DEA Systèmes d'Information
Diplôme Européen de III^{ème} cycle en Systèmes d'Information

Stage effectué au laboratoire
Logiciels Systèmes et Réseaux
IMAG

Test de conformité des programmes JAVA

Projet réalisé par :
Mehdi KESSIS
Mehdi.kessis@imag.fr

Sous la direction de :
Yves LEDRU
Olivier MAURY

Année Universitaire
2002-2003

Table des matières

Table des illustrations

Remerciement

Tout d'abord je voudrais adresser ma profonde reconnaissance à Monsieur Yves Ledru, Professeur à l'Université Joseph Fourier qui m'a accueilli dans son équipe de recherche et a accepté d'être mon directeur de recherche. Ses idées, son savoir-faire et ses précieux conseils ont énormément contribué à ce travail de recherche.

Je tiens à remercier également tous les membres de l'équipe VASCO, et tout particulièrement Olivier Maury, Pierre Bontron, Catherine Oriat, Lydie du Bousquet qui m'ont fait l'honneur de participer, par leurs efforts et leur temps, à l'élaboration de ce travail.

Je remercie ma famille pour le soutien moral et l'encouragement constant qu'elle m'a prodigué durant cette première année. Je suis très reconnaissant à tous ceux qui m'ont encouragé et aidé dans ce travail.

Mehdi

Introduction

Contexte et motivation

Le logiciel joue un rôle de plus en plus important pour les entreprises impliquées dans les *technologies de l'information*. Nous comptons de plus en plus sur des systèmes contrôlés par des logiciels pour obtenir l'information à partir de laquelle nous prenons nos décisions qu'elles soient personnelles, commerciales ou administratives.

L'omniprésence des logiciels, leur complexification croissante et leurs interconnexions toujours plus nombreuses rendent les entreprises chaque jour davantage tributaires de ces systèmes. Avec tous les bienfaits qu'apportent ces systèmes sont également apparus les méfaits de leur défaillance.

On estime à cinq mille milliards de dollars le coût du bogue de l'an 2000 [Xanthakis00]. Cinq millions d'appels téléphoniques ont été bloqués aux Etats Unis suite à une panne logicielle qui a paralysé le réseau de la société AT&T [Xanthakis00]. Plus grave encore est l'émission de gaz toxique par une usine chimique suite à la manifestation d'une situation non prévue dans la conception de son système de contrôle [Norris95]. Les exemples de mauvais fonctionnement de programmes ne manquent pas et s'ajoutent jour après jour à une longue série d'incidents. La criticité de certains systèmes rendent ces défaillances intolérables et mettent l'accent sur le besoin de leur validation [Norris95].

On assiste donc à un intérêt grandissant pour les méthodes de *validation* et de *vérification* permettant d'améliorer la qualité de ces systèmes logiciels. « La validation est un moyen de confirmer le respect d'exigences déterminées pour une utilisation spécifique prévue » [Watkins02]. Les techniques utilisées pour cette activité sont nombreuses. Ce sont pour la plupart des techniques dites fonctionnelles ou de conformité [Xanthakis00] car elles consistent à comparer le comportement d'une réalisation logicielle par rapport à sa spécification.

Jusqu'à une période récente, les outils pour la validation logicielle n'étaient pas sur le plan de la productivité à la hauteur des outils de codage ou de conception. En effet, les générateurs d'applications et les générateurs d'interfaces homme-machines, par exemple, permettent aux développeurs de créer des programmes en moins de temps qu'i n'en faut pour les ingénieurs de test pour les tester et effectuer leur mise au point. Il en résultait un goulot d'étranglement dans le développement des applications.

Le présent travail s'intéresse au test de conformité des logiciels et se propose d'étudier l'automatisation de cette procédure.

L'automatisation des tests a pour objectif principal de remédier à cette difficulté. Elle permet aussi d'améliorer l'efficacité, la précision et la reproductibilité des test [Norris95]. Les outils de test peuvent aboutir à une économie globale de 9 à 18% pour le développement de systèmes [Norris95]. Ceci justifie en grande partie l'intérêt particulier accordé par les chercheurs et les industriels à l'automatisation des tests.

Objectif du Travail

Le présent travail a été réalisé au sein de l'équipe VASCO(spécification, validation et tests de logiciels) au laboratoire Logiciels, Systèmes et Réseaux. L'équipe VASCO s'intéresse, entre autres, depuis quelques années au problème l'automatisation des tests et a récemment participé au projet RNTL COTE consacré à cette problématique.

A cette occasion elle a mis en place un outil de synthèse de cas de test intitulé TOBIAS. Cet outil permet la génération automatique d'un grand nombre de jeux de tests en dépliant systématiquement des patrons de génération. Plusieurs études de cas, dont une menée avec la société Gemplus ont montré les gains de productivité apporté par TOBIAS dans la production de jeux de tests. Elles ont également montré la capacité de l'outil à détecter des erreurs.

Toutefois, le dépliage exhaustif des patrons de génération produit un trop grand nombre de tests pour pouvoir les exécuter dans un temps raisonnable. Il en résulte dans la plupart des cas une explosion combinatoire des tests.

Cette étude vise à pallier cette limite pour les programmes JAVA. L'intérêt porté aux applications JAVA s'accroît, notamment du fait quelles font l'objet d'un marché émergent et prometteur de technologies serveur, en particulier avec la technologie JavaBeans[1]. Une étude

[1] http://java.sun.com/products/javabeans

récente prévoit une croissance des ventes des applications JAVA, au niveau du marché européen, dont le montant devrait atteindre deux milliards de dollars en 2005[2].

Une piste particulièrement prometteuse consiste à associer TOBIAS à JML (Java Modeling Language). JML est un langage de spécification à base d'assertions (invariants, pré- et postconditions) qui contraignent les opérations d'une classe. Le caractère exécutable de ces assertions permet de fournir des « oracles » de test dont le rôle est de juger la conformité d'une implémentation par rapport à sa spécification. Les spécifiassions JML pourraient également servir à filtrer les tests générés par TOBIAS lors de leur exécution. Le but de ce travail serait alors d'explorer cette piste et répondre à la question suivante :

Dans le cadre du test de conformité, la question est de savoir comment l'utilisation de JML pourrait réduire l'explosion combinatoire des tests de conformités produits par TOBIAS ?

La réponse à cette question sera fournie dans ce mémoire en deux grandes parties :

Une première partie, relative à l'état de l'art, servira à l'introduction des deux éléments de base de ce travail. Le premier chapitre sera consacré à la présentation de l'outil TOBIAS. Ce chapitre tentera de mettre en avant l'importance des patrons de génération ou schéma de test dans la production d'un grand nombre de tests. Il permettra, ensuite, d'exposer les limites inhérentes à cet outil. Le deuxième chapitre introduira le langage JML. Il mettra en exergue l'importance de l'aspect exécutable des assertions JML pour fournir des oracles de test.

L'approche proposée et sa mise en œuvre feront l'objet de la deuxième partie de ce mémoire. Le troisième chapitre exposera l'approche proposée pour le filtrage à l'exécution des jeux de tests produits par TOBIAS grâce au caractère exécutable des assertions JML. Les expérimentations feront l'objet d'un quatrième et dernier chapitre. L'étude de cas d'une application de services bancaires y sera effectuée. Les limites et les futures voies de recherches seront discutées dans la conclusion.

[2] Etude de marché européen des logiciels Java (Rapport 3702) : étude effectuée par Frost & Sullivan est un cabinet international d'études de marché et de conseil dont l'expertise couvre un large spectre d'activités high-tech, y compris le secteur informatique : études de marchés, évaluations, stratégies, etc.

Partie 1

Etat de l’art

Chapitre1

TOBIAS :Environnement de génération de cas de test

Nous présentons dans ce chapitre l'outil TOBIAS [PMBY02,Ledru02], un outil de synthèse de cas de test, développé au laboratoire LSR-IMAG. Nous commençons, par une définition de la terminologie à la quelle nous allons nous référer tout au long du chapitre. Ensuite, nous procédons à l'explication de la notion de schéma de test, tout en soulignant l'importance des niveaux d'abstraction, qu'elle offre, pour la synthèse des cas de test. Enfin, nous exposons les problèmes liés à l'utilisation de cet outil.

1.1 TOBIAS : environnement de génération de cas de tests

1.1.1 Présentation de l'outil

TOBIAS (Test Objective desIgn Assistant) est un outil de synthèse de cas de test qui a été développé par l'équipe VASCO du laboratoire LSR-IMAG, dans le cadre du projet RNTL COTE[3]. Il permet d'automatiser la génération des tests, à partir des diagrammes de classes UML et de certains patrons de test (schéma de test), qui lui sont fournis en entrée.

[3] http://www.industrie.gouv.fr/rntl/FichesA/Cote.htm

Cet outil a été conçu dans le but de fournir une aide aux ingénieurs de test afin d'accélérer l'activité de génération de test, jugée assez souvent répétitive et fastidieuse. Son utilisation a occasionné dans pas mal d'occasions des gains de productivité considérables en terme de temps et de coût de génération de test. De tels gains sont particulièrement recherchés par les industriels, vu que le coût très élevé de l'activité de test, souvent évalué entre 40% et 70 % du coût total d'un projet de développement d'un logiciel [Norris95].

Outre la génération d'un grand nombre de cas de test, TOBIAS peut générer de façon combinatoire plusieurs scénarios possibles de test. Ca faisant, TOBIAS peut révéler certaines défaillances du système difficiles à déceler dans le cadre d'un test manuel

A l'heure actuelle, TOBIAS génère des cas de test pour le framework de test JUNIT[4] et l'atelier VDM++ ainsi que des objectifs de test pour l'outil UMLAUT/TGV.

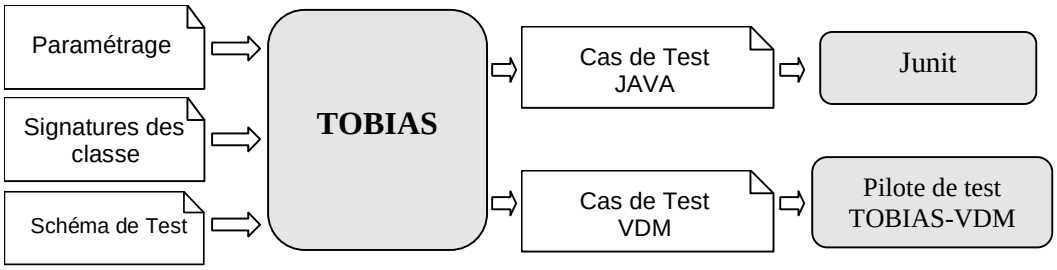


Fig. 1 TOBIAS : environnement de génération de cas de test

1.1.2 Terminologie

Certains termes seront souvent évoqués dans ce chapitre. Nous tenons à les définir de façon précise, dès le départ afin d'éviter tout risque de confusion ou d'ambiguïté.

Définition 1: (Un cas de test/séquence de test) : *Un cas de test est une succession d'appel de méthodes (m_1,m_2,..m_n). Chaque appel est formé d'un couple* $(o, \vec{p})$, *avec o l'objet recevant de l'appel et* $\vec{p}$ *le vecteur contenant les valeurs passées en paramètre à la méthode m.*

[4] http://www.junit.org/index.htm

Définition 2 : **(Un schéma de test) :** *Un schéma de test est une abstraction des séquences de test écrite sous une forme proche des expressions régulières* [PMBY02].

Définition 3 : (Une campagne de test) : *Une campagne de test correspond à la création d'une nouvelle configuration de test* [PMBY02].

1.2 Les schémas de test TOBIAS

Lors de l'élaboration des campagnes de test, l'écriture d'un nombre considérable de cas de test devient nécessaire. Des expérimentations industrielles [Bousuqet01] ont montré que les séquences produites sont loin d'être indépendantes les unes des autres. Elles possèdent souvent des caractéristiques communes.

En effet, certaines d'entre elles se partagent des sous-séquences entières, d'autres contiennent le même nombre d'appels de certaines méthodes. Le schéma de test, est un mécanisme qui permet de factoriser ces caractéristiques communes, et de réduire par conséquent l'effort et le coût de génération des cas de test.

Exemple : Afin d'illustrer l'utilité des schémas de test nous proposons l'exemple de gestion de comptes bancaires. Le diagramme correspondant est présenté dans la Fig.2**.** Ce diagramme de classes décrit une classe Banque composée d'un ensemble de clients et d'un ensemble de comptes. Chaque client de la banque possède un ou plusieurs comptes.

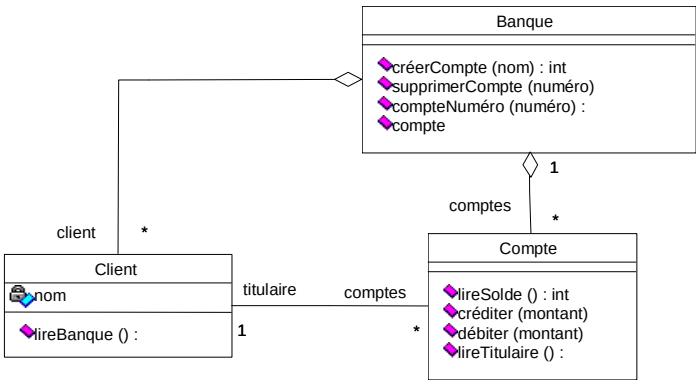


Fig. 2 Diagramme de classe Gestion des Comptes bancaires

Le schéma de test Sh_1 a été crée à partir de ce diagramme de classes. Il se présente comme suit :

Sh_1 := Débiter(m_1)$^{\wedge 1..2}$;Créditer(m_2)$^{\wedge 0..3}$
$m_1 \in \{ 100, 200, -500 \}$
$m_2 \in \{ 1000, 50 \}$

Ce schéma est équivalent à dire que nous voulons effectuer 1 ou 2 appels de la méthode *débiter* avec comme valeur du paramètre m_1 l'un des éléments de l'ensemble {100, 200, -500}, suivi de 0 ou 3 appels de la méthode *créditer* avec comme valeur du paramètre m_2 l'un des éléments de l'ensemble{1000, 50}.

1.2.1 Les niveaux d'abstraction des schémas de test

Les schémas de test intègrent différents niveaux d'abstractions qui constituent des points de repères sur lesquels se base TOBIAS pour générer les différents cas de test. Ces abstractions peuvent être faites sur :

- Les valeurs des paramètres ;
- Les instances ;
- Le nombre des appels ;
- Les groupes de méthodes.

1.2.1.a Abstraction sur les valeurs

Très souvent, dans les campagnes de test, on retrouve des cas de test composés des mêmes méthodes mais avec des valeurs de paramètres différentes. TOBIAS part de ce constat pour offrir la possibilité de factoriser, à travers les schémas de test, l'ensemble des méthodes invoquées avec différentes valeurs de paramètres. C'est que nous désignons par l'abstraction sur les valeurs des paramètres.

Exemple : Considérons les 2 diagrammes de séquences suivants :

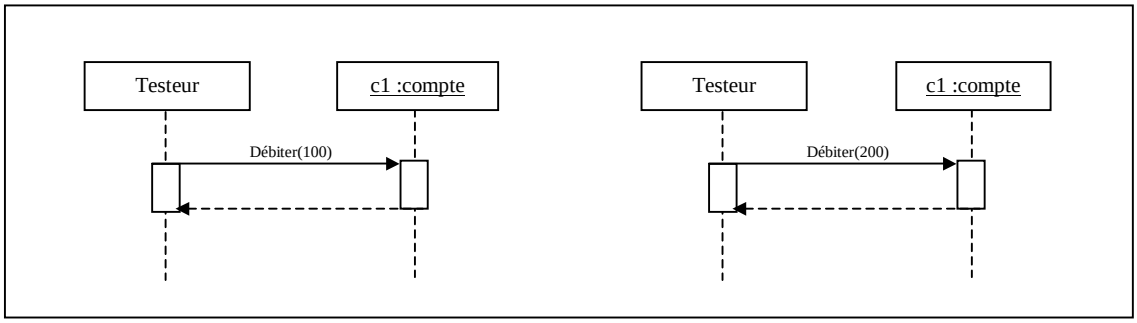


Fig. 3 Cas de test qui diffèrent sur les valeurs

La figure 4 illustre grâce à deux diagrammes de séquences UML, deux cas de test constitués chacun d'un appel de méthode *débiter*. Seule la valeur du paramètre d'appel varie dans les deux cas.

Le schéma de test Sh_2 permet de factoriser les caractéristiques communes de ces deux cas de test. Il fait abstraction sur la valeur du paramètre d'appel de la méthode *débiter(m).*

Sh_2:= débiter(m_1)
$m_1 \in \{ 100, 200 \}$

1.2.1.b Abstraction sur les instances

Parfois, durant les campagnes de test, on veut appliquer les mêmes tests sur plusieurs instances d'une même classe. Le schéma de test nous permet alors de réduire le temps d'écriture des tests en factorisant les cas de test sur les instances invoquées. C'est ce que nous désignons par l'abstraction sur les instances.

Exemple : : considérons les 2 diagrammes de séquences suivants

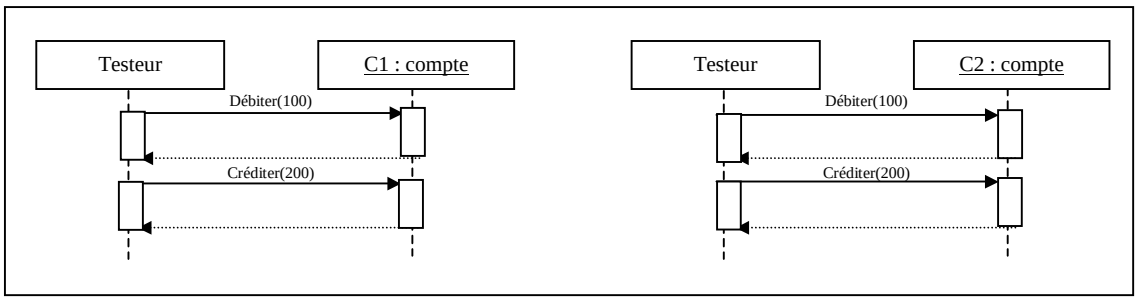


Fig. 4 Cas de test qui diffèrent sur les instances

Le testeur désire exécuter la même séquence sur deux instances de la classe compte C_1 et C_2 de la classe compte. Les deux séquences de test qu'il utilise pour effectuer cette opération peuvent être représentées par le schéma de test Sh_3 suivant :

Sh_3 := *.Débiter(m_1) ; *.Créditer(m_2)
$m_1 \in \{ 100\}$
$m_2 \in \{ 200\}$

Le schéma Sh_3 signifie qu'on peut exécuter la méthode *Débiter(m_1)* avec la valeur de paramètre100 une instance quelconque * puis la méthode *Créditer(m_2)* avec comme

valeur de paramètre 200 sur n'importe quel instance, définie au préalable dans la campagne de test.

1.2.1.c Abstraction sur le nombre des appels

Nombreux sont les tests où des séquences d'appel se répètent un nombre variable de fois. Les schémas de test permettent de faire une abstraction sur le nombre des appels en définissant deux bornes limites : une borne inférieur et une autre supérieur.

