2 secondes.

Pour adresser le problème de l'évolution dynamique du système, l'équipe ADELE du laboratoire LIG a développé une plateforme domotique autonomique appelée H-Oméga [31, 112]. H-Oméga est construite au dessus de OSGi™ [5] et iPOJO [113]. L'environnement OSGi est un système pour Java qui implémente une notion de component dynamique. Le modèle de composant orienté service implémenté par iPOJO facilite la conception d'applications dynamiques.

Spécifier des services à l'aide d'assertions dans un contexte dynamique est un défi supplémentaire, car les assertions doivent s'adapter au contexte changeant. Si les changements d'appareils peuvent être masqués par l'utilisation de framework tels que iPOJO, il reste que la configuration du système peut avoir évolué entre le début et la fin de l'exécution d'une méthode, rendant fausses certaines post-conditions.

Ainsi, pour le service « EcoHeaterMon », nous avons spécifié que le nombre de chauffages actifs devait être conforme à celui imposé par l'écart de température. Supposons maintenant que le service allume tous les chauffages à sa disposition pour réchauffer une pièce très froide, conformément à la spécification du service. Si un nouveau chauffage est ajouté, avant l'évaluation des assertions, le nombre de chauffages allumés sera différent de ce qu'il devrait être. Cette inconsistance conduira à un verdict biaisé si le service est correct, pour des raisons d'observabilité : l'invariant est évalué à un moment où le résultat n'est pas encore établi.

Une façon d'adresser ce problème consiste à s'assurer qu'il est possible d'observer le comportement du système aux moments adéquats. Dans le cas de « EcoHeaterMon », il faut évaluer les assertions sur des variables qui n'évoluent pas entre l'exécution du système et l'évaluation des post-conditions. Par ailleurs, lorsqu'une mise à jour de l'environnement est détectée, l'exécution du service et l'évaluation des assertions doivent être forcées. Dans le contexte spécifique de iPOJO, cela a été rendu possible par la présence de méthodes offertes par le framework qui notifie des changements dans l'environnement du service. Cela permet d'ajuster l'évaluation des assertions au fur et à mesure de l'évolution du système [197].

4.4. *Limites de l'oracle dans le contexte domotique*

Pour ces deux études de cas, toutes les propriétés souhaitées ont pu être exprimées à l'aide d'assertions. Ceci confirme qu'un langage comme JML comporte les constructions nécessaires à l'expression des spécifications de services domotiques.

Toutefois, il est important de noter que l'expression de l'oracle est rendue difficile par la réalité du terrain et non par le langage de spécification choisi [97]. La difficulté se situe au niveau de la perception que le système peut avoir de son environnement et de l'état des appareils qu'il contrôle. Cette perception peut être biaisée.

En effet, un système domotique perçoit son environnement au travers de capteurs ou par l'utilisation d'un modèle. Par exemple, pour le système de Nara/Kobe ne peut pas observer directement l'état de la télévision pour vérifier qu'elle est allumée ou non. L'implantation de ce système est basée sur l'utilisation d'une télécommande universelle pour contrôler les appareils. Pour s'assurer que la télévision est allumée, le système calcule l'état supposé de la télévision en fonction de ses actions. La spécification est exprimée par rapport à cet état supposé.

```
//@ ensures (tv.getStatus().equals("ON") );
```

Si l'état supposé de l'environnement diffère de la réalité (e.g. la télévision est éteinte manuellement), les assertions donnent des résultats biaisés.

Pour limiter la divergence entre l'état supposé du système et l'état réel, on peut utiliser des capteurs, s'ils sont disponibles. Il est alors possible d'exprimer les spécifications directement à partir des valeurs mesurées par les capteurs. Ainsi, pour vérifier que le niveau de luminosité est celui voulu, on peut utiliser la valeur mesurée par un luminomètre :

```
//@ ensures (lu.getBrightness() <10 );
```

Malheureusement, l'utilisation de capteurs peut aussi biaiser le verdict du test. Exprimer la propriété à partir d'un capteur rend l'oracle sensible à des pannes ou des perturbations locales de l'environnement. Ainsi, si le luminomètre donne des informations incorrectes (parce qu'il est défaillant), alors l'évaluation de la luminosité au cours du test peut conduire à une violation de l'oracle. De même, si le luminomètre est recouvert par un objet opaque ou mal positionné (à l'ombre d'un objet), l'information qu'il donne ne sera pas cohérente avec la réalité. Les verdicts relatifs à la luminosité de tests effectués dans ces conditions seront biaisés.

Ma conclusion sur ce travail est que la difficulté majeure pour la création d'un oracle pour le test de services domotiques ne se situe pas dans l'utilisation du langage d'assertion (ici JML). Il se situe dans la perception que le système peut avoir de son environnement, qui peut être différent de la réalité du terrain. Ainsi, le verdict des tests peut être biaisé parce que les propriétés d'oracle sont évaluées dans un contexte inadéquat. Construire un oracle utile dans un environnement réel va bien au-delà de l'expression d'assertions. Cela comprend la mise en *oe*uvre de moyens permettant de s'assurer que la perception du système est compatible avec la réalité.

5. La qualité des assertions comme oracle

Un test est de qualité s'il permet de découvrir des erreurs. S'il n'en découvre pas, soit le système ne contient pas (plus) d'erreur, soit les tests choisis ne permettent pas de les mettre en évidence. Un test ne met pas en évidence une erreur pour deux raisons possibles : les données ne permettent pas de provoquer la défaillance [228], ou l'oracle ne permet pas de détecter la défaillance lorsque celle-ci est observable.

Dans la littérature, de nombreux travaux visent à adresser le problème de la qualité des données de tests. La méthode utilisée la plus classiquement est l'analyse mutationnelle [60]. L'analyse mutationnelle consiste en l'introduction d'un petit changement syntaxique dans le code source du programme sous test dans le but de produire un *mutant*. Par exemple, on peut remplacer un opérateur par un autre ou modifier la valeur d'une constante. Ensuite, le comportement du mutant est comparé à celui du programme original. Si une différence peut être observée sur les sorties des deux programmes, le mutant est *tué*. Sinon, le mutant est vivant. Si un mutant a toujours le même comportement observable que le programme original, on dit que le mutant est *équivalent*. L'ensemble des opérateurs de mutation est décrit par un *modèle de faute*.

Le but originel de l'analyse mutationnelle est l'évaluation de la qualité des données de test. Pour cela, on produit tous les mutants issus d'un modèle de faute donné. Si une suite de test tue tous les mutants non équivalents, la suite de test est déclarée mutation-adéquate. Si c'est le cas, cela signifie que la suite de test permet de mettre en évidence des différences de comportements entre le programme original et des variantes de celui-ci. A noter, l'analyse mutationnelle ne conclut pas sur la correction du programme original. Elle conclut sur la capacité de la suite à *discriminer* les comportements des différentes versions produites.

La qualité des tests dépend non seulement de la qualité des données, mais aussi de celle de l'oracle. Un oracle est de « qualité » s'il permet effectivement de détecter les défaillances lorsque celles-ci sont observables. Pour détecter si c'est bien le cas, il a été proposé d'adapter l'analyse mutationnelle pour s'assurer que l'oracle permettait effectivement de détecter des défaillances produites par les fautes. Il s'agit là d'une variation de l'analyse mutationnelle. L'application d'une analyse mutationnelle pour évaluer la qualité de des assertions comme oracle dans un processus de test n'a pas, à ma connaissance, été explorée de façon systématique. Les travaux tels que [201] et [35] visent à caractériser comment les assertions doivent être définies pour être utiles.

Le principe de l'évaluation de la qualité de l'oracle par l'analyse mutationnelle a été exploré dans [157]. Au lieu de comparer les sorties du mutant avec le système sous test, les sorties sont évaluées par rapport à l'oracle. Si une défaillance est détectée par l'oracle, alors le mutant est tué. Sinon, le mutant reste vivant [157, 70, 82]. Cette approche a essentiellement été proposée pour évaluer la qualité des spécifications avec des techniques de preuve ou de model-checking, qui explorent l'espace des comportements de façon exhaustive.

L'évaluation de la qualité de l'oracle dans un contexte de test pose un problème important : celui de dissocier la qualité des données de tests et celle de l'oracle. Lorsque l'oracle ne détecte pas de défaillance quand le mutant et le programme original diffèrent dans leurs sorties, deux possibilités peuvent être invoquées.

1) La spécification est volontairement incomplète pour laisser une forme de liberté d'implantation, parce que la partie est non critique ou parce qu'il

n'est pas possible de spécifier ce comportement précisément.

2) La spécification est incorrecte, ou incomplète et ce n'est pas ce qui était souhaité.

En ce qui concerne la spécification des services dynamiques tels que « EcoHeaterMon », nous avons souhaité évaluer la pertinence des assertions proposé, notamment dans un contexte dynamique. Pour cela, nous avons considérées les fautes pouvant être générées par le modèle de faute utilisé par MuJava [162]. Nous avons sélectionné 25 mutations concernant les reconfigurations dynamiques. Une suite de 135 cas de test a ensuite été exécutée. 23 des 25 mutations ont été détectées par la suite originale. Les 2 mutants survivants ne pouvaient être tués que par des comportements autres que ceux présents dans la suite de test. Deux cas de test supplémentaires ont été créés et ont permis de tuer les mutants restants.

Dans ce cas particulier, les assertions permettent donc de mettre en évidence les défaillances des 25 mutants. Ceci permet a priori de conclure à la pertinence des assertions.

6. Pertinence des assertions comme oracle

Les assertions peuvent être utilisées comme oracle du test, mais elles peuvent aussi être utilisées dans d'autres contextes de V&V, tels que la preuve. Nous nous sommes donc posé la question de savoir si les assertions décrites pour les différentes activités sont de même nature ou si elles présentent des caractéristiques propres à chaque activité. Dans le premier cas, des assertions produites pour une activité autre que le test pourront donc être réutilisées pendant le test. Des activités de V&V conjointes (test et preuve) pourront être menées pour un coût faible. Si ce n'est pas le cas, des précautions devront être prises pour mener des activités conjointes.

Pour répondre à cette interrogation, j'ai testé une application Java spécifiée en JML et initialement prouvée. Puis, j'ai cherché à prouver une autre application Java spécifiée en JML et initialement testée. La présentation détaillée du travail a été publiée dans [81] (voir Annexe 2). Les conclusions sont que les assertions produites dans un contexte précis (test ou preuve) ne sont pas forcément réutilisables.

L'application d'abord prouvée puis testée est le système `Banking`. Il a été développé par Gemplus dans le cadre du projet RNTL COTE (2000-2002). Elle comprend 8 classes et correspond à plus de 500 lignes de code Java pour plus de 600 lignes de JML (cf. Fig. 10). Le nombre d'assertions JML s'explique par le fait que l'application a été prouvée (avec JACK, outil de preuve développé par Gemplus). De nombreuses assertions ont du être insérées pour aider le prouveur. Ce code et sa spécification JML ont ensuite été testés par l'équipe VASCO. Ainsi, 17 scénarios ont été décrits dans Tobias et dépliés en 1241 cas de tests pour un total de 40 000 lignes de code Java pour JUnit. En parallèle de

Classes	Java lines	JML lines
Transfer_src	116	150
AccountMan_src	105	236
Currency_src	93	28
Balance_src	64	58
Spending_rule	40	42
Saving_rule	40	42
Rule	40	23
Account	30	36
Total	518	615

Figure 10. *Répartition du code et des assertions pour* `Banking`

l'utilisation de Tobias, l'équipe a mené une activité de revue de code combinée à du test aléatoire avec l'outil Jartège, développé par C. Oriat [188].

L'application d'abord testée puis prouvée est le système domotique de Nara et Kobe (versions I1 et I2). Plusieurs outils de preuves ont été expérimentés

– ESC/Java2, un outil de vérification statique étendue pour Java [120, 42],

– Why/Krakatoa, une plateforme générique de vérification de programmes [164, 118], et

– Key, un environnement pour la création, l'analyse et la vérification de programmes [3, 19].

Les propriétés ont été spécifiés a posteriori pour I1 et a priori pour I2, avec l'objectif initial de servir d'oracle du test. Elles correspondent au comportement attendu de l'application et donc peuvent servir de référence pour le test. Toutefois, tout n'est pas aussi simple.

Les vérifications de I1 et I2 avec ESC/Java ont été laborieuses. Bien que les assertions aient été utilisées pour le test, elles ne contenaient pas assez d'informations pour une vérification automatique. Au final, l'ensemble des propriétés locales et globales semblent avoir été validées pour I2 avec l'outil de vérification statique ESC/Java. Toutefois, il convient d'être prudent. Les outils de vérification statique ne sont ni complets ni bien-fondés : ils ne détectent pas toutes les fautes et lèvent parfois de fausses alarmes.

Pour la vérification avec Why/Krakatoa et Key, nous avons choisi de vérifier I2 avant I1, une fois le travail avec ESC/Java terminé. Le résultat n'a pas été à la hauteur des attentes car presque aucun fichier n'a pu être prouvé ni pour I1 ni pour I2, quelques soient les outils. Plusieurs raisons expliquent ce résultat.

Tout d'abord, une partie des assertions issues du test ne sont pas utilisables par les outils de preuve. En effet, même si les outils partagent le même langage de spécification, les constructions utilisables par un outil ne le sont pas forcément pour les autres. Le processus de vérification est limité par la puissance des outils de vérification. Certaines parties du langage JML, supportées par les

outils de tests, ne le sont pas par les outils de vérification. De même, la phase de test exploite le fait que les assertions utilisées doivent être exécutables pour être utilisées comme oracle.

De plus, la spécification issue du test était insuffisante pour mener un processus de preuve. Il faut en effet beaucoup d'assertions supplémentaires pour aider un prouveur, (comme en témoigne la spécification JML pour `Banking`). Un gros effort a donc été nécessaire pour adapter le code de I1 et I2 et les assertions associées pour en produire une version vérifiable par les outils.

Nous résumons ici les leçons apprises pendant ces deux études de cas. Elles concernent la rédaction des spécifications et du programme.

L'utilisation d'un même langage de spécification pour la preuve et le test devrait faciliter un processus de V&V conjoint et une mutualisation de l'effort de spécification. Pourtant, d'un point de vue pratique, chaque outil utilise un sous-ensemble de constructions qui n'est pas tout à fait le même que les autres. Une partie des assertions produites dans le cadre d'une activité de validation était donc inutilisable dans l'autre activité.

Au-delà de cet aspect pratique, il apparaît que les processus de test et de preuve amènent assez naturellement à produire des assertions qui ne sont pas tout à fait de même nature, même si les propriétés que l'on cherche à exprimer représentent la même spécification. Les activités de test et de preuve sont sensiblement différentes. Pendant le test, on recherche des erreurs, pendant la preuve, on cherche à démontrer la correction.

Il en résulte que si l'objectif est de prouver le système, le travail de spécification va porter en priorité sur un sous-ensemble de propriétés jugées importantes et dont on pense qu'il sera prouvable. D'autres assertions seront ensuite ajoutées pour aider le processus de preuve. Ces assertions sont souvent une reformulation du code, et à ce titre ont peu de valeur ajoutée pour le test.

Si, au contraire, les assertions sont produites dans un objectif de test, le travail de spécification va chercher à exprimer un spectre de propriétés potentiellement plus large. L'absence de considération pour les besoins de la preuve fera que les assertions ne seront pas utilisables directement pour la preuve.

Dans les deux cas, un processus conjoint de test et preuve est plus coûteux que l'on peut se l'imaginer naïvement. En plus de la spécification et de la validation originale (test ou preuve), une phase d'évolution de la spécification (ou du code) sera nécessaire pour envisager la seconde étape (preuve ou test).

En ce qui concerne plus spécifiquement les assertions décrites pour le test, il faut insister sur l'écriture des invariants lorsque l'on spécifie après l'écriture du code. En effet, dans ce cas, il y a un risque (non négligeable) que les post-conditions résultent d'une copie du code. C'est un risque plus important quand on spécifie avec des assertions (plutôt qu'avec un modèle indépendant), car il y a toujours la tentation d'analyser le code pour exprimer ce que doit faire l'application. Procéder ainsi rend le processus de test moins pertinent (les résultats observés sont toujours ceux attendus). L'écriture d'invariants néces-

site de prendre du recul et s'abstraire du code en tant que tel. Ce n'est pas une garantie de qualité, mais au moins, les risques de simple copie du code sont plus limités.

En ce qui concerne plus spécifiquement la preuve, il faut souligner que la vérification est un processus qui ne s'improvise pas. On ne peut pas concevoir l'application, la développer et, au dernier moment, décider que l'on va la prouver. Faire cela, c'est s'exposer à devoir tout recommencer. La raison pour cela est que les outils de preuve sont encore limités. Ils ne supportent pas toutes les constructions du langage de programmation. Si l'on souhaite prouver, alors il faut concevoir l'application en conséquence. A l'image du test où l'on cherche de plus en plus à « concevoir pour tester » (*"design for test"*), il faut aussi « concevoir pour prouver » (*"design for verification"*). Cela consiste en particulier à choisir les constructions adéquates qui seront supportées par les outils de preuve.

7. Conclusion et perspectives

Tester, c'est être capable de porter un jugement sur le comportement du système sous test. Être capable de caractériser les comportements attendus est une des multiples difficultés du test. Ce jugement peut être fait de façon manuelle ou automatique. Lorsque l'on produit des tests de façon massive, il faut nécessairement un oracle automatique. Cela peut se présenter sous la forme d'une spécification exécutable ou d'un programme d'une autre nature.

Je me suis particulièrement intéressée à une forme d'oracle automatique consistant en assertions exécutables insérées dans le code. Pour qu'une telle approche soit utilisable, il faut s'assurer que langage d'assertions est adapté pour décrire la spécification du système souhaité, être capable d'évaluer la correction des assertions et leur pertinence.

Pour répondre à ces questions, j'ai exploré l'utilisation d'assertions pour spécifier et tester deux applications du domaine de la domotique. A ce jour, les possibilités offertes par le langage d'assertion utilisé (ici JML) ont permis de décrire toutes les propriétés attendues [243, 244, 86], y compris dans un contexte dynamique [197].

La principale difficulté quant à l'expression des assertions réside dans le fait que l'application domotique peut avoir une connaissance partielle et/ou biaisée de ce qui l'entoure [97]. L'établissement d'un verdict peut alors apparaître biaisé lui aussi. Les spécifications exprimées aux travers des assertions doivent tenir compte de ce contexte. En particulier, l'utilisation de capteurs permet d'ajuster la perception qu'a le système de son environnement, pour autant que ceux-ci fournissent des informations pertinentes.

Pour évaluer la pertinence des assertions, une analyse mutationnelle peut être envisagée. Il s'agit d'évaluer la capacité des assertions à détecter des fautes introduites volontairement dans le code. L'évaluation de la qualité des assertions doit être pondérée par la qualité des données testées [197]. Enfin,

dans le cas où des assertions sont issues d'un processus différent du test, il convient d'être prudent dans leur utilisation. En effet, la nature des activités (dont la preuve) influence le choix des assertions [81].

Le travail de spécification d'application domotique va se poursuivre dans le cadre de la thèse de Y. Grasland, encadré par I. Parissis et R. Groz. Cette thèse sera l'occasion d'étudier d'autres types de systèmes domotiques. Ce sera l'occasion de poursuivre l'analyse des propriétés attendues pour ces systèmes et de confronter mes précédentes observations dans ces nouveaux contextes.

Une autre direction de travail porte sur l'évaluation de la qualité de l'oracle. Comme je l'ai dit précédemment, la qualité d'un test inclut à la fois la pertinence des données et celle de l'oracle. Une perspective de travail consiste donc à évaluer les deux en parallèle. En exécutant le jeu d'essai contre le mutant, on peut confronter les sorties du mutant aux sorties du programme original (analyse mutationnelle classique) et à l'oracle. Alors seulement, on obtiendra une information pertinente sur la qualité de l'oracle.

Le travail envisagé par rapport à ce point consiste en l'évaluation empirique de l'approche. En effet, lorsque l'oracle ne détecte pas une défaillance alors que celle-ci est « détectable » (il existe des sorties différentes entre le mutant et le programme original), il faut déterminer si

– cela dénote d'un choix de conception (la spécification est volontairement abstraite, pour laisser différents choix de conception), ou

– cela correspond à défaut de conception de la spécification.

Etablir dans quel cas de figure on se trouve peut demander beaucoup de temps. Si au cours des expérimentations, on conclut toujours à la situation (1), l'analyse de la qualité des assertions telle que proposée n'est pas satisfaisante. Si par contre, un nombre non négligeable de situations (2) sont observées et que la spécification peut être corrigée en conséquence, alors l'approche a du sens et peut être proposée à plus large échelle.

Chapitre 4

Prédire la testabilité

Question : « Comment utiliser au mieux les travaux relatifs à la prédiction de la testabilité proposés dans la littérature pour produire des logiciels testables ? »

Réponse : « J'ai mené une évaluation de certaines métriques de testabilité afin de déterminer précisément quand elles peuvent être utilisées. »

1. Introduction

Le test est une activité coûteuse. Selon les sources, on obtient entre 20% et 60% du coût total de développement [20, 25]. Les chiffres diffèrent selon les exigences en termes de validation imposées par le contexte, et l'effort qui en résulte. On dépensera plus pour la validation proportionnellement au reste des activités, pour des systèmes critiques que pour des projets scolaires. On dépensera possiblement plus en validation proportionnellement au reste des activités, pour des systèmes qui vont évoluer dans le temps ou qui vont nécessiter de la maintenance que pour les systèmes conçus pour une durée de vie limitée, car à chaque opération de maintenance, il faudra valider/tester à nouveau.

Quelque soit la proportion que représente l'activité de test par rapport aux autres activités de développement, il est nécessaire d'obtenir des moyens de rendre les systèmes plus faciles à tester. La « testabilité » caractérise la capacité d'un système à être testé. Il y a de nombreuses définitions en ce qui concerne la testabilité. Intuitivement plus un système est testable, plus il va être facile de le tester, c'est-à-dire de découvrir ses fautes. C'est en substance ce que dit Binder dans [24]. *« Other things being equal, a more testable system will reduce the time and cost needed to meet reliability goals »*.

Ce même Binder souligne d'ailleurs que le coût du test est influencé par 6 principaux facteurs (cf. Fig. 11) qui sont aussi bien des facteurs internes au système sous test que des facteurs externes. Ainsi, l'utilisation d'outils automatiques (de génération ou d'exécution de tests, d'analyse de traces, de stockage des tests et des résultats ...) est un élément qui permet de rendre la phase de test plus efficiente/efficace et donc moins coûteuse.

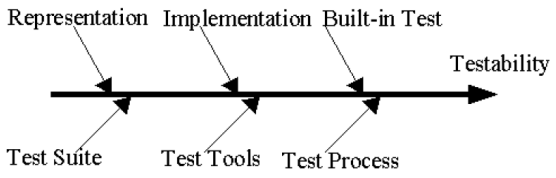


Figure 11. *Les six facteurs influençant la testabilité*

Bennitts définit la testabilité comme : « *the ability to generate, evaluate, and apply tests to satisfy a number of predefined test objectives (for example fault coverage, fault isolation, runtime, time-to-profit) subject to the two fundamental constraints of time and money* » [21]. D'une certaine façon, cette définition rejoint celle de Binder, dans la mesure où elle ne précise pas si la capacité à générer les tests, les évaluer et les appliquer est une caractéristique interne ou externe au système.

D'autres définitions réduisent la testabilité aux seuls facteurs internes du système. Ainsi, l'ISO indique que « *the attributes of software that bear on the effort needed to validate the software product* » [121]. De même, IEEE définit que la testabilité est « *the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met* » [181]. D'autres définitions encore plus spécifiques ont été proposées [228, 23].

Dans la suite de ce document, j'adopte une vision de la testabilité qui correspond à la plupart des travaux de la littérature [121, 181, 228, 23], à savoir une propriété relative au système en lui-même. Ceci correspond essentiellement aux facteurs Représentation, Implémentation et Built-In-Test décrits par Binder (cf. Fig. 11).

Dans l'idéal, une évaluation de la testabilité d'un système doit permettre de repérer les faiblesses qui le rendent peu testable. Une fois identifiées, ces faiblesses pourront être corrigées, permettant ainsi de rendre le système plus testable. Plus les faiblesses de testabilité seront repérées tôt dans le développement, moins coûteuses seront leurs corrections.

La caractérisation de la testabilité du système est très difficile, car elle est liée au contexte de validation. Ce dernier influe sur le choix des stratégies de test mises en œuvre, et chacune a un coût qui lui est propre. De plus, le coût du test peut être compris de différentes façons. Par exemple, on peut s'intéresser aux nombre de tests nécessaires pour atteindre un critère d'adéquation, ou à caractériser la complexité de l'activité [24]. Le nombre de tests et la complexité de leur conception ne sont pas forcément liés. Ainsi, selon les circonstances, une suite de test permettant de couvrir les conditions d'un système un peu complexe, peut être plus difficile à exhiber qu'une suite de test trois fois plus grande, produite de façon aléatoire.

Il résulte de ceci que de nombreuses métriques ont été proposées dans la littérature. Ainsi, pour les systèmes orienté-objet, plus d'une quarantaine

de métriques ont été proposées dans la littérature pour l'évaluation de la testabilité au niveau du code [215]. D'autres métriques ont été proposées pour évaluer la testabilité de ces systèmes sur la spécification [24, 172].

2. Motivations

D'un point de vue ingénierie, une métrique ne peut être utilisée que s'il a été démontré qu'elle mesure bien ce pourquoi elle a été conçue [145]. De plus, l'information qu'elle donne n'est pertinente que si elle est utilisée dans le contexte pour lequel elle a été définie. Plus spécifiquement, pour prédire la testabilité d'un système en particulier, il faut choisir des métriques *validées* et *appropriées* au contexte de validation.

Il existe deux façons de valider des métriques : théoriquement ou empiriquement. La validation *théorique* consiste en la démonstration que la métrique proposée satisfait bien les exigences relatives à un certain nombre de critères proposés, relatifs à la théorie de la mesure ou plus spécifiquement au domaine [237, 110, 51]. Par exemple, Weyuker a proposé 9 axiomes que doit satisfaire une « bonne » mesure de complexité [237]. Par exemple, le premier axiome impose qu'il existe au moins deux programmes différents ayant des valeurs de complexité différentes. Certains de ces axiomes ont par la suite été démontrés comme étant inconsistants [146]. La validation *empirique* cherche à démontrer l'utilisabilité de la métrique en pratique. Elle est nécessaire pour pouvoir être acceptée dans l'industrie.

De façon assez surprenante, très peu de métriques proposées dans la littérature sont validées (aussi bien empiriquement que théoriquement) [215]. De plus, les hypothèses d'applicabilité ne sont pas forcément explicitées dans les articles de recherches. Enfin, certaines métriques n'ont pas été définies de façon opérationnelle : il n'est pas possible d'utiliser directement les définitions proposées dans la littérature [33].

Dans la suite, le problème que j'adresse est la validation empirique de métriques de testabilité. D'abord, je présente l'activité menée autour de la validation de métriques pour les systèmes objet (section 3). Puis, j'aborde le travail mené pour un couple de métriques pour les systèmes synchrones (section 4). Enfin, je dresse quelques perspectives.

3. Validation de DIT dans le contexte objet

Le travail de validation des métriques de testabilité pour les systèmes objets a débuté avec la thèse de M.-R. Shaheen, que j'ai dirigée [210]. Il a débuté par la collecte des métriques de testabilité qui sont évaluables sur le code des applications.

3.1. *De l'utilisation de DIT pour prédire la testabilité*

La métrique DIT (*Depth of Inheritance Tree*) nous a particulièrement intéressés. Cette métrique permet d'évaluer la taille de l'arbre d'héritage d'une classe donnée. Du point de vue de la testabilité, cette métrique est intéressante. D'une part, elle a été préconisée comme métrique prédictive de la testabilité [47, 24]. D'autre part, les expérimentations décrites dans [38] ne permettent pas d'établir une corrélation explicite entre le coût du test et DIT.

En effet, dans cet article, les auteurs cherchent à établir des corrélations entre des métriques calculables sur le code source objet (représentant des coûts prédictifs) et la taille des suites de test (coûts effectifs du test). La taille des suites est mesurée par dLOCC, le nombre de lignes de code des programmes de test (*Lines Of Code for Class*) et par dNOTC, le nombre de cas de test (*Number of Test Cases*).

Dans cette étude, les auteurs constatent que DIT n'est pas corrélée à l'effort de test (notamment à dNOTC), contrairement à ce qu'ils s'attendaient. L'explication donnée par les auteurs est que les méthodes héritées n'ont peut-être pas été systématiquement re-testées.

Selon moi, ces résultats soulignent à quel point il est important de bien identifier quelles sont les hypothèses faites sur la génération des suites de test quand on propose ou que l'on analyse des métriques de testabilité. Cela n'a aucun sens d'évaluer le coût du test avec une métrique de testabilité si les conditions de production des tests ne satisfont pas les hypothèses de la dite métrique. Pour cette raison, établir des corrélations entre des suites de test et des métriques ne peut être fait que dans des situations où la maîtrise la phase de génération de test.

3.2. *Pour la validation de DIT*

Pour valider une métrique, le standard IEEE propose l'approche suivante [216]. Soit M une métrique *prédictive* de testabilité. On commence par caractériser les hypothèses relatives à la phase de test faites pour la définition de M, à l'image de ce qui est proposé dans [172] On identifie une stratégie de test $\mathcal{T}$ compatible avec les hypothèses identifiées. Puis, sur un certain nombre de programmes en adéquation avec contexte (choisis au hasard), on génère des suites de test avec $\mathcal{T}$. Le coût effectif de la production de la suite est ensuite évalué par une métrique C, compatible elle aussi avec les hypothèses. Enfin on compare les coûts prédictifs (donnés par M) et les coûts effectifs (donnés par C). Pour être pertinente, cette analyse doit être reproduite sur un grand nombre de programmes.

Malheureusement, dans le cadre de nos expérimentations au LIG, il n'était pas facile de produire des suites de tests par nous mêmes, en appliquant des stratégies qui respectaient les hypothèses. La difficulté était ici de produire suffisamment de suites de test pour les programmes à étudier. Comme nous

ne pouvions pas garantir la qualité des suites de test, nous avons adopté une variante de cette démarche, qui concerne l'établissement du coût effectif C de l'application de la stratégie de test.

