UNIVERSITÉ JOSEPH FOURIER - GRENOBLE I
UFR INFORMATIQUE ET MATHÉMATIQUES APPLIQUÉES

THESE

pour obtenir le titre de

Docteur de l'Université Joseph Fourier

Discipline : Informatique

présentée et soutenue publiquement

par

Pierre BONTRON

le 1^{er} mars 2005

Les schémas de test : une abstraction pour la génération de tests de conformité et pour la mesure de couverture

Composition du Jury :

J.-C. Fernandez	**Président**
J.-L. Lanet	**Rapporteurs**
T. Jeron	
Y. Ledru	**Examinateurs**
L. du Bousquet	
M.-L. Potet	

Remerciements

Mon principal soutient durant cette thèse a été celui de mes directeurs de thèse qui ont su tout au long de ces années m'encourager, me pousser et me guider tout en préservant mon autonomie. Chacun a su intervenir à sa façon pour m'aider dans la rédaction de cette thèse, ainsi je tiens à remercier :

- Marie-Laure Potet pour sa rigueur et son sens critique qui ont donnés une vraie dimension à ces travaux.
- Lydie du-Bousquet qui m'a apporté ses connaissances du monde du test logiciel et pour ses encouragements constants.
- Yves Ledru pour avoir accepté d'encadrer cette thèse, ainsi que pour sa motivation et ses nombreux retours envers mes travaux.

L'ensemble du personnel administratif du laboratoire Logiciel, Système et Réseaux, Lilianne, Pascale, Martine, Christiane, François et Gilles, m'a permis de travailler dans de très bonnes conditions en toutes circonstances (même lors un effacement impromptu de la thèse).

Je tiens également à remercier M. Jean-Louis Lanet et M. Thierry Jéron pour avoir accepté de rapporter cette thèse et pour les nombreuses remarques qu'ils ont fait et qui ont contribuées à améliorer le document de thèse. Je remercie également M. Jean-Claude Fernandez de m'avoir fait l'honneur de présider le jury de thèse.

Cette thèse représente plus de 4 ans de ma vie. Quatre années où j'ai toujours pu compter sur mes parents Anne et Christian, ainsi que sur mon frère Guillaume avec qui j'ai été en colocation pendant 3 ans, chacun sur sa thèse (thèse qu'il a su mener à bout plus rapidement que moi pour me montrer le chemin à suivre). Enfin je tiens à remercier mes nombreux amis qui m'ont permis de garder une "vraie vie" en parallèle de cette thèse et qui m'ont toujours incité à aller de l'avant, Manu, Aline, Sophie, Magali, Sylvia, Marguerite, Carole, Marilyne, Carine, Clémence, Sébastien, Jérôme (qui a su me seconder dans des moments difficiles), Francis, Cyril, Karim-Cyril, Tanguy, Christophe, Jean-Christophe, Jean-Gilles, Olivier G., Romain, William, Fabien, Thierry, Éric, Rémi.

J'aurais une dernière pensée pour Olivier Maury avec qui j'ai partagé les moments les plus forts de cette thèse (de déprime, mais aussi de motivation et d'exaltation) durant les trois premières années.

II

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Lors du développement d'un logiciel, nous pouvons distinguer quatre phases : la définition des spécifications dans un cahier des charges, le codage du logiciel, la validation du logiciel et éventuellement la maintenance du logiciel après qu'il ait été délivré. Le degré d'investissement dans la phase de validation varie en fonction du temps et des sommes pouvant y être consacrées, ainsi que du niveau de criticité du logiciel. Dans le cadre d'un logiciel qui peut, entre autre, mettre en péril la vie d'un homme, on doit obtenir la meilleure validation possible. C'est le cas par exemple de logiciels de contrôle de centrales nucléaires ou de systèmes ferroviaires [BDM97]. Ces types de logiciels sont appelés des *logiciels critiques*.