Exemple : considérons le diagramme de séquence suivant

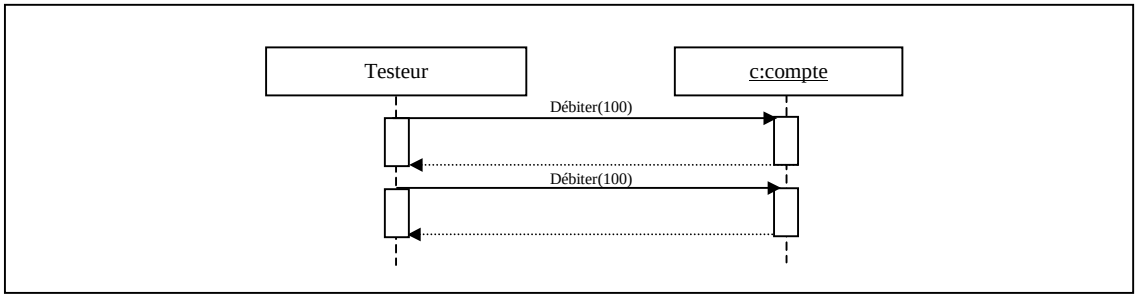


Fig. 5 Cas de test avec des boucles sur les appels de méthodes

Dans ce cas nous pouvons écrire des schémas de test de la façon suivante :

$$Sh_4 := \text{Débiter (m)}^{\wedge 1..2}$$
$$m \in \{100\}$$

D'après ce schéma nous pouvons effectuer au minimum un et au maximum deux appels de la méthode *Débiter(m).*

1.2.1.d Abstraction sur « les groupes »

Le groupe est un mécanisme de groupement des méthodes offert par Tobias pour faire abstraction sur un ensemble de méthodes à tester. Le critère de groupement est choisi par le testeur. Ce dernier essaye souvent de mettre ensemble des méthodes susceptibles d'être appelées au même moment ou des méthodes de même type (exemple : créditer( ) , débiter( )), c'est à dire qui modifient le même état.

Exemple :Supposons que nous voulons tester le schéma de test Sh_1 suivant :

Schéma Sh_1 sans la notion de groupe :

$$Sh_5 : = I_2.\text{CréerCompte}(); I_2.\text{ Débiter } (m)^{\wedge 1..2} ; I_2.\text{Créditer } (m)^{\wedge 1..2}$$
$$m \in \{100,-500,1000\}$$

Schéma Sh_5 avec la notion de groupe:

$Sh_{5'}$: = I_2.CréerCompte(); I_2. $G_1^{\wedge 1..2}$

G1 ={ Débiter(*m*), Créditer(*m*) }

$m \in$ {100,-500,1000}

Ainsi, il s'agit de regrouper pour la gestion des opérations sur les comptes les méthodes créerCompt(m) et supprimerCompte(n) dans un même groupeG1

1.3 Le principe de Fonctionnement

L'outil TOBIAS opère sur deux phases : Une phase de paramétrage et une phase de dépliage.

1.3.1 Phase de Paramétrage

La première étape de cette phase consiste à définir les instances à tester, à sélectionner les différentes méthodes concernées et à les grouper. Le testeur peut par la suite introduire les valeurs des paramètres pour chacune des méthodes.

Une fois ces étapes de préparation sont achevées, l'utilisateur peut commencer à définir les schémas de test. Il peut imaginer et créer plusieurs scénarios de test en se basant sur les différentes possibilités d'abstractions dont il dispose.

1.3.2 Phase de dépliage des schémas de test

Au cours de cette phase, TOBIAS procède par itérations sur les différentes abstractions des schémas de test afin de générer tous les cas de test correspondants au schéma de test [PMBY02]. Cette génération se fait à travers le dépilage systématique des schémas de test, suivant les différents niveaux d'abstraction selon lesquels ils ont été définis à savoir les valeurs, les groupes, le nombre des appels et les instances.

Exemple : Pour bien illustrer le principe de dépliage des patrons de test, nous allons nous baser sur le schéma de test suivant :

Sh_6:=*.$G_1^{\wedge 0..3}$
G_1={Débiter(m_1),Créditer(m_2)}
$m_1 \in$ {100,-100,0}
$m_2 \in$ {200,10}
I ={ I_1,I_2 }

Le dépliage du schéma de test Sh_6 est effectué la façon suivante : tout d'abord les schémas de test sont dépliés selon le nombre des appels. Ce faisant, zéro ou trois seront effectués pour du groupe G_1. Ensuite, chaque groupe est remplacé par l'ensemble de méthodes qu'il contient. Dans notre schéma, TOBAIS va remplacer le groupe G_1 par l'appel des deux méthodes *Débiter(m_1)* et *Créditer(m_2).* Une fois les groupes éclatés, les différentes méthodes seront associées à autant d'instances que possibles(*). Les méthodes *Débiter(m_1)* et *Créditer(m_2)* seront associées aux deux instances I_1 et I_2. La dernière étape consiste à instancier les différentes méthodes avec les valeurs de paramètres qui ont été spécifiées par le testeur au début de la campagne. La méthode *Débiter(m_1)* sera remplacée par *débiter(100), débiter(-100), débiter(0)* et la méthode *Créditer(m_2)* sera remplacée *par Créditer(200), Créditer(10).* A la fin de cette opération de dépliage, nous aurons comme résultat 1111 séquences de test.

$$\text{AN : } Sh_5 := [\ (3\times2+2\times2) + (3\times2+2\times2)^2 + (3\times2+2\times2)^3 + 1\ ]$$
$$= 1111 \text{ cas de test}$$

1.4 Les limites de l'outil

1.4.1 Le problème de l'explosion combinatoire des cas de test

Certes le dépliage systématique des schémas de test par TOBIAS occasionne des gains considérables en terme de temps et de coût d'écriture des cas de test. La grande masse des jeux de données de test fournie en sortie permet de satisfaire des besoins de test pour des systèmes particuliers tels que les *systèmes critiques* et des applications telles que les SGBD. Toutefois, au cours de cette opération, le nombre des cas de test grimpe de façon prohibitive et on se trouve rapidement face à une avalanche de séquences des tests. C'est ce que nous désignons par l'explosion combinatoire des cas de test.

En analysant les cas de test générés, nous avons constaté que dans pas mal de cas elles regroupent un nombre important jeux de tests pertinents c'est à dire qui respectent la spécification de l'application à tester. D'où la nécessité de filtrer les résultats et d'éliminer ces séquences invalides afin de présenter au testeur un nombre optimal de cas de test.

De tels cas peuvent être apprécier dans le cadre d'une campagne de test de *robustesse* dans laquelle l'application sera mise en sollicitations quelconques, y compris erronées ou inopportunes.

Toutefois, le présent travail se situe dans un cadre de test de conformité dans le quel les cas de test doivent être générés conformément à la spécification de l'application testée. Elles ne doivent pas contenir des appels à des méthodes en dehors de leurs spécifications.

Exemple : considérons le schéma de test suivant :

Sh_7:= débiter(m_1)$^{\wedge 1..2}$; créditer(m_2)$^{\wedge 0..3}$
$m_1 \in$ { 100, 200, -100 }
m_2 { 100, -50 }

Le dépliage de ce schéma par l'outil TOBIAS génère 180 séquences. Parmi ces cas de test on trouve les séquences suivantes :

- débiter(-100)
- débiter(-100); débiter(-100)
- débiter(-100); créditer(100); créditer(-50) ;
- débiter(100) ; créditer(-50)
- débiter(200); créditer(-50)

….

Le cas de la non-conformité des cas de test peut se présenter dans le cas où la spécification de la classe *Compte* précise que le montant à débiter doit être toujours positif. Si cette condition avait été définie, tous ces cas de test seront rejetés car ils seront considérés comme non conforme par rapport à la spécification de la classe testée.

En effet sans filtrage de ces cas, il sera insensé de tester la validité d'un cas non conforme à la spécification. Ceci étant, résoudre ce problème, même en partie, revient à améliorer la qualité et optimiser le nombre des cas de test produits et, par conséquent, doter TOBIAS d'une capacité supplémentaire de génération de cas de test en libérant l'espace occupé par les tests non- pertinents.

1.4.2 L'absence d'un pilote de test

L'automatisation du test est un processus qui passe par trois étapes :

1) L'automatisation de la génération des données de test;
2) L'automatisation de l'exécution des données de test;
3) L'automatisation de la génération de l'oracle du test.

L'outil TOBIAS permet d'automatiser toutes ces étapes pour des programmes spécifiés en VDM. Un pilote de test implémenté en TCL lui permet de jouer les cas de test et de récupérer le résultat de leurs exécutions. Ce pilote n'est applicable qu'aux programmes spécifiés en VDM.

Pour pouvoir tester les programmes JAVA, TOBIAS fait recours au framework de test JML-JUNIT. Ce dernier permet d'exécuter les différents cas de test générés et de les statuer en validant leur conformité par rapport aux spécifications. Pour ce faire, Junit requière l'organisation des cas de test sous formes de classes qui dérivent de sa classe centrale (TestCase). L'exécution du test, est déclenché par l'invocation des constructeurs de ces classes avec le nom du test. Les tests proprement dits, sont écrits sous forme de méthodes de cette classe, à l'intérieur desquelles des tests sont principalement réalisés à l'aide des méthodes tel que *assert, assert Equals,* etc. Pour chaque cas de test, des méthodes particulières (SetUp et TearDown) sont appelées, respectivement avant et après l'exécution de chacune de ces méthodes afin d'initialiser et de libérer les ressources partagées par les méthodes de test. Une fois la classe de test est écrite, JML-JUNIT offre la possibilité de regrouper toutes les méthodes à tester dans une suite de test afin d'automatiser leur exécution.

LA principale lacune de JML-JUNIT consiste dans son approche de l'exécution des tests qui ressemble plutôt à une exécution en lot des cas de test. Une telle approche s'avère très coûteuse quand on l'applique sur de cas de test générés par TOBIAS. Tous ces aspects seront détaillés d'avantage dans le chapitre 4.

1.4.3 Le problème de l'oracle

La troisième étape importante après la génération des cas de test et leur exécution, consiste à fournir un oracle pour chaque entrée et sa sortie correspondante. Cette information est indispensable lors de l'évaluation des résultats de l'exécution de tests de manière à pouvoir statuer sur la validité du comportement d'un système vis-à-vis d'un document de référence(cahier de charge, spécification) [Jard01]. D'après [Ham96], on ne peut pas parler de test rigoureux sans l'oracle de test. C'est à lui de prononcer un verdict relatif au test. Ce verdict peut être :

- **Succès** (PASS): le programme satisfait la spécification ;
- **Echec** (FAIL): détection d'une non conformité entre la spécification et le code ;

- **Non-conclusif** (NON-CONC) : aucune décision ne peut être prise concernant la conformité du programme par rapport à sa spécification; la méthode est appelée en dehors de son domaine défini au niveau de sa spécification.

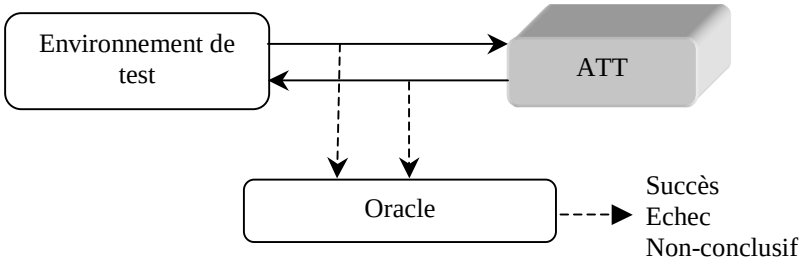


Fig. 6 L'oracle de test

La Figure 6 illustre le rôle de l'oracle qui consiste à vérifier si les sorties produites à l'issue de l'exécution d'une application à tester (ATT) avec des valeurs particulières d'entrée, coïncident avec la spécification de l'application à tester.

Certes, l'automatisation de cette opération contribue à l'amélioration de la qualité du test et contribue à la réduction de son coût. Toutefois, elle est souvent difficile a mettre en oeuvre[Gau95]. Cette difficulté d'interprétation est souvent désigné dans la littérature par le *« problème de l'oracle »*. La difficulté d'aborder ce sujet vient d'abord de l'absence dans la littérature d'une technique générale pour résoudre le problème de l'oracle. La plupart des travaux sur le test se sont intéressés aux techniques de résolution des problèmes relatifs à la génération des tests, leur sélection et aux techniques et méthodes de test. Par contre ils ont fort peu abordé le problème de l'oracle [Ham95]. Typiquement, les théoriciens supposent qu'un oracle existe. Ils parlent alors d'échec et succès sans préciser la façon de l'obtenir.

Traditionnellement, l'oracle n'est personne d'autre que l'ingénieur de test et les résultats de attendu sont calculés «manuellement» [BARESI01]. Avec cette technique, les résultats ainsi obtenus peuvent eux-mêmes être erronés et l'absence d'automatisation rend le processus de test extrêmement coûteux [Ham95,Ham96,Den98,BARESI01]. Dans plusieurs cas, le calcul des résultats attendus est difficile à obtenir [Pat00] ; nous citons, par exemple, le cas où les propriétés du résultat sont connues mais pas sa valeur. De tels cas de figure sont assez répondus dans les cas du calcul numérique [Gau95,PETERS98]. Certains cas sont carrément indécidables ; c'est le cas, par

exemple, la mesure de l'équivalence entre le code source et le programme objet lors du test d'un compilateur [Gau95].

Une autre technique, appelée test dos à dos (*back to back testing*), repose sur le développement indépendant de plusieurs versions d'une même application. Elle n'est guère utilisée car elle cumule les inconvénients : coût élevé, grande incertitude sur la pertinence des conclusions [Jard01].

La méthode des tables de décision offre aussi une réponse à la définition d'un oracle parce qu'elle précise les effets qui doivent être constatés dans les cas définis par les combinaisons de conditions. Néanmoins, l'oracle demeure partiel cependant car si plusieurs actions sont engendrées par une combinaison de conditions, aucune information n'est fournie concernant leur ordonnancement.

D'autre part, certains auteurs pensent que pour obtenir un oracle, il est primordial de disposer des propriétés que doivent satisfaire les sorties ainsi que leurs dépendances vis-à-vis des entrées. Ces informations peuvent être fournies par un document de référence. Les spécifications formelles en sont un exemple [Jard01]. Plusieurs approches peuvent être adopter selon la nature des spécifications (abstraites ou exécutables) ou selon leurs degrés de formalisme (en langage naturel, semi-formel, formel). Nous proposons d'étudier certaines d'entre ces approches.

La première approche se base sur la documentation des modules qui est exprimée sous forme d'expressions tabulaires pour générer l'oracle de test. D. Peters and D. L. Parnas [Den98] ont adopté cette approche en se basant sur des spécifications algébriques et les expressions tabulaires multidimensionnelles. Bien qu'elle a montré son efficacité, cette technique nécessite l'accès aux structures de données. Ceci n'est pas toujours possible et il ne fait pas partie du présent travail qui se base sur la technique *de test boite noire*. [Pat00] et [Den98] préconisent la génération des oracles de test à partir de la documentation si cette dernière est présenté de façon formelle(des spécification algébriques). Mieux encore, [Pat00] valides ces dires par l'implémentation d'un prototype en se basant sur les spécifications algébriques pour générer de façon automatique les oracles de test. [Bernot91] a proposé, aussi, de dériver les oracles à partir de spécifications.

Ainsi, les spécifications formelles fournissent une réponse à la définition de l'oracle en caractérisant les sorties correctes à l'aide de propriétés et de relations liant les sorties aux entrées. Certaines d'entre elles, sont exécutables et permettent la réalisation d'un prototype du programme spécifié, ce qui constitue un oracle. Toutefois, les langages formels sont souvent rejetés par les programmeurs à cause de leur notation souvent jugée trop ardue. Par ailleurs, Hamlet [Ham95] trouve que ce type de spécifications (abstraites) accroît d'avantage la complexité de l'oracle vu que les résultats produits ne sont pas parfaitement équivalent aux résultats spécifiés. Le même problème se pose pour le test fonctionnel (*boite noire*), où le test est basé sur les spécifications, qui sont, assez souvent, plus abstraites que le programme [Ham95].

Récemment, [Cheon02b] a proposé une approche de génération automatique d'oracles de test sur la base de spécifications formelles des interfaces et d'un vérificateur ces spécifications à l'exécution. Le langage de spécification utilisé(JML) pallie les problèmes rencontrés avec le reste des langages formels notamment la difficultés de la notation. Par ailleurs, l'approche proposée, ne se base sur aucun calcul préliminaire des résultats attendus. Il s'agit plutôt d'exécuter des programmes instrumentés et de vérifier la conformité de leur comportement par rapport à la spécification.

1.5 Conclusion

Le but de ce chapitre été d'étudier l'outil de test combinatoire TOBIAS. Deux objectifs ont été visés à travers cette présentation :

- La mise en exergue de la capacité de l'outil à automatiser la génération d'un grand nombre de cas de test à partir des spécifications semi-formelles notamment les diagrammes de classes en UML et des schémas de test;
- La mise en évidence de la difficulté d'atténuer le nombre des cas de test générés et surtout la difficulté de les statuer en jugeant leur validité par rapport à leurs spécifications.

Nous avons conclu le chapitre par une présentation des différentes techniques de génération des oracles de test. L'examen de ces techniques nous a orientés vers une technique très récente qui se base sur les spécifications formelles exécutables en JML pour générer les oracles de test. Nous proposons alors d'explorer cette approche dans le prochain chapitre.

Chapitre 2

JML et les spécifications exécutables

"My ultimate research objective is to help programmers write better programs"

Gary Leavens

Dans ce chapitre, nous présentons le langage de spécification formel des interfaces Java Modeling Langage (JML). Notre objectif n'est pas de faire un exposé exhaustif des capacités de ce langage. IL s'agit plutôt de mettre en évidence l'originalité et la puissance de l'approche de spécification à base d'assertions qu'il propose. Une attention particulière sera accordée à son évaluateur des assertions à l'exécution vue l'intérêt qu'il présente pour le test des programmes JAVA en général et la génération automatique des oracles de test en particulier.

2.1 Présentation du langage

Java Modeling Language(JML), est un langage de spécification formel des interfaces et du comportement des classes JAVA [Leavensb03]. Conçu au début, à l'université de Iowa State par *Gary Leavens*, pour des fins recherche, ce langage a pu rapidement susciter l'intérêt des industriels grâce à sa simplicité et sa puissance. Il fut alors l'objet d'un projet ouvert regroupant des industriels (Compac et Gemplus) et des laboratoires de recherches (MIT, INRIA, etc).

L'idée de base de JML est de permettre l'écriture de spécifications à base de contrats spécifiés en terme de conditions booléennes au niveau des classes (*invariants* et *contraintes historiques*), et des méthodes (*préconditions*, *postconditions et assertions*) afin de décrire deux aspects des modules JAVA :

- **Les interfaces** : l'interface d'une classe correspond à sa signature. Elle est constituée par son nom, les noms et types de ces attributs, et ses méthodes ;
- **Le comportement** :C'est la façon dont un module doit réagir quand il est interpellé. Décrire le comportement d'une méthode revient à décrire les transformations d'états qu'elle peut subir [Leavensb03].