Pour certaines stratégies de test $\mathcal{T}$, il existe des métriques de coût effectif, calculables à partir du code, dont il a été démontré qu'elles représentaient une estimation du nombre de tests à produire. Ainsi, ces métriques permettent d'estimer le nombre de tests à produire (*scope* du test, selon la définition de Binder), sans avoir à construire la suite de test.

Par exemple, pour le critère d'adéquation « couverture des méthodes d'une classe », il faut exécuter au moins une fois chaque méthode définie dans la classe. Le nombre de méthodes représente le nombre de tests maximum suffisant pour atteindre ce critère. De même, le nombre cyclomatique (CC) est égal au nombre d'instructions de décision du programme plus 1 [169, 137]. Cela correspond au nombre de tests recommandés pour assurer la couverture des décisions (i.e. des branches) [233].

D'une certaine manière, cela revient à comparer deux métriques prédictives. Alors pourquoi le faire ? L'idée est de comparer une métrique qui se calcule assez tôt dans le développement (au niveau du modèle/de la spécification) à une autre qui s'évalue sur le code. L'utilisation de métriques sur le modèle permet d'identifier assez tôt les éventuelles faiblesses de testabilité et de les corriger pour un coût raisonnable. Elles sont toutefois peu précises. Au contraire, l'utilisation de métriques sur le code peuvent prédire les faiblesses du système de façon plus précise, mais ne permettent pas vraiment d'envisager des corrections (souvent trop coûteuses). Ainsi, des informations calculées sur le code n'apportent pas les mêmes bénéfices que celles mesurées sur le modèle. Et donc, il est important de s'assurer que les métriques prédictives calculables sur le modèle proposent des informations cohérentes avec celles calculées à partir du code.

3.3. *Expérimentation*

Pour valider DIT, nous avons donc considéré deux types de stratégies de test : celles imposant que les méthodes héritées soient re-testées (T_t) et celles ne l'imposant pas (T_s). Ensuite, le coût effectif a été estimé par le nombre de méthodes à tester. Les hypothèses à valider peuvent donc s'exprimer ainsi :

1) le coût effectif d'une stratégie de test qui n'impose pas le test des méthodes héritées (T_s), estimé par le nombre de méthodes *définies* dans la classe sous test, n'est pas corrélé au DIT de la classe.

2) le coût effectif d'une stratégie de test qui impose le test des méthodes héritées (T_t), estimé par le nombre de méthodes *héritées* dans la classe sous test, est corrélé au DIT de la classe.

Ces hypothèses doivent être raffinées. Pour les langages de programmation tels que Java, il faut tenir compte du fait qu'une partie des classes sont fournies en standard. Si certaines stratégies de test imposent forcément de re-tester les

	# de class / DIT	# de class / DIT_A
0	-	797
1	596	342
2	439	189
3	223	50
4	90	19
5	40	8
6	17	2
7	2	0
total	1407	1407

Tableau 1. *Distribution des classes applicatives étudiées par rapport à DIT et DIT_A*

méthodes héritées [25], elles n'imposent pas nécessairement de re-tester les classes fournies en standard.

Cette distinction nous a amené à raffiner la définition de *DIT* pour distinguer l'ensemble de l'arbre d'héritage (classes standards inclues) et l'arbre restreint aux seules classes applicatives (définies pour les besoins de l'application) (estimé par DIT_A). Appliqué au contexte des classes Java, $DIT(C)$ correspond au nombre de classes héritées (représentées par le nombre d'ancêtres) de C. $DIT_A(C)$ correspond au nombre de classes héritées de C qui ne sont pas des classes standard de Java. Soient AC l'ensemble des classes applicatives et JC les classes standard de Java. $DIT(c) = |Ancestors(c)|$ Si $c \in AC$, $DIT_A(c) = |Ancestors(c) \backslash JC|$. Les hypothèses précédentes peuvent donc être reformulées ainsi :

1-1 le coût effectif d'une stratégie de test qui n'impose pas le test des méthodes héritées (T_s), estimé par le nombre de méthodes *définies* dans la classe sous test, n'est pas corrélé au *DIT* de la classe.

1-2 le coût effectif d'une stratégie de test qui n'impose pas le test des méthodes héritées (T_s), estimé par le nombre de méthodes *définies* dans la classe sous test, n'est pas corrélé au DIT_A de la classe.

2-1 le coût effectif d'une stratégie de test qui impose le test des méthodes héritées (T_t), estimé par le nombre de méthodes *héritées* dans la classe sous test, est corrélé au *DIT* de la classe.

2-2 le coût effectif d'une stratégie de test qui impose le test des méthodes héritées (T_t), estimé par le nombre de méthodes *héritées* dans la classe sous test, est corrélé au DIT_A de la classe.

Pour évaluer ces hypothèses, nous avons collecté 6 applications réelles open-source. Le code de ces applications représente environ 140 packages. Sur les 1900 classes et interfaces étudiées, seules 1785 correspondaient à des classes applicatives et 1407 n'étaient pas des interfaces. Pour chaque classe applicative, ont été collectées la profondeur de l'arbre applicatif et total (cf. Table 1), le nombre de méthodes définies, le nombre de méthodes héritées issues de l'arbre d'héritage applicatif et celles issues de l'arbre d'héritage total.

Application	$rs(\|M_{In}(c)\|$, DIT)		$rs(\|M_{IA}(c)\|$, DIT$_A$)	
	min	max	min	max
NanoXML	0.97	0.97	1	1
EMMA	0.50	1	0.50	1
AspectJ	0.785	1	0.97	1
CEF	0.884	1	0.834	1
Java Groups	0.883	1	1	1
Ant	0.557	1	0.887	1

Tableau 2. *Corrélation de Spearman entre le nombre de méthodes définies et les profondeurs d'héritage (total et applicatif), détaillé pour chaque application*

L'évaluation des hypothèses a consisté à calculer le coefficient de corrélation de Spearman entre le nombre de méthodes définies ($|M_D(c)|$)(resp. héritées de l'arbre total ($|M_{In}(c)|$) ou de l'arbre applicatif ($|M_{IA}(c)|$)) et *DIT* ou *DIT*$_A$. En statistique, le coefficient de corrélation de Spearman est étudié lorsque deux variables statistiques semblent corrélées. Intuitivement, cela consiste à trouver un coefficient de corrélation, non pas entre les valeurs prises par les deux variables mais entre les rangs de ces valeurs. Cela permet de repérer des corrélations monotones. Le coefficient de Spearman (*rs*) est un nombre compris entre -1 (classements inverses) et 1 (classements identiques), la valeur zéro indiquant que les deux classements ne sont pas corrélés.

Nous avons calculé le coefficient de Spearman pour chaque package de chaque application. La table 2 montre les valeurs minimum et maximum obtenues pour l'ensemble des packages de chaque application. On observe que l'on obtient un coefficient de corrélation assez élevé entre le nombre de méthodes héritées et la profondeur d'héritage. Au contraire, le nombre de méthodes définies ($|M_D(c)|$) n'est pas corrélé à l'une des deux profondeurs d'héritage (*DIT* ou *DIT*$_A$). On obtient en effet rs($|M_D(c)|$,DIT) = -0.069 et *rs*(($|M_D(c)|$,DIT$_A$) = 0.173 toutes applications confondues.

Ces expérimentations permettent d'accepter les 4 hypothèses proposées précédemment. Ceci signifie que DIT est corrélé à l'effort de test lorsque celui-ci est estimé par le nombre de méthodes à tester et que la stratégie de test impose le test des méthodes héritées. Ce travail, publié dans [212] et reproduit en annexe 2, permet donc d'expliquer les résultats publiés précédemment, à savoir que DIT peut être utilisé pour prédire *une forme de* testabilité. Mais que cela n'a pas de sens de l'utiliser pour prédire le coût d'une stratégie de test qui ne tient pas compte de l'héritage.

3.4. *Autres études*

Nous avons mené trois autres études pour valider d'autres métriques dans d'autres contextes [211, 214, 213].

Dans [214], nous avons mené une autre étude pour *DIT*. Cette fois-ci le coût effectif du test d'une classe était évalué par $WMC_c c$, une métrique proposée dans [46]. Cette métrique correspond à la somme des complexités des méthodes d'une classe, calculées par le nombre cyclomatique [137]. Cette mesure de coût a été choisie car elle représente une estimation du nombre de tests nécessaires pour atteindre la couverture des branches.

L'analyse a été menée sur plus de 25 applications open-source. Les résultats ont montré que *DIT* (resp. DIT_A) ne permettait pas de prédire un tel coût. Intuitivement, *DIT* est trop abstraite pour représenter la diversité des valeurs obtenues par $WMC_c c$.

Les travaux [211, 213] plus précisément sur l'analyse de la pertinence d'anti-patrons de testabilité. Un anti-patron de testabilité identifie une construction qui peut augmenter l'effort de test par sa présence. Baudry *et. Al* [16] identifient en particulier une construction au niveau du diagramme de classes qui influe négativement sur le test d'intégration. Par les travaux présentés dans [214, 213], nous avons cherché à identifié à quel point ces anti-patrons pouvaient être présents pendant le développement (de la conception au codage) d'une application. Nous avons en particulier noté que plus l'anti-patron était présent tôt dans le développement, plus il était présent au niveau du codage. Ces résultats montrent qu'il est pertinent de réduire au maximum la présence du patron dès les premières phases de conception.

4. Validation de métriques de testabilité pour Lustre

Dans un autre registre, je me suis intéressée à deux métriques de testabilité pour Lustre/Scade proposées par l'équipe ValSys du LCIS [149], dans le contexte du projet ANR SIESTA.

4.1. *Le langage* Lustre

Lustre est un langage de flot de données déclaratif et synchrone [128]. L'hypothèse synchrone considère que le temps de réaction du programme est négligeable par rapport au temps de réaction de l'environnement. L'approche flot de données synchrone consiste en l'association d'une dimension temporelle au modèle flot de données. Ainsi, un flot inclut deux parties : une *séquence de valeurs* d'un type donné et une *horloge* représentant une séquence d'instants (échelle discrète).

Une description Lustre est structurée en un réseau de nœuds (*node*). Les relations entre les entrées et les sorties sont établies grâce à des variables intermédiaires, des constantes et des opérateurs (élémentaires ou de plus haut niveau). Un nœud est défini comme un ensemble d'équations. Chaque sortie et chaque variable locale doivent être définies par une et une seule équation, qui peuvent être écrites dans n'importe quel ordre.

Lustre offre les opérateurs arithmétiques, booléens et conditionnels classiques et des opérateurs spécifiques sur les flots. Il offre aussi plusieurs opérateurs temporels qui permettent (1) de résonner sur les flots comme avec une logique du passé et (2) d'échantillonner les signaux. Par exemple, l'opérateur `pre`, pour « précédent » permet d'obtenir la valeur d'un flot à l'instant précédent. Soient E et F deux expressions du même type dénotées par les séquences $(e_0, e_1, ..., e_n ...)$ et $(f_0, f_1, ..., f_n, ...)$; $\texttt{pre}(E)$ correspond à la séquence $(nil, e_0, e_1, ..., e_{n-1} ...)$ où nil est la valeur indéfinie. L'opérateur $\rightarrow$ (`followed-by`) permet d'initialiser un flot avec une valeur particulière. Ainsi, $E \rightarrow F$ correspond à la séquence $(e_0, f_1, ..., f_n ...)$.

Lustre est un langage textuel, issus des recherches effectuées dans les laboratoires de Grenoble (LGI, puis Vérimag). Sa sémantique a été très précisément définie. Une version graphique et industrielle (Scade) est industrialisée par Esterel Technologie. L'environnement de développement associé à Scade est certifié par rapport à plusieurs standards industriels issus de l'aéronautique (DO-178B), de l'énergie (IEC 61508), du transport ferroviaire (EN 50128) et du nucléaire (IEC 60880),

4.2. *La testabilité de programmes* Lustre/Scade

Dans le cadre d'une utilisation industriel Scade, la question du test et de la testabilité est cruciale. Pour identifier les parties difficiles à tester, Chantal Robach et son équipe ont défini un couple de deux métriques appelées contrôlabilité et observabilité, inspirées des notions du même nom utilisées pour l'évaluation de la testabilité de systèmes matériels [149]. Ces métriques sont notamment utilisées dans le cadre du projet ANR SIESTA pour organiser le processus de test.

La *contrôlabilité* d'un composant à l'intérieur d'un système est définie comme la facilité de transmettre des données, des entrées du système aux entrées du composants. L'*observabilité* d'un composant à l'intérieur d'un système est définie comme la facilité de propager une sortie du composant vers les sorties du système. Ces deux valeurs sont calculées sur le principe de la théorie de l'information pour évaluer la quantité d'information perdue entre les entrées et les sorties d'un système. Cela est basé sur la capacité des modules et des liens entre les modules.

De façon intuitive, les définitions de contrôlabilité et d'observabilité proposées expriment la quantité d'information perdue au cours des calculs dans le système. La contrôlabilité et l'observabilité sont définies par rapport à des *écoulements*. Un écoulement est l'ensemble des chemins entre *une* sortie et les entrées nécessaires à son calcul.

La façon de calculer les métriques de testabilité est détaillée dans [200, 149]. Elles ont été implémentées dans l'outil Satan [61], qui présente les valeurs obtenues de façon graphique ou textuelle, de façon plus ou moins détaillée en fonction des choix de l'utilisateur. Un exemple est donné Fig 12. Il est composé

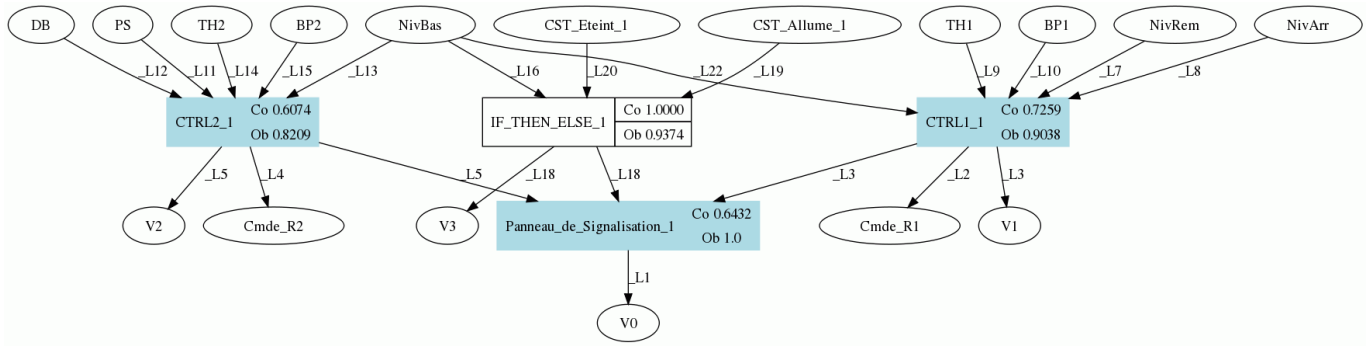


Figure 12. *Contrôlabilité et Observabilité pour le système de pompage*

de 3 sous-systèmes, et implante un système de pompage décrit dans [101]. Les valeurs présentées sont les valeurs minimum observées sur les différents écoulements.

4.3. *Approche de validation*

La contrôlabilité et l'observabilité telles que définies dans [200, 149] visent à l'identification des parties du système difficiles à tester. Il s'agit de mesures de complexité au sens de Binder. De l'avis des utilisateurs de Satan dans la cadre du projet ANR SIESTA, les métriques de contrôlabilité et d'observabilité sont difficiles à interpréter et à exploiter. Intuitivement, une partie du système est difficile à tester s'il est difficile de détecter les fautes présentes dans cette partie. Pour le système de pompage, on obtient que le module `CTRL2_1` semble plus difficile à tester que `CTRL1_1` (Cf. Fig. 12). Si c'est le cas, alors en moyenne, il devrait plus difficile de mettre en évidence des défaillances issues de fautes de `CTRL2_1` que de `CTRL1_1`.

L'objectif du travail présenté ci-après, était de s'assurer que cette interprétation était satisfaisante [76].

Conformément à l'approche proposée dans la section précédente, nous avons d'abord cherché à comprendre les hypothèses faites pour le calcul de ces métriques. Comme les métriques sont calculées sans hypothèse explicite sur la distribution des données pendant le test, nous avons conclu que la stratégie de test correspondait à une génération aléatoire avec une distribution uniforme des entrées.

Pour vérifier si un module est plus difficile à tester qu'un autre, nous introduisons des fautes dans les deux modules et évaluons s'il est facile ou non de les mettre en évidence avec une génération aléatoire des données de test. Ici, pour mesurer la facilité de tuer un mutant, on mesure à quel pas de test le mutant produit une sortie différente du programme original. Comme les stratégies de génération aléatoire de données de test peuvent produire des

résultats très différents d'une exécution à l'autre, il est nécessaire de produire de nombreuses séquences de test pour obtenir une moyenne « fiable » [8].

Les métriques de testabilité d'une part et l'évaluation de la difficulté à tuer les mutants d'un module permettent d'ordonner respectivement les parties du système. Les informations sont consistantes si les classements obtenus sont compatibles (aux approximations de calcul près). Pour confirmer que les mesures et l'interprétation intuitive sont compatibles, il faut faire de nombreuses expérimentations.

A la suite des premières expérimentations, l'évaluation a été suspendue. Effet, les observations montrent que les résultats ne sont pas concordants. Une analyse plus poussée des fautes montre que les plus difficiles à détecter avec un test aléatoire sont le résultat du remplacement des opérateurs « < » par « <= » ou « > » par « >= » (et vice-versa). En effet, avec une stratégie aléatoire, il y a une faible probabilité que les valeurs d'entrées choisies soient exactement celles requises pour provoquer l'égalité des termes.

Ceci est mis en évidence par l'exemple suivant. Soit un nœud Lustre ayant une entrée entières (`a` et une sortie booléenne `x`. Le nœud est défini par l'équation « `x = a <= 10` ». Lorsque ce nœud est considéré seul, sa contrôlabilité et son observabilité sont maximales. Pourtant, le mutant qui remplace « <= » par « < » est tué par l'unique valeur `a = 10`. La probabilité de trouver cette valeur de façon aléatoire est infime. Par contre, si la valeur d'entrée est connue et soumise au mutant, ce dernier sera tué immédiatement.

Le travail effectué permet de démontrer que les hypothèses posées initialement ne sont pas valides. Soit le modèle de faute utilisé pour l'insertion des fautes n'est pas le même que Satan. Soit la génération de test associée aux métriques n'est pas aléatoire.

5. Conclusion et perspectives

Résumé

La testabilité est une qualité logicielle. Nombreuses sont les propositions de métriques visant à estimer la testabilité. Face à la diversité des métriques, il est nécessaire d'exhiber un guide, précisant les contextes possibles d'utilisation de chacune. Une étape nécessaire pour la construction d'un tel guide est la validation des métriques.

Dans ce contexte, j'ai cherché à valider empiriquement des métriques prédictives de testabilité. Nous avons pu démontrer que la métrique *Depth of Inhéritance Tree* était corrélée aux nombre de méthodes à tester, si l'on re-teste toutes les méthodes héritées au sein d'une classe. Mais la mesure est trop abstraite pour prédire le nombre de tests ou la difficulté à produire ces tests. Pour ce qui est des métriques de testabilité des systèmes Lustre/Scade, elles ne sont pas interprétables dans le contexte de validation supposé initialement (i.e. génération aléatoire et modèle de faute sélectionné). De plus, dans les deux cas,

les métriques sont abstraites pour permettre une identification des parties à modifier pour rendre le système plus testable.

A la lumière de mon travail et d'autres précurseurs [38], j'en conclus que les métriques de testabilité proposées n'apportent pas ce qui est attendu : une évaluation prédictive de la testabilité qui soit *vraiment exploitable* pour concevoir testable. C'est pourquoi, d'autres approches doivent être sérieusement envisagées.

Lorsque l'on teste une application, on se dit parfois que cela serait plus facile si le code possédait certaines « bonnes » caractéristiques. Ainsi, Binder dans [24, 25] et Le Traon *et al.* dans [17, 16, 18] ont commencé à identifier des solutions permettant de faciliter le test ou au contraire de le rendre plus compliqué. On appelle ces solutions des *patrons* ou des *anti-patrons* de testabilité. Les premiers doivent être favorisés lors de la conception, et les seconds sont à éviter.

Une variante de cette approche consiste à définir des « transformations » des parties non testables en parties plus testables [130]. Ici, la notion de « testable » est évaluée par rapport à un critère d'adéquation particulier. Dans la suite, je m'intéresse plus particulièrement aux patrons de testabilité.

Patrons et anti-patrons de testabilité

Les patrons de testabilité identifiés par Binder sont élémentaires et visent à augmenter l'observabilité et la contrôlabilité des classes dans le cadre d'une conception objet. Par exemple, Binder recommande les pratiques suivantes.

– Des méthodes `set` ou `reset` doivent être associées à chaque classe pour permettre la modification de l'état interne de la classe (i.e. l'ensemble des attributs de la classe) indépendamment de son état courant. Cela améliore la contrôlabilité de la classe.

– Une méthode `reporter` doit être associée à chaque classe pour pouvoir observer son état à tout moment (i.e. la valeur des attributs).

– S'il n'est pas possible d'implanter une méthode `reporter`, on peut considérer de définir une méthode `get` pour chaque attribut de la classe.

– Pour faciliter la construction et augmenter la maintenance des suites de test, des assertions peuvent être ajoutées pour chaque classe, sous la forme d'invariants, de pré et de post-conditions.

Les anti-patrons mis en évidence par Le Traon et *al.* sont relatifs à l'identification de situations à éviter dans le contexte du test d'intégration. Il faut notamment éviter des conceptions pour lesquelles une classe pourrait en modifier une autre en suivant deux chemins d'associations différents, faute de quoi, des modifications inconsistantes sont possibles.

Mes perspectives de recherche sur le thème de la testabilité se situent autour des patrons de testabilité. Mon objectif est de construire un catalogue de patrons correspondant à un ensemble d'exigences non-fonctionnelles à intégrer

selon les besoins dans le cahier des charges des nouvelles applications. Le point de départ du catalogue sont les patrons issus des travaux [24, 25, 17, 16, 18].

L'intégration des patrons et des anti-patrons dans le catalogue sera conditionnée par une étape de validation qui permettra d'identifier (1) les bénéfices attendus par le patron, (2) les caractéristiques d'une bonne utilisation et (3) les éventuelles restrictions d'usage. Par exemple, une exigence non-fonctionnelle pour faciliter le test, est d'imposer la production de classes « auto-testables », i.e. des classes spécifiées par des assertions utilisables pour le test. Dans cette situation, il faut privilégier les invariants (bonne utilisation) et éviter les copier-coller de code dans les post-conditions (mauvaise usage) (cf. Chapitre 3).

Un autre exemple d'exigence non-fonctionnelle relative a la testabilité est d'imposer la production de classes observables et contrôlables, par description de méthodes de types `set` et `reset`. Ces méthodes doivent permettre de modifier la valeur de tous les attributs de la classe (bonne utilisation). Cette exigence n'est pas compatible dans un contexte où la sécurité doit être importante car les méthodes `set` et `reset` pourraient être utilisées pour des attaques (restriction d'usage). Il vaut mieux alors privilégier l'utilisation de classes de test qui héritent des classes de l'application (bonne utilisation).

En parallèle, on implémentera dans un environnement de développement existant (tel que Eclipse), des méthodes permettant d'analyser le code ou les spécifications (UML par exemple), pour déterminer si les patrons choisis sont bien appliqués systématiquement et si les anti-patrons ne sont pas présents. Dans un premier temps, cela permettra d'évaluer si les exigences de testabilité sont bien implémentées (critère d'adéquation), et si ce n'est pas le cas de repérer les faiblesses à corriger. Par exemple, on peut s'assurer qu'il existe bien une méthode `reporter` dans chaque classe. Dans un second temps, on pourra implémenter des moyens pour évaluer la qualité des solutions mises en œuvre. Par exemple, on pourra vérifier que chaque méthode `reporter` renvoie bien la valeur de tous les attributs de la classe sans les modifier.

Les avantages d'une telle approche sont les suivants. D'abord, c'est une approche « à la carte » qui s'adapte aux contextes. Les exigences de testabilité sont choisies en fonction des besoins et de leurs propriétés : le catalogue indique les bénéfices attendus. Pour chaque patron, on peut associer un critère « d'adéquation » (et donc des mesures[1]) qui décrit s'il a été appliqué partout où il devait l'être. Ces mesures sont faciles à interpréter : elles décrivent la distance entre l'application requise des patrons et ce qui a été effectivement fait. Si seulement 9 classes sur 10 ont une méthode `reporter`, on obtient un score d'adéquation de 0.9. Enfin, l'analyse permet de repérer les endroits où le patron n'est pas utilisé.

La difficulté pour mettre en pratique cette approche est double. D'abord, il faut être en mesure d'identifier les patrons. Pour cela, en plus des patrons

1. Comme je l'ai dit avant, je ne cherche pas absolument à proposer des mesures, mais c'est souvent une exigence d'un point de vue industriel.

déjà identifiés dans la littérature, il est nécessaire de récolter l'expérience des testeurs en milieu industriel.

Une seconde difficulté est la capacité que l'on aura à formaliser un patron de telle sorte que l'on pourra ensuite s'assurer qu'il est (1) appliqué et que (2) l'application est faite de façon satisfaisante. Par exemple, il serait tentant d'associer à chaque classe l'invariant « `true` » pour obtenir un critère d'adéquation maximum sans faire le travail de spécification. Pour les premiers patrons identifiés dans [24, 25, 17, 16, 18], il est possible de vérifier s'ils sont appliqués partout où ils devraient l'être [71]. Des études et expérimentations plus approfondies doivent être menées pour la poursuite du travail.

Chapitre 5

Perspectives

Parce que ceci n'est pas une fin, mais un commencement !

Pour résumer le travail qui a été mené pendant ces dix dernières années, j'ai exploré différentes problématiques autour du test : génération de données, problème de l'oracle, qualité des tests et testabilité. Ces différentes problématiques ont été abordées selon les cas dans des contextes embarqués ou autour des services domotiques, en Java ou en Lustre. Le fil conducteur de ces travaux est la recherche de solutions *automatisées*, si possible *faciles* à mettre en œuvre.

A l'heure des bilans qui sont exigés par la rédaction de cette HDR, la question qui se pose est de s'assurer que c'est réellement le cas. Un ensemble de perspectives s'ouvrent donc sur l'évaluation à plus large échelle des solutions ou propositions que j'ai pu faire. Ces perspectives relèvent en particulier de l'évaluation empirique.

Parmi les éléments à valider, je souhaite m'assurer que le test combinatoire (en général) et de Tobias (en particulier) peuvent s'insérer dans un contexte industriel. en particulier, les propositions de stratégies de réduction des suites de test doivent être évaluées rigoureusement. Le travail commencé pour l'évaluation de la pertinence de la couverture des paires d'appels de méthode va être poursuivi. Une partie du travail dans le projet TASCCC va permettre de compléter les stratégies de réduction de test avec, en particulier, la définition de sélecteurs garantissant de conserver la valeur du critère d'adéquation choisi de la suite originale.

Dans le même ordre d'idée, la diffusion de l'outil Lutess est aussi un travail que j'ai à cœur. Lutess a été développé pendant plusieurs thèses dans l'équipe VASCO, dont la mienne [193, 66, 247, 227, 163, 206, 192]. Il requiert une description de l'environnement du système sous test en Lustre. C'est un frein à sa diffusion. Les travaux qui ont été menés jusqu'à présents ont permis de montrer que le principe de l'outil pouvait être appliqué à d'autres domaines que les systèmes Lustre/Scade [163]. C'est pourquoi le projet ANR-Emergence Lutess-V3 a été déposé en juin 2010 en collaboration avec Ioannis Parissis. L'objectif est de proposer une nouvelle version de l'outil, encore plus puissante et

plus ouverte, permettant l'utilisation de l'outil dans d'autres contextes que les systèmes réactifs synchrones.

Enfin, un large chantier s'est ouvert sur la question des patrons de testabilité. Un premier travail devra être mené pour décrire les patrons de testabilité déjà proposés et en identifier de nouveaux, notamment via des collaborations avec des testeurs issus du milieu industriel. Le club « Diag 21 » est un lieu précieux pour ce type de discussion.

Pour chaque patron, il faudra être en mesure de caractériser son apport. Ce point là sera certainement très difficile à établir. En effet, en appliquant un patron de testabilité, le programme va être modifié, de telle sorte que de nouvelles fonctionnalités vont être disponibles pour le test. Soient T' et T les suites de tests pouvant être écrites pour le système sous test avec et sans l'application des patrons. L'écriture d'un cas de test donné peut varier dans T et T' grâce aux possibilités offertes par les patrons (par exemple l'oracle ne s'écrit pas de la même manière s'il y a ou non des assertions dans le code ou si un `reporter` est disponible). De plus, les patrons vont peut-être permettre d'écrire d'autres cas de tests qu'il ne serait pas possible d'écrire sinon, (par exemple, en utilisant la méthode `reset`). Comparer la qualité des deux suites avec un minimum de biais est donc un défi.

Enfin et comme toujours, l'apport d'une telle approche devra être démontré. Cela ne sera possible que si un environnement est associé au catalogue. Il permettra de reconnaître les différents patrons et de représenter les parties de l'application qui ne satisfont pas les critères choisis. L'étude préliminaire a montré que les patrons déjà identifiés dans la littérature peuvent être utilisés dans un tel environnement. Reste à poursuivre le travail en ce sens.

Annexe 1

Analyse mutationnelle

Question : « Comment s'assurer de la qualité des données des tests ? »

Réponse : « Application des techniques d'analyse mutationnelle »

A. Introduction

L'analyse mutationnelle a été proposée pour évaluer la qualité des données de test [60]. Cette section est dédiée à la présentation de cette approche que j'ai utilisée à plusieurs reprises dans mes travaux présentés chapitres 2, 3 et 4.