Contexte

On peut distinguer la validation statique de logiciel et la validation à l'exécution. La preuve est, par exemple, une technique applicable à des spécifications formelles, telles que des spécifications exprimées en B [Abr96], qui permet de garantir statiquement certaines propriétées. Ce type de technique de validation ne s'applique pratiquement qu'aux systèmes critiques car elle peut s'avérer très coûteuse en termes d'hommes/jours du fait qu'elle demande un grand niveau d'expertise de la part des ingénieurs. Une autre technique à laquelle nous nous intéressons est le test de logiciel. G. J. Myers définit le *test* de la façon suivante :

"Testing is the process of executing a program with the intent of finding errors." [Mye79]

Pour satisfaire à cette définition, le testeur doit construire des séquences d'exécutions qui permettent d'identifier des erreurs du programme. Dans la suite, j'appellerais ces séquences d'exécution *tests*, et l'ensemble des tests utilisés pour tester un logiciel *suite de test*. La difficulté du travail de l'ingénieur de test réside entre autre dans cette phase de création de suites de test, nommée *élaboration des tests*. Cette activité peut s'avérer très longue et le test peut représenter plus de 50% du temps de développement d'un logiciel. Pour réduire cet effort, il existe deux approches : automatiser l'exécution des tests et automatiser leur élaboration. Dans cette thèse, nous nous intéressons à l'élaboration des tests et aux moyens d'exprimer les tests de manière à réduire le temps consacré à leur écriture.

"Tests are formal procedures. Inputs must be prepared, outputs predicted, tests documented, commands executed, and results observed . . ." [Bei90]

B. Beisier présente l'activité de test avec notamment les phases d'élaboration des tests et d'exécution des tests. Il y a toutefois une partie de l'activité de test dont il ne parle pas, la décision de mettre fin à l'activité de test. C'est la *condition d'arrêt*. La fin d'une activité de test peut être commandée par des impératifs d'ordre financier, ou par manque de temps, mais ces conditions ne permettent pas de qualifier et quantifier le résultat. Nous nous intéressons à la définition d'une condition d'arrêt basée sur un critère de couverture des tests sur une spécification du logiciel. Une *couverture de test*, basée sur une spécification donnée, consiste à définir un rapport (souvent exprimé en terme de pourcentage) entre la partie de la spécification testée et l'ensemble de la spécification pour une suite de test suivant un critère considéré. Suivant la criticité du logiciel testé, les critères choisis varient. Par exemple la norme DO-178B [Inc92] impose différents niveaux de couverture pour les niveaux A et B de sécurité.

Contributions

Le contexte de cette thèse met en avant deux facettes du test, l'élaboration des tests ainsi que les critères d'arrêt du test. Les contributions de cette thèse portent sur ces deux points.

Le travail sur l'élaboration des tests a été effectué dans le cadre du projet

RNTL[1] COTE[2]. Le projet COTE est basé sur différents niveaux d'abstraction d'expression des tests. Un constat fait sur les similarités de nombreux tests [Mar01, dBMJ01] nous a incité à définir un niveau d'abstraction nouveau pour exprimer les tests : *les schémas de test*. Pour utiliser les schémas de test nous avons créé *l'outil TObiAs*[3] [BMdB$^+$01a]. Cet outil offre une interface pour décrire les schémas de test et permet de synthétiser de nombreux tests à partir d'un schéma de test. Dans cette démarche, nous avons exploré différents couplages de TObiAs avec d'autres outils. Ces couplages permettent d'aider au choix des valeurs de test avec l'outil Casting, ou d'exécuter les tests générés à travers des outils tels que VDMTool ou JUnit, pour les langages VDM [ML02] et Java [LdBMB04].

La deuxième partie du travail a consisté à définir d'une notion de couverture de la spécification par les schémas de test. Cette démarche s'inscrit toujours dans l'approche permettant de réduire le travail du testeur. En effet, le fait de ne pas avoir de notion de couverture, et donc de critère d'arrêt, au niveau des schémas de test oblige le testeur qui utiliserait cette notion à "naviguer" entre les différents niveaux d'abstractions (schémas de test et tests abstraits ou exécutables) tout au long de la phase de test pour décider du moment où mettre fin à la suite de test. Nous proposons une approche de *couverture a priori* basée sur les schémas de test. Cette couverture, dite "a priori", permet au testeur de savoir s'il peut arrêter sa tâche d'élaboration des tests avant l'exécution même des tests. Le testeur peut donc effectuer tout son travail d'élaboration des tests à un seul niveau, celui des schémas de test.