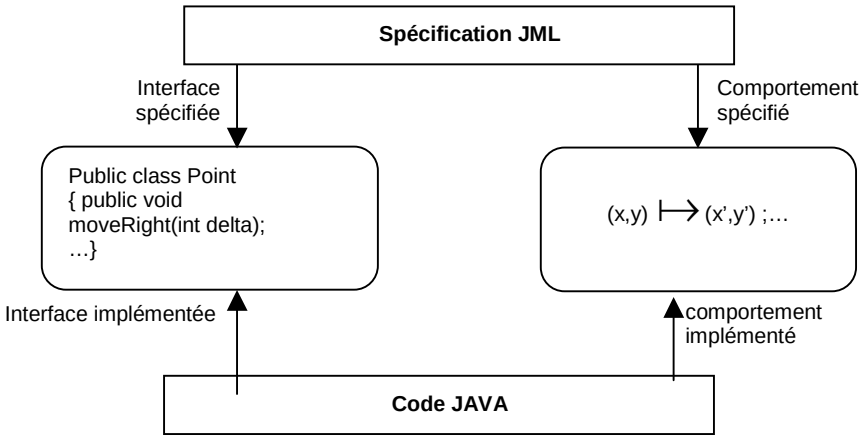


Fig. 7 JML: spécification de l'interface et du comportement

Une telle approche de spécification des modules JAVA présente plusieurs avantages. En effet elle permet de conférer à la spécification:

- **Plus de Lisibilité des spécifications** : Ceci facilite la lecture et la compréhension de la spécification pour les programmeurs. Elle rend plus aisées, par conséquent leur utilisation et leur maintenance;

- **Une indépendance vis-à-vis du code** : JML nous offre la possibilité de séparer les spécifications du code. Des variables dites abstraites ou de modèle, spécifiques à la spécification et qui ne seront pas implémentées, sont utilisées afin de lier la spécification au code correspondant. L'indépendance vis-à-vis du code permet de réutiliser la spécification pour différente implantation et de minimiser, par conséquent, le coût et l'effort de sa maintenance et sa mise à jour pour chaque version du programme ;

- **Les bienfaits du raisonnement formel avec une notation simplifiée :** JML intègre des notions et des idées issues des langages de spécification orientées modèles, tels que VDM ou Z, tout en remédiant aux problèmes issus de la

complexité de leurs notations. Il permet ainsi de considérer, le système comme un modèle mathématique et lui applique par conséquent, le raisonnement formel nécessaire via des opérations ensemblistes par exemple. Avec JML, la complexité de la notation mathématique relative à ce type de raisonnement est cachée derrière une façade des expressions JAVA [Leavensb03].

2.2 Les assertions JML

Définition :(Assertion) "An assertion is a Boolean expression that must be satisfied for associated code to execute properly »[Hoare69].

Les assertions est un moyen permettent d'expliciter les hypothèses formulées par les programmeurs lorsqu'ils élaborent des composants logiciels qu'ils supposent corrects. JML permet d'intégrer des assertions de type invariant ou pré-postconditions au niveau du code Java.

Ces assertions sont décrites sous forme de commentaires particuliers. En effet, ils sont précédés par //@ ou bien compris entre /*@ et *@/. Elles sont ignorées par le compilateur JAVA et c'est le compilateur de JML qui s'en charge. Grâce à ces assertions, JML décrit les deux aspects des classes JAVA : les interfaces des classes et les méthodes.

2.2.1 Spécification des interfaces et des classes

La spécification de l'interface permet de décrire les différents détails de l'interface notamment les paramètres et la visibilité des méthodes et des paramètres. Il fournit ainsi des spécifications plus précises, décrivant les modules dans leurs moindres détails. Ce qui n'est pas possible avec des langages de spécification tels que VDM ou Z qui n'ont pas été conçu pour JAVA. En effet, JML peut spécifier, entre autres, les conditions précises sous lesquelles certaines exceptions peuvent être levées. De telles spécifications sont plus difficile à réaliser avec des langages de spécification qui ne sont pas conçus pour JAVA et qui ne supportent pas la notion d'exception [Leavensb03].

Pour spécifier les interfaces et les classes JAVA, deux types d'assertions sont fournis par JML : les *invariants* et les *contraintes historiques*.

2.2.2 Les invariants

Définition : (Invariant) *Les invariants sont des conditions qui portent sur les attributs de la classe. Ils doivent être préservés non seulement, avant et après l'appel des méthodes mais aussi à chaque appel interne d'une autre méthode.*

Les invariants servent à spécifier les propriétés que toute instance de la classe doit satisfaire à tout instant où les clients sont susceptibles de la considérer. Ils sont décrits en JML par des expressions booléennes introduites par le prédicat **invariant**. Un invariant en JML spécifie une propriété qui doit être établie juste après la création d'un objet et qui doit être préservée par toutes les méthodes. Ceci étant, n'importe quel invariant est inclut implicitement dans chacune des préconditions et postconditions de toutes les méthodes.

JML offre différents niveaux de visibilités (**private**, **public**, **protected**) pour qualifier ses assertions et entre autres les invariants. Ceci facilite le contrôle de la visibilité des spécifications héritées, d'une part, et sa synchronisation avec la visibilité JAVA au niveau des attributs de l'autre.

Exemple : Nous continuons avec l'exemple du compte bancaire

```
1.  public class Compte{
2.  final int MAX_SOLDE;
3.  private int solde;
4.  private int points_fidélité;
5.
6.  //@  public invariant 0<= solde && solde <= MAX_SOLDE ;
7.  //@  private invariant points_fidélité>=0;
8.  //@ public constraint  points_fidélité>= \old(points_fidélité)
9.
10. private byte[] pin
11.
12. /*@ private invariant pin != null & pin.length ==4 &&
13.                (\forall int I; 0<= I && I<4;
14.                  0<= byte[I] && byte[I] <=9);
15. @*/
16. public int retrait(int montant) throws exception{
17. ..}
```

Cet exemple illustre la notion d'invariant. Le premier invariant est décrit au niveau de la ligne 6. Il limite la valeur du solde dans un intervalle borné entre 0 `MAX_SOLDE`. Deux autres invariants sont décrits : 1)les points de fidélité sont toujours positifs ou nuls (ligne7), et 2) chaque élément du code pin de l'utilisateur doit être compris entre 0 et 9 (lignes 12-15).

L'erreur dans cette spécification se situe au niveau de la ligne 6 où le niveau de visibilité de l'invariant (*public*) dans la spécification ne respecte pas le niveau de visibilité au niveau de classe JAVA(*private* (ligne 2)).

2.2.3 Les Contraintes Historiques / Contraintes

Les contraintes historiques, couramment appelées les contraintes, sont des conditions sur les invariants qui doivent être respectées même en cas de *terminaison brutale*[5]. JML permet de les décrire via le prédicat **constraint**.

Les contraintes peuvent par exemple servir à contraindre le changement de leurs valeurs dans le temps. Pour ce faire, elles se servent du prédicat **\old** pour se référer à l'état de l'invariant avant chaque opération. C'est pour cette raison que, les contraintes historiques ne sont pas applicables aux constructeurs, qui n'ont pas de pré-état, ni aux destructeurs, qui n'ont pas de post-état.

Exemple : La ligne 7 de l'exemple précédent contient une spécification de la contrainte. Cette contrainte stipule que la valeur des points de fidélité doit être toujours croissante.

2.2.4 Spécification des méthodes

Le second aspect des classes JAVA que JML peut décrire**,** ce sont les méthodes.

En effet, JML nous permet de spécifier les méthodes via des conditions booléennes. Ces dernières peuvent porter sur l'état des objets, les attributs visibles de la classe concernée ou les paramètres de la méthode spécifiée avant son invocation (son *pré-état*). Nous distinguons entre deux types de conditions : les *préconditions* et les *postconditions*

- **La précondition**: décrit une condition qui doit être respecter avant l'exécution de la méthode. Elle peut concerner, par exemple, la valeur des paramètres de l'appel;

[5] terminaison anormale de l'exécution (erreur ou exception)

- **La postcondition :** est une condition qui se rapporte à l'état de l'objet après l'appel de la méthode. Elle peut se rapporter au pré-état de l'objet pour le comparer au nouvel état résultant de l'exécution de la méthode.

Pour spécifier les préconditions et les postconditions, JML propose les prédicats **requires** et **ensures**. Une spécification JML se présente souvent la forme suivante :

```
/*@ normal_behavior
        requires < précondition> ;
        ensures < postcondition> ;
@*/
```

Cela équivaut à dire que si la précondition est satisfaite lors de l'invocation de la méthode concernée, alors la méthode doit se terminer normalement sans lever une exception et que la *postcondition* va être satisfaite à la fin de l'invocation de cette méthode.

Toutefois, il arrivent que les programmes Java se terminent de façon brutale et lèvent une exception. Ce genre de comportement des méthodes peut être spécifié de la façon suivante :

```
/*@ normal_behavior
        requires   < précondition> ;
        ensures   < postcondition> ;
        signals    ( Exception1 ) <condition1> ;
        …
        signals    ( Exeptionn )  <conditionn> ;
@*/
```

Cela est équivalent à dire que l'exception1 est levée si la condition1 est vérifiée et que nous pouvons spécifier autant d'exceptions que nous le souhaitons. Il. Il importe de noter que les différentes exceptions ($Exception_1$,..,$Exception_n$) sont des sous-classes de la classe *java.lang.Exeption*.

Un troisième cas peut se présenter avec la spécification *exceptional behavior*. En effet, si une spécification est il est quasi certain qu'une exception va être levée. Ce cas de figure peut être spécifié de la façon suivante :

```
/*@  exceptional_behavior
        requires   < précondition> ;
        signals    ( Exception1 ) <condition1> ;
        …
        signals    ( Exeptionn  ) <conditionn> ;
@*/
```

Il faut noter que les spécifications *normal_behavior* et *exceptional_behavior* sont deux cas particuliers de la spécification de la forme *behavior*.

Exemple : Reprenons l'exemple de la classe compte :

```
1.  public class Compte{
2.  final int MAX_SOLDE;
3.  private int solde;
4.  private int points_fidelite;

5.  //@  private invariant 0<= solde && solde <= MAX_SOLDE ;
6.  private byte[] pin
7.  /*@ private invariant pin != null & pin.length ==4 &&
            i.  (\forall int I; 0<= I && I<4;
                    1.  0<= byte[I] && byte[I] <=9);
8.  @*/
9.  /*@ requires montant >=0;
10. assignable solde;
11. ensures solde==\old(solde)- montant &&
12.           \result== solde;
13. signals ( RetraitException) solde ==\old(solde);
14. @*/
15. public int  retrait (int montant) throws exception{
16. ..}
```

La spécification de la méthode *retrait (int montant)* est comprise entre les lignes 9 et 14 de la classe. Elle exprime dans sa précondition que le solde résultant d'une opération de retrait doit être inférieur à l'ancien solde avant l'opération avec une différence égale au montant retiré.

2.2.5 Héritage des spécifications

JML est également capable de prendre en considération le mécanisme d'héritage supporté par JAVA et de l'appliquer aux spécifications. Les différentes assertions des super-classes sont héritées par tous leurs fils. Ces derniers peuvent les affaiblir ou les renforcer. Ils doivent par exemple respecter la conjonction des assertions de niveau classe(invariant et contrainte) et la disjonction des préconditions.

2.2.6 Les spécifications abstraites

JML propose de définir des variables dites des variables de modèles ou abstraites, qui ne seront pas implémentées, et sont visible uniquement au niveau de la

spécification. Ce faisant, JML confère à la spécification une meilleure lisibilité en facilitant pour le lecteur de la spécification la distinction entre le code et la spécification. Il facilite, par ailleurs, l'écriture des spécifications semblables à celles orientés modèles.

L'utilisation des variables abstraites facilite l'écriture des spécification d'une part et lui apporte plus d'abstraction, de concision et d'indépendance vis à vis du code [CHEON03]. Ceci permet d'améliorer la lisibilité et facilite leur maintenance et leur évolution [CHEON03].

Grâce aux clauses **depends** et **represents**, les variables abstraites peuvent dépendre ou représenter des variables concrètes. Ces clauses représentent des fonctions de correspondance qui permettent de relier des variables de différents niveaux d'abstraction. Elles jouent le même rôle que la fonction d'abstraction pour les méthodes de preuve pour les types abstraits de données [Hoare72].

La présence des spécifications abstraites et de la possibilité de les relier au code ou à d'autres spécifications permet de faire du *raffinement* avec JML. Le raffinement ou raffinage se définie comme un processus permettant de transformer une spécification formelle en une implantation réelle[Ruby-Leavens00].

Exemple : Prenons l'exemple de la classe Client

```
Public classe interface Client {
   /*@
      model List list_transcations ;
   @*/
   /*@invariant
            (\forall int i ;
             0<= i && i< list_transcations.size();
              list_transcations.get(i) instanceof transcation);
   @*/
void Get_list_transcations( map transact){..}}
```

Fig. 8 Classe interface Client

```
1. Class ClientDouteux implements Client{
2.      /*@
3.      represents list_transactions <- transaction;
4.      @*/
5. String  nom;
6. String  adresse;
7. map  trasnsaction;
8. …}
```

Fig. 9 Correspondance entre variables abstraites

La Figure 8 illustre l'interface de la classe Client. Nous avons défini au niveau de cette classe une variable modèle `list_transcations` pour désigner l'ensemble des transactions effectuées par un client.

La classe clientDouteux (Fig9), implémente cette interface. Afin de pouvoir relier la spécification abstraite au niveau de l'interface à son implémentation au niveau de la classe clientDouteux nous avons fait appel à la fonction de correspondance représentée en JML par la clause **represents**.

2.3 Les spécifications exécutables

2.3.1 Principe

Il est possible d'évaluer les assertions JML en cours d'exécution, en vu de traquer d'éventuels «bogues». JML Runtime Checker (JMLC) est un vérificateur des assertions JML à l'exécution. Lorsque le contrôle est actif, JMLC évalue une assertion à sa valeur de vérité de façon transparente et sans effet sur la suite de l'exécution. Lorsqu'elle est évaluée fausse, elle déclenche une exception ainsi qu'on le verra plus loin. Cette possibilité de vérification des assertions a pour effet de garantir un puissant mécanisme de test.

JMLC recompile les classes JAVA et les instrumente de façon à ajouter, aux méthodes spécifiées, un code supplémentaire destiné à vérifier la conformité de la spécification par rapport au code. Pour ce faire, il commence par l'instrumentation de la classe en transformant la méthode originale en une méthode privée et la remplace par une méthode enveloppante (*wrapper method)*. Il s'assure que cette dernière ne va déléguer l'appel de la méthode d'origine qu'après avoir fait certaines vérifications.

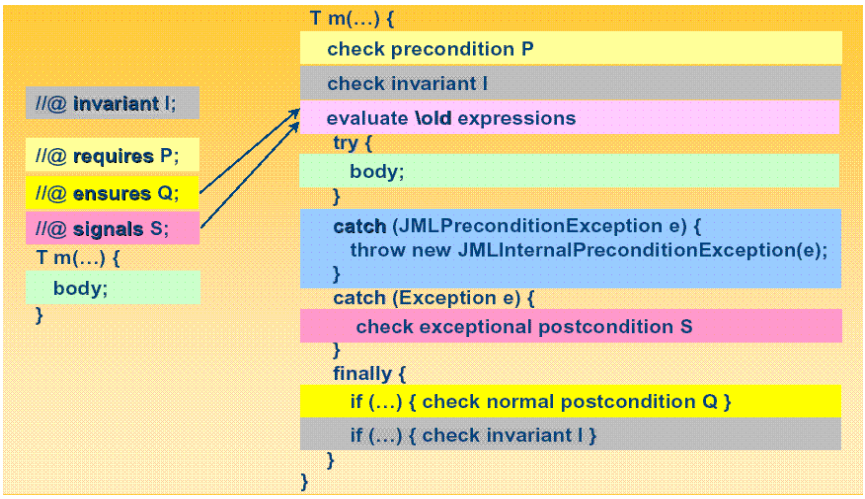


Fig. 10 squelette de la méthode instrumentée

Le code ajouté par JMLC commence vérifie d'abord les *préconditions* et les invariants. En cas de violation des assertions, une exception est levée. Sinon, la méthode d'origine dans le premier bloc *try* est invoquée. La valeur retournée sera sauvegardée car elle servira par la suite au test de la *postcondition*.

Le premier bloc catch sert à propager la violation des assertions qu seront constatées en exécutant la méthode d'origine. Si le contrôle atteint le second catch, alors le traitement de la *postcondition* exceptionnelle (spécifiée au niveau de la *clause signal*) est activé. La dernière étape de cette procédure consiste à vérifier les invariants et les contraintes historiques.

L'instrumentation des classes JAVA par le JML n'a aucun effet de bord et n'affecte aucunement le comportement des méthodes invoquées. Toutefois, des performances en terme de temps d'exécution et de taille seront légèrement influencées. Cette lacune peut être contourner en désactivant les assertions après la livraison de l'application.

2.3.2 Caractéristiques de JMLC

L'évaluateur des assertions à l'exécution JMLC représente une innovation pour l'état de l'art actuel en matière de vérifications dynamique des assertions. Son originalité réside dans le fait qu'il permet de prendre en charge le contrôle des assertions au niveau des spécifications abstraites, des interfaces et des classes qui héritent ou qui implémentent d'autres classes.

L'une des innovations apportées par JMLC consiste dans sa capacité à contrôler les spécifications héritées entre les classes. Il remédie, même, grâce à une approche de délégation dynamique, au problème des conflits de nom entre primitives héritées de parents distincts, un cas inévitable en pratique, surtout lorsqu'il s'agit de classes développées par des contributeurs multiples.

Selon cette approche si la spécification héritée fait appel à une méthode déclarée au niveau de sa super-classe, alors cet appel sera délégué à cette dernière. Une telle procédure permet de prendre en charge les situations d'héritage multiples des classes JAVA.

Par Pailleurs, JMLC se base sur le mécanisme de *réflexion* et d'*introspection* offert par JAVA pour récupérer de façon dynamique les informations concernant la structure des classes. L'usage d'un tel Un tel mécanisme permet à JMLC d'appliquer ses vérifications à l'exécution même avec des les classes compilées séparément [Cheon02].

Outre l'évaluateur des assertions JMLC, plusieurs outils ont été implémentés autour du langage JML. Nous présenterons quelques-uns d'entre eux dans la section suivante.

2.4 Outils de support

L'intérêt particulier porté à JML par les industriels et les laboratoires de recherches précédemment cités s'est traduit par un ensemble d'outils de support :

2.4.1.a Outils de Test unitaire

L'outil JML-JUNIT [Cheon02b] combine le test unitaire et la vérification des assertions à l'exécution. Il offre la possibilité de tester les classes java qui contiennent des assertions JML et constitue une extension du framework populaire de test JUNIT [Beck98]. Ce dernier se présente comme un framework de test pour les programmes Java. L'idée de base de JUNIT est de faciliter, d'accélérer et d'améliorer la qualité des tests unitaires.

2.4.1.b Analyse statique et vérification :

Outre son utilisation dans l'activité de test, JML a été appliqué dans le cadre de l'analyse statique et la preuve logicielle. Nous citons dans cette sections deux exemples d'outil d'analyse statique qui font appel à des démonstrateurs de programmes.