B. Principes de l'approche originale

L'analyse mutationnelle consiste en l'introduction d'un petit changement syntaxique dans le code source du programme sous test dans le but de produire un *mutant* [60]. Par exemple, on peut remplacer un opérateur par un autre ou modifier la valeur d'une constante. Ensuite, le comportement du mutant est comparé à celui du programme original. Si une différence peut être observée sur les sorties des deux programmes, le mutant est *tué*. Sinon, le mutant est vivant. Si un mutant a toujours le même comportement observable que le programme original, on dit que le mutant est *équivalent*. Un mutant est *mort-né* (*stillborn*) s'il est syntaxiquement incorrect. Un mutant est *trivial* s'il est tué par presque tous les cas de tests [187]. L'ensemble des opérateurs de mutations utilisés pour introduire des fautes s'appelle un *modèle de fautes*.

Le but originel de l'analyse mutationnelle est l'évaluation d'une suite de test. Pour décider de la qualité de la suite de test, on produit tous les mutants issus d'un modèle de faute donné. Si une suite de test tue tous les mutants non équivalents, la suite de test est déclarée mutation-adéquate. Si c'est le cas, cela signifie que la suite de test permet de mettre en évidence des différences de comportements entre le programme original et des variantes de celui-ci. A noter, l'analyse mutationnelle ne conclut pas sur la correction du programme

original. Elle conclut sur la capacité de la suite à *discriminer* les comportements des différentes versions produites.

L'adéquation d'une suite de test est évaluée par le score de mutation. Il s'agit du pourcentage de mutants non-équivalents tués par la suite. Soit un programme P. On dénote par M_T le nombre total de mutants produits par rapport à un modèle de faute F. Soient M_E et M_K représentant le nombre de mutants équivalents et le nombre de mutants tués. Le *score de mutation* d'une suite de test T par rapport à un modèle de faute F est défini par : $MS(P,T,F) = \frac{M_K}{M_T - M_E}$. Une suite de test est *mutation-adéquate* si ce score est égal à 1.

L'analyse mutationnelle repose sur deux hypothèses. La première est *l'hypothèse du programmer compétent*. Elle suppose que les programmes sont presque corrects. Le programmeur a compris les exigences mais a pu se tromper sur des éléments de détail. Cette hypothèse justifie le choix du modèle de faute : des fautes élémentaires introduites dans le code.

La seconde hypothèse est *l'hypothèse de couplage*. Elle suppose que si une suite de test permet d'identifier toutes les fautes simples, elle est capable de mettre en évidence des fautes plus complexes. Grâce à cette hypothèse, on insère une *seule* faute *simple* dans chaque mutant. Et on admet que si une suite de test est mutation-adéquate, elle est capable de trouver des fautes plus subtiles. Ceci est confirmé par l'étude de Andrews *et al.*, qui ont démontré que l'analyse mutationnelle permet une bonne indication de la capacité d'une suite de test à détecter des fautes [7].

Cette seconde hypothèse est progressivement remise en cause. Ainsi, on commence à étudier les mutations d'ordre supérieur. Cela consiste en la production de mutants contenant deux fautes ou plus. Y. Jia et M. Harman ont ainsi constaté qu'une suite de test permettant de tuer des mutants de premier ordre n'était pas forcément capable de tuer des mutants d'ordre deux (non équivalents) [141]. Ainsi, en générant aussi des mutants d'ordre 2 ou plus, il est possible de générer des suites de test ayant une plus grande capacité à détecter des fautes.

Au delà de la pertinence des opérateurs de mutation, deux faiblesses de l'analyse mutationnelle ont été identifiées : (1) le coût de l'application de la méthode et (2) la décision quant à l'équivalence d'un mutant. L'analyse mutationnelle est souvent très coûteuse car de nombreux mutants sont produits. Ainsi, Budd estime que le nombre de mutants est globalement proportionnel au nombre de références multiplié par le nombre de données manipulées [39]. Acree *et al.* estiment que le nombre de mutants correspond au carré du nombre de lignes de code source [2].

Deux types de stratégies ont été proposés pour réduire ce coût : la mutation faible et la mutation sélective. La *mutation faible* consiste à comparer les *états internes* (plutôt que les sorties) du programme original et du mutant. Cela permet de détecter des différences de comportements plus tôt dans l'exécution

(dès l'infection de l'état interne plutôt qu'après la propagation vers les sorties). La *mutation N-sélective* est une stratégie qui consiste à éliminer les *N* opérateurs de mutations les plus productifs pour produire un nombre restreint de mutants en conservant le maximum d'opérateur de mutation [186]. Une variante consiste à choisir les mutants de façon aléatoire.

Le second facteur de coût est le problème de l'équivalence des mutants. Décider de l'équivalence d'un mutant est une étape très importante dans le processus d'analyse. En effet, le score de mutation maximum ne peut pas être atteint si un mutant équivalent n'est pas détecté. De nombreux travaux ont été menés pour automatiser en partie la détection de mutants équivalents. Par exemple, des techniques de compilations et d'analyse statique ont été utilisées [183, 185]. Ces stratégies ne peuvent détecter qu'un sous-ensemble de mutants équivalents, car il a été prouvé que « en général il n'y a pas de solution algorithmique pour décider de l'équivalence » [183].

A l'origine, les opérateurs de mutation ont été proposés pour Fortran 77. Depuis, ils ont été adaptés à de nombreux langages de programmation ou de spécifications [242], tels que ADA [59, 187], Java [162], C [1], C#[1] VHDL [179], Petri-Nets, Machines d'états finis (FSM), Statecharts, ... [116, 115, 114, 62].

C. Modèle de faute et opérateurs de mutation

La clef de l'analyse mutationnelle est le modèle de faute. Il est exprimé au travers d'un ensemble d'*opérateurs de mutations*. Le modèle de faute dépend du langage cible. Le modèle original a été proposé pour Fortran 77. Il est adapté à l'évaluation de l'adéquation d'une suite de test *unitaire* et a été dérivé d'une étude des erreurs de programmation des programmeurs. L'ensemble des 22 opérateurs de mutation de Fortran 77 été raffiné pendant plus de 15 ans [59, 184].

Cet ensemble d'opérateurs a été adapté pour d'autres langages. Ainsi, Offutt *et al.* ont défini 65 operateurs pour ADA [187], classifés en 5 sous-ensembles : *operand replacement*, *statement*, *expression*, *tasking* et *coverage*. Le modèle de faute pour Ada est donné Table 13.

Pour évaluer la pertinence d'une suite de *test d'intégration*, un modèle de faute spécifique a été défini dans [58]. L'idée est d'introduire des fautes pour déterminer si les interactions entre les différents modules/composants ont été testées. Ce modèle de faute repose sur l'hypothèse que les erreurs d'intégration apparaissent lorsque des valeurs incorrectes sont échangées entre les modules. Deux ensembles d'opérateurs ont été proposés.

Les opérateurs IM-I correspondent aux mutations appliquées au sein de la fonction appelée. Ces opérateurs sont similaires à ceux de Fortran ou Ada, mais ne sont appliqués que pour provoquer des changements sur les valeurs en entrées ou en sortie de la fonction.

1. http ://www.inria.fr/rapportsactivite/RA2002/triskell/module7.html

Les opérateurs IM-II s'appliquent sur la fonction appelante. Cinq opérateurs sont définis pour remplacer un argument, échanger deux arguments, éliminer un argument, insérer un opérateur unaire devant un argument et éliminer un appel de fonction.

Le modèle de faute proposé pour ADA a été adapté pour d'autres langages. Par exemple, pour Java, le modèle de faute le plus utilisé est celui proposé dans MuJava [162]. Ce dernier comprends les opérateurs applicables au niveau des méthodes (*method level mutation operators*) et d'autres applicables au niveau des classes *class-level mutation operators*). Les opérateurs définis au niveau des classes sont essentiellement des opérateurs *operand replacement* et *expression* adaptés à Java. Les opérateurs définis au niveau des classes permettent d'introduire des fautes spécifiques à l'utilisation de concepts objets, tels que l'héritage ou le polymorphisme.

Operand Replacement Operators

OVV	Variable replaced by a variable.
OVC	Variable replaced by a constant.
OVA	Variable replaced by an array reference.
OVR	Variable replaced by a record reference.
OVP	Variable replaced by a pointer reference.
OVI	Variable initialization elimination.
OCV	Constant replaced by a variable.
OCC	Constant replaced by a constant.
OCA	Constant replaced by an array reference.
OCR	Constant replaced by a record reference.
OCP	Constant replaced by a pointer reference.
OAV	Array reference replaced by a variable.
OAC	Array reference replaced by a constant.
OAA	Array reference replaced by an array reference.
OAR	Array reference replaced by a record reference.
OAP	Array reference replaced by a pointer reference.
OAN	Array name replaced by an array name.
ORV	Record reference replaced by a variable.
ORC	Record reference replaced by a constant.
ORA	Record reference replaced by an array reference.
ORR	Record reference replaced by a record reference.
ORP	Record reference replaced by a pointer reference.
ORF	Record field replaced by a record field.
ORN	Record name replaced by a record name.
OPV	Pointer reference replaced by a variable.
OPC	Pointer reference replaced by a constant.
OPA	Pointer reference replaced by an array reference.
OPR	Pointer reference replaced by a record reference.
OPP	Pointer reference replaced by a pointer reference.
OPN	Pointer name replaced by a pointer name.

Tasking Operators

TEM	ENTRY statement modification.
TAR	ACCEPT statement replacement.
TSA	SELECT alternative replacement.

Statement Modification Operators

SEE	Exception on execution.
SRN	Replace with NULL.
SRR	Return statement replacement.
SGL	GOTO label replacement.
SRE	Replace with EXIT.
SWR	Replace WHILE with repeat-until.
SRW	Replace repeat-until with WHILE.
SZI	Zero iteration loop.
SOI	One iteration loop.
SNI	N iteration loop.
SRI	Reverse iteration loop.
SES	END shift.
SCA	CASE alternative replacement.
SER	RAISE exception handler replacement.

Expression Modification Operators

EAI	Absolute value insertion.
ENI	Neg-absolute value insertion.
EEZ	Exception on zero.
EOR	Arithmetic operator replacement.
ERR	Relational operator replacement.
EMR	Membership test replacement.
ELR	Logical operator replacement.
EUI	Unary operator insertion.
EUR	Unary operator replacement.
ESR	Subprogram operator replacement.
EDT	Domain twiddle.
EAR	Attribute replacement.
EEO	Exception on overflow.
EEU	Exception on underflow.

Coverage operators

CDE	Decision coverage.
CCO	Condition coverage.
CDC	Decision/condition coverage.
CMC	Multiple condition coverage.

Figure 13. *Les opérateurs de mutation pour ADA*

Annexe 2

Trois articles

Trois articles [156, 81, 212] qui couvrent respectivement les chapitres 2, 3, et 4.

Filtering TOBIAS Combinatorial Test Suites

Yves Ledru, Lydie du Bousquet, Olivier Maury, and Pierre Bontron

Laboratoire Logiciels, Systèmes, Réseaux - IMAG
B.P. 72 - F-38402 - Saint Martin d'Hères Cedex - France
{Yves.Ledru,lydie.du-bousquet}@imag.fr

Abstract. TOBIAS is a combinatorial testing tool, aimed at the production of large test suites. In this paper, TOBIAS is applied to conformance tests for model-based specifications (expressed with assertions, pre and post-conditions) and associated implementations. The tool takes advantage of the executable character of VDM or JML assertions which provide an oracle for the testing process. Executing large test suites may require a lot of time. This paper shows how assertions can be exploited at generation time to filter the set of test cases, and at execution time to detect inconclusive test cases.

Keywords: combinatorial testing, model-based specifications, VDM, JML

1 Introduction

Software testing appears nowadays as one of the major techniques to evaluate the conformance between a specification and some implementation. Some may argue that testing only reveals the presence of errors and that conformance may only be totally guaranteed by formal proof, exhaustive testing or a combination of both techniques. Unfortunately, such techniques are often very difficult to apply. In such cases, testing may contribute to increase the confidence that the implementation conforms to its specification. Confidence may result from coverage measurements, from the principles of the test synthesis or selection technique, from the size of the test suite, or from the expertise of the test engineers.

Industrial experiments [5] have shown that test cases within a large test suite often feature a high level of similarity. Many test cases correspond to the same sequence of method calls, with different parameters. Producing these test cases is a repetitive task that reveals the need for appropriate tool support.

From these observations, we have developed the TOBIAS test generator[1] which is aimed at the production of a large set of similar test cases. TOBIAS starts from a test pattern and a description of its instantiations. The tool then unfolds the pattern into a large set of test cases which can be output according to the format of several test tools: calls to VDM operations [12] for VDMTools, Java test cases for JUnit[9] and JML specifications [8,10,11], and test purposes for TGV [7].

[1] TOBIAS was developed within the COTE project, with the support of the French RNTL program. The COTE project gathered Softeam, France Telecom R&D, Gemplus, IRISA and LSR/IMAG.

M. Wermelinger and T. Margaria-Steffen (Eds.): FASE 2004, LNCS 2984, pp. 281–294, 2004.
© Springer-Verlag Berlin Heidelberg 2004

TOBIAS is a typical example of combinatorial testing tool. Its originality is to deal with sequences of method calls, instead of only combination of parameter values. This allows to use the tool with systems that require several interactions before reaching some "interesting" states. It also allows to design test cases in terms of the behavior that has to be exercised.

This paper gives an introduction to TOBIAS. Sect. 2 recalls the principles of conformance testing using executable model-based specifications. Then Sect. 3 gives a quick presentation of the tool and reports on its capability to find errors, on the basis of a simple example, and from the results of industrial experiments. The intrinsic limitation of the tool is that it is subject to combinatorial explosion. Sect. 4 presents two kinds of filters that can be used with TOBIAS to help master the size of test suites. Finally, Sect. 5 draws the conclusions and perspectives of this work.

2 Conformance Testing with Model-Based Specifications

2.1 Checking Conformance with Model-Based Specifications

Model-based specifications describe a system in terms of invariant properties, pre- and postconditions. Some model-based languages, e.g. VDM and JML, have an executable character. It is thus possible to use invariant assertions, as well as pre- and postconditions as oracle for a conformance testing process. VDM assertions can be evaluated in the VDMTools environment against the VDM version of the specified code [6], or compiled into C++ [1]. JML specifications are translated into Java, added to the code of the specified program, and checked against it. The executable assertions are thus executed before, during and after the execution of a given operation (Fig. 1).

One should note that the specification invariants are not exactly checked at the same instants in JML and VDM. In VDM, invariants are evaluated after each statement. In JML, invariants are properties that have to hold in all *visible states*. A visible state roughly corresponds to the initial and final states of any method invocation [8].

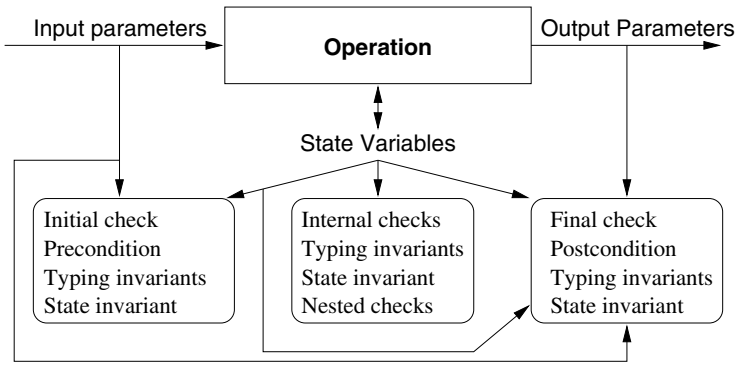


Fig. 1. Dynamic checks associated to operation execution

When an operation is executed in one of those environments, three cases may happen (Fig. 1):

- All checks succeed: the behavior of the operation conforms with the specification for these input values and initial state. The test delivers a PASS verdict.
- An intermediate or final check fails: this reveals an inconsistency between the behavior of the operation and its specification. The implementation does not conform to the specification and the test delivers a FAIL verdict.
- An initial check fails: in this case, performing the whole test will not bring useful information because it is performed outside the specified behavior. This test delivers an INCONCLUSIVE verdict.
  For example, $\sqrt{x}$ has a precondition that x has to be positive. Therefore, a test of a square root method with -1 leads to an INCONCLUSIVE verdict.

2.2 A Small Example in VDM and JML

Let us study a simple example of buffer system (Fig. 2). This system is composed of three buffers. The specification models only the number of elements present in the buffers. A buffer is then modeled with an integer value, which indicates the number of elements in it. The system state is given by the three variables `b1`, `b2` and `b3`.

The maximum size of the system is 40 elements. The system has to distribute the elements amongst the buffers so that: buffer `b1` is smaller than `b2`, which is smaller than `b3`. The difference between `b1` and `b3` should not exceed 15 elements. These constraints leave some freedom on the way to share the elements between buffers. For example, 30 elements can be stored as `b1=5 b2=10 b3=15` or as `b1=8 b2=10 b3=12`.

Three methods are set to modify the systems:

- `Init` resets all buffers to zero.
- `Add(x)` increases the total number of elements of the system of a strictly positive number (`x`) (i.e. it adds `x` elements to the buffers; these elements are distributed in `b1`, `b2`, and `b3`).
- `Remove(x)` decreases the total number of elements in the system of a strictly positive number (`x`) (i.e. it removes `x` elements from the buffers).

The specifications of `Add` and `Remove` keep some implementation freedom: the buffer in which the elements have to be added/removed is not set. For example, if the current state is `8 10 12`, and if 2 elements have to be added, the final state could be `8 10 14`, `8 12 12` , but also `6 12 14`.

2.3 Test Cases

We define a test case as a sequence of operation calls. For example, the following test case initializes the buffer system, adds two elements and removes one of them.

```
TC1 : Init() ; Add(2) ; Remove(1)
```

```
------------------ VDM ------------------
state buffers of
        b1 : nat
        b2 : nat
        b3 : nat

inv mk_buffers(b1,b2,b3) ==
        b1+b2+b3<=40 and 0<=b1 and b1<=b2 and b2<=b3 and b3-b1<=15
init B == B = mk_buffers(0,0,0)
end

operations
Init:() ==> ()
Init() == ...
post b1+b2+b3=0
;
Add: nat ==> ()
Add(x) == ...
pre x<=5 and b1+b2+b3+x<=40
post b1+b2+b3 = b1 +b2 +b3 +x
;
Remove: nat ==> ()
Remove(x) == ...
pre x<=5 and x<=b1+b2+b3
post b1+b2+b3 = b1 +b2 +b3 -x
;
------------------ JML ------------------
public class Buffer{
    public int b1;
    public int b2;
    public int b3;

    /*@ public invariant
      @ b1+b2+b3<=40 && 0<=b1 && b1<=b2 && b2<=b3 && b3-b1<=15; */

    /*@ requires true;
      @ modifies b1, b2, b3;
      @ ensures b1==0 && b2==0 && b3==0; */
    public Buffer(){}

    /*@ requires true;
      @ modifies b1, b2, b3;
      @ ensures b1==0 && b2==0 && b3==0;   */
    public void Init(){...}

    /*@ requires x<=5 && b1+b2+b3+x<=40 && x>=0;
      @ modifies b1, b2, b3;
      @ ensures b1+b2+b3==\old(b1+b2+b3)+x;
    */
    public void Add(int x){...}

    /*@ requires x<=5 && x<=b1+b2+b3 && x>=0;
      @ modifies b1, b2, b3;
      @ ensures b1+b2+b3==\old(b1+b2+b3)-x;  */
    public void Remove(int x){...}
}
```

Fig. 2. Buffer example specification in VDM and JML

Each operation call may lead to a PASS, FAIL or INCONCLUSIVE verdict. As soon as a FAIL or INCONCLUSIVE verdict happens, we choose to stop the test case execution and mark it with this verdict. A test case that is carried out completely receives a PASS verdict.

For example, in the context of the above specification, the test cases TC2 and TC3 should produce an INCONCLUSIVE verdict. If test TC4 is executed against a "correct" implementation, it should produce a PASS.

```
TC2 : Init() ; Add(-1)
TC3 : Init() ; Add(2) ; Remove(3)
TC4 : Init() ; Add(3) ; Remove(2) ; Remove(1)
```

3 TOBIAS

TOBIAS is a test generator based on combinatorial testing [4]. Combinatorial testing performs combinations of selected input parameters values for given operations and given states. For example, a tool like JML-JUnit [3] generates test cases which consist of a single call to a class constructor, followed by a single call to one of the methods. Each test case corresponds to a combination of the parameters of the constructor and a combination of the parameters of the method.

3.1 Principles of TOBIAS

TOBIAS adapts combinatorial testing to the generation of sequences of operation calls. This allows to reach states that do not correspond to a single call to a constructor. It also allows to design tests in terms of behaviors rather than states. For example, in the specification of the buffers, the initial state is fixed (`0 0 0`), and it is not possible to add more than 5 elements at a time. Therefore a rather long sequence is needed (at least 8 operations) to test the behavior of the system at its limits (40 elements).

The input of TOBIAS is composed of a test pattern (also called test schema) which defines a set of test cases. A pattern is a bounded regular expression involving the operations of the VDM or JML specification. TOBIAS unfolds the pattern into a set of sequences, and then computes all combinations of the input parameters for all operations of the pattern.

The patterns may be expressed in terms of *groups*, which are structuring facilities that associate a method, or a set of methods to typical values. For example, let us consider schema `S1`:

$$\begin{cases} \texttt{Init() ; Add_Gr} \\ \text{with } \texttt{Add_Gr} = \{\texttt{Add}(x) | x \in \{1,2,3,4,5\}\} \end{cases}$$

`Add_Gr` is a set of 5 instantiations of calls to the method `Add`. The pattern `S1` is unfolded into 5 test sequences:

```
S1-TC1 : Init() ; Add(1)
S1-TC2 : Init() ; Add(2)
...
S1-TC5 : Init() ; Add(5)
```

Groups may also involve several operations. Let `S2` and `S2'` be two other examples of schemas:

$$\begin{cases} \texttt{S2} = \texttt{Init() ; Modify_Gr \{1..2\}} \\ \texttt{S2'} = \texttt{Init() ; Add(2) ; Modify_Gr \{1..2\}} \\ \text{with } \texttt{Modify_Gr} = \{\texttt{Add}(x) | x \in \{1,2,3,4,5\}\} \cup \{\texttt{Remove}(y) | y \in \{1,3,5\}\} \end{cases}$$

`Modify_Gr` is a set of (5+3)=8 instantiations. The expression `{1..2}` means that the group is repeated 1 to 2 times. The patterns `S2` and `S2'` are unfolded into 8+(8*8)=72 test sequences:

```
S2-TC1 : Init() ; Add(1)
  ...
S2-TC8 : Init() ; Remove(5)
S2-TC9 : Init() ; Add(1) ; Add(1)
  ...
S2-TC72 : Init() ; Remove(5) ; Remove(5)
         ----------------------
S2'-TC1 : Init() ; Add(2); Add(1)
  ...
S2'-TC72 : Init() ; Add(2); Remove(5) ; Remove(5)
```

Group definitions may be reused in several schemas, leading to some level of modular construction.

3.2 Finding Errors with Tobias

Let us consider the buffer problem specification. We have proposed an implementation, containing one error: the `Remove` operation can set one of the three buffers to a negative value while keeping the total number of elements positive, which is forbidden by the specification invariant. This solution was implemented in VDM and Java. We executed the tests corresponding to the schemas `S2`, `S2'`, `S3`, and `S4`.

$$\begin{cases} \texttt{S3} = \texttt{Init() ; Add(5) 7 ; Modify_Gr \{1..2\}} \\ \texttt{S4} = \texttt{Init() ; Add_Gr ; Modify_Gr \{1..3\}} \\ \text{with } \texttt{Modify_Gr} = \{\texttt{Add}(x) | x \in \{1,2,3,4,5\}\} \cup \{\texttt{Remove}(y) | y \in \{1,3,5\}\} \\ \text{and } \texttt{Add_Gr} = \{\texttt{Add}(x) | x \in \{1,2,3,4,5\}\} \end{cases}$$

The schema `S2'` was introduced in order to decrease the number of inconclusive verdicts of schema `S2`. The schema `S3` aims at testing the behavior of the application at the "limits", i.e. when the buffer system is quite full. The schema `S4` was built to produce lots of test sequences (some kind of "brute force approach"). The following table gives the verdicts of the various test cases.

Schema	Test cases	Pass	Inconclusive	Fail
S2	72	39	33	0
S2'	72	48	22	2
S3	72	57	15	0
S4	2920	1887	773	260

As expected, the error is detected (by schemas `S2'` and `S4`). `S3` is aimed at testing full buffers and can not reveal the error; `S2` is a small test suite with a lot of inconclusive test cases, which does not achieve enough exhaustiveness to find the error.

This example shows that TOBIAS test suites are able to find errors. Here the error was not straightforward, and small test suites such as `S2` are not sufficient to detect it (actually, the error may only happen if two `Add` operations have been performed). Longer test sequences are needed, such as the ones generated by `S4`.

We have carried out several experiments with TOBIAS. In [12], we report on a VDM case study. This case study showed that the development of a TOBIAS test suite requires the same amount of effort as a simple manual test suite. It also shows that since TOBIAS test suites achieve more exhaustiveness (by exercising all combinations in the schema), they reveal some errors that are often overlooked by manual test suites.

3.3 Industrial Case Study

Two experiments were also carried out on an industrial case study provided by Gemplus (a smart card manufacturer). The case study is a banking application which deals with money transfers. It has been produced by Gemplus Research Labs and is somehow representative of java applications connected to smart cards. The application user (i.e. the customer) can consult his accounts and make some money transfers from one account to another. The user can also record some "transfer rules", in order to schedule regular transfers. These transfer rules can be either saving or spending rules.

The case study is actually a simplified version of an application already used in the real world. The code length is 500 lines. The specification was given in JML. Most preconditions are set to true. Since the application deals with money, and since some users may have malicious behaviors, the application is expected to have defensive mechanisms. Thus, it is supposed to accept any entry, but it should return error messages or raise exceptions if the inputs are not those expected for a nominal behavior. It is a typical example of *defensive programming* style. This means that test cases do not produce INCONCLUSIVE verdicts.

Two testing experiments with TOBIAS were carried out from this case study. The first one was carried out by a Gemplus team. They have first used their internal testing methodology to elaborate an informal test plan. It includes 40 nominal "scenarios" (a scenario is an informal description of a test case). It was possible to abstract those scenarios and express them with only 5 TOBIAS schemas, which were unfolded into 1900 executable test cases. This experiment shows that TOBIAS schemas are more compact than test cases. Moreover, by abstracting test cases into schemas, we ended up with more general schemas than the original scenarios, resulting into 50 times more test cases.

This experiment was considered as a success by our industrial partner. On the one hand, TOBIAS schemas were perceived as an interesting structuring mechanism for the design of tests. On the other hand, the tool allows to complete the original test suites by achieving some kind of exhaustiveness.

The second experiment was carried out by our research team. From the informal requirements, we deduced 17 TOBIAS schemas, mainly to simulate malicious behaviors. They were unfolded into 1100 test sequences, representing 40 000 Java code lines (for JUnit). It took 6 person-day to analyze the specification, produce the abstract scenarios, execute the tests and analyze the traces. (The test suite execution time by itself takes only 1 hour.) The execution of the test cases revealed 16 errors, in either the Java code or in the corresponding JML specification. A discussion with the Gemplus team after the experiment showed that we discovered most of the errors in the code. The only remaining error was impossible to detect because the JML specification did not address this feature of the system.

3.4 Conclusion

TOBIAS is a combinatorial tool that instantiates a large set of test sequences from an abstract description. It aims to be a simple and easy to use tool for combinatorial testing which supports and amplifies the creative work of a test engineer. The tool can also be used in order to express existing test sequences in a more abstract way, which helps the test engineer to structure his test suite.

Several experiments have shown that it is well-suited for a conformance testing activity, in conjunction with executable model-based specification. These experiments include both research and industrial case studies. In all these case studies, the tool allowed to detect errors in the implementation under test.

4 Handling Large Test Suites

The major strength of TOBIAS is also its main weakness. The combinatorial approach allows to produce large test suites, whose systematic character helps to detect errors. But the size of the test suite may also become a problem when too many resources are needed to run the tests and analyze their results. The first way to avoid combinatorial explosion is to design test schemas with great care. By avoiding useless calls in the schema and by keeping the possible values of a parameter to a minimum, we were able to control the number of generated test cases in the experiments we led so far. Nevertheless, two additional mechanisms have proved to be useful to reduce the amount of tests. They filter the set of test cases either at execution or at generation time.

The typical size of a TOBIAS test suite ranges from hundreds to thousands of tests. Today, the largest test suite generated by the tool counts about 40 000 test cases. Several experiments have shown that such test suites include a large number of inconclusive test cases. For instance, schema `S4` leads to 773 INCONCLUSIVE test cases. Although these are useful to test preconditions, their execution may require a significant amount of time, and it makes sense to try to eliminate some of them.

This section will discuss two techniques, based on predicates, that are used to control or cope with the size of TOBIAS test suites.

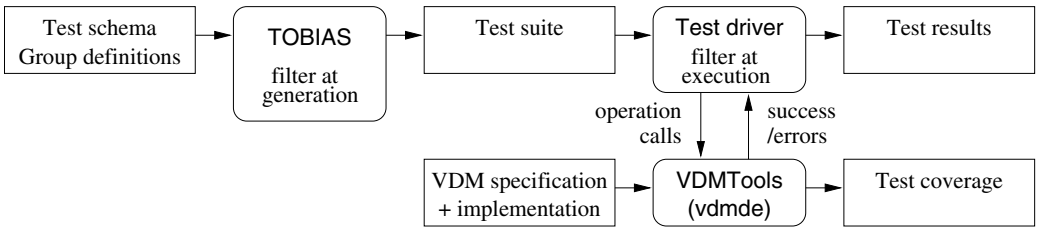


Fig. 3. Exploitation of TOBIAS generated tests in the VDM environment

- Filtering at the execution time: a test driver takes into account the results of the oracle and filters out test cases with a prefix that has already failed one of the checks.
- Filtering at the generation time: the test case generator can take into account a predicate which filters out test cases whose input parameters do not fulfill the predicate.

4.1 Filtering at Execution Time

By construction, TOBIAS test suites are made up of similar test cases. One of the possible similarities is that several test cases may share a common prefix. For example, schema `S2'` includes 9 test cases which start with prefix `init() ; Add(2) ; Remove(5)`. If the execution of the prefix is erroneous (or INCONCLUSIVE) for any of the 9 test cases, and if the implementation is deterministic, the 8 remaining test cases will also exhibit an erroneous (or INCONCLUSIVE) prefix. It is useless to execute the test cases with this prefix.