Pour valider cette notion de couverture, nous avons effectué une étude formelle mettant cette notion de couverture en relation avec des notions de couverture établies au niveau des tests abstraits [BP03, BP04]. Tout comme nous avons outillé les schémas de test, nous avons aussi outillé notre notion de couverture avec *l'outil CoPAS*.

---

[1] Réseau National des Technologies Logicielles

[2] "COmponent TEsting". Ce projet s'est déroulé d'octobre 200 à octobre 2002 et était précompétitif.

[3] Test OBjectif desIgn ASsistant

Plan du mémoire

Ce mémoire est structuré en trois parties. La première partie présente dans le chapitre 2 le contexte du test avec les diverses formes de test existantes ainsi que certaines techniques de synthèse de test. Nous positionnerons notre approche à la fin de ce chapitre. Le chapitre 3 est consacré à la présentation du projet COTE et permettra de mieux comprendre certains choix faits au cours de cette thèse.

La deuxième partie correspond à la première contribution de la thèse, à savoir la réduction de l'effort d'élaboration des tests grâce aux schémas de test. Les travaux présentés dans cette deuxième partie ont été effectués en collaboration avec un autre doctorant : O. Maury. Dans le chapitre 4 nous présentons les schémas de test avec leur syntaxe et leur sémantique, ainsi que le processus qui permet de générer des objectifs de test à partir d'un schéma de test [BMdB+02]. Nous introduisons aussi l'outil TObiAs [BMdB+01a] qui permet de structurer les schémas de test et de générer des tests à un niveau d'abstraction moindre. Le chapitre 5 présente différentes utilisations de l'outil TObiAs, comme avec le langage VDM [ML02] ou en le couplant avec l'outil Casting pour générer automatiquement des valeurs de test. Ce chapitre fait aussi le point sur certaines améliorations possibles telle qu'une meilleure couverture des diagrammes UML, ou la mise en place de filtrage des tests générés [LdBMB04] pour éviter un problème d'explosion combinatoire. Le dernier chapitre de cette partie présente deux études de cas menées avec l'outil TObiAs : une étude de cas menée par l'entreprise Gemplus et par notre laboratoire dans le cadre du projet COTE [dBLM+04] et une étude de cas d'un système d'ascenseurs simplifié qui sera poursuivie dans la partie 3. Ces études permettent d'évaluer les apports méthodologiques de TObiAs et d'analyser les résultats produits pour déterminer l'efficacité de TObiAs et ses lacunes.

La troisième partie est consacrée à la seconde contribution de la thèse : la définition d'une couverture *a priori*[4] au niveau des schémas de test. Le chapitre 7 présente la notion de couverture de test ainsi que différents critères de couverture possibles. Nous préciserons la notion de *couverture des branches* que nous utilisons pour ces travaux en justifiant ce choix. Dans le chapitre 8 nous formalisons les différents niveaux d'abstraction des tests (tests abstraits, objectifs de test et schémas de test) en choisissant de représenter les tests sous la forme de

[4]La couverture est évaluée avant l'exécution des tests.

Input Output Labelled Transition Systems (IOLTS) réduits à des séquences, c'est à dire non branchus. Nous présentons pour chaque niveau d'abstraction comment s'exprime la couverture de test. Le chapitre 9 est consacré à la définition d'une seconde notion de couverture associée aux schémas de test [BP03]. Cette notion est indépendante des tests abstraits et objectifs de test, elle se base sur une abstraction de la spécification obtenue en utilisant les mêmes mécanismes d'abstraction que ceux utilisés dans les schémas de test. Nous définissons une relation, que nous démontrons, entre les deux notions de couverture que nous avons définies. Le chapitre 10 reprend la deuxième étude de cas du chapitre 6 en introduisant l'outil CoPAS. Ce chapitre présente aussi les avantages et inconvénients que peut avoir l'utilisation de CoPAS sur le couple d'outils TObiAs/TGV pour la génération de tests abstraits.

Le dernier chapitre de ce mémoire est consacré à une synthèse du travail effectué et aux perspectives qu'ouvre ce travail.

Première partie

Le cadre : le projet COTE et le test

Chapitre 2

Le test de logiciel : état de l'art

Les pratiques associées au test sont toutes aussi diverses que les logiciels peuvent l'être. Ainsi nous n'utilisons pas les mêmes méthodes pour tester des systèmes synchrones ou des systèmes asynchrones, des systèmes d'information ou des systèmes embarqués, un compilateur ou un site de commerce électronique.