- **L'outil LOOP**(*Logic of Object-Oriented Programming*)

  L'outil LOOP permet de traduire les annotations JML en obligations de preuve. Ces dernières seront transmises à un démonstrateur de théorèmes tel que PVS ou Isabelle afin de vérifier si les implémentations Java sont correctes [Burdy03]. Cela a demandé un effort considérable de formalisation des détails de la sémantique de JAVA et de JML. L'université Nijmengen, qui a développé, cet outil a combine l'utilisation de l'outil LOOP avec JML dans le cadre d'une étude de cas de la nouvelle génération des cartes java intelligentes (Java Smart card) [Poll00] ;

- **Extended Static Checker (ESC/ Java)**

  Le projet ESC [Flangan02] est un outil de test qui a été développé par Compaq Systems Research Center. Il se sert d'un démonstrateur de théorèmes pour détecter les erreurs java les plus couramment commises par les programmeurs telles que les pointeurs nuls ou le débordement des tableaux, etc. Les vérifications sont faites de manière statique et automatique sans exécution le code ni intervention. ESC/Java utilise un sous-ensemble des spécifications JML qui lui servent détecter d'autres types d'erreurs. En effet, cet outil peut détecter des erreurs supplémentaires en vérifiant l'inadéquation entre les paramètres d'appel d'une méthode et les conditions spécifiées sur ces paramètres par exemple.

2.4.1.c Assistants de génération de spécifications

Les deux outils précédents qui assurent les tâches de validation et de vérification sont complétés par des assistants dont la fonction est d'aider à l'écriture des spécifications :

- L'outil Daikon [Ernst01] : a été conçu afin de détecter les invariants en observant le comportement du programme lors de son exécution ;

- L'outil Houdini [Flanagan01] permet d'inférer les annotation pour le code ;
- L'outil jmlspec [Burdy03] produit des squelette de spécification à partir du code source JAVA.

2.4.1.d Générateur de documentation

L'outil de documentation fourni avec JML s'appuie sur les assertions pour produire, à l'intention des développeurs de classes clientes, des informations décrivant le comportement des classes.

Cet outil permet de générer une documentation en ligne à partir des annotations JML et des fichiers JML [Raghvan00] . Il procède d'une façon similaire à *Javadoc*[6] avec et facilite, par conséquent, l'utilisation des spécifications JML dans la documentation des modules Java.

2.5 Travaux connexes

Nombreux sont les efforts qui ont été déployés pour automatiser l'évaluation des assertions à l'exécution. Le premier travail qui a présenté cette approche a été avec Meyer en 1992 avec son approche de *conception par contrats* [Meyerb92] a développé au moyen du langage Eiffel[Meyer92]. La principale lacune de Eiffel est son incapacité à prendre en charge les quantificateurs existentiels et par conséquent de fournir des spécifications orientées modèles. Cette approche, a été ensuite adopté par plusieurs autres langages notamment C, C++, Smaltalk, Python, etc [Cheon 02] mais aucun des évaluateurs des assertions pour ces langages n'a été utilisé dans l'activité du test. Il en est de même pour les évaluateurs d'assertions pour les modules JAVA [Duncan98,Karmer98,Karaorman99,Bartetzko01,BARESI01] qui se sont, dans la plupart des cas, situés dans un cadre de programmation par contrat plutôt que dans un cadre d'automatisation du test.

L'un des premier outils qui a intégré les annotations est l'outil *iContract* [Karmer98]. Il permet d'intégrer dans les modules JAVA des spécifications sous forme d'annnotations. *iContract*, commence par recompiler les classes java pour les instrumenter en introduisant les vérifications nécessaires au niveau du code. Les vérifications n'ont aucun effet de bord. Pour vérifier les assertions *iContract* établit un

[6] http://java.sun.com/j2se/javadoc/

registre de contrats qu'il consulte pour propager les contrats selon la hiérarchie des classes. Cela est équivaut au mécanisme d'héritage des assertions. Cet outil a la particularité de se servir de son propre langage de spécification pour exprimer les conditions booléennes. L'utilisation des étiquettes telles que *@pre, @post* facilite la génération de la documentation des modules avec des outils comme l'outil *javadoc,* le standard de *JDK*. Toutefois, *iContact* ne supporte que quelques quantificateurs au niveau des assertions et il nécessite, dans la plupart des cas, l'accès au code source ce qui n'est pas toujours possible.

JContractor [Karaorman99] est un autre outil qui vise le même objectif mais adopte une philosophie légèrement différente de *iContract*. En effet, il se présente comme une bibliothèque java et un ensemble de règles de nommage. Les patrons de contrats (*contract* patterns) seront reconnus et interprétés durant le chargement des classes. Il procédera à des modifications des classes afin de respecter ces contrats. Chaque *invariant* ou *pré-postcondition* est décrit par une méthode booléenne. Cette technique de description des assertions se révèle cependant limitée à mesure que le nombre de méthodes augmente. De plus, cet outil ne supporte pas les quantificateurs existentiels, ni la spécification des types abstraits ce qui limite. Son statut se limite donc à un simple vérificateur des *préconditions, postconditions* et des *invariants*.

Handshake [Duncan98] se présente comme une bibliothèque dynamique de lien et compilateur des contrats. Son fonction consiste à intercepter les appels des méthodes émis par la machine virtuelle Java (*JVM*). Il compile les spécifications et les affecte aux classes chargées dans la mémoire, sans affecter le code d'origine. La principale lacune de cette approche réside dans l'expression des spécifications dans un langage ainsi que dans l'utilisation d'une syntaxe différente de celle de JAVA.

[Findler00] a présenté les limites des trois travaux [Duncan98,Karmer98, Karaorman99], précédemment évoqués. Il a démontré que *iContract*, *Jcontrator et handshake* sont incapables de traiter correctement les cas de spécifications d'implémentation d'interfaces multiples.

Dans le même contexte, *le précompilateur Jass* **(J**ava with **ass**ertions) [Bartetzko01] supporte les assertions au niveau du code Java. Il offre la possibilité de spécifier des contrats en termes de *pré-postconditions* et invariants. Il supporte aussi quelques quantificateurs au niveau des spécifications *(Forall* et *exists*). De plus, il

permet l'héritage des assertions et le raffinement. Ce *pré-compilateur*, instrumente les code des modules spécifiés pour traduire les assertions Jass en code Java.

Baresi et Young [BARESI01] ont critiqué toutes ces approches et ont présenté leurs limites. Nous proposons dans ce travail une synthèse de leurs critiques dans Le tableau comparatif suivant :

	iContract	Jass	Handshake	jContractor	JML
Invasive	+	+		+	+
Pré-compilation	+	+		+	+
Commentaire	+	+			+
Pré-postCondition	+		+	+	+
Check					+
Quantificateur	+				+
Old	+			+	+
return	+			+	+

Fig. 11 Les systèmes d'assertion pour Java

- « *Invasive* »: les insertions sont intégrées dans le code source ;
- « *Pre-processor* »: c'est un mécanisme qui recompile les classes java afin de les instrumenter et intégrer le code qui les force à tester les assertions ;
- *« Commentaires »*: les assertions sont intégrées au niveau du code sous forme de commentaires ;
- *« Pre-postcondition »*: la possibilité d'intégrer des *pré-postconditions* ;
- *« Check »*: une assertion écrite par le programmeur pour activer un flot de contrôle ;
- *« Old »*: la référence à la valeur des variables avant l'exécution du programme ;
- Return : la possibilité de se référer au résultat retourné dans les assertions .

D'après cette étude *iContract* et *Jassert* sont les mieux placés, vu les fonctionnalités qu'ils offrent. Toutefois, ses dernières mises à jour, JML se place en tête des langages de spécifications exécutables pour JAVA.

2.6 Conclusion

L'objectif de ce chapitre été de présenter le langage JML et de mettre en évidence sa capacité à fournir des spécifications à la fois abstraites, non ambiguës, lisibles et faciles à maintenir.

Une attention particulière a été accordée à l'évaluateur des assertions JMLC qui confère aux spécifications JML un aspect exécutable particulièrement intéressant dans le cadre de test automatisé et notamment la génération des oracles de test pour les programmes JAVA.

Partie 2

Propositions Réalisation et Expérimentations

Chapitre 3

Approche Proposée

L'objectif de ce chapitre est de présenter une approche de génération automatique des oracles de test pour les programmes JAVA sur la base des spécifications exécutables JML. L'application de cette approche sera axée sur l'outil TOBIAS. Notre ambition est d'améliorer ses performances et de réduire tant que faire se peut l'explosion combinatoire des cas de test qu'il génère.

3.1 Les problèmes à résoudre

Comme nous l'avons déjà expliqué au niveau du chapitre 1, les défis à relever sont de 3 ordres :

i) **Le problème de l'explosion combinatoire des cas de test** : C'est un problème propre à l'outil TOBIAS. Il est occasionné par le dépliage exhaustif des schémas de test TOIBIAS au cours de la génération des cas de test. Il en résulte un nombre très élevé des jeux de test avec des séquences non-conformes. Ces cas de test encombrent la mémoire et exigent un temps supplémentaire pour leur traitement. Le filtrage de ces résultats permettrait d'en réduire le nombre et d'en améliorer la qualité. Surtout, cela procurerait à l'outil TOBIAS de meilleures performances. Toutefois, ceci ne sera possible qu'en le dotant en entrée des spécifications ;

ii) **L'absence d'un pilote de test :** Une fois les cas de test obtenus, encore faut-il pouvoir les appliquer à une implantation du système afin d'en vérifier la conformité. A l'heure actuelle, TOBIAS ne dispose pas d'un tel mécanisme. Pour exécuter les cas de test qu'il génère recourt à des outils externes notamment Junit et JML [Beck98]. Ces outils ne lui sont pas parfaitement adaptés et ne prennent pas en considération les caractéristiques spécifiques des cas de test TOBIAS ;

iii) **Le problème de génération automatique de l'oracle de test** : La revue de l'état de l'art en matière de génération d'oracle de test, nous a révélé la sensibilité du problème de l'oracle et a permis de mettre en exergue la difficulté de l'obtenir. L'automatisation de l'évaluation des assertions JML à l'exécution nous semble être une piste particulièrement prometteuse pour résoudre ce problème.

Notre hypothèse de départ est que les spécifications peuvent être utiles à la fois pour la résolution du problème de l'oracle ainsi que pour la maîtrise de l'explosion combinatoire des cas de test. Ce chapitre tentera de combiner TOBIAS à JML et vérifier la validité de cette piste.

3.2 Les spécifications exécutables comme oracle de test

Une spécification est *un contrat* entre les implémenteurs d'une méthode et ses utilisateurs [Meyer92,Liskov01]. La précondition engage les appelants; la postcondition engage la routine. De ce fait, doter les spécifications d'une capacité d'exécution fournira une aide vitale dans la recherche des erreurs.

En effet, aiguiller le système, tandis qu'il s'exécute, vers l'évaluation d'assertion et déclencher une exception lorsqu'il détecte une violation, permettent de contrôler la cohérence entre ce que le système fait (au travers des corps de routine) et ce que l'on croit qu'il fait(formulé dans les assertions).

Nombreux sont les travaux qui se sont intéressé à l'automatisation de l'évaluation de ces spécifications. Cet intérêt a commencé au début des années 90 avec les travaux de Meyer et le langage Eiffel [Meyer92] dans le cadre de la *conception par contrat*. Ce n'est que récemment que l'automatisation de l'exécution des spécifications a intégré l'activité dans le but d'intégrer l'activité de test unitaire[Cheon02b].

qui ont donnée naissance à l'évaluateur des assertions JMLC . Cet outil, qui a été présenté dans le chapitre 3, a été combiné avec un framework de test assez connu (JUNIT) dans le but d'intégrer l'activité de test unitaire[Cheon02b].

Tout en s'inscrivant dans la même lignée, notre travail s'efforcera de montrer l'utilité des spécifications exécutables pour l'activité du test et, par la même, de révéler travers leur application, leur apport pour les cas de tests générés par Tobias.

Comme nous l'avons précédemment signalé, [Cheon02b] a déjà proposé d'adopter les spécifications exécutables de JML comme oracles de test dans le cadre d'une activité de test unitaire. Le présent travail adopte une approche qui diffère de celle proposée dans le travail cité. La spécificité de notre approche réside dans la façon dont les résultats de l'exécution des spécifications sont traités. En effet, le feed-back immédiat fourni par les spécifications exécutables JML est pour nous un outil très précieux dans la mesure où il permet de filtrer à lors de l'exécution le nombre de cas de test à exécuter.

A notre connaissance aucune approche n'a encore considéré les assertions exécutables et les oracles exécutables comme nous l'avons fait. Le problème de l'explosion combinatoire des tests nous a été un terrain très fertile pour l'application et le test de la validité de notre approche et notre façon de traiter les verdicts produits suite à l'exécution des tests.

3.2.1 L'exemple du Buffer

Nous présentons dans cette section l'exemple du *Buffer* qui sera référencé tout au long de ce chapitre pour illustrer nos propositions. Le Buffer étudié, dans cet exemple possède deux attributs *x* et *y* qui doivent, satisfaire les conditions suivantes, en toutes circonstances:

- $x>=0$
- $y>=0$
- $x-y>=0$

L'interface du *Buffer* est constituée des méthodes suivantes :

- **void init ( )** : correspond à l'opération d'initialisation du Buffer ;
- **void Add (int a)** : correspond à l'opération d'ajout au Buffer d'une valeur positive ou nulle a et la départager entre les *x* et *y* ;

- **void Del (int a)** : correspond à retrancher du Buffer une valeur positive ou nulle_ *b* de sorte que la nouvelle valeur de (*x*+*y*) soit égal à son ancienne valeur moins la valeur de a ;
- **int GetX( ), int GetY( ) :** Des opérations de récupérations de la valeur courante de *x* et *y*;
- **void SetX ( ), void SetY ( ) :** Des opérations de mise à jour de la valeur de *x* et *y*;
- **void PrintBuffer( ) :** Méthode d'affichage des éléments du buffer. Elle requière que la valeur de (*x*-*y*) soit supérieure ou égal à 3.

Nous présentons, dans un premier temps, une spécification correcte de ces méthodes (Fig.13). Toutefois, par la suite, des erreurs seront insérées au niveau de la spécification et du code afin d'illustrer des situations de non conformité.

```
1.   public interface Class buffer {

2.   //@  model int X;
3.   //@  model int X;
4.   //@* invariant X>=0 && Y>=0 && X>Y;

5.   /@* public normal_behavior
6.   *requires a >=0;
7.   *ensures X+Y==\old(X+Y)+a;
8.   *@/
9.   void  add( int a);

10.  /@* public normal_behavior
11.  *requires b >=0;
12.  *ensures X+Y==\old(X+Y)-a;
13.  *@/
14.  void  del(int b);
15.  void init();
16.  void GetX();
17.  void GetY();
18.  void SetX();
19.  void SetY();/@* public normal_behavior
20.
21.  /@* public normal_behavior
22.  *requires X-Y >=3;
23.  *ensures true;
24.  *@/
25.  void PrintBuffer ( )
```

Fig. 12 Spécification de l'interface du buffer

Les lignes deux et trois décrivent les variables de modèle *x* et *y* qui sont visibles uniquement au niveau de la spécification. La ligne 4 de la spécification décrit la

condition globale à respecter (l'invariant) sous forme d'une conjonction de 3 conditions. Les spécifications des méthodes *add, delete* et *printBuffer* sont effectués au niveau des trois clauses *public normal_behavior*.

3.2.2 L'approche JML-JUNIT

3.2.2.a Principe

Cette approche consiste à doter le framework JUNIT de la capacité de prendre en charge des assertions JML. L'outil résultant JML-JUNIT [Cheon02b], qui a été présenté dans le chapitre 3, permet d'exécuter les différents cas de test générés par TOBIAS pour des classes JAVA spécifiées en JML.

En effet, TOBIAS permet de générer des cas de test en JAVA, de les regrouper sous forme d'appels de méthodes et de les intégrer dans des classes respectant le format des cas de test JUNIT. Les différents cas de test seront, par la suite, invoqués de façon à garantir leur indépendance les uns des autres et d'éviter, par conséquent, les éventuels effets de bord liés à une manipulation de la même instance par différents cas de test.

JML-JUNIT exécute, alors les séquences à tester. Il récupère, ensuite, les exceptions levées par le vérificateur des assertions JML et affiche les verdicts correspondants à l'exécution des chaque cas de test.

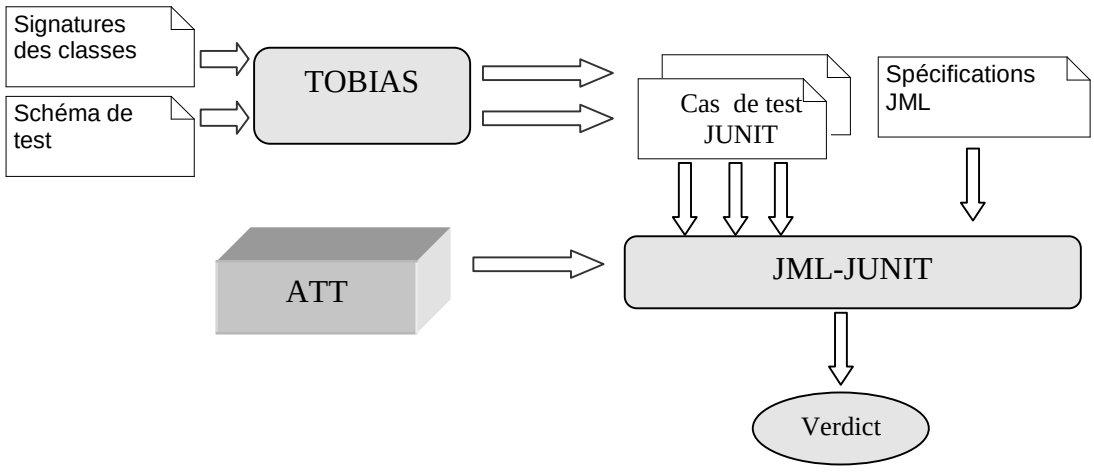


Fig. 13 JML-JUNIT et TOBIAS

Jusqu'à présent, JML-JUNIT joue le rôle du pilote de test pour TOBIAS. Il permet d'automatiser l'exécution de ses cas de test et la délibérations des verdicts correspondants.

3.2.2.b Limites de l'approche

La principale lacune de cette approche provient du fait que JML-JUNIT exécute les cas de test de façon séquentielle. Il n'est pas capable d'effectuer des ajustements de sa stratégie d'exécution des tests en fonction des erreurs et des exceptions qui surgissent au cours de leur exécution. Une telle façon de procéder s'avère très coûteuse en terme de temps d'exécution des tests.

Par ailleurs, JML-JUNIT ne traite pas les cas de non-terminaison et les boucles infinies. Les conséquences de cette faiblesse peuvent êtres grave sur les performances de l'outil en terme de temps d'exécution. En effet, l'apparition de cas de non-terminaison au cours du test peut prolonger d'avantage leur temps de l'exécution.

Toutes ces limites JML-JUNIT nous ont amené à délaisser cet outil et lui substituer un autre pilote de test qui a pour ambition de résoudre le problème de l'oracle et de réduire le nombre des cas de test généralement élevé généré par TOBIAS.

3.3 Notre Approche

Dans cette section nous présentons une approche d'optimisation du nombre de cas de test exécutés. Nous expliquons, dans un premier temps, l'importance de structurer nos cas de test afin de faciliter, lors de leur exécution, l'élimination de ceux qui ont échoué et ceux qui sont non conforme à la spécification.