Therefore, we have developped test drivers that take this property into account. Every erroneous prefix is stored during the execution of the test suite. Before playing a new test case, it is compared to the stored erroneous prefixes and discarded if it matches any of them.

The nice property of this filtering scheme is that it takes advantage of the executable VDM/JML assertions (mainly preconditions) to help filter the test suite. It does not require additional input from the user. Of course, this filtering scheme is better suited to specifications that include strong preconditions. It will not provide any benefit for specifications which adopt a defensive programming style, with all preconditions set to true.

Buffer Case Study

The following table shows the execution time for the 4 test suites of the buffer problem. The tests were executed on a Pentium III/500MHz/128Mb linux machine.

					VDM		JML	
Schema	Test cases	Pass	Incon-clusive	Fail	Exec. Time	Exec. Time with filtering	Exec. Time[2]	Exec. Time with filtering
`S2`	72	39	33	0	2 s.	1.205 s.	0.709 s.	0.134 s.
`S2'`	72	48	22	2	2 s.	1.674 s.	0.524 s.	0.157 s.
`S3`	72	57	15	0	6 s.	4.596 s.	0.400 s.	0.269 s.
`S4`	2920	1887	773	260	2min 05 s.	9.930 s.	14.307 s.	1.352 s.

[2] With JML-Junit environment.

The tests were executed with filtering and non-filtering test drivers. As expected, the optimized drivers execute the test suites quicker than the original test drivers. The speed up is more important when there are many INCONCLUSIVE (or FAIL) verdicts. Both kinds of drivers reveal the implementation error.

Banking Application

The banking application is a typical example of defensive programming. The preconditions of operations are usually set to `true`, in order to face all kinds of unexpected inputs. With such applications, the test cases never end up with an INCONCLUSIVE verdict. Therefore, filtering at execution time can only take into account the prefixes which lead to a FAIL verdict. In the banking application, this corresponds to a small number of tests (at most 1%). Hence, filtering at execution time does not lead to a significant speed-up.

The following experiment was led to make sure that the filtering mechanism did not slow down execution significantly when there are no INCONCLUSIVE test cases. We have executed the tests with both JUnit and our driver for Java, on a Pentium III/500MHz/128Mb Windows machine. This one has some limitations. For instance, it is not possible to set several instantiations for a constructor method in the same test suite (a test suite here is the set of test cases derived from one schema). As a result, some the test suites were not executable with our driver. The following table shows the execution time for the tests. As it can be noticed, the execution time with our driver is shorter than with JUnit.

Schema	nb of tests	with Junit	with our driver	Speedup
One account creation	162	0.671 s.	0.410 s.	0.39
Several account creations	96	0.401 s.	0.030 s.	0.93
One account deletion	30	0.160 s.	0.060 s.	0.63
Several account deletions	512	2.553 s.	0.641 s.	0.75
Several transfers	1	0.050 s.	0.060 s.	-0.20
Incorrect transfer (1)	16	0.251 s.	0.240 s.	0.04
Incorrect transfers (2)	60	0.520 s.	0.601 s.	-0.16
Incorrect transfers (3)	2	0.040 s.	0.030 s.	0.25
Use of infinity values	12	0.141 s.	0.110 s.	0.22
12 digit numbers	4	0.070 s.	0.060 s.	0.14
Transfers and account deletion	12	0.140 s.	0.080 s.	0.43
Transfer rules (1)	120	2.163 s.	2.204 s.	-0.02
Transfer rules (2)	96	1.733 s.	1.592 s.	0.08
Transfer rules (3)	12	0.341 s.	0.180 s.	0.47
Transfer rules (4)	8	0.301 s.	0.110 s.	0.63
Saving rule and account deletion	3	0.591 s.	0.050 s.	0.91
Spending rule and account deletion	3	0.711 s.	0.080 s.	0.87

Our experiments (the banking application and the buffers) show that our test driver is faster than JUnit. There are several reasons:

- algorithmic reasons: when a large number of tests have the same prefix, and when this prefix leads to a FAIL or an INCONCLUSIVE verdict, these tests (which are amongs the longest of the test suite) are not executed with our driver. For example, in the `S4` schema, 760 tests are discarded, which corresponds to a quater of the tests.
- technical reasons: JUnit has a generic character and uses introspection/reflection facilities to discover th tests stored in a class. Our test driver is directly compiled from the test suite and does not have to find this information. Moreover, we suspect that the graphical interface of JUnit (which were used during our tests) also slows down the execution. The banking experiment, which never leads to INCONCLUSIVE verdicts, shows that these technical reasons alone result in significant speedups.

4.2 Filtering Test Cases at Generation Time

The previous section has shown that preconditions and other assertions could filter a lot of INCONCLUSIVE test cases at execution time. TOBIAS provides an other mechanism which allows to eliminate some test cases at generation time, using a VDM predicate as a filter.

Let us consider again the schemas `S2` to `S4`. A lot of the inconclusive verdicts are due to the fact that the total number of removed elements is greater than the total number of added elements. One idea is to complete the schema definitions with a constraint on the combination of parameters. Schema `S2` was defined as:

$$\begin{cases} \texttt{S2} = \texttt{Init()} \ ; \ \texttt{Modify_Gr} \ \{1..2\} \\ \text{with } \texttt{Modify_Gr} = \{\texttt{Add}(x) | x \in \{1,2,3,4,5\}\} \cup \{\texttt{Remove}(y) | y \in \{1,3,5\}\} \end{cases}$$

and unfolds into 72 test cases:

```
S2-TC1 : Init() ; Add(1)
...
S2-TC72 : Init() ; Remove(5) ; Remove(5)
```

Let `add1` be the sequence of `x` parameters associated to each call to `Add` in a given test case, and `del1` be the sequence of `y` parameters associated to each call to `Remove`. For example, test case `S2-TC1` corresponds to `add1 = [1]` and `del1 = []`, and test case `S2-TC72` corresponds to `add1 = []` and `del1 = [5,5]`. The following VDM constraint expresses that the sum of the elements of `add1` is greater than or equal to the sum of elements of `del1`.

```
S2_constraint : () ==> bool
S2_constraint() == (
  dcl sommeAdd : nat:=0;
  dcl sommeDel  : nat:=0;
  for a in add1 do (sommeAdd:= sommeAdd+a;);
  for d in del1 do (sommeDel:= sommeDel+d;);
return(sommeAdd>=sommeDel) )
```

TOBIAS has been extended to generate sequences `add1` and `del1` for each unfolded test case, and then pass them to a VDM interpreter which evaluates the constraint. Test cases which fail to verify the constraint are discarded from the generated test suite.

With schema `S2` and `S2_constraint`, the resulting test suite only features 48 test cases, among wich only 9 lead to INCONCLUSIVE verdict (instead of 33).

This first example has shown that constraints can get rid of INCONCLUSIVE tests at generation time. But this technique requires the test engineer to write the constraint, while filtering at execution time took advantage of the existing preconditions. Still, filtering at generation time is an interesting mechanism, because constraints can be motivated by other concerns than simply ruling out INCONCLUSIVE tests, as will be shown in the following example.

Application to the Banking Problem

It was already mentioned that the banking problem does not lead to INCONCLUSIVE verdicts. Still, constraints can be used in this case study to master combinatorial explosion by adding further test hypotheses.

One of the 17 schemas is named "Several account deletions". It is unfolded into 512 test cases, which is actually the highest number of test cases in this experiment. This schema is defined as follows:

```
Create {2..2}; Delete {3..3}; CreateErr; Delete
```

where `Create` has only one instance, `CreateErr` has two instances and `Delete` has four instances corresponding to four possible values of its only integer parameter. This schema is unfolded thus into 1*1*4*4*4*2*4= 512 test cases.

In order to reduce this size, one may wish to express additional test hypotheses. For example, `Delete` can be instantiated as `Del(10)`, `Del(11)`, `Del(12)`, or `Del(13)`. A first test hypothesis may be that the order of the first three instances of `Delete` is not significant. Therefore the following test sequences are equivalent for the tester:

```
Create; Create; Del(10); Del(11); Del(12); CreateErr; Del(10)
Create; Create; Del(12); Del(11); Del(10); CreateErr; Del(10)
```

Let `del1` be the sequence of parameters associated to the first three calls to `Delete`, the following constraint expresses that only the sequence where the parameters appear in ascending order will be kept:

```
forall val1, val2 in set inds del1 &
              val1<val2 => del1(val1)<=del1(val2)
```

Another test hypothesis (here a regularity hypothesis) is that it does not make sense to try to delete the same account more than twice. This hypothesis can be enforced if the four `Delete` calls refer to at least three different accounts. Let `del2` be the single element sequence corresponding to the fourth call to `Delete`, this constraint can be expressed as:

```
card(elems(del1) union elems(del2))>=3
```

These hypotheses are then grouped into the following constraint.

```
Delete_C : () ==> bool
Delete_C () == (
 return(
        (forall val1, val2 in set inds del1 &
              val1<val2 => del1(val1)<=del1(val2))
        and
         card(elems(del1) union elems(del2))>=3
        )
)
```

When TOBIAS takes this constraint into account, the number of unfolded test cases is reduced from 512 to 80. From a test engineer point of view, this reduction may be interesting since it results in a better balance of the whole test suite. Thus this test schema no longer appears as the most significant one.

5 Conclusion

This paper has presented TOBIAS, a test case generator based on the combinatorial unfolding of test schemas. It has shown how the tool can be combined with executable model-based specifications in a conformance testing process. TOBIAS aims to be a simple and easy to use tool for combinatorial testing which supports and amplifies the creative work of a test engineer. The tool has proved to be useful to detect errors in several case studies, including an industrial experiment where Java code was tested against a JML specification.

Other tools adopt a combinatorial testing approach in combination with model-based specifications. Korat [2] and JML-JUnit [3] generate combinations of a call to a constructor followed by a single call to one of the methods of the class. Korat uses an elaborate generator to cover a wide range of calls to the constructor. TOBIAS adds the possibility to express a sequence of method calls in the test schema, allowing to reach states that cannot be created with the constructor and to express tests on the basis of a behavior.

This paper has also presented filters that help master the size of the generated test suites. Filtering at execution time is an interesting feature because it does not require additional inputs from the test engineer. It allows to filter a significant percentage of the tests for specifications with strong preconditions.

Filtering at generation time requires that the test engineers express some constraints on the schema parameters. But it is a more flexible filtering mechanism and allows to translate test hypotheses into filtering constraints.

Perspectives. Several improvements may be considered when filtering at generation time. On the one hand, several typical constraints could be added as primitives of the schema language. For example, a variant of iteration of a method could mandate parameter values to be all different, or to appear in ascending order. On the other hand, a library of constraints could be developed to express frequently used testing constraints. Moreover, since constraints translate test hypotheses, the library could be structured in terms of these higher level concerns.

Still, improvements in filtering capabilities should not prevent the test engineers from handling combinatorial explosion by a careful design of their test schemas. Further methodological advances are needed to guide the elaboration of test schemas. We expect that further experiments with TOBIAS with help us to progress in that direction.

References

1. B.K. Aichernig. Automated black-box testing with abstract VDM oracles. In M. Felici, K. Kanoun, and A. Pasquini, editors, *SAFECOMP'99*, LNCS 1698, pages 250–259. Springer, 1999.
2. 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 July 2002. IEEE.
3. 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.
4. D.M. Cohen, S.R. Dalal, J. Parelius, and G.C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 13(5):83–88, 1996.
5. 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.
6. The VDM Tool Group. VDM-SL Toolbox User Manual. Technical report, IFAD, October 2000. ftp://ftp.ifad.dk/pub/vdmtools/doc/userman_letter.pdf.
7. T. Jéron and P. Morel. Test Generation Derived from Model-checking. In *Computer Aided Verification (CAV)*, LNCS 1633. Springer, 1999.
8. The Java Modeling Language (JML) Home Page. http://www.cs.iastate.edu/~leavens/JML.html.
9. JUnit. http://www.junit.org.
10. 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.
11. 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, June 2002.
12. O. Maury, Y. Ledru, P. Bontron, and L. du Bousquet. Using TOBIAS for the automatic generation of VDM test cases. In *Third VDM Workshop (in conjunction with FME'02)*, july 2002.

J Autom Reasoning
DOI 10.1007/s10817-009-9132-y

Reusing a JML Specification Dedicated to Verification for Testing, and Vice-Versa: Case Studies

Lydie du Bousquet · Yves Ledru · Olivier Maury · Catherine Oriat · Jean-Louis Lanet

Received: 5 October 2008 / Accepted: 6 May 2009
© Springer Science + Business Media B.V. 2009

Abstract Testing and verification are two activities which have the same objective: to ensure software dependability. In the Java context, the Java Modelling Language (JML) has been proposed as specification language. It can be used both for verification and test. Usually, the JML specification is designed with a specific purpose: test or verification. This article addresses the question of reusability of a JML specification provided for one activity (resp. verification or test) in the other context (resp. test or verification). Two different case studies are considered.

Keywords Software dependability · Java · JML · Verification · Testing

1 Introduction

Testing and verification are two classical approaches used in order to achieve software dependability. As it has been noticed in the *Test And Proof* conference series (TAP 2007, 2008, and 2009), verification and testing have been pursued by distinct communities using rather different techniques and tools. During the last decade, an increasing number of efforts were carried out to combine both approaches.

In this article, we focus on the validation and verification of Java programs. The Java Modelling Language (JML) has been proposed as specification language for those applications. It can be used both for verification and test activities, and a large amount of tools have been proposed to support both of them.

L. du Bousquet (✉) · Y. Ledru · O. Maury · C. Oriat
Laboratoire d'Informatique de Grenoble (LIG Labs),
Universités de Grenoble, BP 72, 38402 Saint Martin d'Hères cedex, France
e-mail: lydie.du-bousquet@imag.fr
URL: http://www.liglab.fr/

J.-L. Lanet
Laboratoire XLIM, Université de Limoges, 123,
avenue Albert Thomas, 87060 LIMOGES CEDEX, France

Published online: 02 June 2009

One would think that using the same language for test and verification would ease the combination of both activities. In reality, it still seems not to be so common. Indeed testing and verification have two different objectives, and a (JML) specification designed with a specific purpose (test or verification), may not be appropriate for the other activity. Verification aims at assessing the correctness of some properties for all inputs in all "situations". On the other hand, testing is an incomplete validation approach. It aims at detecting faults but it cannot guarantee that all faults are discovered. Since verification is difficult, it requires a good knowledge of the application code, and is limited by the power of the tools (for instance, it is very difficult to verify some properties on floats). Hence, properties written for verification may specify a subset of the application, and this is not appropriate to detect faults on the unspecified parts of the application. Since testing is easier to carry out (it requires less knowledge of the code and is less restricted by the power of tools), testers may pay less attention to the way the specification is written, making it less usable for verification.

This article addresses the question of reusability of a JML specification provided for one activity (resp. verification or test) in the other context (resp. test or verification). Two different case studies are considered. The first one is a Banking application, which has first been verified then tested. The second one is a Home Network Services application, which has been tested and then verified. In both cases, evolution of the JML specification had to be undertaken in order to ease the second step of the validation (resp. test or verification).

In the following, Section 2 presents JML. Section 3 details the work done to test the Banking application after the verification. Section 4 describes the work done to verify the Home Network Services application after it was tested. Section 5 concludes with the lessons learnt during both experiments.

2 JML: Language and Tools

This section briefly presents JML and some associated tools.

2.1 The JML Language

Java Modelling Language (JML) is designed to specify Java programs by expressing formal properties and requirements on classes and their methods. The Java syntax of JML makes it easier for Java programmers to read and write specifications. The language is based on Java, with some additional keywords and logical constructions. Examples of JML assertions are given in Figs. 2 and 6. For more details, see [14, 16].

The JML specification appears as special purpose Java comments: between /*@ and @*/ or starting with //@. The specification of each method precedes its interface declaration. This follows the usual convention of Java tools, such as JavaDoc, which put such descriptive information in front of the method.

JML annotations adopt a "design by contract" style of specifications, which relies on three types of assertions: class invariants, preconditions and postconditions.

– *Invariants* are properties that have to hold in all visible states. A visible state roughly corresponds to the initial and final states of any method invocation [16].

- *Preconditions* in the *requires* clause give the assertions that must hold before this method can be called. If that is not true, then the method is under no obligation to fulfil the rest of the specified behaviour.
- *Postconditions* are expressed in the *ensures* clauses. They express the results and the properties expected to hold just after the method execution. The *exsures* clause is a special kind of postcondition (signal clause) for exception specification.

JML extends the Java syntax with several keywords.

- `\result` is the value returned by the method. It can only be used in *ensures* clauses of a non-void method.
- `\old`. An expression of the form \old(Expr) refers to the value that the expression Expr had in the initial state of a method.
- `\forall` and `\exists` are universal and existential quantifiers.

JML can also be used to define *annotation statements* that may be interspersed with Java statements in the body of a method. For instance, a loop statement can be annotated with *loop invariants* or *variant functions*, that are written above the loop itself. Both are used to help verification of the partial correctness and the termination of a loop statement. Moreover, various assertions can be used to specify abstract data types. For example, the *initially* clause allows one to define properties that must be established by constructor methods of a class.

JML is more expressive than the Java assertion mechanism. The assertion mechanism was introduced in version 1.4 of the Java language. An assertion is a boolean condition that can be evaluated at run-time. In Java, options to the compiler allow turning the evaluation of assertions on and off. Java assertions are a simpler mechanism than JML:

- Java assertions are pure Java expressions and do not benefit from the additional constructs of JML (e.g. `\old`, `\result`, `\forall` and `\exists`).
- While JML features various kinds of assertions (invariants, pre- and postconditions), Java assertions are of a single kind. With JML, an invariant is written once and checked after each method invocation. To obtain a similar result with Java assertions, the invariant must be copied at all places where it must be checked.
- The only tool supporting Java assertions is the Java compiler, while JML is associated with several verification and testing tools.

2.2 JML for Testing: Principles and Tools

The JML release consists of several tools to check the syntax and typing of specifications [5]. It also includes the `jmlrac` tool (JML runtime assertion checker), which uses the JML annotations to add runtime assertions to the compiled Java code [8].

The assertions are executed before, during and after the execution of a given method or constructor call. When a method (or constructor) is executed, three cases may happen.

All checks succeed: the behaviour of the method conforms to the specification for these input values and initial state. The test delivers a *Pass* verdict.

An *intermediate or final check fails*: this reveals an inconsistency between the behaviour of the method and its specification. The implementation does not conform to the specification and the test delivers a *Fail* verdict.

An *initial check fails*: in this case, performing the test will not bring useful information because it is performed outside of the specified behaviour. This test delivers an *Inconclusive* verdict. For example, $\sqrt{x}$ has a precondition that requires x being positive. Therefore, a test of a square root method with a negative value leads to an *Inconclusive* verdict. But, if the square root method is called with a negative value inside a method under test, then a *Fail* verdict is delivered.

The code generated by `jmlc` can be used in combination with JUnit [15] in a testing process. The JML-JUnittool [9] is a combinatorial testing tool which generates simple test cases consisting of a call to one of the constructors of the given class, followed by a single call to one of the methods of the object under test. The tool generates combinations of selected values of the constructor and method parameters to result in a large set of test cases. The tool then exploits JUnit to run the tests and `jmlc` to provide an executable oracle.

For the testing experiment in Section 3 of this article, we have mainly used two tools: Jartege and Tobias. Both use JML as test oracle.

Jartege allows random generation of unit tests for Java classes specified in JML [24]. As in the JML-JUnit tool, JML assertions are used as a test oracle. Jartege randomly generates test cases, which consist of a sequence of constructor and method calls for the classes under test. The random aspect of the tool can be parameterized by associating weights to classes and operations, and by controlling the number of instances which are created.

Tobias is a tool for combinatorial testing. Unlike JML-JUnit that generates test cases which consist of a *single* call to a class constructor, followed by a *single* call to one of the methods (see [8]), Tobias supports combination of calls to the methods [18, 21]. Tobias is available as an Eclipse plug-in [17].

2.3 JML and Verification

Several tools are available for formal verification of Java programs specified in JML [4, 5]. The ESC/Java tool aims at identifying (and correcting) errors early in the development (static validation) [7, 13]. It does not aim to provide a formal verification of the code. JACK [2, 6], Why/Krakatoa [12, 20] and KeY [1, 3] are three available environments for verification of Java programs specified in JML.

2.3.1 ESC/Java

ESC/Java[1] is an *Extended Static Checking* tool. The verification is *static* since the code is verified without being executed in a Java Virtual Machine. It is *extended* since the tool detects more errors than can be detected with traditional static analysis.

ESC/Java uses the Simplify prover to reason about the program semantics. It raises warnings in case of classical runtime errors, such as null dereferences, array bound errors, type cast errors, etc. It also warns about synchronization errors

[1] ESC/Java can be downloaded at http://kind.ucd.ie/products/opensource/ESCJava2/download.html

in concurrent programs (race conditions and deadlocks). Finally, ESC/Java issues warnings if the source code violates the JML assertions.

2.3.2 JACK

JACK[2] (Java Applet Correctness Kit) is a tool for the validation of Java applications annotated in JML [2]. JACK implements a weakest precondition calculus to automatically generate proof obligations for each path of the control flow. They can both be discharged to automatic and interactive theorem provers such as Coq or Simplify. Proof obligations are first expressed in an intermediate representation, and are then translated into the adequate language for the chosen prover. The tool is integrated into Eclipse.

2.3.3 Why/Krakatoa

"Why"[3] is a generic platform for deductive program verification [12]. The core of the platform ("Why" tool) produces verification conditions and sends them to existing provers. Several provers can be used: proof assistants such as Coq, PVS, Isabelle/HOL, HOL 4, HOL Light, Mizar and decision procedures such as Simplify, Alt-Ergo, Yices, Z3, CVC3, etc. Krakatoa is dedicated to the translation of Java programs annotated in JML, into the input language of "Why" (similar to ML) [20].

2.3.4 KeY

The KeY[4] System is a formal software development tool that aims to integrate design, implementation, formal specification, and formal verification of object-oriented software as seamlessly as possible. At the core of the system is a theorem prover for a first-order Dynamic Logic for Java. The tool has an easy-to-use graphical interface and seamlessly integrates automated and interactive verification [1, 3]. KeY also uses Simplify.

3 From Verification to Testing

This section focuses on the work done to reuse a specification dedicated to verification for testing activities. Section 3.1 deals with the presentation of the case study, a banking application. Section 3.2 briefly describes the JML specification produced for verification. Section 3.3 details the work done for test, and especially focuses on the errors found. Section 3.4 proposes a partial conclusion.

3.1 The Banking Application Case Study

The banking application case study, proposed by Gemplus, deals with money transfers [10]. The application administrator (the bank officer) can create accounts. The

[2]The current version can be downloaded at http://www-sop.inria.fr/everest/soft/Jack/jack.html under Cecill C licence.

[3]"Why" can be downloaded at http://why.lri.fr/

[4]KeY can be downloaded at http://www.key-project.org/download/

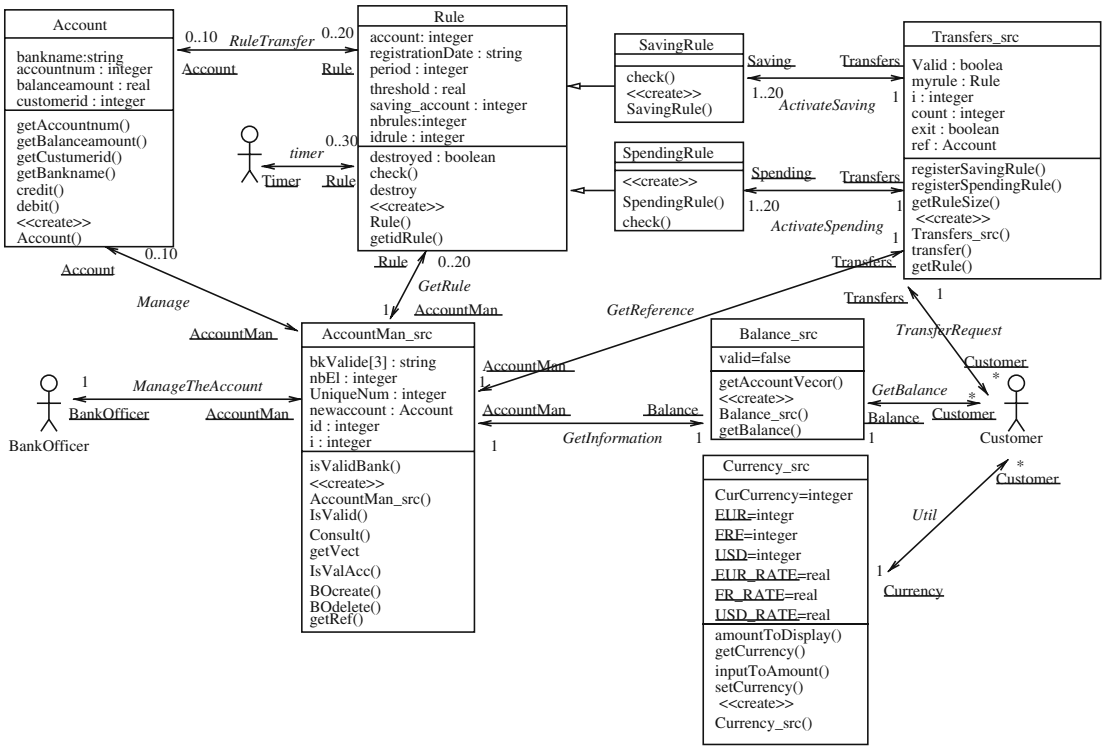


Fig. 1 Class diagram of the banking application

application user (i.e. the customer) can consult his accounts and make some money transfers from one account to another. The user can also record some "transfer rules", in order to schedule periodical transfers. These transfer rules can be either saving or spending rules. Moreover, the application includes some features to convert money from one currency to another.

The case study is a simplified version of an application already used in the real world. This application is running on a central server, which is linked to several smart card terminals. For the simplified case study, the smart card terminals have been withdrawn.

The banking application code is composed of eight classes (Fig. 1), among which: the `Account` class, the `AccountMan_src` class to create and delete accounts, the `Transfers_src` class to define spending and saving rules to transfer money from an account to another according to different thresholds, the `Balance_src` class that allows the customer to have access to his accounts, and the `Currency_src` class to convert currencies. The three remaining classes are dedicated to the definition of the transfer rule principles.

3.2 The Banking Application JML Specification for Verification

For this application, the JML annotations were originally designed to support a verification process. Both the Java application and the JML code were written by engineers at Gemplus. The application was first informally specified and modelled in UML. The Java code was then produced. Finally, the JML assertions were added to

Fig. 2 JML annotation for the method register spending rule

```
/*@ requires true ;
  @ ensures (threshold > 0 && period >= 0
   && account != spending_account
   && account >= 0 && spending_account >= 0
   && (\exists int i; i >= 0 &&
      i< accman.LocalVector.size() &&
      ((Account)(accman.LocalVector.
                 elementAt(i))).accountnum
                        == account)
  && (\exists int i; i >= 0 &&
      i< accman.LocalVector.size() &&
      ((Account)(accman.LocalVector.
                 elementAt(i))).accountnum
                    == spending_account))
  ==>(\result == 0 &&
     (rules.size()==(\old(rules.size())+1)));
  @ ensures ...
  @ exsures (Exception e) false;  @*/
  public int registerSpendingRule(String date,
       int account, float threshold,
       int spending_account, int period)
   { ... }
```

the code. The JML specification was originally designed to evaluate Jack, Gemplus's prover for JML [2, 6].

The application was not totally verified at this stage: some proof obligations were not satisfied and some parts of the code were not annotated. The JML assertions, the informal requirements and the code were then directly used for the testing phase. An example of those annotations is given Fig. 2.

This was the first use of JML for verification purposes by the Gemplus Research team. Some metrics of this application are given in Table 1. The fact that the number of JML annotation lines is larger than the Java code length is mainly due to the verification process: annotation statements were inserted to guide the prover.

In the banking example, 362 of the 615 lines of JML assertions are distributed as shown in Table 2. Postconditions (the *ensures* clause) represent most of the JML assertions, especially in classes `Balance_src` and `AccountMan_src` where they

Table 1 Some metrics of the banking application

Classes	Java lines	JML lines
Transfer_src	116	150
AccountMan_src	105	236
Currency_src	93	28
Balance_src	64	58
Spending_rule	40	42
Saving_rule	40	42
Rule	40	23
Account	30	36
Total	518	615

Table 2 JML assertion distribution in the banking example

Classes	Number of methods	Number of lines of			
		Invariant	Requires	Ensures	Exsures
Transfer_src	7	5	6	108	6
AccountMan-src	8	17	8	9	7
Currency_src	7	7	7	6	7
Balance_src	3	1	2	37	2
Spending_rule	2	20	13	6	1
Saving_rule	2	20	13	4	1
Rule	5	3	6	6	2
Account	7	5	8	9	7
Total	41	81	63	185	33

are dedicated to the specification of error codes. The remaining 253 lines of JML correspond to loop invariants, to additional keywords such as the *modifies* clauses, or to comments.

3.3 Testing the Banking Application

Two testing campaigns were performed by two different teams in the Laboratoire d'Informatique de Grenoble (LIG). The code and annotations were re-used without modification by the LIG testing team. During this work, we tried to answer the following questions:

1. Can the banking application JML specification dedicated to verification be used for testing?
2. Is the JML specification detailed enough to allow accurate validation by test?

Both teams worked separately during a bounded time period (3 days). For both teams, the testing work consisted in producing some test data sequences and executing them.

The first team made a critical code review and then used random testing (with Jartege). The code review phase took one person-day. It allowed the detection of four errors (see Fig. 3). Those were corrected before the random testing phase. Information from the code review was used to target random tests to suspicious parts of the code. This testing phase revealed five new errors or suspicious situations in one day.

The second team applied a combinatorial testing approach based on the informal requirements: the requirements document was used as a basis for the design of test inputs. First, seven general properties from the requirements were identified. Then, some "abstract scenarios" were expressed to define sets of similar test cases. The Tobias tool was used to instantiate the abstract scenarios into executable JUnit test cases. Seventeen abstract scenarios were produced, which were unfolded into 1,241 test cases. Those represented 40,000 Java code lines (for JUnit).