Dans la première section de ce chapitre nous présentons les différentes étapes liées à l'élaboration des tests. Les deuxième et troisième sections introduisent des techniques d'automatisation des étapes de construction des test que nous nommons synthèse de test. Nous présentons la synthèse de test pour deux types de spécification que nous avons manipulés : les spécifications à base de pré et post conditions, et les spécifcations à base d'IOLTS. Enfin la quatrième section introduit plus précisément les techniques et formes de test utilisées dans le travail présenté dans cette thèse.

2.1 Formes de test

En introduction à cette thèse nous avons présenté l'activité de test comme étant découpée en trois phases : l'élaboration des tests, l'exécution des tests et la décision d'arrêter la phase de test. Parmi ces trois phases, celle de l'élaboration des test est sans doute la plus critique, c'est durant cette phase qu'est décidé de quelle façon le logiciel va être testé. Cette phase peut être découpée en plusieurs étapes. Une première étape consiste à définir quelle *phase de test* nous souhaitons mettre en place (test unitaire, d'intégration, ...) en fonction de la phase de développement du logiciel. Une deuxième étape consiste à définir le *type de test* à appliquer (conformité, robustesse, ...). Une troisième étape consiste à définir des valeurs de test répondant à certaines stratégies ou hypothèses de test.

2.1.1 Les phases de test

Généralement le développement d'un logiciel se compose de plusieurs phases de conception. A chacune de ces phases, on peut associer une catégorie de test adaptée.

Test unitaire

"Testing of individual software units or groups of related units." [IEE90]

Un système à tester est souvent composé d'éléments qui interagissent entre eux. Le test unitaire consiste à tester ces éléments de manière indépendante. Il existe différentes façons de concevoir des tests unitaires. Dans certaines applications la granularité est le test d'une méthode. Dans d'autres applications on considère un composant/une classe comme une unité. Le test unitaire peut aussi bien être d'ordre structurel que fonctionnel. Le testeur doit décider de ce qu'il considère comme une unité : une procédure, une méthode, une classe, un composant, ...

Le test unitaire est habituellement effectué avant de tester le système dans sa globalité. Il est en effet plus simple de trouver une erreur située dans un composant en testant ce seul composant plutôt que tout le système.

Test d'intégration

"Testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them." [IEE90]

Le test d'intégration fait suite au test unitaire. Le test d'intégration a pour but de trouver des erreurs liées à la mise en interaction des éléments unitaires. Le test d'intégration permet de détecter des erreurs associées à une mauvaise communication entre les interfaces, telle que des passages de valeurs hors limites, des mauvaises signatures d'interface ou bien des problèmes de visibilité entre les composants.

Test système

"Testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements." [IEE90]

Le test système offre une vision globale du système là où le test d'intégration s'intéressait aux connexions d'interfaces entre éléments du système. Les tests systèmes sont effectués par les concepteurs du système et s'assurent de la cohérence globale de celui-ci d'un point de vue fonctionnel.

Test de recette

"Test effectué par l'utilisateur pour voir si l'application répond à ses besoins."

Le test de recette est une activité contractuelle. Ce type de test considère l'ensemble du système du point de vue du client ou de l'utilisateur final, c'est le pendant du test système. Les tests de recettes permettent de s'assurer du bon fonctionnement du logiciel par rapport aux spécifications du cahier des charges. Le cahier des charges peut contenir des scénarios d'utilisation pré-établis qui servent de test de recette. Le test de recette conditionne l'acceptation du produit par le client.

2.1.2 Les types de test

Les types de test se rapportent aux aspects du logiciel à tester (fonctionnel, performances, ...).

Test de conformité

"Protocol conformance testing is a kind of testing where an implementation of a protocol entity is tested with respect to its specification. The aim is to gain confidence in the correct functioning of the implementation with respect to a given specification, ..." [Tre91]

Le test de conformité consiste à tester le système considéré vis-à-vis de sa spécification. Le but du test de conformité est de valider les comportements de l'implantation vis-à-vis des comportements attendus, et repose sur une notion de *relation de conformité*. Le choix de cette relation dépend du contexte, du type de système étudié et des hypothèses de test mises en place. Le test de conformité peut aussi bien s'effectuer en ayant connaissance ou non du code de l'application, et plusieurs stratégies pour définir les tests peuvent être adoptées. Par exemple pour la sélection des valeurs, nous pouvons utiliser les stratégies de test aléatoire ou aux limites, présentées ci-après.