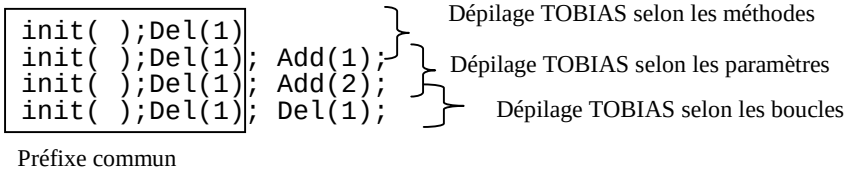
L'idée de base consiste à organiser les données de test de façon à ce que l'élimination d'un cas de test, soit accompagnée systématiquement par une élimination de ceux qui constituent son prolongement.

3.3.1 Organisations des cas de test en arbre n-aire

L'examen des différents cas de tests générés par TOBIAS, nous a permit de constater qu'elles possèdent, dans la plupart des cas, certaines caractéristiques communes.

Exemple : Prenons le cas du schéma de test suivant :

Sh_1:=MTC.$G_1^{\wedge 0..3}$
G_1={initt( ),Del(m_1),Add(m_2)}
$m_1 \in \{1,2\}$
$m_2 \in \{1\}$



Ces quatre cas de test sont des séquences possibles générées à partir du schéma *sh1*. Elles commencent toutes par la même séquence d'appel d'appels de méthodes [init( ) ; Del(1)], qui ne satisfait pas l'invariant (x>=0 et y>=0). L'exécution ces cas de test avec l'outil JML-JUNIT aboutit à quatre messages d'erreur qui signalent la même erreur pour chaque cas de test alors qu'en réalité il s'agit de la même erreur. Il est possible d'éviter l'exécution des trois derniers cas de test en procédant autrement et en prenant en considération les similarités entre les cas de test TOBIAS.

L'approche que nous préconisons a pour ambition de réduire le nombre d'exécutions des cas de test et par conséquent le nombre de ces messages. Ainsi, au lieu de quatre dans le cas de l'exemple cité, un seul message sera affiché. Il signalera que la séquence [init( ) ; Del( )] mène toujours à une erreur. Pour que ceci soit vrai, nous supposons alors que le comportement des méthodes est déterministe afin de garantir la répétition de ce comportement chaque fois que la séquence d'appel [init() ;Del( )] est invoquée.

L'idée de base consiste à organiser les cas de test de façon à commencer par en exécuter les plus « courts[7] ». Si une erreur est rencontrée au niveau de ces séquences, l'exécution de celles qui sont plus longues ne sera pas effectuée**.** De cette façon la propagation des séquences sera stoppée erronées dès leur première apparition.

Pour que cette technique soit applicable, les cas de test doivent être organisés de façon hiérarchique permettant de mettre en évidence la relation de composition entre les cas de test. Nous proposons alors d'organiser les séquences de test un arbre n-aire dont la construction obéit aux étapes suivant :

[7] Constitués du nombre le plus petit possible de séquences d'appels de méthodes

1. Créer un nœud racine dont le rôle consiste à initialiser les cas de test avant leur exécution ;
2. Récupérer les instances, les méthodes et les paramètres de méthodes à partir des cas de test générés par TOBIAS et les placer dans une même structure ;
4. Insérer les nœuds dans l'arbre de façon à obtenir des séquences d'appels de méthodes qui commencent par l'appel du nœud racine (init) et se terminent par l'appel des nœuds feuilles ou par des nœuds particuliers marquant la fin de cas de test.

Exemple : Prenons le schéma de test sh_1 suivant :

$$Sh_1 := b_1.\text{Add}(x)^{\wedge 1..2} ; b_2.\text{Del}(y)^{\wedge 0..1}$$
$$x \in \{ 1, -1 \}$$
$$y \in \{ 1 \}$$

La Fig.14 représente l'arbre correspondant au schéma Sh_1. La racine de cet arbre est la méthode *init( )* qui initialise et rafraîchie les instances pour chaque cas de test. Chaque nœud de l'arbre correspond à l'appel d'une méthode sur une instance avec des paramètres bien déterminés. Les branches de l'arbre correspondent à des cas de test. Un cas de test commence par l'appel de la méthode, d'initialisation des cas de test, init( ) suivi, de l'appel d'une séquence de ces fils. Il se termine par un nœud qui marque la fin de la séquence ou par un nœud feuille.

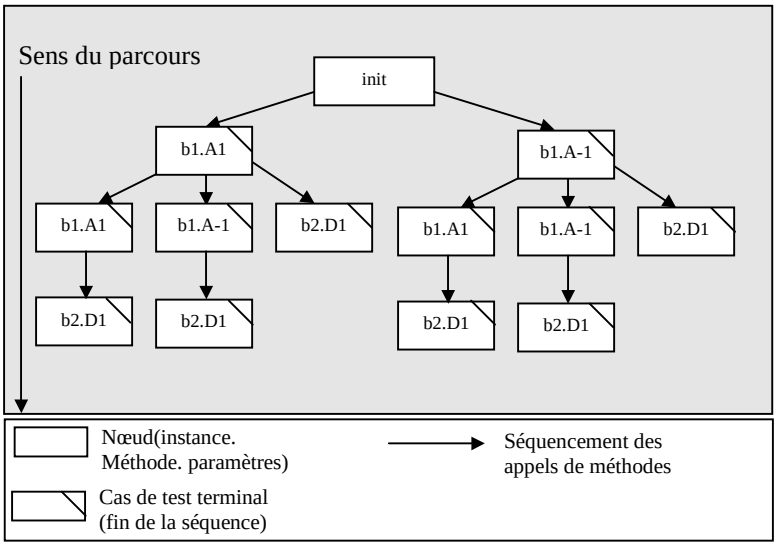


Fig. 14 Arbre des cas de test

L'examen de la figure 14 montre que la taille des cas de test, de point de vue nombre d'appel, croit au fur et à mesure qu'on avance vers les feuilles en suivant le chemin le plus à gauche.

Exemple : Un cas de test possible est :

```
init;b1.Add(1); b1.Add(1); b2.Del(1)
```

Il correspond à l'appel de la méthode Add(1), sur l'instance b1, deux fois, suivi de l'appel de la méthode Del(1) sur l'instance b2.

3.3.2 La sélection des cas de test

L'organisation des cas de test TOBIAS de façon à mettre en évidence l'aspect hiérarchique qui les caractérisent joue un rôle crucial dans la tâche de sélection des cas de test.

En effet, la sélection commence par les cas les plus courts qui sont situés en haut de l'arbre, puis, progressivement, sont sélectionnés des séquences de plus en plus longues. La stratégie de sélection des données de test que nous proposons, dans un premier temps, consiste à parcourir l'arbre des cas de test en profondeur.

Exemple : Le résultat du parcours total de l'arbre de la Fig. 15 se présente comme suit :

```
1. init;b1.Add(1) ;
2. init;b1.Add(1) ; b1.Add(1)
3. init;b1.Add(1) ; b1.Add(1)  ; b2.Del(1)
4. init;b1.Add(1) ; b1.Add(-1)
5. init;b1.Add(1) ; b1.Add(-1) ; b2.Del(1)
6. init;b1.Add(1) ; b2.Del(1)
7. init;b1.Add(-1);
8. init;b1.Add(-1); b1.Add(1)
9. init;b1.Add(-1); b1.Add(1)  ; b2.Del(1)
10.init;b1.Add(-1); b1.Add(-1)
11.init;b1.Add(-1); b1.Add(-1) ; b2. Del(1)
12.init;b1.Add(-1); b2..Del(1)
```

3.3.3 Exécution des cas de test

Une fois les cas de test sont sélectionnés et générés, encore faut-il les exécuter de façon automatique. C'est le pilote de test qui va se charger de cette tâche.

Après l'initialisation des cas de test, le pilote de test parcourt les branches et exécute les méthodes selon une stratégie incrémentale. Il affiche par la suite le résultat de l'exécution.

Jusqu'à présent, TOBIAS ne dispose pas d'un pilote de test pour les programmes JAVA. Nous avons donc implémenté un pilote de test permettant de récupérer les cas de test générés par TOBIAS, de créer les arbres de cas de test correspondant, les parcourir et les exécuter et afficher les verdicts.

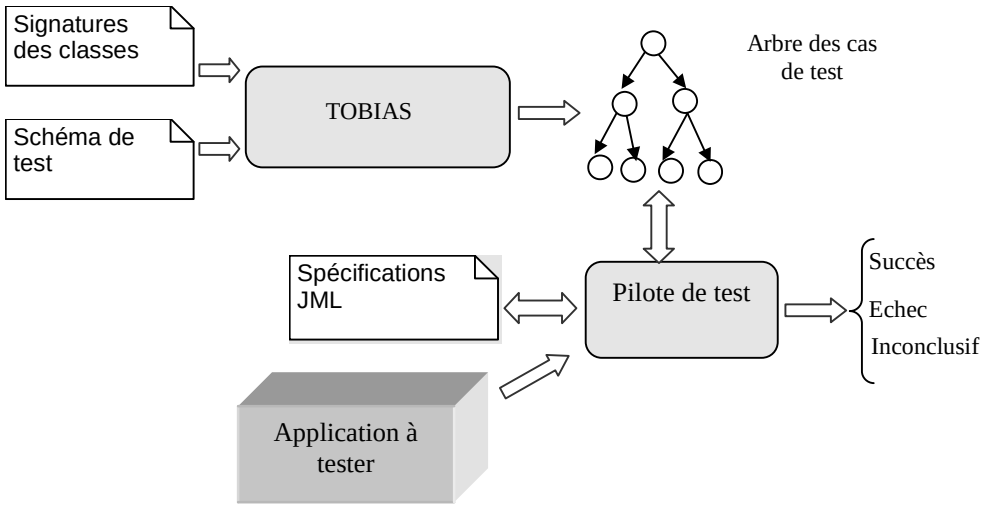


Fig. 15 Le pilote de test de TOBIAS

Le pilote de test parcourt l'arbre en profondeur en commençant par les branches les plus à gauche. Il exécute, le contenu de chaque nœud qui consiste en une méthode avec les paramètres d'appel et l'instance concernée par l'exécution.

Nous rappelons que notre travail est situé dans un cadre de test de conformité et que nos classes JAVA sont spécifiées en JML et instrumentées par le vérificateur des assertions à l'exécution de JML. Cette instrumentation oblige les classes à effectuer certains contrôles sur la conformité par rapport à la spécification avant et après l'exécution de chaque méthode. Dés lors, une exception sera levée chaque fois qu'une contrainte de la spécification est violée. La figure16 illustre la hiérarchie des exceptions JML qui peuvent être rencontrées.

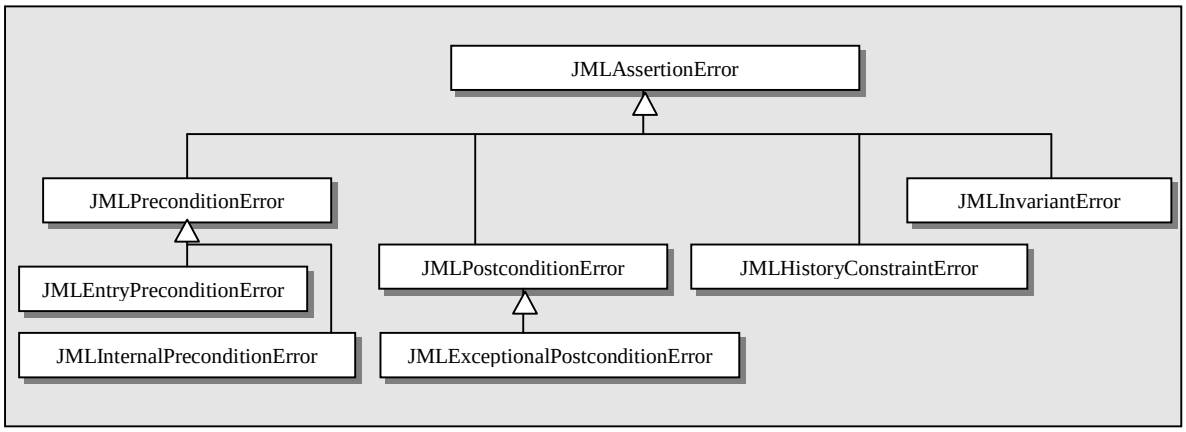


Fig. 16 Hiérarchie des exceptions JML

En tête de cette hiérarchie se trouve la classe *abstraite* « JMLAssertionError ». Toutes les autres classes héritent de cette classe. Nous proposons de détailler cette hiérarchie afin de mieux appréhender les différents types d'erreurs JML. Ces données seront utile pour les sections suivantes dans lesquelles nous proposons des optimisations d'exécution selon le type d'erreur rencontrée.

i) **JMLPreconditionError**: Cette exception est déclenchée lors de l'appel d'une méthode en dehors de sa précondition. Il existe deux variantes de cette erreur ;

- **JMLEntryPrecondition :** Cette exception est levée si l'appel d'une méthode est effectué en dehors de son domaine ; la précondition est une obligation que le client doit satisfaire. Rien n'est garanti du moment où la spécification n'est pas respectée ;
- **JMLIntenalPreconditionError**: Cette exception est levée si une méthode *m* en cours de son exécution appelle une autre méthode *f* en dehors de sa précondition ;

ii) **JMLPostconditionError**: Cette exception est déclenchée lorsque les résultats de l'exécution d'une méthode ne sont pas conformes par rapport à sa spécification. Cette erreur peut être Exceptionnelle ;

- **JMLExceptionalPostcondition** si la précondition d'une méthode est respectée, et la postcondition ne l'est pas alors une exception est levée par JMLC. Cette exception peut être relative à la clause **ensures**. Elle est traitée au niveau de la clause **signal** de la spécification.

iii) **JMLInvariantError**: Cette exception est déclenchée lors de la transgression de l'invariant d'une classe ;

iv) **JMLHistoryConstraintError**: Cette exception est déclenchée lors de la violation d'une contrainte historique.

Le Pilote de test récupère ces exceptions JML et affiche à l'ingénieur de test le verdict statuant le cas de test. Trois situations sont possibles :

INCONCLUSIF: Ce verdict est délivré dans le cas où l'appel d'une méthode *m* du cas de test est effectué en dehors de son domaine. On dit que le cas de test est insensé et il est rejeté. Dans cette situation l'exception *JMLEntryPrecondition* sera levée ;

ECHEC: Ce verdit est délibéré dans le cas de n'importe quelle violation des assertions JML, sauf celle des préconditions à l'entrée (*JMLEntryPrecondition).* Il correspond à l'échec de l'implémentation à satisfaire sa spécification ;

SUCCES: Ce verdict est délivré si l'exécution de toutes les méthodes du cas de test se termine sans lever une exception. Un cas de test est dit réussi si l'exécution du code des méthodes invoquées est conforme à leurs spécifications.

Exemple : Si nous reprenons les cas de tests correspondants au schéma sh_1 avec une implantation de la méthode *Del( )* non conforme à sa spécification nous pourrons avoir la liste des verdicts suivante *:*

Cas de test	Verdict
1. init ; b1.Add(1);	SUCCES
2. init; b1.Add(1); b1.Add(1)	SUCCES
3. init; b1.Add(1); b1.Add(-1)	INCONC
4. init; b1.Add(1); b2.Del(1)	ECHEC
5. init; b1.Add(1); b1.Add(1); b2.Del(1)	ECHEC
6. init; b1.Add(1); b1.Add(-1); b2. Del(1)	INCONC
7. init ; b1.Add(-1);	INCONC
8. init; b1.Add(-1); b1.Add(1)	INCONC
9. init; b1.Add(-1); b1.Add(-1)	INCONC
10. init; b1.Add(-1); b2..Del(1)	INCONC
11. init; b1.Add(-1); b1.Add(1); b2.Del(1)	INCONC
12. init; b1.Add(-1); b1.Add(-1); b2. Del(1)	INCONC

Préfixe commun

Dans cet exemple nous avons 2 cas de test qui ont réussi, 2 qui ont échoué et 8 qui ont été « rejetés ».

Le tableau de l'exemple illustre le résultat de l'exécution des cas de test tels qu'ils ont été générés par TOBIAS (sans leur réorganisation sous forme d'arbre). Si nous exécutons ces tests selon l'ordre de leur génération, nous allons commettre la même erreur que l'approche JML-JUNIT. Et nous n'allons pas tirer parti de la spécificité des tests générés par TOBIAS et de la possibilité de factoriser la vérification de leur validation.

C'est pour cette raison que nous partons du constat que les cas de test TOBIAS peuvent être organisés de façon hiérarchique et que nous construisions notre arbre des

cas de test correspondant et nous proposons une série d'optimisations dans le but d'atténuer le nombre des tests.

3.3.4 Découpage des branches et réduction du nombre des cas de test

La première optimisation que nous proposons dans le cadre de ce travail consiste à découper « les branches contaminées[8] » de l'arbre ; Autrement dit, nous arrêtons l'exécution des cas de test qui constituent des prolongements des cas de test erronés ou rejetés.

Dans l'exemple précédent, les cas de test 7 à 12 ont tous le même préfixe. Lors du parcours de l'arbre et de l'exécution des cas de test, dès que l'appel d'une méthode *Add(-1)* génère une erreur de précondition, l'exécution du cas de test correspondant est arrété. Cette opération s'accompagne systématiquement par une renonciation à l'exécution de tous les cas de test commençant par la même chaîne d'appel : [*init ; add(-1)*] .

Exemple : Si nous reconsidérons l'arbre correspondant au schéma sh_1 présenté dans la section, nous aurons un arbre de cette allure :

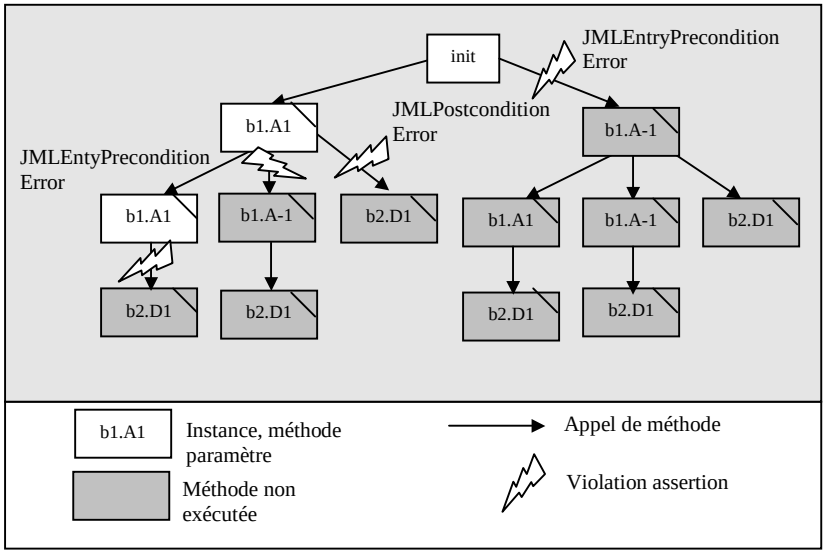


Fig. 17 découpage des branches de l'arbre des cas de test

La branche découpée est dite infectée car elle est composée de séquences invalides.

[8] Une branche contaminée : branche de l'arbre de cade test dont la séquence initiale mène le système à un état incohérent.

Notre hypothèse de travail est que le découpage des branches « infectées » de l'arbre à lui seul assurera par rapport à l'approche JML JUNIT de meilleures performances en terme de temps d'exécution des cas de test. Ce gain sera d'autant plus visible avec des arbres de tailles importantes où les branches coupées correspondent à la renonciation à l'exécution de centaine ou de milliers de cas de test.