In parallel with the test execution, the second team performed a critical analysis of the execution results. This helped us to find some cases where the code and the JML specification were consistent, but were different from the requirements or contrary to common sense. It took 6 person-days to analyze the specification, produce

	team 1		*team 2*		
Err.	Code review	Random testing	With Tobias	Type of error	Method of detection
1			X	limit	human oracle
2			X	limit	human oracle
3	X		X	floating-point	code rev + JML or.
4		X	X	floating-point	JML oracle
5		X	X	floating-point	JML oracle
6		X	X	floating-point	JML oracle
7			X	postcondition	JML oracle
8			X	postcondition	JML oracle
9		X	X	design	JML oracle
10	X			design	code review
11			X	limit	human oracle
12			X	limit	human oracle
13	X		X	design	code rev + Java ex.
14	X			postcondition	code review
15		X	X	several*	Java exception
16			X	counter-intuitive	human oracle
17			X	counter-intuitive	human oracle
18			X	floating-point	human oracle

*precondition mistake, under specification, or design mistake

Fig. 3 Errors detected

the abstract scenarios, execute the tests and analyze the traces. Sixteen errors or suspicious situations were discovered by the execution of these tests.

The testing efforts of both teams aimed at discovering inconsistencies between informal requirements, JML specification and code. Three cases were identified: JML specification and code are inconsistent (1); JML specification and code are consistent and both are inconsistent with informal requirements (2); JML specification, code and informal requirements are consistent but overlook common sense requirements (3).

At the end of both processes, 18 different errors or suspicious situations were identified (Fig. 3). We say that there is an error when the JML assertion checker raises an exception. Java exceptions also often reveal errors in the code[5] (case 1).

[5] Or reveal a missing *exsures* clause.

We call suspicious situations the cases where the formal specification and the code have the same behaviour, but do not correspond to the informal requirements or to common sense (cases 2 and 3).

Each error was carefully analysed in order to classify them. Figure 3 lists all errors, with their types and the way they were discovered. Errors 3, 10, 13 and 14 were fixed between code review and random testing, in order to facilitate the random testing process.

- *Floating-point approximations*
  There are five cases related to floating-point approximations (errors 3, 4, 5, 6, 18). The `float` type is used to represent the account balance. The errors are revealed when the postcondition and the code compute the same "value" in different ways. For example, $(x + y) - z$ is not always equal to $x + (y - z)$ when x, y, z are float numbers. With float numbers, + and - operations are not commutative due to their limited precision[6] *(case 1: JML specification and code inconsistent)*.
- *Erroneous JML specification*
  Three cases are in the postconditions, typically several \old arguments were forgotten (err. 7, 8, and 14). For instance, error 14 is due to an assertion indicating that the new value of an attribute is equal to itself ($a == a$). The correct assertion should have been ($a == \backslash old(a)$), expressing that the value of the attribute has not been changed.[7] This specification error is a typical example of error that can not be discovered with a black-box testing approach, since the assertion is always true. It was actually detected by code review *(JML specification and code both inconsistent with informal requirements)*.
- *Limit*
  There are four cases that are dealing with "limits" (err. 1, 2, 11 and 12), i.e. boundary values. Let us detail two examples.
  A transfer rule can be registered with a time period of 0, which is forbidden in the informal requirements, but not in the code and in the JML specification. *(JML specification and code both inconsistent with informal requirements)*.
  One informal requirement says that there is no limit amount for a credit. So testers tried to credit one account with the Java pre-defined constant POSITIVE_INFINITY. The fact that this operation is accepted was considered as a suspicious situation. This is a typical example where the success of a test actually reveals a problem. *(JML specification, code and informal requirements inconsistent with common sense)*.
- *Design mistake*
  Errors 9, 10, and 13 have been classified as design mistakes. One critical attribute is `public` instead of being `private` (err. 10). It is possible to assign the same identifier to two different accounts if two account managers are created (err. 9)*(JML specification and code inconsistent)*. The banking application deals with threads, but there is no protection (i.e. critical section) to prevent a concurrent

[6] During the verification process, the approximation problem was not addressed.

[7] These properties could have been expressed with the JML keyword *not_modify*.

access an account (err 13). These errors were revealed either by the tests or by code review.

– *Counter-intuitive behaviour*
  Errors 16 and 17 denote counter-intuitive behaviours. In fact, it is possible to delete an account on which there are some active saving or spending transfer rules. This case is neither specified informally nor formally. So, it is not possible to conclude whether the application behaviour is correct or not. Intuitively, one can imagine that the removal of an account, which is a transfer destination may create some access conflict if the rule is not deactivated before. *(JML specification, code and informal requirements inconsistent with common sense)*.
– *Several classifications*
  Error 15 falls into several categories. The method `inputToAmount` of the `Currency_src` class needs a parameter to be a string representing a float.

```
/*@ requires true;
  @ ensures input == null ==> \result == 0;
  @ exsures (Exception e) false; @*/
public float inputToAmount(String input) {
...
if (input == null) {... return 0; }
            else { amount = new Float(input); ... } }
```

  If this method is called with an incorrect string (for instance `inputToAmount("aaa")`), it will raise an exception when calling `new Float(input)`. This can therefore be considered as an error in the specification: the `exsures` clause should be modified to allow this exception or the precondition should be stronger in order to exclude illegal input values. *(JML specification and code inconsistent)*
  The fact that the input should be a string is not indicated in the informal requirements. This error can thus be considered as a design mistake (the parameter should have been typed as `float`).

Expressiveness of JML Using JML, seven out of 18 errors were detected. Many other errors correspond to properties which could have been expressed formally using JML. The case study has thus revealed the incompleteness of the available specification. Table 3 shows which kind of properties were actually detected using JML and which ones could have been detected if the specification was more complete.

Table 3 Potential to detect more errors with JML assertions

Error type	Detected by JML assertions	Detectable by JML assertions
Limit	No	Yes
Floating point	Yes	
Postcondition	Yes	
Design	No	One of the three errors
Counter-intuitive	No	Yes

3.4 Partial Conclusion

At the beginning of this section, two questions were asked. This section will now try to answer them, and add some lessons about the choice of a testing strategy.

- *Can the banking application JML specification dedicated to verification be used for testing?*
  It is very attractive to reuse the same specification with several tools. Here the specification was first created for verification purposes then reused for testing purposes. Before the verification process, one writes the specification focusing on the main parts, i.e. what has to be verified (invariants, pre and post-conditions). Then, during the verification process, some new assertions are added (mainly *annotation statements*), to help the verification tool.
  For the testing process, tools take advantage of the fact that a large subset of JML assertions are executable. Non-executable assertions are expressions that can not be translated to Java due to various factors. For example, `\forall` or `\exists` have to be iterated over a finite range of integers or a JML set to be executable. Since verification tools do not need JML assertions to be executable, only a subset of the specification can be reused.
  About the executable part of the specification, one should notice that the annotation statements are often too close to the Java code to help find errors. But although they do not contribute to the test oracle, they do not harm the testing process. These elements of the specification can even be useful for regression testing, provided they are sufficiently abstract to express the functionalities and not how they are implemented. The only negative influence of these specification statements is that they increase the size of the specification and tend to give some misleading confidence that the specification is complete.
  In summary, only the executable part of the JML specification can be reused for testing. This may include annotation statements which are too close to the code to reveal errors.
- *Is the JML specification detailed enough to allow accurate validation by testing?*
  JML has a good expressiveness to cover most of the requirements of the banking application. Unfortunately, like most formal languages it faces the risk of incomplete specifications: while writing specification for verification purposes, the type of properties and the way they are written are implicitly influenced by the ability of the verification tools. For example, for the Banking application, engineers deliberately chose not to describe properties about the float values, since they knew that JACK could not handle them. Thus, only seven of the 18 errors were found because they violated JML assertions. But 80% to 90% of the errors could have been detected if adequate JML assertions had been available (see Table 3). This reveals the incomplete character of the provided specifications which reduces the testability of the application.
  We divided the errors into three categories. Category 1 corresponds to the seven errors we discovered where code and specification were inconsistent. Five of them are related to floating point errors and could not be detected by the verification process because JACK does not support float variables. The two other errors (nine and 13) could have been detected by a verification process.
  The remaining errors (categories 2 and 3) could not be revealed by the specification. Category 2 corresponds to the incompleteness of the specification with

respect to the requirements. Here, we believe that the systematic use of traceability techniques can help reduce this incompleteness by clearly marking the requirements not covered by the specification. Finally, category 3 corresponds to the incompleteness of the requirements documents. This problem should be addressed with adequate requirements engineering techniques but definitely remains a difficult issue. It must be noted that the use of more sophisticated specification-based testing techniques or the use of verification techniques would not have improved our capability to detect these errors since such techniques only find errors covered by the specification.

– *About the choice of a testing strategy*
  The two testing approaches have not revealed exactly the same errors. The first approach, combining code review (human validation) and random testing (automated data selection and oracle), allows one to detect two errors unfound by the second approach. On the other hand, the second approach based on the study of the informal requirements and on combinatorial testing (manual data selection, human and automated oracle) detects nine errors unfound by the first approach. It is important to notice that nine of these 11 errors were detected thanks to human analysis and correspond to categories 2 and 3. This makes us think that the ability to find faults automatically (revealed by runtime errors or JML assertion violation) does not depend on the testing approach but on the accuracy of the JML specification.

4 From Testing to Verification

This section focuses on the work done to reuse a JML specification dedicated to testing, for verification activities. Section 4.1 deals with the presentation of the case study, a Home Network Services application. Section 4.2 briefly describes the testing work. Section 4.3 details the work done for verification, and especially focuses on the refactoring of the code and the specification. Section 4.4 proposes a partial conclusion.

4.1 The HNS Case Study

The second case study deals with Home Network Services (HNS). HNS consist of one or more networked appliances connected to a LAN at home. One of the major HNS applications is the integrated services of networked home appliances (called *integrated service* in the following). An integrated service orchestrates several home appliances via a network in order to provide more comfortable and convenient living for the users. For instance, the *DVD Theater Service* turns on a DVD player, switches off the lights, selects 5.1ch speakers and adjusts the volume automatically (Fig. 4).

Nakamura et al. have proposed a framework that adapts the legacy appliances with conventional infrared remote controllers [22, 23]. The key ideas are (1) to use a programmable infrared remote controller to control the different appliances, and (2) to rely on a service-oriented architecture (SOA) (see [19, 25]).

For each appliance, a self-contained component is implemented in Java and deployed as web service (using Apache AXIS) (Fig. 5). Methods like `On()` and `Off()` are open interfaces for accessing basic features of the appliance. They use a

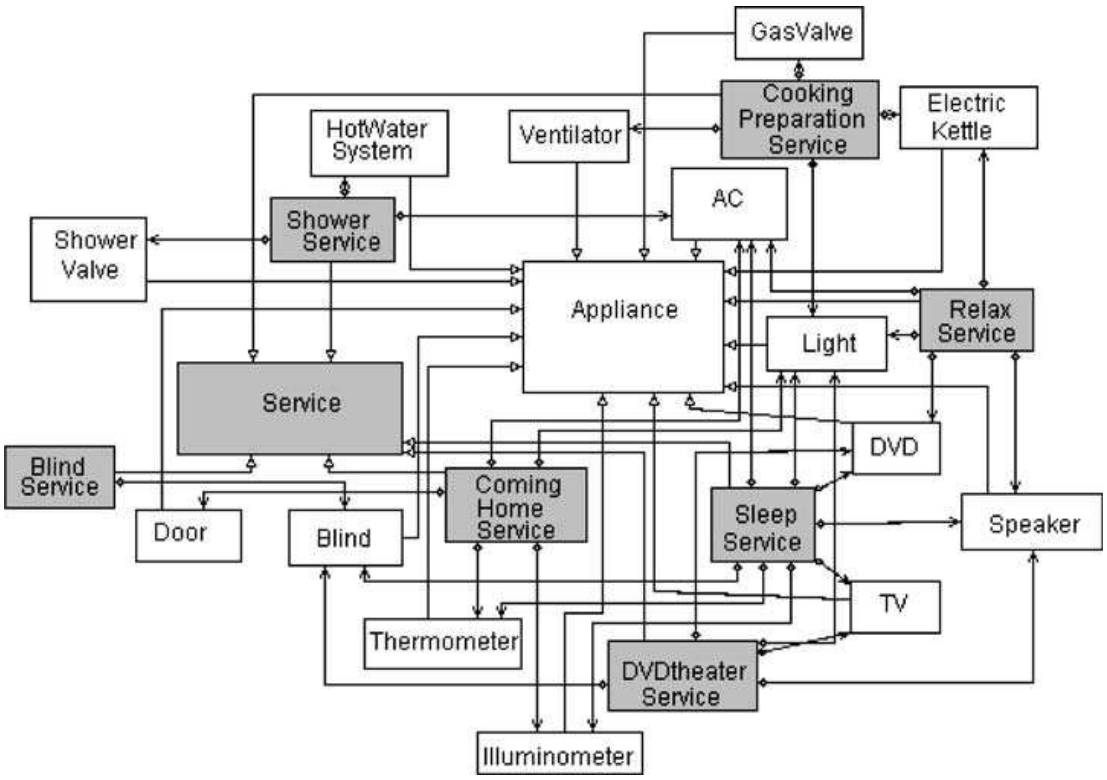
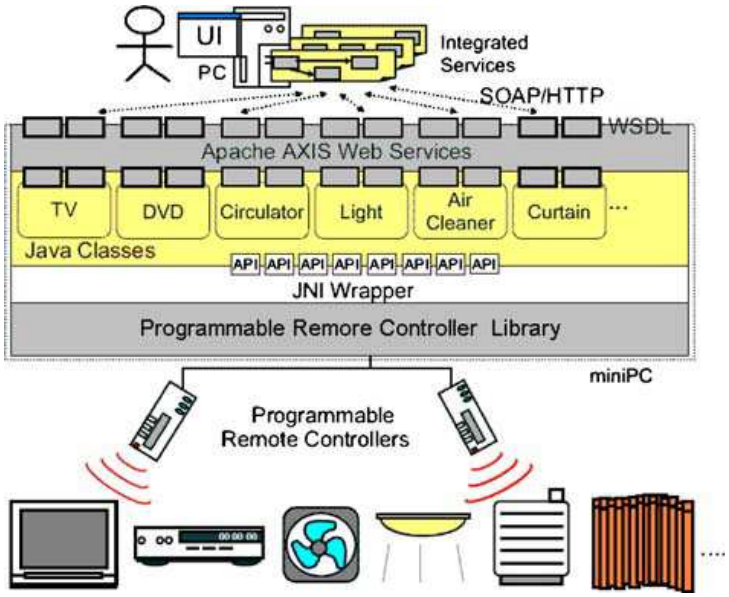


Fig. 4 Class diagram of the HNS application, restricted to appliances and integrated services (*in grey*)

set of APIs by which the PC can send infrared signals to the appliances (Ir-APIs). Ir-APIs have been implemented by wrapping the programmable infrared remote controller with a Java Native Interface (JNI Wrapper).

Integrated services can be implemented in this framework as client applications. An integrated service invokes the methods of the appliance components. The application was mainly developed in Nara and Kobe universities. The core of the

Fig. 5 HNS



application, consisting of appliance and integrated services, has been developed under several versions. The one which is studied here was developed by a Master student at the LIG labs. The Java code and the JML assertions were written in parallel by this student.

The initial version was composed of 25 classes among which 14 were appliance components and seven, integrated services (Fig. 4). One level of inheritance was introduced for the appliances and the integrated services. A specific class (`HomeEnvironment.java`) describes the sensor values (temperature, light level,...). The version has 2,000 lines of code. In this code, 209 JML annotations were inserted (17 preconditions, 150 postconditions, and 42 invariants).

4.2 Testing the HNS Services

To test the appliance and integrated services, we adopted a combinatorial approach based on the informal requirements as for the Banking application. More than 30 test schemas were described and unfolded by Tobias [11]. Each schema has between 500 and 5,000 test cases. Tests cases were then translated in the JUnit format and executed within the Eclipse environment. 10 errors were found (at the appliance and service levels). These errors reveal mainly inconsistencies between code and JML specifications.

4.3 Verifying the HNS Services

Two types of works were done to verify the HNS classes: one with ESC/Java and the other with deductive verification tools (JACK, KeY, Why/Krakatoa). The code and annotations were modified during the verification process, in order to correct the errors or to detail incomplete parts, and to continue the verification process. During the work, we tried to answer the following questions:

1. Can the HNS specification dedicated to test be used for verification?
2. If no, how should the testing specification be modified to support verification?

4.3.1 Using ESC/Java

ESC/Java was the first tool to be used during the verification process. The verification was carried out on the code that was tested and corrected. The verification of the Java classes was long and uneasy. The JML assertions were not sufficient to allow an automatic verification, even if they allowed to perform testing.

The first ESC warnings were obtained for `Appliance.java`. They were related to the use of set methods of `HomeEnvironment.java`. The use of these methods in `Appliance.java` could lead to a violation of `HomeEnvironment` assertions. In order to solve the problem, `Appliance.java` class had to be refactored. This operation has impacted all subclasses of `Appliance.java`.

Then, to remove several warnings, several JML assertions had to be inserted. Indeed, several postconditions were added to specify the returned result. For instance, for the method `public String getPower()`, the following postcondition was added `//@ ensures \result.equals(powerState);`.

Several warnings were related to the problem of null values. Some assertions were added in order to specify that the variables or the attributes (of type String) were not

```
public  class Appliance {
  protected /*@  spec_public @*/ String Name; // used for printing messages
  protected /*@  spec_public non_null @*/ HomeEnvironment currentEnv ;
  protected /*@  spec_public non_null @*/ String  powerState = "OFF";
  protected /*@  spec_public non_null @*/ String  internalState = "OFF";
  //@ public invariant (!powerState.equals("OFF") ==> powerState.equals("ON"));
  //@ public invariant (!powerState.equals("ON") ==>  powerState.equals("OFF"));
  //@ public invariant (!internalState.equals("OFF") ==> internalState.equals("ON"));
  //@ public invariant (!internalState.equals("ON") ==> internalState.equals("OFF"));

  //@ public invariant minConsumption<=maxConsumption;
  //@ public invariant minConsumption >= 0 ;
  //@ public invariant applianceCurrentConsumption>=0 ;
  //@ public invariant applianceCurrentConsumption<=maxConsumption;
  //@ public invariant powerState.equals("ON")==>(applianceCurrentConsumption>=minConsumption);
  //@ public invariant powerState.equals("OFF")==>(applianceCurrentConsumption==0);
  protected /*@  spec_public  @*/ int maxConsumption =0;
  protected /*@  spec_public  @*/ int minConsumption=0;
  protected /*@  spec_public  @*/ int applianceCurrentConsumption=0;
 [..]
  //@ modifies \everything;
  //@ requires powerState.equals("ON");
  //@ ensures powerState.equals("OFF");
  //@ ensures internalState.equals("OFF");
  //@ ensures applianceCurrentConsumption == 0;
  public /*@  spec_public  @*/ void  powerOff(){
        internalState = "OFF";
        powerState="OFF";
        applianceCurrentConsumption = 0;
        System.out.println(Name + " is powered off");
  }
  public /*@ pure @*/ int getConsumption(){return applianceCurrentConsumption;}

  //@ ensures \result != null;
  //@ ensures \result.equals(powerState) ;
  public /*@ pure @*/ String getPower(){ return powerState; }

  //@ ensures \result != null;
  //@ ensures \result.equals(internalState) ;
  public /*@ pure @*/ String getInternalState(){ return internalState;}

  //@ requires powerState.equals("ON");
  //@ requires internalState.equals("ON");
  //@ ensures internalState.equals("OFF");
  protected /*@  spec_public  @*/ void  switchOff(){
        setApplianceMinConsumption();
        internalState="OFF";
        System.out.println(Name + " is switched off");
  }

  //@ requires powerState.equals("ON");
  //@ requires internalState.equals("OFF");
  //@ ensures internalState.equals("ON");
  protected /*@  spec_public  @*/ void  switchOn(){
        internalState="ON";
        setApplianceMaxConsumption();
        System.out.println(Name + "is switched on");
  } [..] }
```

Fig. 6 JML annotations for the appliance java class

null. Moreover, all constructors of appliances were modified in order to initialize all attributes explicitly.

After code refactoring, the code size represents 2,400 lines of code. The new JML assertions represent more than 600 lines of code (see the code of Appliance Fig. 6). At the end of the process, all appliance and service properties seem to be validated:

there was no remaining warning. However, one has to be careful. ESC/Java is neither complete nor sound. Some errors may not be reported and false alarms may be issued.

From a general point of view, the verification of the code with ESC/Java required more work than expected. Indeed, the effort spent to complete the specification to help the tool was underestimated for two reasons. First, the code of the appliance API and integrated services is quite simple (no loop for instance). Second, the testing phase did not reveal inconsistencies between the assertions and the code. So it was expected that verification would be easy. Actually the verification process did not detect additional errors.

4.3.2 Using JACK, Why/Krakatoa and KeY

JACK, Why/Krakatoa and KeY were successively used in order to perform the verification. The process was difficult, and the result was not as good as expected (for the three tools).

A difficulty was that the JML version used for the project was no longer compatible with JACK. Moreover, Krakatoa (in the version used) did not accept the whole syntax of JML. For instance, assertions such that `non_null` or `pure` were not accepted. Several files had to be modified.

Regarding the use of KeY, the verification could be carried out only for file `HomeEnvironment.java`. The main reason was that the JML assertions deal with strings, which are currently not supported by KeY.

After the use of ESC/Java, the class `HomeEnvironment.java` has 27 methods (1 constructor, 11 get methods, 15 set methods) and three invariants. Each get method has been declared "`pure`". Half of the set methods were associated with a precondition. None of them has a postcondition. KeY produced between 3 and 5 proof obligations for each method. All of them were verified. Most of them were verified automatically with Simplify or Yices provers. For three methods, the "elementary arithmetic strategy" had to be used. For one method, we had to increase the number of computing steps (1,100 instead of 1,000 by default).

A new refactoring of the code was carried out. The attributes of type String were in fact used to implement an enumerated type. The code and the JML assertions were modified so that integers were used to implement those enumerated types.

KeY was then used again on a small part of the application. It was possible to verify automatically more than one half of the proof obligations related to appliances and integrated services, with the help of Simplify and Yices provers. However, some proof obligations are still pending. No additional error was found.

4.4 Partial Conclusion

At the beginning of this section, two questions were asked. We may now try to answer them.

– *Can the HNS specification dedicated to testing be used for verification?*
  Properties which were stated for the test process were properties we wanted to be verified. In that sense, the JML specification can be reused. However, some elements must be taken into account.

First, the verification process is limited by the power of the tools: some parts of the JML language that is supported by the testing tools are not currently supported by the verification tools. In fact, the same properties written during the verification process would probably have been designed differently, in order to help the verification tools. For instance, some properties about the initial states of the appliances or services could have been specified using the *initially* clause (which is not executable). So, some parts of the testing specification are not usable.

Moreover, the testing specification is possibly incomplete. Properties were written with the objective to be used for testing. So, the tester implicitly chose to write properties with respect to an appropriate (executable) subset of JML. With a larger subset, other properties could have been specified. In the HNS specification for instance, quantifiers and the `initially` clause are never used.

– *If no, how should the testing specification be modified to support validation by verification?*

It must be noticed that in this experiment, a large effort was required to adapt the code and the JML specification to the verification process. Improvements only restructured the code and increased the redundancy of the specification. We were not aware that it corrected any bug in the code or in the specification, since no failure was exhibited.

We can distinguish two types of works to support verification process. First, one has to provide code and specification compatible with the tool abilities. During our experiment, we had to refactor both the code and the JML specification in order to carry out the verification. A first refactoring was carried out for ESC/Java. It mainly consisted of (1) a simplification of the coupling of the methods and classes, and (2) a multiplication by four of the size of the JML specification. A second refactoring was needed by the deductive provers, in order to translate enumerated type (from String to Integer).

Second, one may have to add specific assertions to help the tools. For this application, the assertions to be inserted were mainly related to null values. It also concerned some indications of the value of the returned results. In this application, there were no loops, so no loop assertions were required. However, it is quite usual to add those types of assertions to help the verification process.

5 Conclusion and Lessons Learnt

This article reports on two case studies implemented in Java and specified in JML (Java Modelling Language). We specially address the problem of reusing a JML specification produced for one activity (resp. verification or test) for the other (resp. test or verification). These two case studies bring about interesting lessons on the use of JML in a Java validation process.

Writing a Specification (in JML) In the HNS case study, the JML assertions were written during coding, for testing purposes. A part of the assertions were devoted to express the internal consistency of the classes. For instance, we expressed the expected value of one attribute with respect to the values of the other attributes. Those assertions were really useful during the whole development process, as a way to maintain consistency among the classes during the different evolutions.

In the banking application case study, since the code was taken from an existing application, JML assertions have been added after the coding phase. As a result, some postconditions may have been influenced by the code. Actually, it is tempting to simply copy-paste the code of the method in the JML assertion and then to replace "=" with "==" and add some "\old" keywords. Unfortunately, this often results in copying coding errors into the specification. Therefore, care should be taken to express the specification in a different, and often more abstract, way. This should increase implementation freedom, and result in specifications which are more robust to evolution.

The copy-paste effect is a problem especially for testing. From the two case studies, we noticed that the copy-paste effect had less impact on invariants than on postconditions. This is due to the fact that writing invariants requires to step back, since it will concern the class in a global way and not only a method, as it is the case for postconditions. For this reason, one should favour the identification of invariants when writing a JML specification, especially for existing code.

An important point to notice is that JML is not completely supported by the different tools. For instance, verification tools such as ESC/Java, JACK, Why/Krakatoa and Key support only a part of the JML constructions. Similarly, the JML runtime assertion checker (`jmlrac`), used for test, supports only executable features of JML (for instance, a `\forall` is not executable if the following expression does not concern a JML set or an integer interval). So, assertions are written with an adequate subset of JML, with respect to the approach plan to be used.

From a methodological point of view, it seems more appropriate to write the specification independently of the code and to write first the invariants to reduce the copy-paste effect, then pre and postconditions. If testing is planned, one should check if all the assertions are executable. If it is not the case, an executable expression should be added. If verification is planned, annotation statements should be added.

Using a Specification for Testing or Verification For a JML specification to be usable for testing or verification, one should pay attention to two elements. First, one should keep in mind that the specification is never complete and can possibly be inconsistent with informal requirements. In the first case study, the analysis shows that the two testing approaches have comparable outcomes: they detect quite the same inconsistencies between the formal specification and the code. However, the human analysis of respectively the code and the test results raise additional errors.

A second point that needs attention is the fact that the code and the specification should be designed so that the verification and both testing tools could be used. It is especially critical for the verification process, which is possible only if the construction (code and assertions) are supported. It is also the case for the testing process, for instance, results should be observable. So in both case, during the whole development, one should "design for test" and/or "design for verification", that we can summarize as "design for validation and verification".

Reusing a Formal Specification Formal specifications are aimed to be used several times during software development, and it is often the case that the intended use of the specification influences its style and contents. But, the way the specification will be used (resp. the tools that are going to be used) has an influence of how the specification has to be written. Reusing a specification should then be done carefully.

A JML specification designed for verification contains several assertions, such as invariants associated with constants or loop invariants, which were added specifically for verification in order to help the tools. Unfortunately this over-specification can become an obstacle to evolutions of the system.

Over-specification is not spread uniformly in the specification. For the banking case study, several methods were clearly under-specified (the postcondition is not stated) and it was not possible to make a judgement on the correctness of their execution.

A specification designed for test tends to express a judgement about the results. It generally specifies the expected behaviours. The main requirement is that the assertions have to be executable. In order to verify such a specification, refactoring may have to be carried out for both code and assertions. In the case of the HNS application, it is still not clear for us if the refactoring had to be carried out in order to help the tools or in order to correct some remaining errors, since no failure was demonstrated before the refactoring.

As a conclusion, since specification description is motivated by different concerns, it should be interesting to use structuring and documentation mechanisms that identify parts of the specification according to their rationale and intended use. In particular, each assertion of the specification should either be linked to the requirement it expresses or marked as a proof annotation.

Acknowledgements The work on the first case study was partially supported by the COTE RNTL project (http://www.irisa.fr/cote/). The work on the second case study was partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Young Scientists (B) (No. 18700062), Scientific Research (B) (No. 17300007), and Comprehensive Development of e-Society Foundation Software program. It is also supported by JSPS and MAE under the Japan-France Integrated Action Program (PHC-SAKURA). A special thanks to Natasha King who corrects the English phrasing.