Test de robustesse

"Un système/composant logiciel est dit robuste vis-à-vis d'une propriété si et seulement si celle-ci est préservée lors de toutes exécutions, y compris en dehors des conditions nominales. Plus précisément, le système garde un fonctionnement acceptable en présence d'entrées invalides, de dysfonctionnements internes, de conditions de stress, de pannes, etc." [Pac03]

Le test de robustesse est à opposer au test de conformité. Le but n'est plus de s'assurer que l'implantation respecte la spécification, le but est d'observer le comportement du logiciel dans des conditions non spécifiées. Certains logiciels doivent pouvoir faire face à des données "hors normes" sans se bloquer. Le test de robustesse consiste à utiliser le logiciel avec des valeurs qu'il ne devrait pas recevoir.

Ce type de test peut s'appliquer notamment dans le cadre des logiciels critiques afin de s'assurer que des données hors normes n'impliquent pas le blocage du système et que les conditions de sécurité sont préservées. Un autre exemple est celui de logiciels bancaires, ou de logiciels grand public, où l'utilisation candide ou malveillante ne doit pas faire de dégât suite à une fausse manœuvre.

Test en charge

"Testing conducted to evaluate the compliance of a system or component with specified performance requirements." [IEE90]

Le test en charge consiste à recréer les conditions d'utilisation du logiciel dans le cadre d'une utilisation intensive ou typique (pour mesurer des performances). Par exemple dans le cas d'un système d'exploitation, ou d'un intergiciel, le testeur simule de nombreuses requêtes à des informations ou encore des changements dans une base de données. Ce type de test permet de s'assurer que l'architecture, aussi bien matérielle que logicielle, du système peut supporter la charge correspondant par exemple à un envoi de données d'une taille conséquente.

Test de non régression

"Regression testing is that testing that is performed after making a functional improvement or repair to the program. Its purpose is to determine if the change has regressed other aspects of the program." [Mye79]

Dans le cadre du développement d'un logiciel, on peut être amené à en développer plusieurs versions. Chaque version est testée avec des tests appropriés aux fonctionnalités ajoutées, mais aussi avec un ensemble de tests déjà utilisés dans la version précédente. Ces tests sont appelés tests de non régression, ils permettent de s'assurer que les fonctionnalités sont conservées malgré la modification de certaines parties du système.

Ce type de test est différent des autres car il ne constitue pas un "type de test" au même sens que le test de conformité, de robustesse ou en charge. Un test dit de robustesse peut être désigné comme étant aussi un test de non régression. Les tests de non régression sont un sous-ensemble des tests défini pour chaque type de test créé.

2.1.3 Les stratégies de sélection des valeurs

Quand tous les paramètres définissant le type de test à mettre en œuvre sont définis, il reste à définir les valeurs que peuvent prendre les tests. Pour cela différentes approches peuvent être combinées.

Test aux limites et classes d'équivalence

"Experience shows that test cases that explore boundary conditions have a higher payoff than test cases that do not. Boundary conditions are those situations directly on, above, and beneath the edges of input equivalence classes and output equivalence classes." [Mye79]

Ce type de test revêt la forme d'une stratégie de test qui porte sur les données, en partant du principe qu'un logiciel a plus de chance de dévoiler ses erreurs lorsqu'on utilise des valeurs limites de classes d'équivalence.
Les classes d'équivalence regroupent des valeurs sur lesquelles est faite l'hypothèse que toutes les valeurs produisent les mêmes erreurs. Donc si une valeur amène à détecter une erreur, toutes les valeurs de cette même classe auraient mené

à cette erreur. Si au contraire cette valeur ne permet pas de détecter une erreur, aucune valeur de la classe considérée ne le permet.
Les intervalles de valeurs sont des classes d'équivalence particulières. Un test aux limites correspond à tester les valeurs extrêmes de l'intervalle considéré. Par exemple, si une variable entière peut appartenir à l'intervalle $[-1000 \ldots 1000]$, l'ingénieur de test testera le logiciel avec les valeurs -1000 et 1000 dans le cadre du test de conformité[1]. On peut espérer ainsi détecter des débordements de tableau ou de boucles par exemple. Les valeurs -1, 0 et 1 sont aussi des valeurs intéressantes à tester du fait que ces valeurs ont un comportement "particulier" vis-à-vis des opérateurs arithmétiques. Dans le cas précédent l'intervalle des valeurs serait partitionné en trois classes d'équivalences : $[-1000 \cdots -1]$, $[0]$ et $[1 \ldots 1000]$.