Toutes ces optimisations ont été implémentées, testées et validées par des expérimentions effectuées sur l'outil TOBIAS. Nous discuterons les résultats de ces expérimentations dans le chapitre suivant.

Bien qu'elles aient montré leur efficacité, aux cours des expérimentations, les optimisations proposées jusque là nous semblent encore insuffisantes pour le filtrage d'un arbre de cas de test de grande taille (le cas des arbres générés par TOBIAS). En effet, ces derniers même avec le découpage des branches contaminées, demeurent d'une assez grande taille. L'intérêt de leurs appliquer des optimisations est clair d'autant plus que certains d'entre eux renferment dans leurs branches des sous-séquences ou des appels de méthodes qui ont été rejetées lors du parcours des premières branches.

Nous proposons alors des solutions complémentaires à celles qui ont été déjà présentées en partant d'une analyse supplémentaire de la spécification et/ou des types d'erreurs et exceptions rencontrées lors du parcours et de l'exécution des cas de test.

Les propositions qui seront présentées dans les prochaines sections, n'ont pas été implémentées, faute de temps. Nous nous limitons à des propositions théoriques qui nécessitent une étude empirique pour être validées.

3.3.5 Optimisations avancées

L'objectif de cette section est de proposer une approche de filtrage des cas de test exécutés en fonction des erreurs qui surviennent lors du parcours de l'arbre des cas de test. Cette approche requiert des informations supplémentaires que nous puisons dans une analyse approfondie du contenu de la spécification JML. Une attention et un traitement particuliers seront accordés à chaque type d'erreur généré lors de l'exécution des cas de test.

3.3.5.a Cas des erreurs de précondition

La principale lacune de l'approche de découpage des branches, présentée dans les sections précédentes, consiste à faire des retours vers la racine afin de ré-exécuter toute la chaîne d'appel pour les nouveaux cas de test plus sur la même branche et de même préfixe. Ces retours peuvent ralentir le temps d'exécution de l'ensemble des cas de test et pénaliser les performances du pilote de test si les branches de l'arbre sont trop longues.

Exemple : soit le schéma de test suivant :

$Sh_1 := b_1.\text{Add}(a)^{\wedge 1..5}; b_2.\text{Del}(b)^{\wedge 0..1}$

$a \in \{ 1, 200 \}$

$b \in \{ 5, -3 \}$

Avec un tel schéma de test, TOBIAS va générer 186 cas de test. Nous présentons dans la figure 18 la portion d'une branche de l'arbre généré à partir de ce schéma de test.

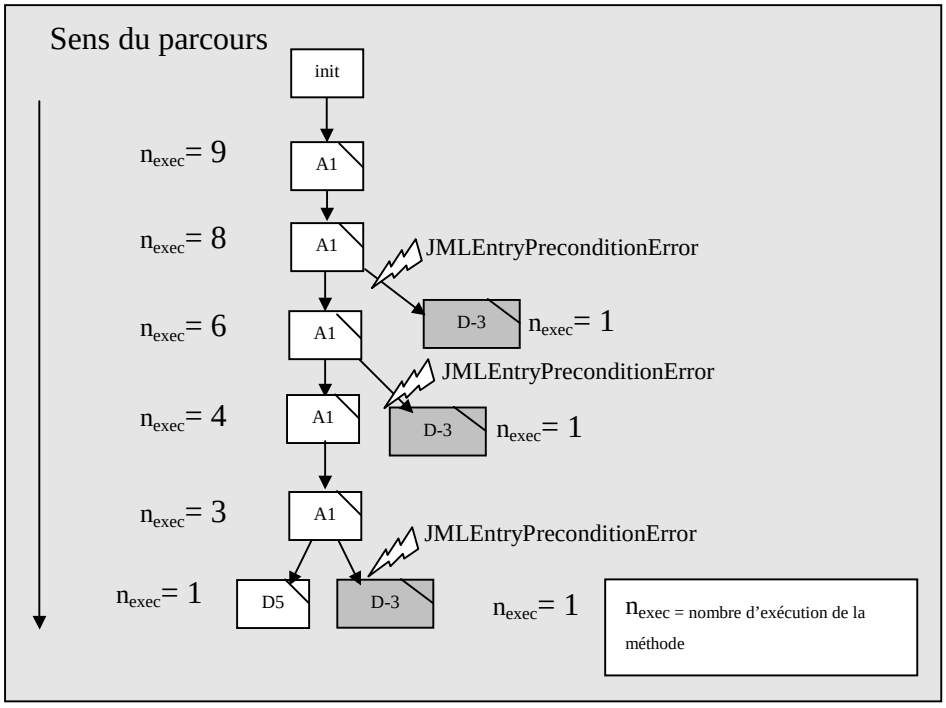


Fig. 18 Limite du parcours en profondeur de l'arbre

Dans cet exemple, nous aurions pu éviter le coût de remonter jusqu'au nœud *Add(1)* (premier fils de la racine) et la ré-exécution de toute la chaîne d'appels menant

jusqu'au nœud *Delete(-3)* si nous avions testé quand on était au niveau de son père immédiat la conformité de ce nœud.

Afin d'optimiser le comportement du pilote de test face à de telles situations, nous proposons de combiner le parcours en profondeur avec un parcours en largeur pour évaluer les préconditions des fils de chaque nœud atteint par l'exécution. Nous estimons qu'une telle stratégie peut assurer des gains de performance considérables et une réduction du nombre des retours en arrière.

La figure 19 illustre un exemple de parcours en profondeur combiné avec un parcours en largeur. Le pilote évalue les préconditions de tous les fils du second nœud A1. Il réalise que l'appel du nœud D-3 n'est pas conforme à la spécification. Il marque alors ce nœud et ne l'exécutera plus. Il gagne ainsi en temps d'exécution en évitant d'exécuter inutilement la chaîne [init ; A1 ;A1].

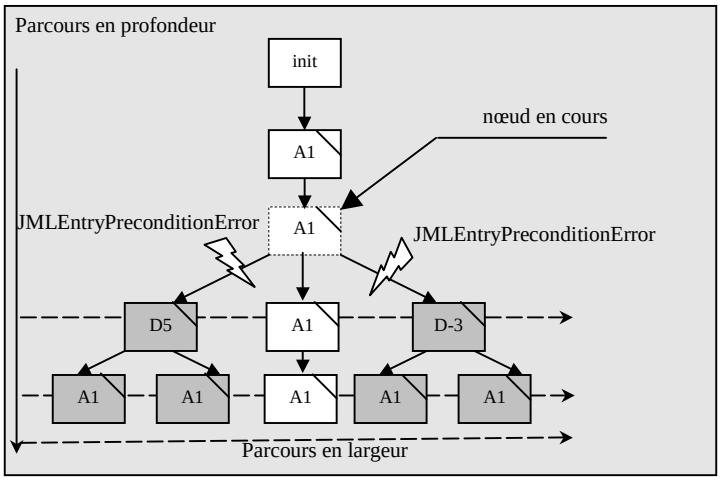


Fig. 19 balayage horizontal des fils du nœud visité et évaluation des préconditions

Nous estimons qu'une telle stratégie peut occasionner des gains de performance significatifs et de réduire le nombre des retours en arrière.

Nous distinguons dans la suite du travail, deux variantes d'erreurs de précondition :

- **Les erreurs de précondition à l'entrée** (JMLEntryPreconditionErrror): relatives aux appels effectués en dehors du domaine de la méthode ;

- **Les erreurs de précondition internes** (JMLInternalPreconditionError): relatives à l'échec des appels internes d'autres méthodes par la méthode exécutées.

3.3.5.a.i Les erreurs de précondition à l'entrée (JMLEntryPrecondition)

Si une erreur de précondition à l'entrée a été levée, l'une des deux situations suivantes peut se présenter :

- La précondition est exprimée en terme de dépendance vis à vis des seuls paramètres ;
- La précondition est exprimée en terme de dépendance vis à vis de l'état de l'objet.

Nous proposons alors de procéder de la façon suivante, selon que l'on rencontre l'un de ces deux cas.

A. Cas de la dépendance vis à vis des seuls paramètres d'appel

La seconde optimisation que nous proposons, se base sur une analyse de la spécification et du schéma de test. Elle est applicable dans le cas où les précondition sont exprimées en terme de dépendance vis à vis des paramètres d'appel de la méthode.

Elle consiste à vérifier la conformité des paramètres d'appel aux contraintes spécifiées sur la valeur des paramètres d'appel. En cas de non-conformité, nous proposons de procéder selon l'une de ces 2 possibilités :

i) Supprimer les paramètres non-conformes et régénérer un nouveau schéma de test avec les « bons » paramètres restants. Cette approche permet de réduire d'avantage le temps d'exécution des cas de test en éliminant les cas où une erreur de précondition relative à la non-conformité des paramètres surviendrait ;

Exemple :

$Sh_1 := b_1.\text{Add}(a)^{\wedge 1..2} ; b_2.\text{Del}(b)^{\wedge 0..1}$

$a \in \{ 1, -1 \}$

$b \in \{ 3, -30 \}$

⇨ *Résultat = 36 séquences*

Re-génération du schéma de test Sh_1*(précondition de Add : a>=0 et précondition de Del : b>=0)* donne le schéma Sh_2 suivant :

$Sh_1 := b_1.\text{Add}(a)^{\wedge 1..2} ; b_2.\text{Del}(b)^{\wedge 0..1}$

$a \in \{ 1 \}$

$b \in \{ 3 \}$

⇨ *Résultat = 9 séquences*

ii) Demander à l'utilisateur de saisir de nouvelles valeurs de paramètres qui soient conformes à la spécification ;

B. Cas de la dépendance vis à vis de l'état de l'objet

Par dépendance vis à vis de l'état de l'objet nous voulons faire référence au cas des spécifications où les préconditions décrivent des conditions sur la valeur des attributs de l'objet en cours.

Contrairement au type d'erreur précédent, la non-conformité de l'état de l'objet par rapport à la spécification ne peut pas être détectée lors de la génération du schéma.

Exemple : Nous reprenons dans le fragment de la spécification de l'interface suivant :

```
1.  /@* public normal_behavior
2.  @ requires X-Y >=3;
3.  @ ensures X=\old(X) && Y=\old(Y);
4.  *@/
5.  void PrintBuffer ( );
6.
7.  /@* public normal_behavior
8.  @ requires b < X+Y ;
9.  @ ensures X=\old(X+Y)-b && Y=\old(X+Y)-b;
10. *@/
11. Void delete(int b);
```

La spécification de la méthode *printBuffer* stipule que cette méthode ne change pas l'état de l'objet en cours. D'autre part, la spécification de la méthode *Del(m)* décrit une dépendance vis à vis de l'état de l'objet.

L'optimisation que nous proposons pour ce type de spécification consiste à propager la vérification au niveau suivant de l'arbre à partir du nœud courant. Nous procédons par un parcours en largeur des fils du nœud dont la spécification décrit un effet nul sur l'état de l'objet courant(Fig 20).

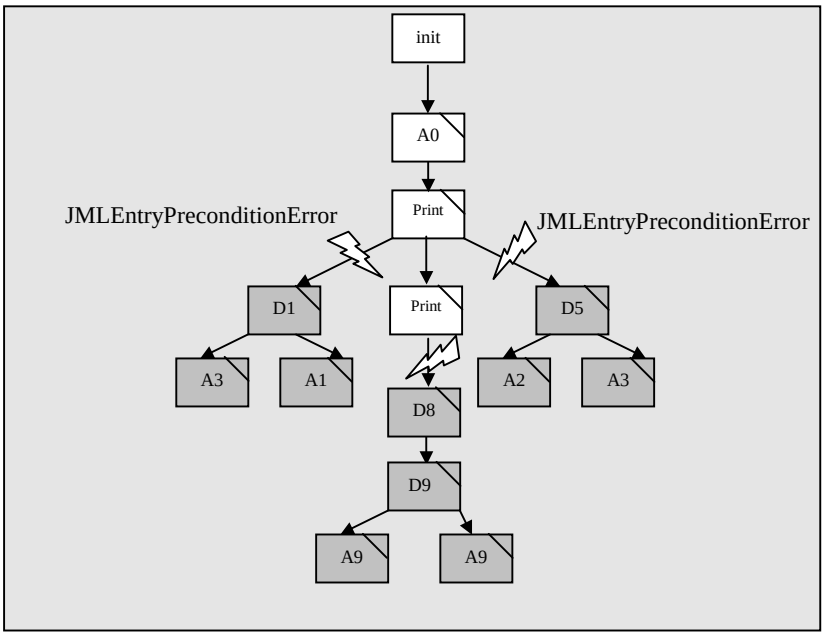


Fig. 20 propagation de l'analyse des *préconditions*

Pour pouvoir examiner la portée de la méthode, du point de vue objets ou variables modifiés, nous utiliserons la clause ***assignable*** contenue dans JML. Cette clause non définie, la méthode est alors considérée comme sans effet sur l'état de l'objet. En effet, une méthode ne doit pas avoir un *effet de bord* [9]sur les *« locations »* qu'elle n'est pas sensée modifier [Ruby-Leavens00] .

Pour garantir une plus grande optimisation de la validation des cas de test, nous suivrons les chemins où les clauses ***assignable*** des méthodes ne sont pas spécifiées(Fig.22). Chaque fois que cette condition est vérifiée, nous propageons la vérification au niveau suivant. Plus les branches et les clauses ***assignable*** signalent un effet nul sur l'état plus le bénéfice de cette approche sera profitable.

[9] Une méthode un effet de bord si elle affecte une location sans que le client d'en rende compte[Ruby-Leavens00]

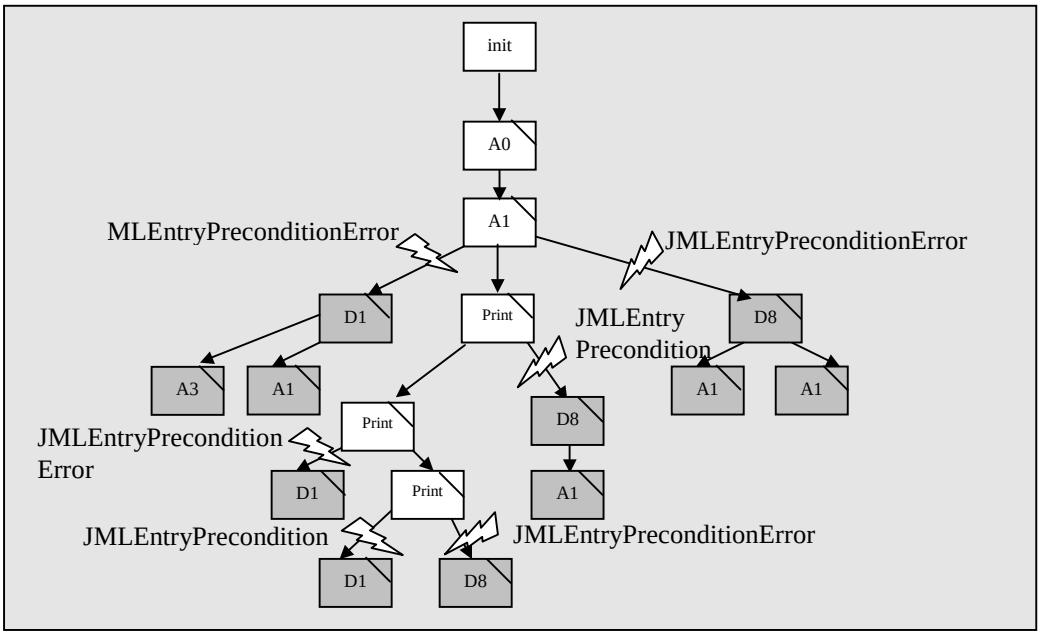


Fig. 21 propagations à plusieurs niveaux

b) Cas des d'erreurs de précondition internes(JMLInternalPrecondition)

Si une méthode *m* appelle une autre méthode *f* en dehors de sa précondition, nous pouvons conclure qu'il y a une faute dans le corps de la méthode *m. N*ous disons qu'elle échoue à implémenter la spécification de ce cas de test et il faudra se demander, à ce moment là, s'il est utile de la tester davantage.

Le choix de continuer ou d'arrêter l'exécution d'une méthode relève d'un choix du testeur et du cadre de test dans le quel il se situe. Dans le cadre d'une approche test XtremeProgramming[10] selon laquelle des tests unitaires sont effectués au fur et à mesure de l'avancement dans la programmation par exemple, il serait intéressant à ce moment là de stopper l'exécution de ces méthodes. Par contre, nous pouvons imaginer une autre approche qui consiste à continuer l'exécution des méthodes bien qu'elles soient non conformes afin de faciliter la tâche du débogueur, par la suite, ou bien pour faire des statistiques sur les types d'erreurs qui surgissent plus souvent.

Nous proposons dans la figure 22 une situation de marquage d'un nœud infecté. Cette dernière est suivie systématiquement par la désactivation des séquences d'appel

[10] http://www.xprogramming.com

qu'elle effectue. Cela a pour effet de réduire davantage la taille et le nombre des cas de test à exécuter par le pilote de test.

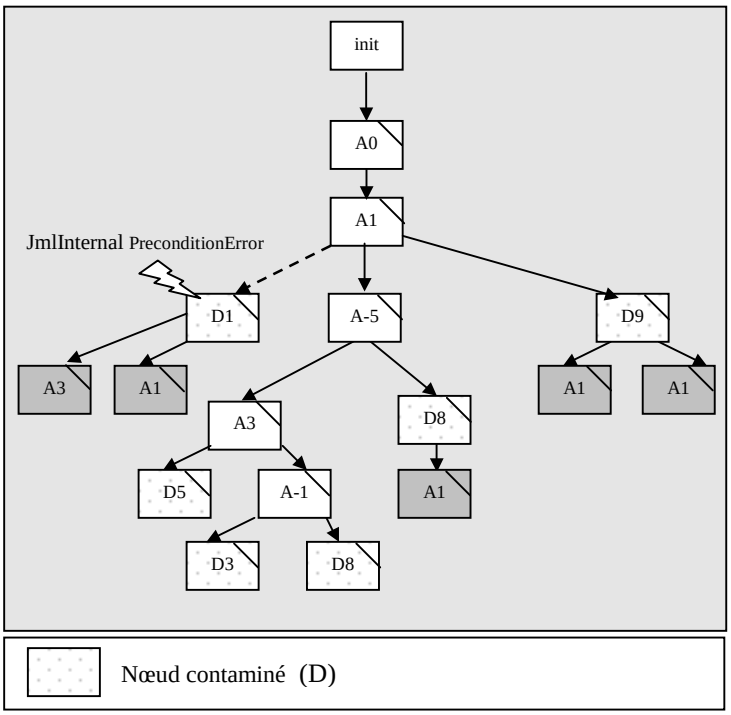


Fig. 22 Marquage des nœuds infectés

3.3.5.b Les erreurs de postcondition /contraintes historiques/invariant

Ces types d'erreurs sont rencontrés dans le cas où la sortie une opération ne correspondrait pas à la sortie attendue décrite par la spécification. Pour garantir que cette non conformité se répète chaque fois qu'on fait un appel nous posons l'hypothèse qu'on travail dans un cadre déterministe. Comme dans le cas des erreurs de préconditions internes, le testeur aura le choix entre arrêter l'exécution de ces méthodes ou continuer à le faire pour des besoins statistiques ou pour de débogage.

3.3.5.c Les erreurs et les exceptions JAVA

Jusqu'à maintenant, nous n'avons discuté que les exceptions levées par JMLC tout en ignorant les erreurs set les exceptions JAVA levées par la machine virtuelle JAVA.