References

1. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY tool. Softw. Syst. Model. **4**, 32–54 (2005)
2. Barthe, G., Burdy, L., Charles, J., Grégoire, B., Huisman, M., Lanet, J.-L., Pavlova, M., Requet, A.: JACK—a tool for validation of security and behaviour of Java applications. In: 5th International Symposium Formal Methods for Components and Objects (FMCO). Lecture Notes in Computer Science, vol. 4709, pp. 152–174. Amsterdam, The Netherlands (2006)
3. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: the KeY Approach. Springer, New York (2007)
4. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. In: Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS'03). Electronic Notes in Theoretical Computer Science, vol. 80, pp. 73–89 (2003)
5. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. STTT **7**, 212–232 (2005)
6. Burdy, L., Requet, A., Lanet, J.-L.: Java applet correctness: a developer-oriented approach. In: The 12th International FME Symposium, Pisa, Italy (2003)
7. Chalin, P.: Early detection of JML specification errors using ESC/Java2. In: Proceedings of the 2006 Conference on Specification and Verification of Component-Based Systems (SAVCBS), pp. 25–32. Portland, Oregon (2006)
8. Cheon, Y., Leavens, G.T.: A runtime assertion checker for the java modeling language (JML). In: Arabnia, H.R., Mun, Y. (eds.) International Conference on Software Engineering Research and Practice (SERP '02), pp. 322–328. Las Vegas, Nevada (2002)

9. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: the JML and JUnit way. In: 16th European Conference on Object-Oriented Programming (ECOOP'02), pp. 231–255 (2002)
10. du Bousquet, L., Ledru, Y., Maury, O., Oriat, C., Lanet, J.-L.: A case study in JML-based software validation (short paper). In: Automated Software Engineering (ASE). Linz, Austria (2004)
11. du Bousquet, L., Nakamura, M., Yan, B., Igaki, H.: Using formal methods to increase confidence in one home network system implementation. Case study. In: Workshop on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2007). Revue des Nouvelles Technologies de l'Information, vol. RNTI-SM-1. Poitiers, France (2007)
12. Filliâtre, J.-C., Marché, C.: The why/krakatoa/caduceus platform for deductive program verification. In: 19th International Conference Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 4590, pp. 173–177. Berlin, Germany (2007)
13. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: PLDI, pp. 234–245 (2002)
14. JML: The java modeling language (JML) home page (2008). http://www.cs.iastate.edu/~leavens/JML.html
15. JUnit: JUnit (2008). http://www.junit.org
16. Leavens, G., Baker, A., Ruby, C.: JML: a notation for detailed design. In: Kilov, H., Rumpe, B., Simmonds, I. (eds.) Behavioral Specifications of Businesses and Systems, pp. 175–188. Kluwer, Dordrecht (1999)
17. Ledru, Y., Dadeau, F., du Bousquet, L., Ville, S., Rose, E.: Mastering combinatorial explosion with the tobias-2 test generator. In: 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), pp. 535–536. USA (2007)
18. Ledru, Y., du Bousquet, L., Maury, O., Bontron, P.: Filtering TOBIAS combinatorial test suites. In: Fundamental Approaches to Software Engineering (FASE'04). LNCS, vol. 2984. Barcelona, Spain (2004)
19. Loke, S.W.: Service-oriented device ecology workflows. In: First International Conference on Service-Oriented Computing (ICSOC 2003). Lecture Notes in Computer Science, vol. 2910, pp. 559–574. Trento, Italy (2003)
20. Marché, C., Paulin-Mohring, C., Urbain, X.: The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. J. Log. Algebr. Program. **58**(1–2), 89–106 (2004)
21. Maury, O., Ledru, Y., du bousquet, L.: Using TOBIAS for the automatic generation of VDM test cases. In: Third VDM Workshop (in conjunction with FME'02) (2002)
22. Nakamura, M., Tanaka, A., Igaki, H., Tamada, H., Matsumoto, K.: Adapting legacy home appliances and web services. In: Int. Conf. on Web Services (ICWS 2006), pp. 849–858 (2006)
23. Nakamura, M., Tanaka, A., Igaki, H., Tamada, H., Matsumoto, K.: Constructing home network systems and integrated service using legacy home appliances and web services. Int. J. Web Serv. Res. **5**(1) (2009)
24. Oriat, C.: Jartege: a tool for random generation of unit test for java classes. In: First International Conference on the Quality of Software Architechtures and Second International Workshop of Software Quality (QoSa/SOQUA). Lecture Notes in Computer Science, vol. 3712, pp. 242–256 (2005)
25. Papazoglou, M.P., Georgakopoulos, D.: Special issue: service-oriented computing. Introduction. Commun. ACM **46**(10), 24–28 (2003)

2008 International Conference on Software Testing, Verification, and Validation

Relation between Depth of Inheritance Tree and Number of Methods to Test

Muhammad Rabee Shaheen Lydie du Bousquet
Universités de Grenoble
Laboratoire d'Informatique de Grenoble (LIG)
BP 72, 38402 Saint Martin d'Hères cedex, France
{Muhammad-Rabee.Shaheen, lydie.du-bousquet}@imag.fr

Abstract

Depth of inheritance tree (DIT) is considered as a factor influencing the cost of testing. Test is supposed to be more expensive if DIT is high. This paper relates the analysis of DIT with respect to the number of methods to test in each class. Our study based on more than 1700 classes from 6 Java applications.

1. Introduction

Inheritance is one of the main features of object-oriented programming paradigm. There are several measures to characterize the inheritance tree complexity [4], among which Number Of Root classes (NOR), Fan In (FIN), Number Of Children (NOC) and Depth of Inheritance Tree (DIT). DIT belongs to the Chidamber and Kemerer metrics suite [11], which has been proposed in early nineties. The depth of a class within the inheritance hierarchy is the maximum length from the class node to the root of the tree, measured by the number of ancestor classes. The deeper a class within the hierarchy, the greater the number of methods it is likely to inherit [11, 18].

Since it has been demonstrated that inheritance may be abused in many ways [2], several testing strategies have been dedicated to inheritance testing. Basically, two types of strategies were proposed. It is either suggested to test all new methods and to retest all inherited methods (for instance, see Binder's book [5]). Or, it is suggested restricting testing to validate only changes in the inherited features (methods and attributes) [16, 10, 18]. In both cases, the number of methods to be tested is supposed to be proportional to DIT [11, 4]. That is why DIT is generally considered as a way to estimate the testing effort.

Moreover, since the inheritance tree is often designed during the design (and sometimes before the definition of the application methods), DIT is often considered to be predictive measure. In [9], authors have evaluated the correlation between a set of OO source code metrics (among which DIT) and their capabilities to predict the effort needed for testing, expressed as dLOCC (Lines Of Code for Class) and dNOTC (Number of Test Cases). Somewhat surprisingly, DIT was not correlated to dNOTC. This was explained by the fact that inherited methods were probably not systematically re-tested. However, if all inherited methods are re-tested, it is expected that the number of test cases should increase with respect to DIT.

The aim of the work we are presenting here is to evaluate experimentally how accurate is the assumption that "the number of methods to be tested is proportional to the depth of inheritance tree". We focus our work on Java applications. For each class, we have evaluated its number of defined and inherited methods and compared it to the inheritance depth.

Formally, the depth of inheritance includes (1) standard classes (for instance the Java Development Kit classes) and (2) applications classes (those developed in the context of the application under test). To be able to distinguish both cases, we have first introduced two notions of inheritance: considering the standard classes or restricted to the application.

In the following, section 2 describes our motivations and the related works. Section 3 presents two notions of depth of inheritance tree. The first one (classical one) takes the standard classes into account. The second one restricts the inheritance tree to the application classes. In section 4, we briefly present the applications we have studied. Section 5 is dedicated to the analysis of the number of inherited methods with respect to the two DIT definitions. Section 6 is dedicated to the analysis of the number of defined methods. Section 7 concludes and draws some perspectives.

2. Motivation and related works

Promises and challenges in object-oriented methodology have received the attention of both practitioners and researchers. Lots of work in this domain focused on the

978-0-7695-3127-4/08 $25.00 © 2008 IEEE
DOI 10.1109/ICST.2008.34

IEEE computer society

understanding of software systems in terms of objects and their properties. Thus, several set of metrics have been proposed such as the Chidamber and Kemerer set [11, 13], The understanding of these metrics and especially their relevance in improving the outcomes of software developments led us to a large set of researches focusing on the validation of the different metrics [3, 7, 24, 6, 8, 15, 25, 17, 26, 9, 14]. Our work lies in this context.

The primary focus of our research is on the understanding of how the Depth of Inheritance Tree (DIT) can be used to predict the cost of unit testing. DIT is the length of the longest path from a given class to the root class of the inheritance hierarchy. Unit testing is a procedure used to validate that individual units of code are working properly. In OO programming, unit can be either method or class. In the following we focus on the testing of class.

Our work is motivated by the different feedbacks from the literature about the influence of inheritance on the quality of the developed systems. A large amount of work has been done to study the relation between OO metrics and fault-proneness of classes [3, 7, 24, 6, 8, 15, 25, 17, 26]. Most of the studied systems were implemented in C++. For half the cited works, DIT was not significant of the fault-proneness of classes. The Weighted Method per Class (WMC) and the Source Line of Code (SLOC) were found more positively significant of the fault-proneness of classes.

The aim of testing is to find some defects [21]. Testing strategies should then focus especially on parts of the systems that are known to be fault proneness. Several testing strategies have been proposed in order to find those defects introduced because of the usage of inheritance paradigm. In the following, we consider to type of testing strategies with respect to the inherited methods. Some strategies focus only on the test of methods that are newly defined in the class. Let us call T_s the set of those strategies. Other strategies suggest that all the inherited methods should be re-tested. Let us call T_t the set of those strategies. Two hypotheses we would like to check are:

1. For testing strategies that do not consider inheritance (T_s), the cost of testing is not influenced by DIT.

2. For testing strategies that consider inheritance (T_t), the cost of testing is influenced by DIT.

In [9], M. Bruntink *et al.* have evaluated the correlation between a set of OO source code metrics (among which DIT) and their capabilities to predict the effort needed for testing. In this study, test cases were all written as JUnit class [19]. The effort for testing was expressed as the number of Lines Of Code for the JUnit Class (dLOCC) and the Number of Test Cases (dNOTC). Five large open-source applications were studied. They were available with their JUnit tests. One difficulty here was the fact that the method used for the production of the test was not under control. Coverage criteria were different from one application to another, and sometimes different from one package to another from the same application. The result that the testing effort was not related to the DIT is explained by the fact that inherited methods were probably not systematically re-tested.

In the following, we have decided to estimate the cost of testing by the estimating the number of methods to be tested. This choice was done to be as independent as possible of the testing method/criteria. This idea is equivalent to the one which estimates the Weighted Methods by Class (WMC) by a weighting factor of unity. We do not use WMC since we want to evaluate both the number of defined methods in the class (WMC) and the number of inherited methods. Thus the previous hypotheses can be re-written in:

1. For testing strategies that do not consider inheritance (T_s), the number of defined methods in a class is not influenced by DIT.

2. For testing strategies that consider inheritance (T_t), the number of inherited methods in a class is influenced by DIT.

3. Depth of Inheritance of complete Tree or application Tree

3.1. Inheritance in Java

Object-oriented programming allows classes to inherit commonly used state and behavior from other classes. Inheritance is the capability of a class to use the properties and methods of another class while adding its own functionality. In the Java programming language, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of subclasses. The `Object` class is the superclass of Java. All other classes are subclasses of the `Object class`. The `Object` class includes 11 methods among which `clone()`, `finalize()`, `toString()`, or `equals(Object src)`.

The superclass contains elements and properties common to all of the subclasses. Often, a superclass will be set up as an abstract class which does not allow objects of its prototype to be created. Abstract methods are methods with no implementation. Subclass of an abstract class must provide the implementation of the abstract methods or should be declared as an abstract class.

Java does not allow multiple inheritance for classes (i.e. a subclass being the extension of more than one superclass). To tie elements of different classes together Java uses an interface. Interfaces are similar to abstract classes but all methods are abstract and all properties are static final. Interfaces can be inherited (i.e. you can have a sub-interface).

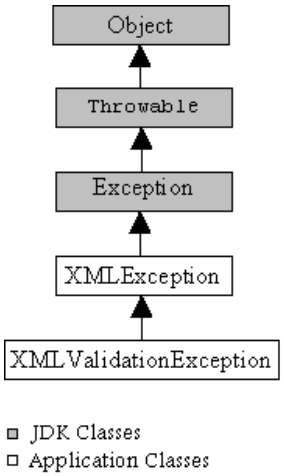


Figure 1. Sub tree of NanoXML inheritance tree

It is not possible to create object from interfaces, and thus they cannot be tested. Moreover, interfaces do not inherit of the `Object` class.

3.2. Dealing with inheritance when testing OO systems

Inheritance mechanism leads one to suppose that well-known super classes can be reused with confidence in subclasses. However, Binder underlines that *even if a superclass has been shown to be reliable, it is not guarantee equal reliability its subclasses*: reliable superclass methods can fail in the context of the subclasses [5].

For testing classes in the context of inheritance, Binder distinguishes two cases [5]. Either the class under test inherits from a trusted development environment (Java Development Kit - JDK - in our case), either it inherits from a class that does not warrant this level of confidence. In the first case, testing inherited methods is a problem of integration. Otherwise, it is suggested to retest the superclass methods in the context of the subclass.

For instance, let us consider NanoXML, a small XML parser for Java [22]. Figure 1 shows a sub part of NanoXML inheritance tree. The classes `XMLException` and `XMLValidationException` are both project classes. The other three classes are standard Java classes.

In the following, we compare DIT to the number of methods to be tested (considering or not inheritance). The methods inherited from the JDK are rarely being considered for testing. Therefore, to be as accurate as possible for our study, we distinguish the complete inheritance tree (including the JDK classes) and the inheritance tree restricted to the application. For the NanoXML example, DIT for `XMLValidationException` class is 4 if we take into account the complete inheritance tree. It is only 1 if we take into account only the application's classes. The following subsection formalizes these two definitions of DIT.

3.3. Two definitions for DIT

An object-oriented system consists of a set of classes, C. For every class $c \in C$ we have $Ancestors(c) \subset C$ the set of classes from which c inherits either directly or indirectly. The following formal definition of Depth of Inheritance Tree was given in [9].

$$DIT(c) = |Ancestors(c)|$$

This definition of DIT relies on the assumption that the considered OO programming language allows each class to have at most one parent class. Only then the number of ancestors of c will correspond to the depth of c in the inheritance tree. Java complies with this requirement.

This definition of DIT corresponds to the complete inheritance tree, i.e. it includes standard Java classes. Let us introduce DIT$_A$ as the Depth of Inheritance Tree restricted to the Application. Let JC be a set of all standard Java classes and AC be a set of all application classes, $JC \cap AC = \emptyset$ if $c \in AC$,

$$DIT_A(c) = |Ancestors(c) \backslash JC|$$

Let us now consider the methods. Let $M_D(c)$ be the set of methods that c newly declares, and let $M_{In}(c)$ be the set of methods that c inherits. Let $M(c)$ denote the set of methods of c. $M(c) = M_D(c) \cup M_{In}(c)$.

Those definitions include standard Java classes. To restrict the study to the application methods, we consider $M_{IA}(c)$ as the set of methods that c inherits, and which are defined in $Ancestors(c) \backslash JC$. We also consider the set of application methods of c as $M_A(c) = M_D(c) \cup M_{IA}(c)$.

Thus, the hypotheses given in previous section can be re-written in:

1-1 For testing strategies that do not consider inheritance (T_s), the number of defined methods in a class is not influenced by DIT.

1-2 For testing strategies that do not consider inheritance (T_s), the number of defined methods in a class is not influenced by DIT$_A$.

2-1 For testing strategies that consider inheritance (T_t), the number of inherited methods in a class is influenced by DIT.

2-2 For testing strategies that consider inheritance (T_t), the number of inherited methods in a class is influenced by DIT$_A$.

A testing strategies in T_s will require to test $|M_D(c)|$ methods for class c. A testing strategies in T_t will require to test $|M(c)|$ or $|M_A(c)|$ methods for class c, depending if one considers the complete inheritance tree or the inheritance tree restricted to the application classes.

4. Case study

4.1. Data Source

This study based on data collected from 6 real open-source applications. They represent approximately 140 packages and more than 1900 classes and interfaces. Here is a brief description of these projects.

NanoXML project is a small XML parser for Java (source code could be downloaded from http://nanoxml.cyberelf.be). It consists of 1 package and about 25 classes and interfaces. It is composed of three different components `NanoXML/Java`, `NanoXML/SAX` and `NanoXML/Lite`.

EMMA project is an open source toolkit for measuring and reporting Java code coverage. We studied the version 2.1, available on `http://emma.sourceforge.net/`. It contains 322 classes and interfaces in 31 packages. For instance, `com.vladium.emma.report` contains 44 classes and interfaces, `com.vladium.emma.data` contains 19 classes and interfaces, and `com.vladium.emma.instr` contains 27 classes, etc.

AspectJ project is a seamless aspect-oriented extension to the Java programming language. We studied the version available February the 28th 2007 on `http://www.eclipse.org/aspectj/`. It contains 127 classes and interfaces in 15 packages.

Chemical Evaluation Framework 1.001 (CEF) is a molecular structure based software to assist in hazard assessment. It is available on `http://sourceforge.net/projects/chemeval/`. It consists of 128 classes and interfaces in 3 packages.

Java Groups-2.5.0 is reliable group communication based on IP multicast and configurable protocol stack. It is available on `http://sourceforge.net/projects/javagroups/`. It contains 530 classes and interfaces in 15 packages.

Apache **Ant project** is an open source Java-based build tool. It looks like "Make", but it is more powerful. The Ant source code is kept in a public CVS repository. We studied the version 1.7 available on `http://ant.apache.org/`. It contains 778 classes and interfaces in 75 packages.

4.2. Data collection

In order to collect data from subject systems, we have produced a Java program that takes a path of the application, and navigates through the different packages and classes. It loads each class of the application dynamically, navigating from one class to another in the inheritance tree in order to reach to the `Object` class. During this navigation the program collects the following data:

- the total number of inherited methods ($|M_{In}(c)|$),
- the total number of inherited methods from standard Java classes,
- the total number of inherited methods from the application classes ($|M_{IA}(c)|$),
- the number of declared methods in the class ($|M_D(c)|$),
- the classical DIT, starting from the root (`Object` class),
- the DIT$_A$, starting from the first parent application class.

This program focuses only on the inheritance of classes and ignores the inheritance of interfaces. The calculations produced by the program are stored in an excel file.

4.3. Quantitative figures

We analyzed 1785 application classes in all the packages. Among these, 378 were interfaces, and were excluded of the analysis. Therefore, we studied 1785-378=1407 classes. Table 1 displays the distribution of these classes with respect to DIT and DIT$_A$ values. For the classical Depth of Inheritance Tree, there is no class with DIT=0 since all application classes inherits at least from `Object` class. For the Depth of Inheritance Tree restricted to application classes, DIT$_A$ is 0 its superclass is a standard JDK class. The number of classes with DIT$_A$ = 0 is not equals to the number of classes with DIT=1 since an application class can inherit from several standard JDK classes (cf. the `XMLException` class given Fig. 1).

As it can be noticed on table 1, the population is for DIT 6 and 7 (and DIT$_A$ 4, 5 and 6) can be considered as not representative. Each of them represents less than 2 % of the total population.

5. Inherited methods and DIT/DIT$_A$

In this section, we carried out a first analysis to evaluate the influence of the number of inherited methods with respect to the depth of inheritance.

	# of class / DIT	# of class / DIT$_A$
0	-	797
1	596	342
2	439	189
3	223	50
4	90	19
5	40	8
6	17	2
7	2	0
total	1407	1407

Table 1. Distribution of the studied application classes w.r.t DIT and DIT$_A$

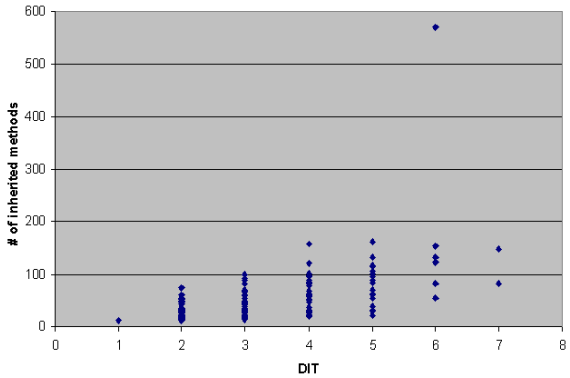


Figure 2. Number of inherited methods w.r.t. DIT

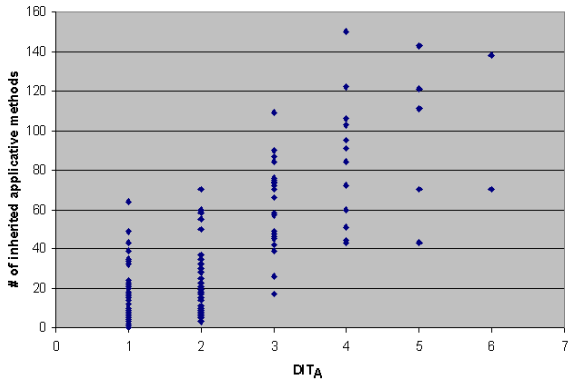


Figure 3. Number of inherited application methods w.r.t. DIT$_A$

Application	$rs(\|M_{In}(c)\|$, DIT)		$rs(\|M_{IA}(c)\|$, DIT$_A$)	
	min	max	min	max
NanoXML	0.97	0.97	1	1
EMMA	0.50	1	0.50	1
AspectJ	0.785	1	0.97	1
CEF	0.884	1	0.834	1
Java Groups	0.883	1	1	1
Ant	0.557	1	0.887	1

Table 2. Spearman's rank-order correlation between the number of methods and depth of total and application inheritance tree for each application

5.1. Correlation analysis

It is expected that the deeper a class is within the hierarchy, the greater the number of methods it is likely to inherit. To do that, we express two questions:

1. Is the number of inherited methods ($M_{In}(c)$) influenced by DIT?

2. Is the number of inherited application methods ($M_{IA}(c)$) influenced by DIT$_A$?

A first informal verification consists in drawing the scatter plot diagrams between the number of inherited methods and the DIT. Fig 2 displays the number of inherited methods with respect to the DIT. Fig. 3 displays the number of inherited methods *from the application classes* with respect to the DIT$_A$. The scatter plot diagrams show that there is positive relation between the number of inherited methods and the DIT.

In order to carry out a more formal verification, we calculate Spearman's[1] rank-order correlation coefficient, rs, for the number of inherited methods ($|M_{In}(c)|$) (resp. ($|M_{IA}(c)|$) and DIT (resp. DIT$_A$). We use $rs(n, d)$ to denote Spearman's rank-order correlation between the number of methods n and depth of inheritance d.

Spearman's rank-order correlation coefficient is a measure of association between two variables that are measured in at least an ordinal scale [23]. The measurements are ranked according to both variables. Subsequently, the measure of association is derived from the level of agreement of the two rankings on the rank of each measurement. The value of rs can range from -1 (perfect negative correlation) to 1 (perfect positive correlation). A value of 0 indicates no correlation. We decided to use this correlation measurement (and not the more common Pearson correlation), since

[1] Spearman's and Kendall usually produce very similar results and there is no strong reason for preferring one over the other.[12]

rs can be applied independent of the underlying data distribution, and independent of the nature of the relationship (which need not be linear).

We have calculated rs for each package of each application. As expected, for all cases, rs is positive and between 0.5 and 1. This indicates a strong positive correlation. Table 2 shows for each application, the minimum and maximum rs values observed in its packages. This means that the number of inherited methods increase with the DIT or DIT$_A$.

5.2. Distribution analysis

So the number of inherited methods is correlated to the depth of inheritance tree. This was predictable and predicted. The next question is "how does the number of inherited methods increase with the DIT?"

To have a first idea, we have drawn the histograms of the number of inherited method distribution for a fixed DIT (resp. DITA). Fig. 4(a-c) draws these histograms for DIT = 2, 3 and 4 and Fig. 4(d-f) draws them for DIT$_A$ = 1, 2 and 3. For DIT = 1, all classes inherit from Object and thus have 11 inherited methods. For DIT$_A$ = 0, application classes do not inherit from another application class, thus they have 0 inherited application methods.

We have also computed the number of inherited method in average for each of those distributions. For DIT = 2, 3 and 4, the average number of inherited methods is 22, 37 and 56. For DIT$_A$ = 1, 2 and 3, the average number of inherited application methods is 11, 25 and 62.

When considering DIT$_A$, one can notice a gap between average values at DIT$_A$ = 2 and DIT$_A$ = 3. When considering DIT, the gap occurs between DIT=3 and DIT=4. The gap occurs at only one level of difference because most of the application class inherits always from `Object` class.

The reason why there is a gap between DIT$_A$ = 2 and DIT$_A$ = 3 (or DIT=3 and DIT=4) is not understood yet. However, it seems to confirm a practical design rule stating there should be no more that 3 levels of inheritance otherwise test becomes very difficult[2].

6. Defined methods and DIT/DIT$_A$

Previous analysis has confirmed that the number of inherited methods was increasing with the DIT, both considering the total inheritance tree or tree restricted to the application classes. Another question is: does the number of defined method increase with DIT?

One can expect that the number of methods defined in a class is increasing with the depth of inheritance tree since

[2]An application class with DIT$_A$ = 3 is the fourth in the application inheritance tree, since the root of the application tree has DIT=0.

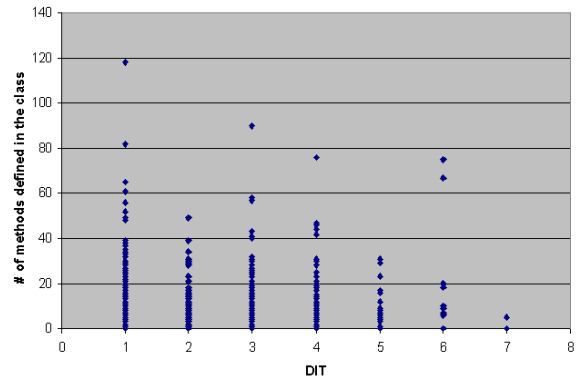


Figure 5. Number of defined methods w.r.t. DIT

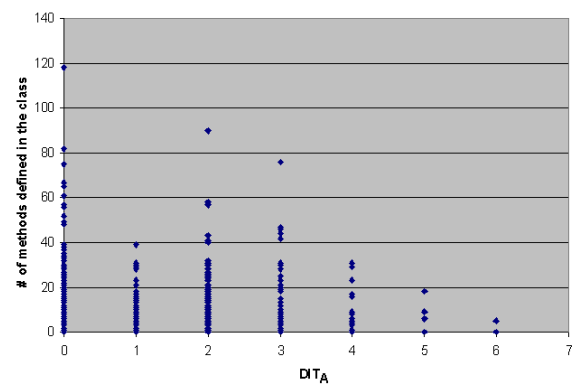
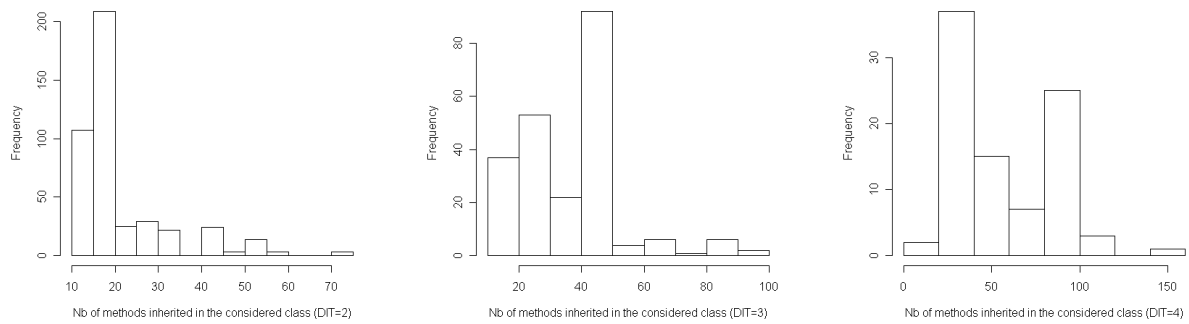
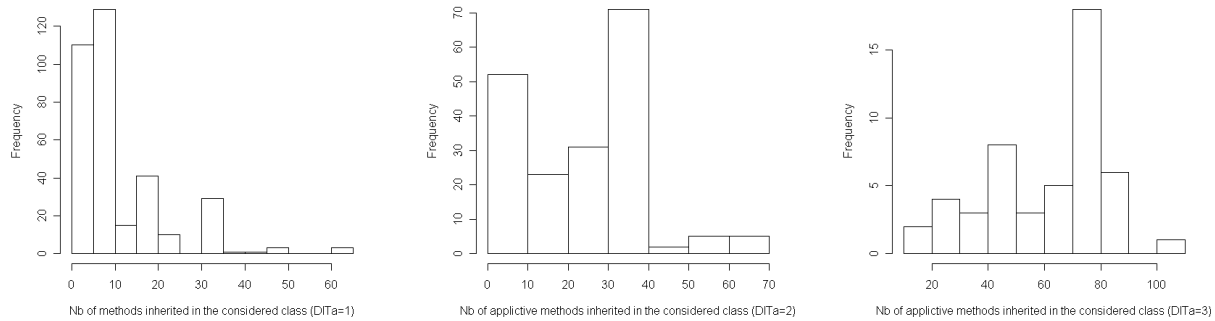


Figure 6. Number of defined methods w.r.t. DIT$_A$



(a-c) Number of inherited methods distribution for DIT = 2, 3 and 4



(d-f) Number of inherited application methods distribution for DIT$_A$ = 1, 2 and 3

Figure 4. Number of (inherited) application methods w.r.t. DIT (and DIT$_A$)

the behavior of the object is expected to be more complex (or at least, more detailed). Or, one can expect that fewer methods are required since most of the behavior has been previously defined. And finally, one can expect that there is no relation between them.

Again, as a first informal verification, we have drawn the scatter plot diagrams between the number of defined methods and the DIT. Fig 5 and 6 display the number of defined methods with respect to the DIT and DIT$_A$. The scatter plot diagrams show that there is no evident relation between the number of defined methods and the DIT or DIT$_A$.

We have calculated the Spearman's rank order for the classes considered all together. We have obtained rs($|M_D(c)|$,DIT) = -0.069 and $rs(|M_D(c)|$,DIT$_A$) = 0.173. This indicates that there is no correlation between the number of declared methods and the depth of inheritance.

To visualize the distribution law, we have drawn the histograms of the number of declared methods distribution for a fixed DIT and a fixed DIT$_A$ (see Fig. 7). These histograms give the impression that the different distributions follow the same probabilistic law. To check if the different distributions follow the same law, we have used the Wilcoxon test. This is a nonparametric test that compares two paired groups. It has an associated null hypothesis. Generally, one rejects the null hypothesis if the p-value is smaller than or equal to the significance level. If the level is 0.05, then the results are only 5% likely to be as extraordinary as just seen, given that the null hypothesis is true.

	1	2	3	4	5
1	-	2.28 10^{-12}	0.404	0.282	0.763
2	-	-	2.51 10^{-8}	0.016	0.002
3	-	-	-	0.181	0.834
4	-	-	-	-	0.326

Table 3. P-value of Wilcoxon test for the declared method distribution at a fixed DIT

Results for DIT and DIT$_A$ are given tables 3 and 4. For DIT$_A$ = 2, 3 and 4, it is reasonable to accept that the samples come from the same law. It is also reasonable to accept that samples from DIT$_A$ = 0 and 1 come from the same law, which is different from the previous ones. For DIT, it is reasonable to accept that all the sample come from the same law, expect the second sample (DIT=2), which is different.