Test aléatoire

"[..] the process of testing a program by selecting, at random, some subset of all possible input values." [Mye79]

L'apport du test aléatoire est controversé dans la communauté scientifique du test [Mye79]. Néanmoins le test aléatoire a des avantages qui en font un bon complément à d'autres formes de test :
- le test aléatoire est peu coûteux et facile à mettre en place. Il permet de produire de grandes quantités de valeurs de test ;
- le test aléatoire peut détecter un nombre d'erreurs conséquent pour un moindre coût [DN81, HT90] ;
- le test aléatoire peut permettre d'évaluer la fiabilité d'un programme [HT90, Ham94].

Avec cette technique de test on peut aussi définir une distribution statistique sur les valeurs pouvant être sélectionnées. Cette technique permet d'orienter les tests vers des séquences plus intéressantes pour l'ingénieur de test. On parle de test statistique [TFWY95].

Test combinatoire

"Combinatorial testing performs combinations of selected input parameters values for given operations and given states."[LdBMB04]

[1]Dans le cadre du test de robustesse le testeur aurait essayé les valeurs -1001 et 1001.

Cette technique de test a pour but de tester de la façon la plus exhaustive possible le système étudié en utilisant dans les tests produits toutes les combinaisons possibles d'ensembles de valeurs. Cette approche a comme faiblesse le fait de conduire naturellement à une explosion combinatoire du nombre de tests, qui la rend vite impraticable pour des systèmes pouvant utiliser des ensembles de valeurs de grande taille.

2.2 Synthèse de tests à partir de spécifications à base d'assertions

Certaines spécifications se présentent sous la forme de prédicats décrivant des pré et post conditions associées à des méthodes, des invariants, ou des gardes associées à des transitions. Ces prédicats peuvent être utilisés pour calculer des valeurs d'entrées valides du programme, ou une séquence d'appels valides.

Ces spécifications sont le propre de langages tels que B, Z ou VDM, et plus récement OCL ou JML. Des travaux ont été menés par Dick et Faivre [DF93] sur une technique automatique d'analyse des partitions obtenues par la mise sous forme normale disjonctive des pré et post conditions. Ces travaux ont permis de mettre au point des environnements de test automatique tel que ProTest [SLB05] et BZ-Testing-Tools [LPU02] que nous allons présenter dans la section suivante. Nous présenterons aussi Casting qui se base sur des techniques de décomposition de prédicats portant sur des diagrammes d'états-transitions, ce qui permet d'utiliser cet outil avec d'autres types de spécifications que des spécifications formelles comme B ou Z.

2.2.1 L'environnement BZ-Testing-Tools

BZ-Testing-Tools[2] utilise des spécifications exprimées en B ou en Z. Cet environnement offre deux modes de fonctionnement : l'animation du modèle et la génération de tests fonctionnels aux limites [LP01]. Ces deux fonctionnements utilisent le même moteur [BLP00, BLPP00] formé des deux solveurs de contraintes : CLPS et CLPS-BZ. CLPS met en œuvre les algorithmes de Dick et Faivre [DF93] qui permettent de partitionner les contraintes en les passant sous une forme normale disjonctive. CLPS-BZ [BLP02] est dédié à l'interprétation du format BZP (B/Z Prolog format) et traduit les expressions et opérations du langage

[2]Par la suite nous nommerons cet environnement BZTT

BZP en contraintes primitives qui forment les entrées du solveur CLPS. L'utilisation de BZTT peut se découper en quatre points que nous allons détailler avant de présenter les limitations liées à l'outil.

Traduction de la spécification
Durant cette phase le fichier source est traité de façon à extraire les informations nécessaires au format BZP. C'est à partir de ces informations que l'animateur contraint de BZTT peut être exécuté [PLT00].