La Figure 23 illustre la hiérarchie, simplifiée, des exceptions en JAVA.D'après cette figure, toutes les exceptions descendent de *throwable*, puis la hiérarchie se sépare immédiatement en deux branches : une branche *Error* et une branche *Exception*.

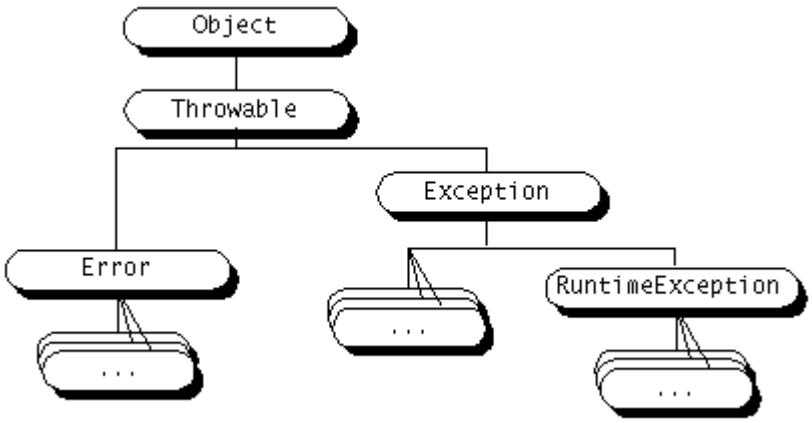


Fig. 23 extrait de la hiérarchie des exceptions JAVA[11]

Exception : Un événement exceptionnel risquant de compromettre le bon déroulement du programme [SUN].

Erreur : la sous classe « Error » regroupe les erreurs graves de la machine virtuelle : Machine virtuelle dans un état instable, récursivité infinie, classe en chargement non trouvée, etc [SUN].

La classe RuntimeException regroupe les divers incidents potentiels tels qu'un envoi de message à une référence nulle, la division par zéro, un indice hors des bornes d'une collection indexée, etc. Tous ces types d'erreur sont récupérés par le pilote de test et le message d'erreur correspondant sera affiché à l'utilisateur.

Nous prêtons, dans cette section, une attention particulière au cas de non-terminaison. Ces erreurs qui ne sont pas détectées par le framework de test JUNIT peuvent influencer considérablement les performances de l'outil en terme temps d'exécution des tests.

Ces erreurs peuvent être évitées par une analyse de la clause ***@@diverge*** de JML. Cette clause renseigne sur la possibilité de non-terminaison de l'appel.

Nous proposons alors de résoudre ce problème en déclenchant un compteur d'exécution dès la rencontre de cette clause. Nous pouvons, aussi, définir ce compteur pour tous les appels de méthodes afin d'éviter les situations de non-terminaison qui ne sont pas prévues dans l'absence de cette clause.

[11] http://java.sun.com

3.4 Conclusion

Dans ce chapitre, nous avons présenté une approche de filtrage des tests à l'exécution. Cette approche tire parti de l'aspect exécutable des assertions JML et des similarités qui caractérisent les tests TOBIAS pour atténuer le nombre des tests produits et pour éviter l'exécution des tests non conformes. Afin de pouvoir tester la validité de cette approche, nous avons été amenés à réaliser un pilote de test. Cet outil ainsi que les résultats de son application sur des tests générés par TOBIAS feront l'objet de la seconde du prochain chapitre.

Chapitre4

Réalisation et expérimentations

Le but de ce chapitre est de mettre en œuvre et de tester la validité de l'approche présentée dans le chapitre 3. Dans un premier temps, nous présenterons le pilote de test que nous avons réalisé conformément à notre approche. Ensuite, nous appliquerons cet outil à l'étude d'une application de gestion bancaire « BankingGemplus » qui nous a été fournie par l'industriel Gemplus[12].

4.1 Présentation du pilote de test

Dans le présent travail nous avons réussi à concevoir et à mettre en place un pilote de test pour les programmes JAVA spécifiés en JML. Son rôle consiste à automatiser l'exécution des cas de test générés par l'environnement de synthèse des cas de test TOBIAS. Combiné avec JML, cet outil permet de résoudre le problème ed l'oracle rencontré avec TOBIAS. Il permet par conséquent de résoudre le problème de l'oracle rencontré avec TOBIAS. Nous avons essayé à travers cet outil de concrétiser l'approche de réduction du nombre des cas de test que nous avons présentée dans le chapitre précédent.

La figure 24 illustre les différentes classes du pilote réalisé.

[12] http://www.gemplus.com/

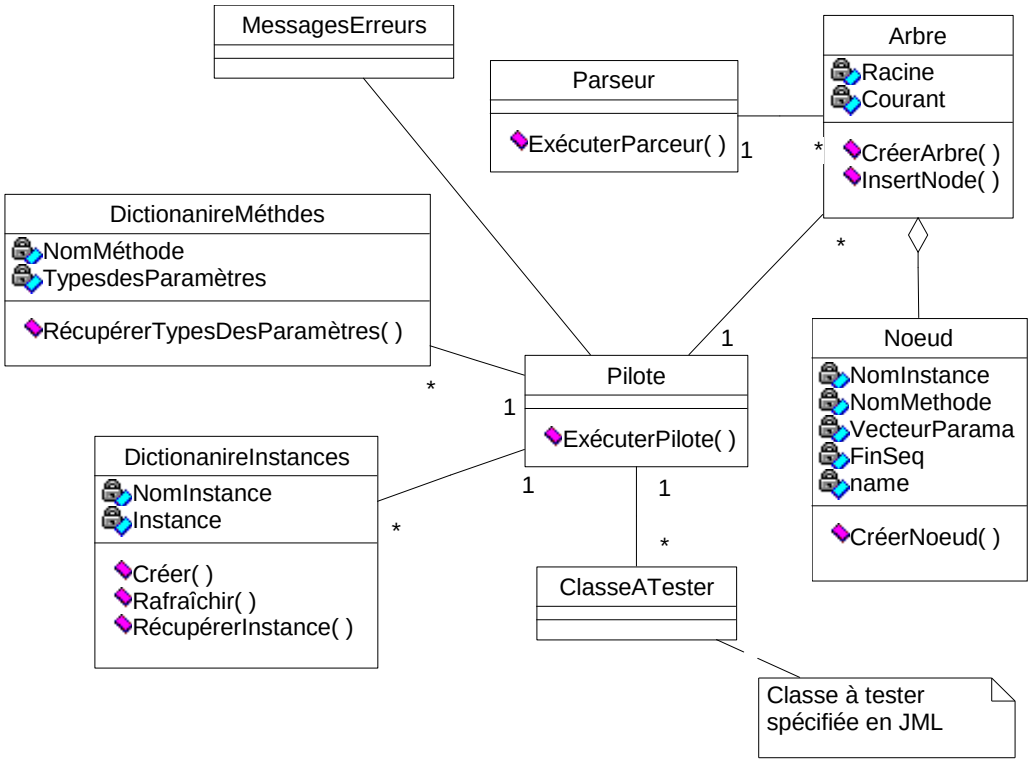


Fig. 24 Diagramme de classes du pilote de test réalisé

Le pilote de test est composé de 7classes :

- ***Pilote*** : C'est la classe centrale de l'outil qui permet de parcourir l'arbre des cas de test et d'exécuter les différentes méthodes. Pour ce faire, elle fait appel aux services de la classes *Dictionnaire des méthodes* pour récupérer les types des paramètres des méthodes. Elle utilise aussi, les services de la classe *Dictionnaire des instances*, pour créer les instances et les rafraîchir ;
- ***Parceur*** :Il parce le fichier *« ots »* et crée l'arbre de cas de test correspondant ;
- ***Arbre*** : Une instance de cette classe correspond à une suite de test (l'ensemble des cas de test générés par TOBIAS à partir d'un schéma de test donné) ;
- ***Nœud*** : Il enveloppe la méthode à invoquer, les paramètres d'appel et l'instance concernée par l'appel ;
- ***Dictionnaire des méthodes :*** Une instance de cette classe regroupe l'ensemble des méthodes concernées par une configuration de test ainsi que le type des paramètres correspondant ;
- ***Dictionnaire des instances :*** Une instance de cette classe regroupe l'ensemble des instances concernées par une campagne de test ;
- ***MessagesErreurs :*** Cette classe regroupe l'ensemble des messages d'erreurs qui seront affichés au cours de l'exécution des tests.

4.2 Principe de fonctionnement

En effet, le pilote commence par récupérer es cas de test générés par TOBIAS et construit l'arbre de cas de test correspondant. Il procède, par la suite, à l'exécution de ces cas de test en parcourant en profondeur l'arbre. Au cours de ce parcours, le pilote fait des retours en arrière vers la racine à la fin de chaque fin de cas de test pour réinitialiser les instances.

L'originalité de cet outil réside dans les optimisations qu'il effectue lors de son parcours et son exécution de l'arbre. En effet, nous avons doté cet outil de la capacité de renoncer à l'exécution des branches non conformes de l'arbre dès que l'exécution de l'appel d'un nœud échoue.

L'exécution des méthodes est effectuée de façon dynamique grâce au mécanisme de l'introspection JAVA. Dès qu'une exception JML est levée, le pilote la récupère et affiche le message d'erreur correspondant. Si un message d'erreur est levé, le cas de test sera considéré comme non valide. Il peut s'agir d'une situation d'échec ou de non conformité du code par rapport à la spécification. Dans tous ces cas, le pilote de test arrête l'exécution sur la branche en cours à partir de l'appel qui a généré l'exception et poursuit son exécution des tests suivants s'ils existent.

Le diagramme de séquence illustrant le fonctionnement du pilote est présenté dans la figure 25.

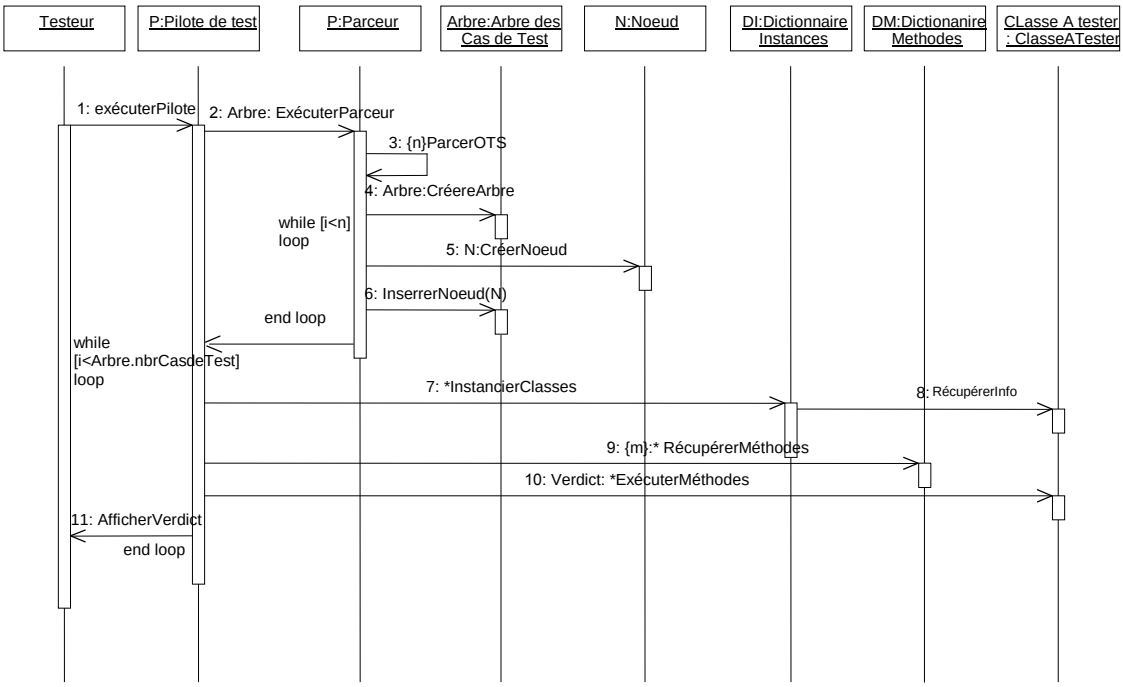


Fig. 25 Diagramme de séquence du pilote de test TOBIAS

4.3 Etude de Cas « Banking- Gemplus »

Nous spécification dans cette section une application de service bancaire notamment l'étude de cas « Banking-Gemplus » qui a fait l'objet d'étude au cours du projet RNTL COTE.Le diagramme de classe correspondant à cette application est présenté dans le la figure 26.

En examinant ce diagramme, nous constatons que l'environnement de l'application est constitué de 3 acteurs à savoir :

- ***Bank Officer (Le banquier) :*** Cet acteur a pour fonction de créer et de détruire des comptes. Sa responsabilité étant de vérifier auparavant la validité de cette création , toute création d'un compte est valide . De la même manière il détruit les comptes ;
- ***Customer (le client)***: Cet acteur peut accéder à son compte (ou ses comptes) en visualisant les soldes. Il peut définir des règles de transfert de fonds en précisant si leur exécution est répétitive ou unique. Il peut aussi demander à visualiser les sommes des règles dans plusieurs devises différentes ;
- ***Timer (Horloge)*** : Cet acteur permet de simuler un mécanisme d'horloge.

Le diagramme de classe « baking-Gemplus » présenté est composé de 7 classes :

- ***AccountMan_sr*** : Cette classe gère la création et la destruction des comptes des utilisateurs par l'acteur *Bank Officer*. C'est dans cette classe que se gère l'unicité de l'identifiant du compte ainsi que la liste des banques autorisées. Par un souci de simplification, nous ne gérons au maximum que10 comptes d'où une association de 0..10 ;

Transfers_src. Cette classe gère les règles de transfert de fonds d'un compte à l'autre ou bien une demande de transfert immédiate. Il s'assure que les règles sont valides avant de les accepter ;

- ***Rule*** : Cette classe permet de faciliter la création de nouvelles règles et surtout leur validation ;
- ***Balances_src*** : Cette classe permet la consultation des soldes des comptes détenus par un utilisateur ;
- ***Currency_src*** : Cette classe joue le rôle de convertisseur de monnaie;.
- ***Account*** : Cette classe définit un compte bancaire. Les comptes sont gérés par un gestionnaire de compte, en l'occurrence la classe *AccountMan_src* ;
- ***SpendingRule*** : Cette classe hérite de *rule* et définit une règle dite "de dépense". Suite à sa création et sous certaines conditions, elle transfère un montant d'un compte à un autre.

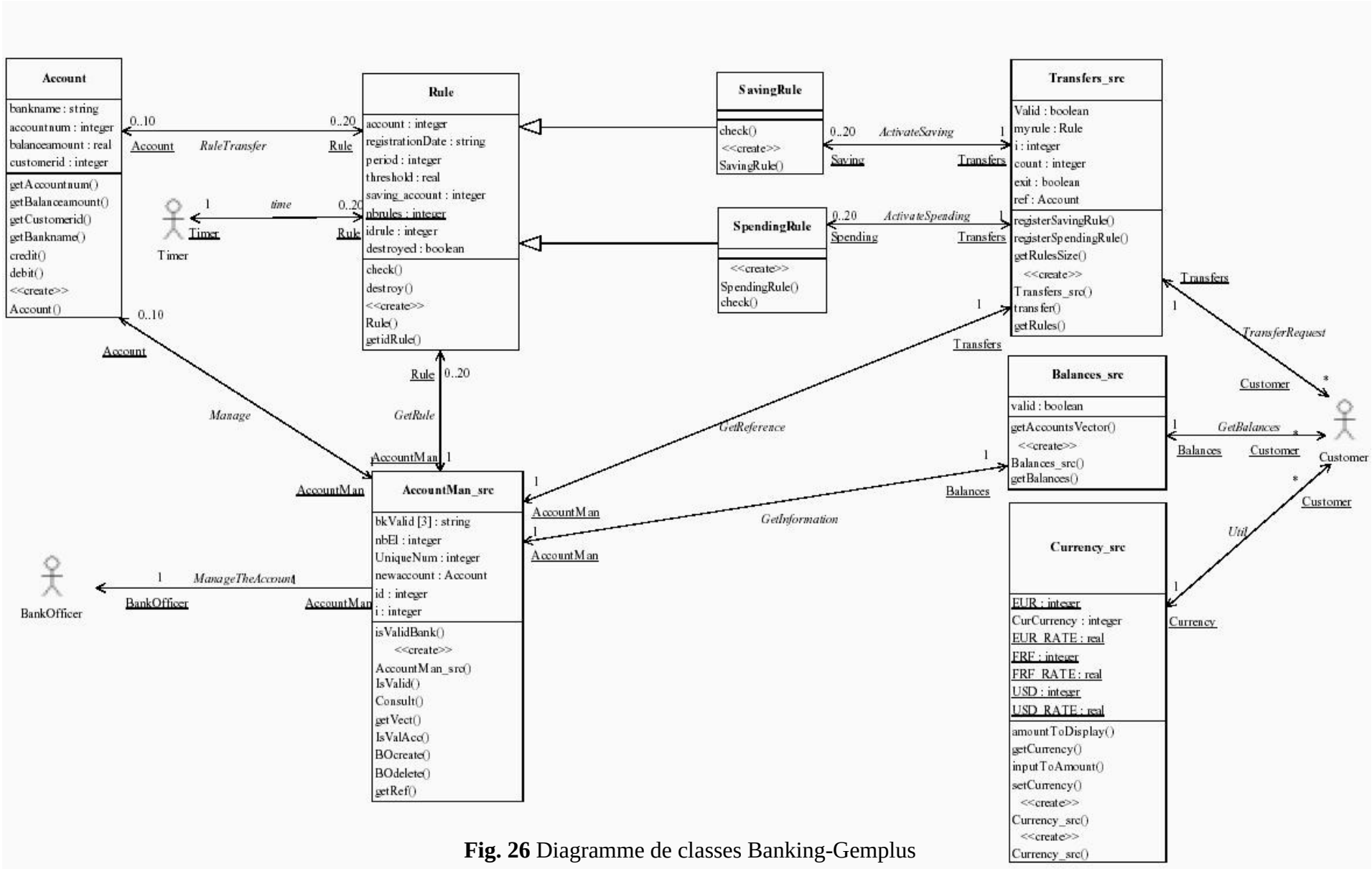


Fig. 26 Diagramme de classes Banking-Gemplus

4.4 Mise en œuvre et expérimentations

Dans cette section nous allons effectuer quelques expérimentations afin de vérifier la validité de notre approche de réduction du nombre des cas de test exécutés. Pour ce faire, nous avons choisi de tester une application de gestion de services bancaires : « Banking-Gemplus »

4.4.1 Configuration du test

4.4.1.a Création des instances :

Au cours cette campagne de test, nous allons tester les 3 classes : *AccountMan_src , Transfers_src, Balances_src* . Nous avons donc créé une instance pour chacune de ces classes.

Classe	Instance
AccountMan_src	CentraleBancaire
Transfers_src	CentraleTransfers
Balances_src	Solde

4.4.1.b Sélection des méthodes :

Les méthodes au quelles nous allons faire référence sont les suivantes :

Méthode	Classe
public int BOcreate(int Cust_id, String bn, float balance)	***AccountMan_src***
public int BOdelete(int AccountNum)	***AccountMan_src***
public int transfer(int from_account, int to_account, float amount)	***Transfers_src***
void prettyPrintBalances(int Cust_id)	***Balances_src***

4.4.1.c Définition des schémas de test

Pour les besoins des expérimentations, nous avons créé les 5 schémas (Fig27, Fig28). Les deux premiers tests de la figure27 sont constitués des même sous séquences et avec le même nombre d'appels pour chaque séquence. Les trois schémas de la figure28 représentent des schémas de différentes longueurs.