	0	1	2	3	4
0	-	0.046	$3.23\ 10^{-7}$	0.0003	0.21
1	-	-	$4.14\ 10^{-10}$	8.710^{-6}	0.05
2	-	-	-	0.48	0.68
3	-	-	-	-	0.53

Table 4. P-value of Wilcoxon test for the declared method distribution at a fixed DIT$_A$

7. Conclusion and perspectives

The primary focus of our research is on the understanding of how the Depth of Inheritance Tree (DIT) can be used to predict the cost of unit testing. Our work is motivated by the different feedbacks from the literature about the influence of inheritance on the quality of the developed systems. In this paper, we express the cost of unit testing as the number of methods to be tested. We have distinguished two types of testing strategies: those which consider that inherited methods should be re-tested and those which only consider defined methods to be tested. Moreover, we have distinguished two types of inheritance tree: the classical one (DIT) which considers the entire inheritance tree, another one (DIT$_A$) restricted to the application classes. The reason for this distinction is that from a testing point of view, the standard (JDK) methods are quite never retested.

In this paper, we consider four hypotheses:

1-1 For testing strategies that do not consider inheritance, the number of defined methods in a class is not influenced by DIT.

1-2 For testing strategies that do not consider inheritance, the number of defined methods in a class is not influenced by DIT$_A$.

2-1 For testing strategies that consider inheritance, the number of inherited methods in a class is influenced by DIT.

2-2 For testing strategies that consider inheritance, the number of inherited methods in a class is influenced by DIT$_A$.

Statistical analysis have been carried out on 6 open-source applications, corresponding to more than 1700 classes. Section 5 focused on hypotheses 2-1 and 2-2. It showed that both can be accepted. Section 6 focused on hypotheses 1-1 and 1-2. It also showed that both can be accepted.

The originality of the work is in its distinction of testing strategies and inheritance tree. It is a first step before being able to use correctly depth of inheritance tree as a predictive measure for testing cost evaluation.

As a perspective of this work, new studies should be carried out to compare DIT and DIT_A with test cases, as it was done in [9]. An important requirement on the test cases is that they have to be created with a testing strategy that requires inherited method to be re-tested[3].

Inheritance is a mean to re-use code. Several propositions were done to re-use also tests produced in the inherited classes such as in [20, 1]. It would also be interesting to evaluate how much is the testing effort (in term of test case creation and code adaptation) when tests from a superclass are re-used for subclass. It is probably different from the effort to create test at the first time. Moreover, it would be interesting to evaluate how much companies really re-test inherited methods and how much standard require it.

Acknowledgments
Part of this research work has been funded by the ISLE cluster of the French Rhône-Alpes regions.

References

[1] Jehad Al-Dallal and Paul G. Sorenson. Reusing class-based test cases for testing object-oriented framework interface classes. *Journal of Software Maintenance*, 17(3):169–196, 2005.

[2] J.M. Armstrong and R.J. Mitchell. Uses and abuses of inheritance. *Software Engineering Journal*, 9(1):19–26, january 1994.

[3] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Software Eng.*, 22(10):751–761, 1996.

[4] R. V. Binder. Design for testability in object-oriented systems. *Communications of the ACM*, 37(9):87–101, September 1994.

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

[6] Lionel C. Briand, Jürgen Wüst, John W. Daly, and D. Victor Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, 51(3):245–273, 2000.

[7] Lionel C. Briand, Jürgen Wüst, Stefan V. Ikonomovski, and Hakim Lounis. Investigating

[3] One should notice that classical coverage tools evaluate coverage only with respect to methods defined in the class under test.

quality factors in object-oriented designs: An industrial case study. In *ICSE*, pages 345–354, 1999.

[8] Lionel C. Briand, Jürgen Wüst, and Hakim Lounis. Replicated case studies for investigating quality factors in object-oriented designs. *Empirical Software Engineering*, 6(1):11–58, 2001.

[9] M. Bruntink and A. van Deursen. An empirical study into class testability. *Journal of Systems and Software*, 79(9):1219–1232, September 2006.

[10] T. J. Cheatham and L. Mellinger. Testing object-oriented software systems. In *ACM Conference on Computer Science*, pages 161–165, 1990.

[11] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Eng.*, 20(6):476–493, 1994.

[12] D. J. Colwell and J. R. Gillett. Spearman versus kendall. *The Mathematical Gazette*, 66(438):307–309, 1982.

[13] Fernando Brito e Abreu and Rogério Carapuça. Candidate metrics for object-oriented software within a taxonomy framework. *Journal of Systems and Software*, 26(1):87–96, 1994.

[14] Fernando Brito e Abreu and Walcélio L. Melo. Evaluating the impact of object-oriented design on software quality. In *IEEE METRICS*, pages 90–99, 1996.

[15] Khaled El Emam, Saïda Benlarbi, Nishith Goel, and Shesh N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. Software Eng.*, 27(7):630–650, 2001.

[16] S. P. Fiedler. Object-oriented unit testing. *Hewlett-Packard Journal*, 40(2):69–75, April 1989.

[17] Tibor Gyimóthy, Rudolf Ferenc, and István Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Software Eng.*, 31(10):897–910, 2005.

[18] M.J. Harrold, J.D. McGregor, and K.J. Fitzpatrick. Incremental testing of object-oriented class structures. In *International Conference on Software Engineering (ICSE)*, May 1992.

[19] JUnit. http://www.junit.org.

[20] Leesa Murray, David A. Carrington, Ian MacColl, and Paul A. Strooper. Extending test templates with inheritance. In *1997 Australian Software Engineering Conference (ASWEC '97)*, pages 80–87, Sydney, Australia, September 1997. IEEE Computer Society.

[21] G. Myers. *The Art Of Software Testing*. Wiley-Interscience, 1979.

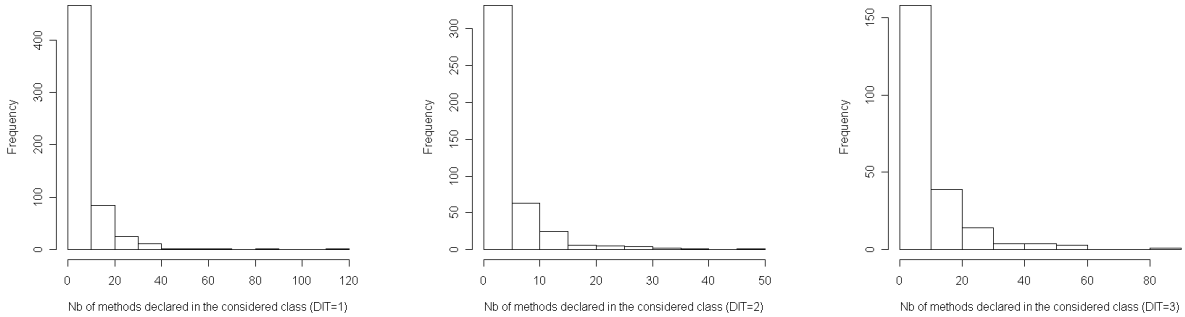
[22] Marc De Scheemaecker. About nanoxml, 2007.

[23] S. Siegel and N.J.C. Castellan Jr. *Nonparametric Statistics for the Behavioral Sciences*. McGraw-Hill Book Company, New York, 1998.

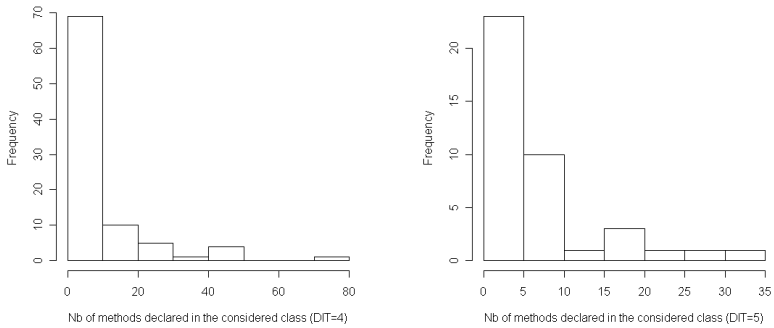
[24] Mei-Huei Tang, Ming-Hung Kao, and Mei-Hwa Chen. An empirical study on object-oriented metrics. In *6th IEEE International Software Metrics Symposium (METRICS'99)*, pages 242–249, Boca Raton, FL, USA, November 1999. IEEE Computer Society.

[25] Ping Yu, Tarja Systä, and Hausi A. Müller. Predicting fault-proneness using oo metrics: An industrial case study. In *6th European Conference on Software Maintenance and Reengineering (CSMR 2002)*, pages 99–107, Budapest, Hungary, March 2002. IEEE Computer Society.

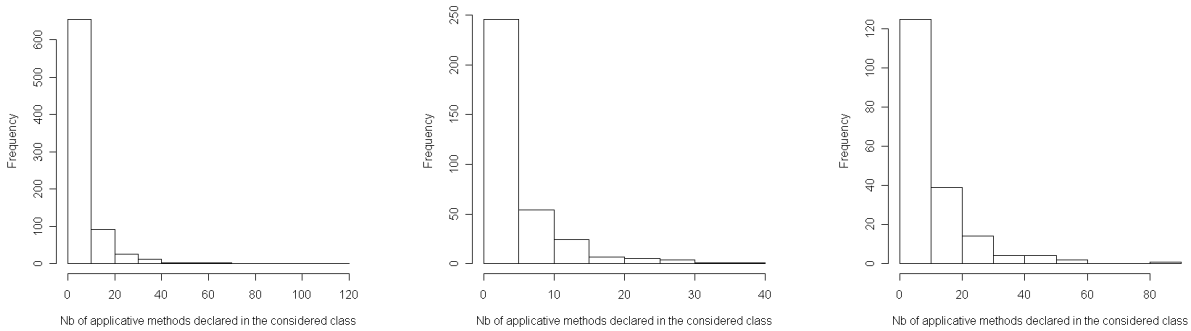
[26] Yuming Zhou and Hareton Leung. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Trans. Software Eng.*, 32(10):771–789, 2006.



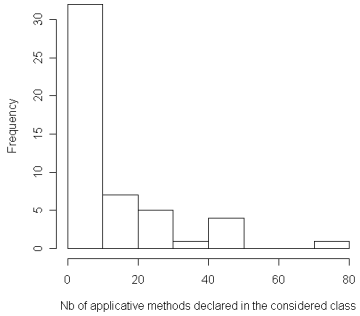
(a-c) Number of defined methods distribution for DIT = 1, 2 and 3



(d-e) Number of defined methods distribution for DIT = 4 and 5



(f-h) Number of defined application methods distribution for DIT_A = 0, 1 and 2



(i) Number of defined application methods distribution for DIT_A = 3

Figure 7. Number of defined methods w.r.t. DIT and DIT_A

Annexe 3

Curriculum Vitae

née le 2 juillet 1973
MdC à l'université J. Fourier, Grenoble I.
Laboratoire d'Informatique de Grenoble (LIG)
BP. 72, 38402 Saint-Martin d'Hères cedex, FRANCE
27ème section au CNU
lydie.du-bousquet@imag.fr

A. Parcours professionnel

2000- MdC à l'université J. Fourier, Grenoble I
Laboratoire LSR puis LIG, équipe Vasco
Titulaire de la PEDR depuis octobre 2008.

1999-00 Postdoc à l'IRISA, Rennes, France, projet Pampa (resp. Claude Jard)
En collaboration avec Gemplus Research Labs, équipe de P. Paradinas
sujet : Test de conformité basé sur des spécifications UML

1996-99 Doctorante sous la direction de F. Ouabdesselam et J.-L. Richier
Laboratoire LSR, IMAG, Grenoble
Test fonctionnel statistique de systèmes spécifiés en Lustre, application à la validation de services téléphoniques

B. Encadrement d'étudiants

	2000-01	02-03	2004	2005	2006	2007	2008	2009	2010
Doctorants	——— PB ———								
				SH					
					——— MRS ———				
DEA		SB, HB, MK	BB	FA	MSB				AA, TT
CNAM					SV				

PB : P. Bontron (33%), directeurs Y. Ledru et M.-L. Potet, soutenue le 1er mars 2005
SH : S. Housseno (50%), directeur Ch. Robach, non autorisé à poursuivre
MRS : Muhammad Rabee Shaheen (90%), directeur F. Ouabdesselam, soutenue le 7 octobre 2009

AA : A. Amiar (60%)
BB : B. Baldassari (50%)
FA : F. Allouti (50%)
HB : H. Bouldjeri (50%)
MK : M. Kessis (50%)
MSB : M.-S. Bouaten (100%)
SB : S. Beghdadi (50%)
TT : T. Taha (40%)

SV : S. Ville (50%)

C. Participation et montage de projets

Nom	Année	Type	Responsabilité	Etat
IO32	2010	FUI	Participant	Déposé
TASCCC	2009-11	ANR	Participant	En cours
SIESTA	2008-11	ANR	Participant	En cours
COST	2009-20	PHC Aurora	Responsable Français	En cours
i-POTEST	2007-08	Projet UJF	Participant	Terminé
DVV-DHNS-2	2007 et 08	PHC Sakura	Responsable Français	Terminé
POSE	2005-07	RNTL	Participant	Terminé
COCOVI	2006	projet IMAG	Participant	Terminé
CONTEST	2004	projet IMAG	Co-responsable	Terminé
COTE	200-02	RNTL	Participant	Terminé
IO32	2009	FUI	Participant	Labélisé, non financé
DVV-DHNS-1	2006	PHC sakura	Responsable Français	Non accepté
MINT	2004	BQR INP	Co-responsable	Non accepté
ATELIER-J	2003	RNTL	Participant	Non accepté
COMPOSTAGE	2002	RNTL	Responsable	Non accepté

*DVV-DHNS = Design, Validation and Verification of Dependable Home Network Services.
PHC : Partenariats Hubert Curien, Egide.

D. Service à la communauté académique et scientifique

Animation de réseau

– En 2004, j'ai été porteur de l'AS 161 "Testabilité des systèmes informatiques".

Organisation de congrès et Comité de sélection

– Comité d'organisation du *First International Workshop on Testability Assessment* (IWOTA 2004)

– PC Co-chair de l'*International Conference on Feature Interactions in Software and Communication Systems* (ICFI 2007).

– PC Co-chair du *5th International Workshop on Mutation Analysis*

– membre du comité de programme de IWOTA 2004

– membre du comité de programme de ICFI 2007 et 2009

– membre du comité de programme de ICSEA 2007, 2008, 2009 et 2010

– membre du comité de programme de ICST 2009 et 2010

– membre du comité de programme de ICSOFT 2008, 2009, 2010

– membre du comité de programme de VALID 2009 et 2010

Autre

– Membre nommée au CNU en Janvier 2009

E. Invitations

– J'ai été invitée par l'université de NARA pour un **séjour** de 2 semaines en 2006.

– J'ai été invitée au **Panel** de la conférence ICFI 2005.

– J'ai été invitée pour une **présentation** lors de l'école d'été TAROT (juillet 2007).

– J'ai fait partie du **Jury de thèse** de O. Maury, soutenue à l'université J. Fourier (déc. 2005).

F. Enseignement

En terme d'enseignement, mon service a varié entre 200 et 300 heures par an depuis septembre 2000. J'enseigne essentiellement en Licence et Master, sur des thèmes tels que l'introduction à la programmation, le génie logiciel, le

test et l'architecture logicielle. Je détaille ci-après les responsabilités en terme d'enseignement.

– Je suis reponsable du Master 2 Génie Informatique en Apprentissage depuis septembre 2007 (12 à 16 alternants par ans).

– Je suis co-Responsable de l'UE INF112 (Informatique Instrumental et multimédia) , depuis 2003. Cette UE s'addres à des étudiants en première année de Licence de Biologie, Chimie-Biologie et Science de la Vie et de la Terre, et concerne entre 14 et 16 groupes de 30 étudiants chaque année.

– Entre septembre 2004 et septembre 2007, j'ai été co-responsable d'un module de Travail et de Recherche (TER) destiné à des étudiants de M1. Ce module est une introduction à la recherche, qui correspond à 9 ECTS. Il se déroule sous la forme d'un stage dans un laboratoire à raison d'un jour et demi par semaine au second semestre, suivi de 4 semaines à temps plein.

G. Publications

Tableau récapitulatif

	1996-99	2000-01	2002-03	2004-05	2006-07	2008-09	2010
RI	[95]					[81, 86]	
RN	[94]	[103]			[244]		
EA					[99]		
CI	[100, 65] [**?**, 93, 91] [88, 89]	[155, 104] [203, 98]	[96]	[77, 156]	[73, 74] [243, 69]	[213, 214] [97, 212]	[220]
WI	[90]	[68, 84] [166, 83]	[87, 78] [165, 168]	[63, 105] [80, 12]	[85, 152] [54, 79] [153]	[117, 191] [211]	[71, 82] [197]
CN	[92]	[28]		[147]			
Au	[67]				[189]		

RI : Revue internationale
RN : Revue nationale ou article court dans une revue internationale
EA : édition d'actes
CI : Conférences internationales avec comité de programme
WI : Workshops internationaux ou articles courts dans une conférence internationale
CN : Conférences nationales avec comité de programme
Au : Autres types de publications (Thèse, Dépôt APP).

Références

[1] H. Agrawal, R. Demillo, R. Hathaway, Wm. Hsu, W. Hsu, E. Krauser, R. J. Martin, A. Mathur, and E. Spafford. Design of Mutant Operators for the C Programming Language. Technical Report SERC-TR-41-P, Soft. Eng. Research Center, Dep. of Computer Science, Purdue Univ., Indiana, 1989.

[2] A. T. Agree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical report git-ics-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, USA, September 1979.

[3] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4 :32–54, 2005.

[4] M. Aiello. The role of web services at home. In *International Conference on Internet and Web Applications and Services, WEBSA*, pages 164–ff, 2006.

[5] OSGi Alliance. *OSGi Service Platform : Release 3, March 2003*. IOS Press, 2003.

[6] A. Amiar. Evaluation de stratégies de réduction de suites de test combinatoire. Rapport de M2R, Université J. Fourier, Grenoble, France, 2010.

[7] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments ? In *27th International Conference on Software Engineering (ICSE'05)*, pages 402–411, St. Louis, Missouri, USA, May 2005. ACM.

[8] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. Technical Report (2010-10), Simula Research Laboratory, 2010.

[9] A. Arcuri, M. Z. Iqbal, and L. Briand. Formal analysis of the effectiveness and predictability of random testing. In *19th international symposium on Software testing and analysis (ISSTA)*, pages 219–230, NY, USA, 2010. ACM.

[10] J. Bach and P. J. Schroeder. Pairwise testing : A best practice that isn't. In *22nd Annual Pacific Northwest Software Quality Conference*, pages 180–196, 2004.

[11] R. Bache and M. Müllerburg. Measures of testability as a basis for quality assurance. *Software Engineering Journal*, pages 86–92, March 1990.

[12] B. Baldassari, C. Robach, L. du Bousquet, and J Brosse. Early metrics for object oriented designs. In *1st Int. Workshop on Testability Assessment (IWoTA) (in conjunction with ISSRE04)*, Rennes, France, November 2004.

[13] L. Baresi, C. Ghezzi, and S. Guinea. Smart monitors for composed services. In *International Conference on Service-Oriented Computing (ICSOC)*, pages 193–202, 2004.

[14] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass - java with assertions. *Electr. Notes Theor. Comput. Sci.*, 55(2), 2001.

[15] I. Bashir and A. L. Goel. *Testing Object-Oriented Software. Life cycle solutions*. Springer, 1999.

[16] B. Baudry, Y. Le Traon, and G. Sunyé. Testability analysis of a uml class diagram. In *8th IEEE International Software Metrics Symposium (METRICS 2002)*, pages 54–, Ottawa, Canada, June 2002.

[17] B. Baudry, Y. Le Traon, G. Sunyé, and J.-M. Jézéquel. Towards a 'safe' use of design patterns to improve oo software testability. In *12th International Symposium on Software Reliability Engineering (ISSRE 2001)*, pages 324–331, Hong Kong, China, November 2001. IEEE Computer Society.

[18] B. Baudry, Y. Le Traon, G. Sunyé, and J.-M. Jézéquel. Measuring and improving design patterns testability. In *9th IEEE International Software Metrics Symposium (METRICS 2003)*, pages 50–, Sydney, Australia, September 2003.

[19] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software : The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[20] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1983.

[21] R. G. (Ben) Bennitts. Progress in design for test : A personal view. *IEEE Design & Test of Computers*, 11(1) :53–59, 1994.

[22] A. Bertolino, G. De Angelis, L. Frantzen, and A. Polini. Model-based generation of testbeds for web services. In *Testing of Software and Communicating Systems, 20th IFIP TC 6/WG 6.1 International Conference, TestCom 2008, 8th International Workshop (TestCom/FATES)*, volume 5047 of *Lecture Notes in Computer Science*, pages 266–282, Tokyo, Japan, June 2008. Springer.

[23] A. Bertolino and L. Strigini. On the use of testability measures for dependability assessment. *IEEE Trans. Software Eng.*, 22(2) :97–108, 1996.

[24] R. V. Binder. Design for Testability in Object-Oriented Systems. *Communications of the ACM*, 37(9) :87–100, September 1994.

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

[26] B.W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[27] P. Bontron. *Les schémas de test : une abstraction pour la génération de tests de conformité et la mesure de couverture*. PhD thesis, Université Joseph Fourier, Grenoble, France, March 2005.

[28] 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 *International Conference on Software and Systems Engineering and their Applications (ICSSEA)*, Paris, France, December 2001.

[29] A. Bouchachia, R. Mittermeir, P. Sielecky, S. Stafiej, and M. Zieminski. Nature-inspired techniques for conformance testing of object-oriented software. *Applied Software Computing*, 10(3) :730 – 745, 2010.

[30] F. Bouquet, B. Legeard, F. Peureux, and E. Torreborre. Mastering test generation from smart card software formal models. In *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Device, (CASSIS)*, volume 3362 of *Lecture Notes in Computer Science*, pages 70–85, Marseille, France, 2004. Springer.

[31] J. Bourcier, A. Chazalet, M. Desertot, C. Escoffier, and C. Marin. A dynamic-soa home control gateway. In *IEEE International Conference on Services Computing (SCC 2006)*, pages 463–470, Chicago, Illinois, USA, September 2006.

[32] L. C. Briand. A critical analysis of empirical research in software testing. *Empirical Software Engineering and Measurement, International Symposium on*, 0 :1–8, 2007.

[33] L. C. Briand, J. W. Daly, and J. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Software Eng.*, 25(1) :91–121, 1999.

[34] L. C. Briand, Y. Labiche, and S. He. Automating regression test selection based on uml designs. *Information & Software Technology*, 51(1) :16–30, 2009.

[35] L. C. Briand, Y. Labiche, and H. Sun. Investigating the use of analysis contracts to improve the testability of object-oriented code. *Softw., Pract. Exper.*, 33(7) :637–672, 2003.

[36] E. Brinksma and J. Tretmans. Testing Transition Systems : An Annotated Bibliography. In F. Cassez, C. Jard, B. Rozoy, and M. Ryan, editors, *Summer School MOVEP'2k – Modelling and Verification of Parallel Processes*, pages 44–50, Nantes, July 2000.

[37] E. Brinksma, J. Tretmans, and L. Verhaard. A framework for test selection. In *Protocol Specification, Testing and Verification XI*, 1991.

[38] M. Bruntink and A. van Deursen. An empirical study into class testability. *Journal of Systems and Software*, 79(9) :1219–1232, September 2006.

[39] T. A. Budd. Mutation analysis of program test data. Phd thesis, Yale University, New Haven, CT, USA, 1980.

[40] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness : a developer-oriented approach. In *the 12th International FME Symposium*, Pisa, Italy, September 2003.

[41] A. Causevic, D. Sundmark, and S. Punnekkat. An industrial survey on contemporary aspects of software testing. In *Third International Conference on Software Testing, Verification and Validation (ICST)*, pages 393 –401, 6-10 2010.

[42] P. Chalin. Early detection of jml specification errors using esc/java2. In *conference on Specification and verification of component-based systems (SAVCBS)*, pages 25–32, New York, NY, USA, 2006. ACM.

[43] T. Chatain and C. Jard. Symbolic diagnosis of partially observable concurrent systems. In *24th IFIP WG 6.1 International Conference Formal Techniques for Networked and Distributed Systems (FORTE)*, volume

3235 of *Lecture Notes in Computer Science*, pages 326–342, Madrid, Spain, 2004. Springer.

[44] 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, June 2002. CSREA Press.

[45] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing : The JML and JUnit way. In *16th European Conference on Object-Oriented Programming (ECOOP'02)*, number 2374 in LNCS, pages 231–255. Springer, June 2002.

[46] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object oriented design. In *OOPSLA*, pages 197–211, 1991.

[47] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Eng.*, 20(6) :476–493, 1994.

[48] T. S. Chow. A generalized assertion language. In *2nd International Conference on Software Engineering (ICSE)*, pages 392–399. IEEE, 1976.

[49] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. Stg : A symbolic test generation tool. In *8th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS*, volume 2280 of *Lecture Notes in Computer Science*, pages 470–475, Grenoble, France, 2002. Springer.

[50] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 13(5) :83–88, 1996.

[51] S. D. Conte, H. E. Dunsmore, and V. Y. Shen. *Software engineering metrics and models*. Benjamin-Cummings Publishing Co., Inc., USA, 1986.

[52] D. Coppit and J. M. Haddox-Schatz. On the use of specification-based assertions as test oracles. In *29th Annual IEEE/NASA Software Engineering Workshop (SEW)*, pages 305–314, Greenbelt, Maryland, USA, April 2005. IEEE.

[53] F. Dadeau, Y. Ledru, and L. Du Bousquet. Directed random reduction of combinatorial test suites. In *RT '07 : Proceedings of the 2nd international workshop on Random testing*, pages 18–25, New York, NY, USA, 2007. ACM.

[54] F. Dadeau, Y. Ledru, and L. du Bousquet. Measuring a java test suite coverage using jml specifications. In *3rd Workshop on Model-Based Testing (MBT)*, Braga, Portugal, March 2007.

[55] F. Dadeau and R. Tissot. jsynopsys - a scenario-based testing tool based on the symbolic animation of b machines. *Electr. Notes Theor. Comput. Sci.*, 253(2) :117–132, 2009.

[56] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, Christopher M. Lott, Gardner C. Patton, and Bruce M. Horowitz. Model-based testing in practice. In *ICSE*, pages 285–294, 1999.

[57] M. D. Davis and E. J. Weyuker. Pseudo-oracles for non-testable programs. In *Proceedings of the ACM'81 conference*, pages 254–257, New York, NY, USA, 1981. ACM.

[58] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface mutation : An approach for integration testing. *IEEE TSE*, 27(3) :228–247, 2001.

[59] R. DeMillo, D. Guindi, K. King, M. M. McCracken, and J. Offutt. An extended overview of the mothra software testing environment. In *2nd Workshop on Software Testing, Verification, and Analysis*, pages 142–151, Banff, Canada, July 1988.

[60] 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.

[61] H.-V. Do, M. Delaunay, and C. Robach. Integrating testability into the development process of reactive systems. In *25th conference on IASTED International Multi-Conference : Software Engineering (SE'07)*, pages 322–327, Anaheim, CA, USA, 2007. ACTA Press.

[62] S. do Rocio Senger de Souza, J. C. Maldonado, S. C. P. F. Fabbri, and W. Lopes de Souza. Mutation testing applied to estelle specifications. In *HICSS*, 2000.

[63] G. J. Doherty, L. du Bousquet, J. Creissac Campos, E. M. El Atifi, G. Falquet, M. Massink, and C. Santoro. Ambience and mobility. In *Interactive Systems, Design, Specification, and Verification (DSV-IS'05)*, volume 3941 of *Lecture Notes in Computer Science*, page 264, Newcastle upon Tyne, UK, July 2005. Springer.

[64] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7) :1165 – 1178, july 2008.

[65] L. du Bousquet. Feature interaction detection using testing and model-checking, experience report. In *World Congress on Formal Methods*, volume 1708 of *LNCS*, pages 622–641, Toulouse, France, September 1999. Springer Verlag.

[66] L. du Bousquet. *Test fonctionnel statistique de systèmes spécifiés en Lustre*. PhD thesis, Université Joseph Fourier, Grenoble, France, September 1999.

[67] L. du Bousquet. Validation incrémentale de services téléphoniques. Rapport technique, to be published, Université Joseph Fourier, Grenoble, France, septembre 1999.

[68] L. du Bousquet. An Approach to Evaluate Testability. In *2nd Int. Workshop on Automated Program Analysis, Testing, and Verification (WAPATAV)*, Toronto, Canada, 2001.

[69] L. du Bousquet. Evaluating behavior correctness of synchronous systems through time to service distribution analysis : Tools required. In *International Conference on Software Engineering Advances (ICSEA'06)*, Papeete, Tahiti, French Polynesia, 2006. IEEE Computer Society.

[70] L. du Bousquet. Mutation analysis to evaluate lustre program specifications in the context of model-checking. Research Report RR-LIG-007, LIG, Grenoble, France, 2010.

[71] L. du Bousquet. A new approach for software testability - fast abstract. In *Testing : Academic and Industrial Conference Practice and Research Techniques*, page à paraitre, Windsor, UK, 2010. IEEE.

[72] L. du Bousquet and M. Delaunay. Mutation analysis for lustre programs : Fault model description and validation. In *Testing : Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 176–184, Windsor, UK, 2007. IEEE Computer Society.

[73] L. du Bousquet and M. Delaunay. Towards mutation analysis for lustre programs. In *Model-driven High-level Programming of Embedded Systems (SLA++P)*, Braga, Portugal, March 2007.

[74] L. du Bousquet and M. Delaunay. Using mutation analysis to evaluate test generation strategies in a synchronous context. In *Second International Conference on Software Engineering Advances (ICSEA)*, page 40, Esterel, France, August 2007. IEEE.

[75] L. du Bousquet and M. Delaunay. Towards mutation analysis for lustre programs. *Electr. Notes Theor. Comput. Sci.*, 203(4) :35–48, 2008.

[76] L. du Bousquet, M. Delaunay, H.-V. Do, and Robach Ch. Analysis of testability metrics for Lustre/Scade programs. In *Second International Conference on Advances in System Testing and Validation Lifecycle VALID*, Nice, France, August 2010. IEEE.

[77] L. du Bousquet and O. Gaudoin. Telephony feature validation against eventuality properties and interaction detection based on a statistical analysis of the time to service. In *Int. Conference on Feature Interactions in Telecommunications and Systems Software*, Leicester, UK, June 2005.

[78] L. du Bousquet, J.-L. Lanet, and H. Martin. Enhancing Java Card applet validation process : a methodology and its associated tools. In *e-SMART*, Sophia Antipolis, France, 2003.

[79] L. du Bousquet, Y. Ledru, F. Lydie Dadeau, and F. Allouti. A Case Study in Matching Test and Proof Coverage. In *3rd Workshop on Model-Based Testing (MBT), Satellite workshop of ETAPS 2007*, Braga, Portugal, March 2007.

[80] L. du Bousquet, Y. Ledru, O. Maury, C. Oriat, and J.-L. Lanet. A case study in JML-based software validation (short paper). In *Automated Software Engineering (ASE)*, Linz, Austria, September 2004.

[81] L. du Bousquet, Y. Ledru, O. Maury, C. Oriat, and J.-L. Lanet. Reusing a JML specification dedicated to verification for testing, and vice-versa : case studies. *Journal of Automatic Reasoning*, published online on June 2009.

[82] L. du Bousquet and M. Lévy. Proof process evaluation with mutation analysis. In *4th International Conference on Tests & Proofs (TAP)*, volume 6143 of *Lecture Notes in Computer Science*, pages 55–60, Málaga, Spain, July 2010. Springer.

[83] L. du Bousquet and H. Martin. Automatic test generation for java card applets. In *FM-tools wokshop*, Reisenburg, Germany, July 2000.

[84] L. du Bousquet, H. Martin, and J.-M. Jézéquel. Conformance Testing from UML specificationd, Experience Report. In Gesellschaft für Informatik (GI), editor, *p-UML workshop, Lecture Notes in Informatics (LNI)*, volume P-7, pages 43–56, Toronto, Canada, 2001.

[85] L. du Bousquet, M. Nakamura, B. Yan, and H. Igaki. Using formal methods to increase confidence in one home network system implementation. case study. In *workshop on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2007)*, Poitiers, France, décembre 2007.

[86] L. du Bousquet, M. Nakamura, B. Yan, and H. Igaki. Using formal methods to increase confidence in a home network system implementation : a case study. *Innovations in Systems and Software Engineering*, 5(3), September 2009.

[87] L. du Bousquet, F. Ouabdesselam, I. Parissis, J.-L. Richier, J. Vassy, and N. Zuanon. Black-box testing of reactive synchronous software. In *Soft-Test : UK Testing Research II*, Department of Computer Science, University of York, September 2003.

[88] L. du Bousquet, F. Ouabdesselam, I. Parissis, J.-L. Richier, and N. Zuanon. Lutess : a testing environment for synchronous software. In *Tool support for System Specification, Development, and Verification, Advances in Computing Science*, pages 48–61, Malente, Germany, 1998. Springer.

[89] L. du Bousquet, F. Ouabdesselam, and J.-L. Richier. Expressing and implementing operational profiles for reactive software validation. In *9th International Symposium on Software Reliability Engineering*, Paderborn, Germany, 1998.

[90] L. du Bousquet, F. Ouabdesselam, and J.-L. Richier. Telecommunication software validation using a synchronous approach. In *First IEEE Workshop on Application-Specific Software Engineering and Technology (ASSET'98)*, Dallas, USA, 1998.

[91] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Incremental feature validation : a synchronous point of view. In *Feature Interactions in Telecommunications Systems V*, pages 262–275. IOS Press, September 1998.

[92] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Environnement pour le test d'applications synchrones. In *2ème Congrès sur la Modélisation des systèmes réactifs*. Hermes, 1999.

[93] 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*, pages 267–276. ACM Press, May 1999.

[94] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Test et approche synchrone pour la détection d'interactions de services téléphoniques. *Calculateurs Parallèles, Systèmes Répartis, Réseaux*, 32(4) :419–432, 1999.

[95] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Feature interaction detection using synchronous approach and testing. *Computer Networks and ISDN Systems*, 11(4) :419–446, 2000.

[96] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Testing against some eventuality properties of synchronous software : a case study. In *Synchronous Languages, Applications, and Programming (SLAP'03)*, Porto, Portugal, July 2003. to be published electronically by Electronic Notes in Theoretical Computer Science (ENTCS), volume 88.

[97] L. du Bousquet, A. Rajan, C. Oriat, J.-L. Richier, and G. Vega. Service specification and validation in the context of the home. In *10th International Conference on Feature Interactions (ICFI 2009)*, pages 207–219, Lisbon, Portugal, June 2009. IOS Press.

[98] L. du Bousquet, S. Ramangalahy, S. Simon, C. Viho, A. Belinfante, and R. de Vries. Formal test automation : the conference protocol with tgv/torx. In *Int. Conf. on Testing of Communicating Systems (Testcom)*, Ottawa, Canada, september 2000.

[99] L. du Bousquet and J.-L. Richier, editors. *Feature Interactions in Software and Communication Systems IX, International Co nference on Feature Interactions in Software and Communication Systems, ICFI 2007, 3-5 September 2007, Grenoble, France*. IOS Press, 2007.

[100] L. du Bousquet and N. Zuanon. An overview of lutess, a specification-based tool for testing synchronous software. In *14th IEEE International Conference on Automated Software Engineering*. IEEE, October 1999.

[101] B.-M. Duc. *Conception et modélisation objet des systèmes temps réel*. Eyrolles, 1998.

[102] P. Dupont and L. Miclet. Inférence grammaticale régulière : fondements théoriques et principaux algorithmes. Technical report, IRISA, juillet 1998.

[103] S. Dupuy and L. du Bousquet. A Multi-formalism Approach for the Validation of UML models. *Formal Aspects of Computing*, 12(4) :228–230, 2000.

[104] S. Dupuy and L. du Bousquet. Validation of UML models thanks to Z and Lustre. In *Formal Methods Europe*, Berlin, Germany, march 2000.

[105] 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 (DSV-IS)*, volume 3941 of *LNCS*, pages 102–113. Springer, 2005.

[106] J. Duran and S. Ntafos. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering*, 10(4) :438–444, 1984.

[107] G. Durrieu, H. Waeselynck, and V. Wiels. Leto - a lustre-based test oracle for airbus critical systems. In *13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, volume 5596 of *Lecture Notes in Computer Science*, pages 7–22, L'Aquila, Italy, September 2008. Springer.

[108] M. Dyer. *The cleanroom approach to quality software development*. J. Wiley, 1992.

[109] J. Edvardsson. A survey on automatic test data generation. In *Second conference on Computer Science and Engineering in Linköping (ECSEL)*, pages 21–28, October 1999.

[110] Lem O. Ejiogu. Five principles for the formal validation of models of software metrics. *SIGPLAN Notices*, 28(8) :67–76, 1993.

[111] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3) :35–45, 2007.

[112] C. Escoffier, J. Bourcier, P. Lalanda, and Jianqi Yu. Towards a home application server. In *5th IEEE Consumer Communications and Networking Conference (CCNC 2008)*, pages 321–325, January 2008.

[113] C. Escoffier, R.S. Hall, and P. Lalanda. iPOJO : an extensible service-oriented component framework. In *IEEE International Conference on Services Computing (SCC 2007)*, pages 474–481, July 2007.

[114] S. C. P. F. Fabbri, J. C. Maldonado, M. E. Delamaro, and P. C. Masiero. Mutation Testing applied to Validate Specifications Based on Statecharts. In *10th International Symposium on Software Reliability Engineering*, Boca Radon, FL, USA, 1999.

[115] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, and M. E. Delamaro. Proteum/fsm : A tool to support finite state machine validation based on mutation testing. In *19th International Conference of the Chilean Computer Science Society (SCCC)*, pages 96–104, Talca, Chile, November 1999. IEEE Computer Society.

[116] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, M. E. Delamaro, and W. E. Wong. Mutation testing applied to validate specifications based on petri nets. In *Formal Description Techniques VIII, Proceedings of the IFIP TC6 Eighth International Conference on Formal Description Techniques (FORTE)*, volume 43 of *IFIP Conference Proceedings*, pages 329–337, Montreal, Canada, October 1995. Chapman & Hall.

[117] L. Ferro, L. Pierre, Y. Ledru, and L. du Bousquet. Generation of test programs for the assertion-based verification of tlm models. In *Design and Test Workshop, 2008. IDT 2008. 3rd International*, pages 237–242. IEEE, 2008.

[118] J.-C. Filliâtre and C. Marché. The why/krakatoa/caduceus platform for deductive program verification. In *19th International Conference Computer Aided Verification (CAV)*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. Springer.

[119] C. Fischer and D. Meemken. Jawa : Java with assertions. In *Java-Informations-Tage*, pages 49–59, 1998.

[120] C. Flanagan, K. R. M. Leino, Lillibridge M., G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245. ACM Press, 2002.

[121] International Organization for Standardization. Iso 9126 : Information technology software product evaluation quality characteristics and guidelines for their use. Technical report, ISO, Geneva, 1991.

[122] G. Garfinke. History's worst software bugs. *Wired.com*, Novembre 2005. http ://www.wired.com/news/technology/bugs/0,2924,69355,00.html.

[123] A.M. Geras, M.R. Smith, and J. Miller. A survey of software testing practices in alberta. *Canadian Journal of Electrical and Computer Engineering*, 29(3) :183 –191, july 2004.

[124] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.

[125] A. Gotlieb. Euclide : A constraint-based testing framework for critical c programs. In *Second International Conference on Software Testing Verification and Validation (ICST)*, pages 151–160, Denver, Colorado, USA, April 2009. IEEE Computer Society.

[126] M. Grindal, A. J. Offutt, and S. F. Andler. Combination testing strategies : a survey. *Softw. Test., Verif. Reliab.*, 15(3) :167–199, 2005.

[127] L. Groves, R. Nickson, G. Reeve, S. Reeves, and M. Utting. A survey of software development practices in the new zealand software industry. In *Australian Software Engineering Conference*, pages 189 –201, 2000.

[128] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Data Flow Programming Language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320, 1991.

[129] D. Hamlet and R. Taylor. Partition Analysis Does Not Inspire Confidence. *IEEE Transactions on Software Engineering*, pages 1402–1411, december 1990.

[130] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30 :3–16, 2004.

[131] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3) :270–285, 1993.

[132] R. M. Hierons. Testing from a Z Specification. *Softw. Test., Verif. Reliab.*, 7(1) :19–33, 1997.

[133] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan. Using formal specifications to support testing. *ACM Computing Surveys*, 41(2) :1–76, 2009.

[134] R. M. Hierons, S. Sadeghipourb, and H. Singhc. Testing a system specified using Statecharts and Z. *Information and Software Technology*, 43(2) :137–149, February 2001.

[135] W.-M. Ho, J.-M. Jézéquel, A. Le Guennec, and F. Pennaneac'h. Umlaut : An extendible uml transformation framework. In *ASE*, pages 275–278, 1999.

[136] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *Computer*, 30(4) :75–82, 1997.

[137] McCabe T. J. and Butler C. W. Design complexity measurement and testing. *Communication of the ACM*, 32(12) :1415–1425, 1989.

[138] E. Jaffuel and B. Legeard. Leirios test generator : Automated test generation from b models. In *7th International Conference of B Users*, volume 4355 of *Lecture Notes in Computer Science*, pages 277–280, Besançon, France, January 2007. Springer.

[139] P. Jalote. *An Integrated Approach to Software Engineering*. Springer-Verlag, 1991.

[140] C. Jard and T. Jéron. TGV : theory, principles and algorithms. *Software Tools for Technology Transfer*, 7(4) :297–315, 2005.

[141] Y. Jia and M. Harman. Higher order mutation testing. *Inf. Softw. Technol.*, 51(10) :1379–1393, 2009.

[142] The Java Modeling Language (JML) Home Page. http ://www.cs.iastate.edu/ leavens/JML.html.

[143] S. Jungmayr. Identifying test-critical dependencies. In *International Conference on Software Maintenance (ICSM)*, pages 404–413, Montreal, Quebec, Canada, 2002. IEEE Computer Society.

[144] JUnit. http ://www.junit.org.

[145] C. Kaner and W. P. Bond. Software engineering metrics : What do they measure and how do we know ? In *10th IEEE International Software Metrics Symposium (METRICS)*, Chicago, USA, September 2004. IEEE.

[146] B. Kitchenham, S. L. Pfleeger, and N. E. Fenton. Towards a framework for software measurement validation. *IEEE Trans. Software Eng.*, 21(12) :929–943, 1995.

[147] A. Lakehal, I. Parissis, and L. du Bousquet. Critères de couverture structurelle de programmes Lustre. In *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL)*, France, Juin 2004.

[148] J.C. Laprie. Informatique Sûre de Fonctionnement : Des Concepts Aux Limites. In *International Seminar held at LAAS-CNRS for its 25th anniversary*, pages 43–54, Toulouse, France, May 1993.

[149] Y. Le Traon and C. Robach. Testability Measurements for Data Flow Design. In *Proceedings of the Fourth International Software Metrics Symposium*, pages 91–98, Albuquerque, New Mexico, November 1997.

[150] 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.

[151] 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, June 2002.

[152] Y. Ledru, F. Dadeau, L. du Bousquet, S. Ville, and E. Rose. Mastering combinatorial explosion with the tobias-2 test generator. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pages 535–536, USA, November 2007. ACM.

[153] Y. Ledru and L. du Bousquet. Tobias-Z : An executable formal specification of a test generator. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 353–354, Tokyo, Japan, Septembre 2006. IEEE Computer Society.

[154] Y. Ledru and L. du Bousquet. De l'utilisation d'une spécification z comme prototype dans le redéveloppement d'un projet. In *10es Journées Francophones Internationales sur les Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL)*, Poitiers, France, juin 2010.

[155] Y. Ledru, L. du Bousquet, P. Bontron, O. Maury, C. Oriat, and M.-L. Potet. Test purposes : adapting the notion of specification to testing. In *16th IEEE Int. Conf. on Automated Software Engineering (ASE)*, San Diego, CA, USA, November 2001.

[156] Y. Ledru, L. du Bousquet, O. Maury, and P. Bontron. Filtering tobias combinatorial test suites. In *7th Int. Conf. FASE, Held as Part of ETAPS*, volume 2984 of *LNCS*, pages 281–294, Barcelona, Spain, 2004. Springer.

[157] T.-C. Lee and P.-A. Hsiung. Mutation coverage estimation for model checking. In *Second International Conference on Automated Technology for Verification and Analysis (ATVA)*, volume 3299 of *Lecture Notes in Computer Science*, pages 354–368, Taipei, Taiwan, ROC, October 2004. Springer.

[158] P. Leelaprute, T. Tsuchiya, T. Kikuno, M. Nakamura, and K.-I. Matsumoto. Describing and verifying integrated services of home network systems. In *12th Asia-Pacific Software Engineering Conference (APSEC'05*, pages 549–560, December, Taipei, Taiwan 2005. IEEE Computer Society.

[159] B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from z and b. In *International Symposium of Formal Methods Europe (FME)*, volume 2391 of *Lecture Notes in Computer Science*, pages 21–40, Copenhagen, Denmark, July 2002. Springer.

[160] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. IPOG/IPOG-D : efficient test generation for multi-way combinatorial testing. *Softw. Test., Verif. Reliab.*, 18(3) :125–148, 2008.

[161] D.C. Luckham and F.W. Von Henke. An overview of anna, a specification language for ada. *IEEE Software*, 2(2) :9 –22, March 1985.

[162] Y.-S. Ma, J. Offutt, and Y. R. Kwon. Mujava : an automated class mutation system. *Softw. Test., Verif. Reliab.*, 15(2) :97–133, 2005.

[163] L. Madani. *Utilisation de la programmation synchrone pour la spécification et la validation de services interactifs*. PhD thesis, Université Joseph Fourier, Grenoble, France, October 2007.

[164] C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certificationof JAVA/JAVACARD programs annotated in JML. *J. Log. Algebr. Program.*, 58(1-2) :89–106, 2004.

[165] H. Martin, F. Combret, L. du Bousquet, P. Bontron, and O. Maury. Toward testing automation. In *Gemplus Developer Conference*, Singapore, 2002.

[166] H. Martin and L. du Bousquet. Automatic test generation for Java-Card applets. In *the Java-Card Workshop*, Cannes, France, September 2000.

[167] O. Maury. *Outils pour la synthèse de tests et la maîtrise de l'explosion combinatoire*. PhD thesis, Université Joseph Fourier, Grenoble, France, December 2005.

[168] O. Maury, Y. Ledru, and L. du Bousquet. Using TOBIAS for the automatic generation of VDM test cases. In *Third VDM workshop (in conjunction with FME2002)*, Copenhagen, Danemark, 2002.

[169] T. J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4) :308–320, 1976.

[170] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10) :40–51, 1992.

[171] J. Meyer and A. Poetzsch-Heffter. An Architecture for Interactive Program Provers. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1785 of *LNCS*. Springer, 2000.

[172] S. Mouchawrab, L. C. Briand, and Y. Labiche. A measurement framework for object-oriented software testability. *Information & Software Technology*, 47(15) :979–997, 2005.

[173] G. Myers. *The Art Of Software Testing*. Wiley-Interscience, 1979.

[174] M. Nakamura, H. Igaki, and K.-I. Matsumoto. Feature interactions in integrated services of networked home appliances : An object-oriented approach. In *Feature Interactions in Telecommunications and Software Systems VIII, (ICFI'05)*, pages 236–251, Leicester, UK, June 2005. IOS Press.

[175] M. Nakamura, A. Tanaka, H. Igaki, H. Tamada, and K. Matsumoto. Adapting Legacy Home Appliances to Home Network Systems Using Web Services. In *Int. Conf. on Web Services (ICWS 2006)*, pages 849–858. IEEE, September 2006.

[176] B. A. Nejmeh. Npath : a measure of execution path complexity and its applications. *Communication of the ACM*, 31(2) :188–200, February 1988.

[177] S. Nelson and J. Schumann. What makes a code review trustworthy ? In *37th Annual Hawaii International Conference on System Sciences*, page 10 pp., 5-8 2004.

[178] S. P. Ng, T. Murnane, K. Reed, D. Grant, and T. Y. Chen. A preliminary survey on software testing practices in australia. In *15th Australian Software Engineering Conference (ASWEC)*, pages 116–127, Melbourne, Australia, April 2004. IEEE Computer Society.

[179] T. B. Nguyen and C. Robach. Mutation Testing Applied to Hardware : the Mutants Generation. In *Proceedings of the 11th IFIP International Conference on Very Large Scale Integration*, pages 118–123, Montpellier, France, December 2001.

[180] Ch. Nie, H. Leung, and B. Xu. The minimal failure-causing schema of combinatorial testing, A paraître.

[181] Institute of Electrical and Electronics Engineers. IEEE Standard Computer Dictionary : A Compilation of IEEE Standard Computer Glossaries. Technical report, IEEE, New York, USA, 1990.

[182] A. J. Offutt. A practical system for mutation testing : Help for the common programmer. In *Proceedings IEEE International Test Conference (ITC'94)*, pages 824–830, Washington, DC, USA, October 1994.

[183] A. J. Offutt and W. M. Craft. Using compiler optimization techniques to detect equivalent mutants. *Softw. Test., Verif. Reliab.*, 4(3) :131–154, 1994.

[184] A. J. Offutt and J. H. Hayes. 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, January 1996.

[185] A. J. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *Softw. Test., Verif. Reliab.*, 7(3) :165–192, 1997.

[186] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *ICSE*, pages 100–107, 1993.

[187] A. J. Offutt, J. Voas, and J. Payne. Mutation Operators for Ada. Technical Report ISSE-TR-96-06, George Mason University, 1996.

[188] C. Oriat. Jartege : A Tool for Random Generation of Unit Tests for Java Classes. In *2nd International Workshop on Software Quality (SOQUA 2005)*, pages 242–256, Erfurt, Germany, September 2005. LNCS 3712, Springer.

[189] Y. Ledru P. Bontron, L. du Bousquet and O. Maury. Tobias. Référencement par l'Agence de Protection des Programmes IDDN.FR.001.230021.000.R.P.2005.000.10600, déposé par l'Univ. Joseph Fourier, 2005.

[190] S. Packevicius, A. Usaniov, and E. Bareisa. Software testing using imprecise ocl constraints as oracles. In *International Conference on Computer Systems and Technologies (CompSysTech)*, page 121, Rousse, Bulgaria, June 2007. ACM.

[191] V. Papailiopoulou, L. Madani, L. du Bousquet, and I. Parissis. Extending structural test coverage criteria for lustre programs with multi-clock operators view. In *13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, L'Aquila, Italy, September 2008.

[192] V. Papailliopoulou. *Test automatique de programmes Lustre/SCADE*. PhD thesis, Université Joseph Fourier, Grenoble, France, Février 2010.

[193] I. Parissis. *Test de logiciels synchrones spécifiés en Lustre*. PhD thesis, Université Joseph Fourier, Grenoble, France, september 1996.

[194] B. Pettichord. Success with Test Automation. In *Quality Week*, San Francisco, May 1996. http ://www.io.com/ wazmo/succpap.htm.

[195] R. Plösch. Evaluation of assertion support for the java programming language. *Journal of Object Technology*, 1(3) :5–17, 2002.

[196] R. S. Pressman. *Software engineering : a practitioner's approach*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[197] A. Rajan, L. du Bousquet, Y. Ledru, G. Vega, and J.-L. Richier. Assertion-based test oracles for home automation systems. In *7th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES), C collocated with ASE*, Antwerp, Belgium, Septembre 2010.

[198] S. Reiff-Marganiec and M. Ryan, editors. *Feature Interactions in Telecommunications and Software Systems VIII, ICFI'05, 28-30 June 2005, Leicester, UK*. IOS Press, 2005.

[199] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In *14th international conference on Software Engineering (ICSE)*, pages 105–118, New York, NY, USA, 1992. ACM.

[200] C. Robach and S. Guibert. Information Based Testability Measures. In *Proceedings of Silicon Design Conference*, pages 429–438, Wembley, GB, 1986.

[201] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Softw. Eng.*, 21(1) :19–31, 1995.

[202] G. Rothermel and M.J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8) :529–551, August 1996.

[203] V. Rusu, L. du Bousquet, and T. Jéron. An approach to symbolic test generation. In *Integrated Formal Methods (IFM), LNCS 1945*, pages 338–357, Dagstuhl, Germany, 2000. Springer.

[204] A. Santos. Overture : Combinatorial test automation support for vdm++. In *Fourth VDM-Overture Workshop at FM'08*, Turku, Finland, May 2008.

[205] D. Seifert. Conformance testing based on uml state machines. In *10th International Conference on Formal Engineering Methods (ICFEM)*, volume 5256 of *Lecture Notes in Computer Science*, pages 45–65, Kitakyushu-City,Japan, October 2008. Springer.

[206] B. Seljimi. *Test de logiciels synchrones avec la PLC*. PhD thesis, Université Joseph Fourier, Grenoble, France, July 2009.

[207] N. Sethi and C. Barrett. Cascade : C assertion checker and deductive engine. In *18th International Conference on Computer Aided Verification (CAV)*, volume 4144 of *Lecture Notes in Computer Science*, pages 166–169, Seattle, WA, USA, August 2006. Springer.

[208] S. R. Shahamiri, W. M. N. Wan Kadir, and S. Z. Mohd-Hashim. A comparative study on automated software test oracle methods. In *The Fourth International Conference on Software Engineering Advances (ICSEA)*, pages 140–145, Porto, Portugal, September 2009. IEEE.

[209] M. Shahbaz and R. Groz. Inferring mealy machines. In *Second World Congress on Formal Methods (FM)*, volume 5850 of *Lecture Notes in Computer Science*, pages 207–222, Eindhoven, The Netherlands, November 2009. Springer.

[210] M.-R. Shaheen. *Validation de Métriques de Testabilité Logicielle pour Les Programmes Objets*. PhD thesis, Université Joseph Fourier, Grenoble, France, October 2009.

[211] M.-R. Shaheen and L. du Bousquet. Quantitative analysis of testability antipatterns on open source java applications. In *12th ECOOP Workshop on Quantitative Approaches on Object Oriented Software Engineering*, 2008.

[212] M.-R. Shaheen and L. du Bousquet. Relation between depth of inheritance tree and number of methods to test. In *1st International Conference on Software Testing, Verification and Validation*, Lillehammer, Norway, April 2008. IEEE.

[213] M.-R. Shaheen and L. du Bousquet. Analysis of the introduction of testability antipatterns during the development process. In *Fourth International Conference on Software Engineering Advances ICSEA*, Porto, Portugal, September 2009. IEEE.

[214] M.-R. Shaheen and L. du Bousquet. Is depth of inheritance tree a good cost prediction for branch coverage testing ? In *First International Conference on Advances in System Testing and Validation Lifecycle VALID*, Porto, Portugal, September 2009. IEEE.

[215] M.-R. Shaheen and L. du Bousquet. Survey of source code metrics for evaluating testability of object oriented systems. Research Report RR-LIG-005, LIG, Grenoble, France, 2010.

[216] J.W. Sheppard and M. Kaufman. Formal specification of testability metrics in ieee p1522. *AUTOTESTCON Proceedings, 2001. IEEE Systems Readiness Technology Conference*, pages 71–82, 2001.

[217] A. J.H. Simons, N. Griffiths, and C. Thomson. Feedback-based specification, coding and testing with jwalk. In *Academic & Industrial Conference on Testing : Practice And Research Techniques (TAIC-PART)*, pages 69–73. IEEE, 2008.

[218] B. H. Smith and L. Williams. On guiding the augmentation of an automated test suite via mutation analysis. *Empirical Software Engineering*, 14(3) :341–369, 2009.

[219] I. Sommerville. *Software Engineering (7th Edition) (International Computer Science Series)*. Addison Wesley, May 2004.

[220] G. Sindre T. Stalhane and L. du Bousquet. Comparing safety analysis based on sequence diagrams and textual use cases. In *22nd International Conference on Advanced Information Systems Engineering (CAiSE)*, Hammamet, Tunisia, June 2010. Springer.

[221] K.-C. Tai and Y. Lei. A test generation strategy for pairwise testing. *IEEE Trans. Software Eng.*, 28(1) :109–111, 2002.

[222] G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology (NIST), May 2002.

[223] H. Toth. Abs++ : Assertion based subtyping in c++. *Journal of Object Technology*, 5(6), 2006.

[224] T. Triki. Création d'un langage d'entrée pour l'outil de test combinatoire Tobias. Rapport de M2R, Université J. Fourier, Grenoble, France, 2010.

[225] T. H. Tse, F. C. M. Lau, W. K. Chan, P. C. K. Liu, and C. K. F. Luk. Testing object-oriented industrial software without precise oracles or results. *Communications of the ACM*, 50(8) :78–85, 2007.

[226] J. van den Berg and B. Jacobs. The LOOP Compiler for Java and JML. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *LNCS*. Springer, January 2001.

[227] Jérôme Vassy. *Génération automatique de cas de test guidée par des propriétés de sûreté*. PhD thesis, Université Joseph Fourier, Grenoble, France, October 2004.

[228] J.M. Voas. PIE : A dynamic Failure-Based Technique. *IEEE Transaction on Software Engineering*, 18(8) :41–48, August 1992.

[229] J.M. Voas and K. Miller. Semantic Metrics for Software Testability. *J. Systems Software*, 20 :207–216, 1993.

[230] W3C. Web service activity. `http ://www.w3.org/2002/ws/`.

[231] F. Wang and K. J. Turner. Policy conflicts in home care systems. In *IX International Conference on Feature Interactions in Software and Communication Systems (ICFI)*, pages 54–65, Grenoble, France, 2007. IOS Press.

[232] Z. Wang, B. Xu, and C. Nie. Greedy heuristic algorithms to generate variable strength combinatorial test suite. In Hong Zhu, editor, *Proceedings of the Eighth International Conference on Quality Software, (QSIC'08)*, pages 155–160, Oxford, UK, August 2008. IEEE Computer Society.

[233] A. H. Watson and T. J. McCabe. Structured testing : A testing methodology using the cyclomatic complexity metric. NIST Special Publication 500-235, National Institute of Standards and Technology, August 1996.

[234] M. Weiglhofer and F. Wotawa. Asynchronous input-output conformance testing. In *33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC*, volume 1, pages 154 –159, 20-24 2009.

[235] M. Weiser. Some computer science issues in ubiquitous computing. *Commun. ACM*, 36(7) :74–84, 1993.

[236] E. J. Weyuker. On testing non-testable programs. *Comput. J.*, 25(4) :465–470, 1982.

[237] E. J. Weyuker. Evaluating software complexity measures. *IEEE Trans. Softw. Eng.*, 14(9) :1357–1365, 1988.

[238] L. White and E. Cohen. A Domain Strategy for Computer Program Testing. *IEEE Transactions on Software Engineering*, pages 247–257, May 1980.

[239] A.W. Williams and R.L. Probert. A measure for component interaction test coverage. In *Computer Systems and Applications, ACS/IEEE International Conference on. 2001*, pages 304–311, 2001.

[240] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler : automatic generation of path tests by combining static and dynamic analysis. In *Proc. European Dependable Computing Conference*, volume 3463 of *LNCS*, pages 281–292. Springer, 2005.

[241] M. Wilson, M. Kolberg, and E. H. Magill. Considering side effects in service interactions in home automation - an online approach. In *IX International Conference on Feature Interactions in Software and Communication Systems (ICFI)*, pages 172–187, Grenoble, France, 2007. IOS Press.

[242] W. E. Wong. *Mutation Testing for the New Century*. Kluwer Academic Publishers, June 2001.

[243] B. Yan, M. Nakamura, L. du Bousquet, and Ken ichi Matsumoto. Characterizing safety of integrated services in home network system. In *5th International Conference On Smart Homes and Health Telematics (ICOST)*, volume 4541 of *Lecture Notes in Computer Science*, pages 130–140, Nara, Japan, June 2007. Springer.

[244] B. Yan, M. Nakamura, L. du Bousquet, and K.-I. Matsumoto. Validating safety for integrated services of home network system using jml. *Journal of Information Processing (IPSJ)*, 16 :38–49, 2008.

[245] Qian Yang, J. Jenny Li, and David M. Weiss. A survey of coverage-based testing tools. *Comput. J.*, 52(5) :589–597, 2009.

[246] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4) :366–427, 1997.

[247] N. Zuanon. *Test de spécifications de services de télécommunication*. PhD thesis, Université Joseph Fourier, Grenoble, France, Juin 2000.