Le dernier schéma de la figure29 est l'exemple type d'un schéma *« explosif »*. En effet son dépliage donne lieu à 32736 cas de test.

Sh_1:=MTC!CentraleBancaire. BOcreate (m1,m2,m3)^1..2 ; MTC!CentraleTransfer. transfer(m4,m5,m6)^0..2 ; MTC!Solde. PrettyPrintBalances (m7)^0..2

Sh_2:= MTC!CentraleBancaire. BOcreate (m1,m2,m3)^1..2 ; MTC!Solde. PrettyPrintBalances (m7)^0..2 ; MTC!CentraleTransfer. transfer(m4,m5,m6)^0..2

$m_1 \in \{104\}$
$m_2 \in \{« bnp », « poste »\}$
$m_3 \in \{5000, 1000000\}$
$m_4 \in \{11,10\}$
$m_5 \in \{50,100\}$
$m_6 \in \{-500, 1000\}$
$m_7 \in \{104\}$

Fig. 27 schémas de test avec permuation des appels

Sh_3:= MTC!CentraleBancaire. BOcreate (m1,m2,m3)^1..2 ; MTC!CentraleTransfer. transfer(m4,m5,m6)^1..2

Sh_4:=MTC!CentraleBancaire. BOcreate (m1,m2,m3)^1..2 ;MTC!CentraleTransfer. transfer(m4,m5,m6)^1..4

Différence de taille entre les schémas

$m_1 \in \{104\}$
$m_2 \in \{« bnp »\}$
$m_3 \in \{100000, 1500\}$
$m_4 \in \{104\}$
$m_5 \in \{1000,2000\}$
$m_6 \in \{10000, -200\}$
$m_7 \in \{104\}$

Fig. 28 schéma de test de différentes tailles

Sh_5:= MTC!CentraleBancaire. BOcreate (m1,m2,m3)^1..2 ; MTC!Solde. PrettyPrintBalances(m7)^0..3 ; MTC!CentraleTransfer transfer(m4,m5,m6)^1..5

$m_1 \in \{104\}$
$m_2 \in \{« bnp »\}$
$m_3 \in \{100000, 1500\}$
$m_4 \in \{104\}$
$m_5 \in \{1000,2000\}$
$m_6 \in \{10000, -200\}$
$m_7 \in \{104\}$

Fig. 29 exemple de schéma "explosif"

4.4.1.d Résultats et discussion

Les schémas de test sh1, sh2 ont été dépliés par TOBIAS et ensuite testés avec JML-JUNIT et notre pilote de test. Pour des besoins d'expérimentation nous avons inséré une erreur au niveau du code de méthode *transfer*. Le Tableau.1 illustre les résultats de ce test.

	Sh_1		Sh_2	
	JML-JUNIT	Pilote TOBIAS	JML-JUNIT	Pilote TOBIAS
Nbr total de cas de test	**2900**		**2900**	
Nbr cas de test exécutés	2900	1220	2900	1620
Nbr cas de test échoués	2080	400	2080	800
Temps exécution total*	**615.375**	**205.875**	**768.985**	**335.891**
Taux d'amélioration	**66.54%**		**56.32%**	

Tableau 1 tableau comparatif des résultats du test

La première constatation que nous pouvons faire en regardant ce tableau concerne les performances de notre pilote par rapport à JML-JUNIT. En effet, le nouveau pilote de TOBIAS arrive à réduire significativement le temps d'exécution des tests. Ceci est du à la renonciation à l'exécution des séquences dont la non-conformité est prévisible.

La seconde remarque que nous pouvons faire concerne la position du premier appel erroné dans l'arbre. Il semble que plus cet appel est situé dans un niveau élevé de l'arbre plus les éliminations sont importantes.

	Sh_3		Sh_4	
	JML-JUNIT	Pilote TOBIAS	JML-JUNIT	Pilote TOBIAS
Nbr total de cas de test	**120**		**2040**	
Nbr cas de test exécutés	120	72	2040	360
Nbr cas de test échoués	84	36	126	252
Temps exécution total*	**136.015**	**40.188**	**1111,5**	**200,843**
Taux d'amélioration	**59.6%**		**81.93%**	

Tableau 2 tableau comparatif de schémas de différentes longueurs

Le Tableau 2 illustre deux schémas de test de différentes longueurs. En examinant les résultats, nous constatons un taux d'amélioration croissant en fonction de la tailles des arbres des cas de tests.

Le dernier tableau qui sera présenté est le tableau 3. Il décrit un exemple typique d'un schéma « explosif » dont le dépliage donne lieux à 32736 cas de test. Dans un premier temps, nous avons testé cette suite de test avec notre pilote sans lui appliquer

l'optimisation de découpage de branche, comme le fait JML-JUNIT. Ensuite nous avons testé la même suite en lui appliquant l'optimisation de découpage des branches infectées.

Les résultats de l'expérimentation ont dépassé nos attentes. Nous avons réussi à diviser par 10 le temps de traitement de la suite de test.

Ce tableau vient confirmer l'utilité de notre approche pour le filtrage des tests de très grandes tailles.

	Sh_5	
	Pilote TOBIAS Sans découpage des branches	Pilote TOBIAS
Nbr total de cas de test	**32736**	
Nbr cas de test exécutés	32736	2976
Temps exécution total*	**10146.750**	**820.344**
Taux d'amélioration	**91,91%**	

Tableau 4 schéma "explosif"

4.5 Conclusion

Le but de ce chapitre été d'expérimenter le pilote de test que nous avons implémenté sur des tests qui ont été générés par TOBIAS. Les expérimentations ont porté sur une application de gestion de services bancaires.

Les résultats de l'expérimentation ont été très prometteurs. En effet, au cours de tests effectués, l'outil a occasionné des gains de temps très significatifs. Les gains sont d'autant plus importants avec les schémas explosifs dont le dépliage génère des milliers de tests.

*temps d'exécution en seconde

Conclusion

Le présent travail repose essentiellement sur l'outil TOBIAS. Il s'agit d'un outil de synthèse de tests de conformité qui a permis, lors des expérimentations effectuées dans le cadre du projet RNTL COTE, des gains de productivité en terme de qualité, de coût et de précision des tests produits.

L'idée de base de cet outil consiste à effectuer un dépliage exhaustif de certains patrons de génération (schémas de test) pour produire un grand nombre de jeux de tests en un temps relativement court et à un coût moindre.

Durant notre étude de l'outil TOBIAS, nous avons constaté que le dépliage des schémas de test selon la technique combinatoire ou qui lui est propre, occasionne, dans la plupart des cas, la production d'un très grand nombre de jeux de tests. C'est ce que nous désignons par l'explosion combinatoire des cas de test. Ce phénomène s'accompagne souvent d'un nombre relativement élevé de tests non conformes à la spécification. L'exécution de cette pléthore de tests non conformes ralentit le processus de test et, qui plus est, ne fournit aucune valeur ajoutée à l'ingénieur de test.

Notre objectif a alors été de remédier à cette lacune en filtrant les jeux de tests produits par TOBIAS afin de fournir au testeur un nombre raisonnable de tests pertinents. L'approche que nous avons proposée consiste à filtrer les tests produits au fur et à mesure de leur exécution. Elle part du constat que les tests générés par TOBIAS présentent certaines similarités qui peuvent servir à factoriser certains traitements et filtrer, par conséquent, les tests. L'examen de ces caractéristiques nous a permis de dégager une structure arborescente pouvant être appliquée aux jeux de test produits.

Cette structure favorise l'exécution des séquences les plus courtes d'abord. Elle permet également d'éviter l'exécution des plus longues qui commencent par des séquences erronées détectées lors des premières exécutions.

Afin de pouvoir juger la conformité des tests, nous avons combiné TOBIAS à JML. Ce dernier se présente comme un langage de spécification à base d'assertions (*invariant, précondition et postcondition*) exécutables permettant grâce à un évaluateur d'assertions à l'exécution de tester la conformité du code par rapport à la spécification. Ce langage nous a fourni d'autres pistes pour réduire d'avantage les tests de TOBIAS. Ces pistes nous ont permis de procéder à la propagation des vérifications à plusieurs niveaux de profondeur dans l'arbre en naviguant entre les cas de tests dont l'exécution ne modifie pas l'état de l'objet en cours.

Afin de pouvoir exécuter ces jeux de test, nous avons réalisé un pilote qui construit des arbres de test, les exécute et récupère les résultats de l'exécution. Nous avons doté ce pilote de la capacité de couper les branches invalides ou non conformes de l'arbre, au cours de son parcours.

Cependant, nous avons relevé une lacune dans cette approche. Il s'agit des retours en arrières à la fin de chaque test pour exécuter les prédécesseurs du dernier nœud du test courant. Pour pallier cette faiblesse, nous avons proposé de combiner les parcours en profondeur avec des parcours en largeur afin d'évaluer la validité des préconditions du fils du nœud courant.

L'autre limite de notre approche réside dans le fait qu'elle n'est applicable que dans un cadre déterministe.

Nous envisageons, au cours des prochains travaux, de mettre en œuvre les autres optimisations proposées et d'en tester la validité. Nous pensons notamment à la re-génération des schémas de test conformément à la spécification, à l'évaluation en largeur des préconditions et à la navigation à travers les appels qui ne modifient pas l'état de l'objet testé lors de leur invocation.

Nous comptons également, généraliser notre approche et l'étendre à des cas où le comportement est non déterministe. Il s'agit en particulier des systèmes parallèles.

Les techniques qui seront utilisées seront pour la plupart des techniques de **validation**. Nous citons essentiellement :

-**Des techniques de test de conformité** : Nous avons l'intention, au cours des prochains travaux, de mettre en œuvre les autres optimisations proposées et de tester leur validité. Il s'agira notamment à la re-génération des schémas conformément à la spécification avant l'exécution des tests et de la propagation des évaluations à plusieurs niveaux de profondeur dans l'arbre ;

-**Des techniques probabilistes** : Nous envisageons aussi d'utiliser les propriétés de la spécification afin de réduire le nombre des tests produits en sélectionnant des sous-ensembles de tests grâce à des techniques de tirage au sort ou de choix heuristique.

Nous projetons de combiner ces techniques avec d'autres **techniques de vérification** afin de démontrer la conformité de certaines branches de l'arbre et d'éviter par conséquent son test. Nous estimons que ces techniques permettront d'une part d'atténuer l'explosion combinatoire des tests TOBIAS et d'acquérir un degré de confiance plus élevé vis à vis de la conformité de l'implantation de l'autre.

Bibliographie

[BARESI01] L. Baresi and M. Young,***"Test Oracles".***Technical Report CIS-TR-01-02, August 2001.

[Bartetzko01] D. Bartetzko, C. Fischer, M. M¨oller , H. Wehrheim , ***"Jass – Java with Assertions"***. In K. havelund and G. Rosu, editor, ENTCS, volume55(2). Elsevier Publishing, 2001.

[Beck98] K. Beck,E. Gamma,***"Test infected: Programmers love writing tests"***.Java Report, 3(7):37–50, 1998.

[Bernot91] G. Bernot,M-C. Gaudel et B. Marre ***"Softwrae Testing Based on Formal Speifications :A theory and a Tool"***. *Software Engineering Journal*, November 1991.

[Bousuqet01] L. du Bousquet, H. Martin, and J-M. Jézéquel, "***Conformamance Testing from UML specification"***, Experience Report.In Gesellschaft fur Informatic, editor, *p-UML workshop*, Lecture Notes in informatics, volume P-7, page 43-56, Toronto, 2001.

[Burdy03] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. Rustan, M. Leino, and E. Poll," ***An Overview of JML Tools and Applications***". Department of Computer Science, University of Nijmegen, NIII-R0309, Nijmegen, March 2003. See http://www.cs.kun.nl/research/reports.

[Cheon 02] Y. Cheon, G. T. Leavens, ***"A Runtime Assertion Checker for the Java Modeling Language (JML)".*** In Hamid R.Arabnia and Youngsong Mun (eds.), International Conference on Software Engineering Research and Practice (SERP '02), *Las Vegas, Nevada*. CSREA Press, June 2002, pages 322-328.

[Cheon02b] Y. Cheon and G. T. Leavens, ***"A Simple and Practical Approach to Unit Testing: The JML and JUnit Way".*** In Boris Magnusson (ed.), *ECOOP 2002 - Object-Oriented Programming, 16th European Conference, Malaga, Spain, June 2002, Proceedings*. Volume 2374 of *Lecture Notes in Computer Science*, pages 231-255. Springer-Verlag, 2002.

[CHEON03] Y. Cheon, Gary T. Leavens, M. Sitaraman, and S. Edwards. ***"Model Variables: Cleanly Supporting Abstraction in Design By Contract"***. Department of Computer Science, Iowa State University, TR #03-10, March 2003.

[Cheon03] Y. Cheon,***"A Runtime Assertion Checker for the Java Modeling Language".*** Department of Computer Science, Iowa State University, TR #03-09, April 2003.

[Den98] D. Peters and D. L. Parnas," ***Using test oracles generated from program documentation".*** *IEEE Transactions on Software Engineering*, 24:161–173, 1998.

[Duncan98] A. Duncan, U. Hölzle, ***"Adding Contracts to Java with Handshake".*** Technical Report TRCS98-32, UC Santa Barbara, Compaq Digital Systems Research Center, December 1998.

[Ernst01] M. D. Ernst, J. Cockrell, Wi.G. Griswold, and D. Notkin, ***"Dynamically discovering likely program invariants to support program"***. evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, 2001.

[Findler00] R. B. Findler, M. Felleisen,***" Behavioral Interface Contracts for Java".*** (Rice University CS TR00-366), Department of Computer Science; Rice University , USA, August, 2000.

[Flanagan01] C. Flanagan , K. Rustan M. Leino,***"Houdini, an annotation assistant for esc/java"***. In J. N. Oliveira and P. Zave, editors, FME 2001, volume LNCS, 2021, pages 500 – 517. Springer, 2001.

[Flangan02] C.Flanagan, K. Rustan M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R.Stata,***" Extended static checking for Java".*** *In Programming Language Design and Implementation (PLDI'2002),* pages 234–245, 2002.

[Gau95] M. Gaudel,***"Testing can be formal, too".*** In Proceedings of TAPSOFT'95, volume 915 of LNCS, Springer, 1995.

[Ham95] Dick Hamlet,***" Software quality, software process, and software testing".*** *In Marvin V. Zelkowitz, editor,* Advances in Computers, vol.\ 41*, pages 191-229. Academic Press, 1995.*

[Ham96] D. Hamlet, ***"Predicting dependability by testing".*** *International Symposium on Software Testing and Analysis*, 1996.

[Hoare69] C.A.R. Hoare. "***An Axiomatic Basis for Computer Programming***". *Communications of the* ACM, 12(10), October 1969.

[Hoare72a] C. A. R. Hoare,***"Proof of correctness of data representations".*** *Acta Informatica*, 1(4):271--281, 1972.

[Jard01] Auteurs : C. Jard, T. Jensen, T. Jéron, JM. Jézéquel, S. Pickin, L. Van Aertrick, L. Du Bousquet, Y. Ledru, ***"Etat de l'art sur la synthèse(génération automatique) de tests" .*** Rapp Prg. RNTL-COTE, Septembre 2001.

[Karaorman99] M. Karaorman, U. Holzle, and J. Bruno, ***"jContractor: A reflective Java library to support design by contract".***In P. Cointe, editor, *Meta-Level Architectures and Reflection, Second International Conference on Reflection* '99, Saint-Malo, France, July 19–21, 1999, Proceedings, volume 1616 of Lecture Notes in Computer Science, pages 175–196.Springer-Verlag, July 1999.

[Karmer98] R. Kramer, ***"iContract – the Java design by contract too"***. *TOOLS 26: Technology of Object-Oriented Kanguages and Systems, Los Alamitos, California*, pages 295–307, 1998.

[Leavens00] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs, ***"JML: Notations and tools supporting detailed design in Java"***. In OOPSLA 2000".Companion, Minneapolis, Minnesota, pages 105–106. ACM, October 2000.

[Leavensb03] G.T. Leavens, A. L. Baker, and C. Ruby,***"Preliminary design of JML: A behavioral interface specification language for Java".***Technical Report 98-06u, Iowa State University,Department of Computer Science, April 2003.

[Ledru02] Y. Ledru. "***The TOBIAS Test Generator and Its Adaptation to Some ASE Challenges"*** (position paper). In *Workshop on the State of the Art in Automated Software Engineering, ICS Technical Report UCI-ICS-02-17*, Université de Californie à Irvine, USA, 2002.

[Liskov01] B. Liskov, J. Guttag, ***"Program development in JAVA : Abstraction, Specification and Object-Oriented Design***".Addison Wesley,2001.

[Maury02] O. Maury, Y. Ledru,***"Using TOBIAS for automatic generation of VDM test cases"***. *In VDM workshop, Copenhagen*, Danemark, 2002.

[Meyer92] B. Meyer " ***Eiffel, le langage*** ". InterEditions Paris, 1994.

[Meyer92b] B. Meyer, ***"Applying Design by Contract"***. Computer, 1992. 25(10): p. 40-51.

[Meyerb92] Meyer, B., *Applying* ***"Design by contract"***. Computer, 1992. 25(10): p. 40-51.

[Norris95] M. Norris et P. Rigby, ***"Conception et qualité du logiciel".*** ed. AFNOR, 1995.

[Pat00] Patricia D.L. Machado, ***"Testing from Structured Algebraic Specifications: The Oracle Problem"***, Université Edinburgh, 2000.

[PETERS98] D.K. Peters and D.L. Parnas, "***Using test oracles generated from program documentation".*** *Proc. Third, IEEE Transactions on Software Engineering*, 24(3):161–173, March 1998.

[PMBY02] P. Bontron and O. Maury and L. du Bousquet and Y. Ledru and C. Oriat and M.-L. Potet. "***TOBIAS : un environnement pour la création d'objectifs de test à partir de schémas de test"***.In *14th International Conference Software & Systems Engineering and their Applications-ICSSEA'2000*, Paris, France, 2001.

[Poll00] E. Poll, J. van den Berg, and B. Jacobs, "***Specification of the JavaCard API in JML"***, In the Fourth Smart Card Research and Advanced Application Conférence(CAEDIS). Kluwer Academic Publishers, 2000.

[Raghvan00] A. D. Raghavan, "***Design of a JML documentation generator***". Technical Report 00-12, Iowa State University, department of Computer Science, July 2000.

[Ruby-Leavens00] C. Ruby ,G. T. Leavens**,** "***Safely Creating Correct Subclasses without Seeing SuperclassCode***". In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, pp. 208-228. Volume 35, number 10 of *ACM SIGPLAN Notices*, October, 2000.

[SUN] SUN **Sun** Microsystems
www.java.sun.com

[Watkins02] J . Watkins, " ***Test logiciel en pratique***", Vuibert , Paris, 2002.

[Xanthakis00] S. Xanthakis, P. Régnier, C. Karapoulinos, "***Le test des logiciels***", HERMES Sceinces pulications, paris, 2000.