Calcul des valeurs aux limites
Les variables d'état de la spécification et les paramètres des opérations sont contraints par des prédicats (invariant, pré-conditions). Ces expressions sont traduites en contraintes primitives traitées par le solveur CLPS. Ces contraintes sont passées sous la forme normale disjonctive, chaque terme de la forme disjonctive correspond à une partie du domaine des variables ou des paramètres d'entrée. Les valeurs extrêmes de ces parties sont calculées par maximisation/minimisation. Ces valeurs sont les buts aux limites.

Génération des tests abstraits
Les tests se décomposent en quatre parties :
- **Le préambule :** la partie du test abstrait qui amène le système de son état initial jusqu'à un état limite. Un état limite est un état où au moins une variable d'état a une valeur limite.
- **Le corps :** la partie contenant l'appel d'une opération avec les valeurs limites comme entrées.
- **L'identification :** la partie du test consistant à observer certains aspects du système en appelant des méthodes d'observation. C'est en fonction des valeurs observées que l'oracle peut associer un verdict au test.
- **Le postambule :** la partie du test qui ramène le système dans son état initial. Cela permet d'exécuter de façon automatique plusieurs tests à la suite.

La figure 2.1 présente un test abstrait avec ses différentes parties.

Génération automatique de tests exécutables
L'un des points forts de BZTT est d'automatiser la génération de scripts de test à partir des tests abstraits générés [BL03]. Pour cela l'ingénieur de test doit tout de même définir deux entrées : un patron de script de test et une table de correspondance. Le patron de script est un fichier exprimé dans le langage cible comportant

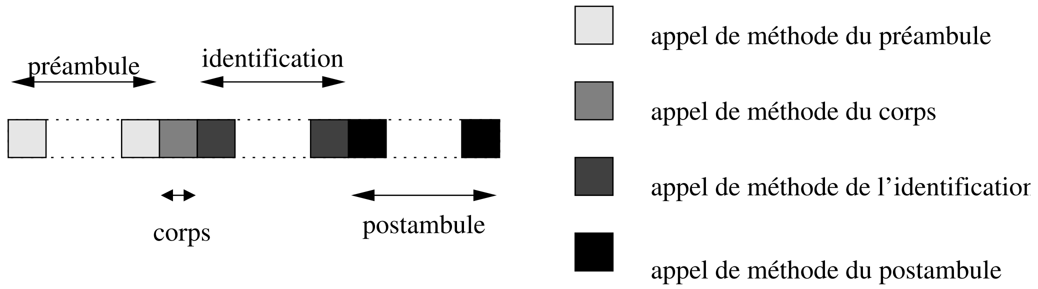


FIG. 2.1 – architecture d'un test abstrait

des balises indiquant où insérer les différentes parties d'appels de méthode du test qui a été défini (préambule, corps, identification et postambule). La table de correspondance comporte trois types d'informations : les méthodes et variables substituées du model abstrait (obtenu de manière automatique), les variables et méthodes observables, et le corps des opérations exprimé dans le langage cible.
La figure 2.2 présente les différentes étapes de la génération de script de test exécutables. Les boites en pointillé sont produites par l'ingénieur de test (ou avec son aide), les boites rectangulaires correspondent à des fichiers et les boites ovales à des programmes.

Les restrictions d'utilisation
BZTT impose certaines restrictions sur la spécification en entrée. Les structures de données doivent être finies et de tailles raisonnables. Dans le cadre d'une spécification exprimée en B, une seule machine peut être définie. Une spécification Z doit être exprimée sous la forme d'un seul schéma comprenant l'initialisation et les opérations de la spécification. Toutes les opérations doivent avoir des pré-conditions explicites. Cette restriction implique en Z de calculer les pré-conditions de chaque opération au préalable et de les ajouter au schéma. La spécification doit aussi être déterministe, afin de procéder à la génération de test, et être au même niveau d'abstraction que le programme à tester.

2.2.2 L'outil Casting

L'outil Casting [ABM97b, ABM97a, Aer98] a pour objectif de construire des tests à partir de spécifications à base de prédicats complétées par un diagramme d'états-transitions. Celui-ci exprime les séquences d'appel valides.

Cet outil se base sur un diagramme d'états transitions du composant à tester.

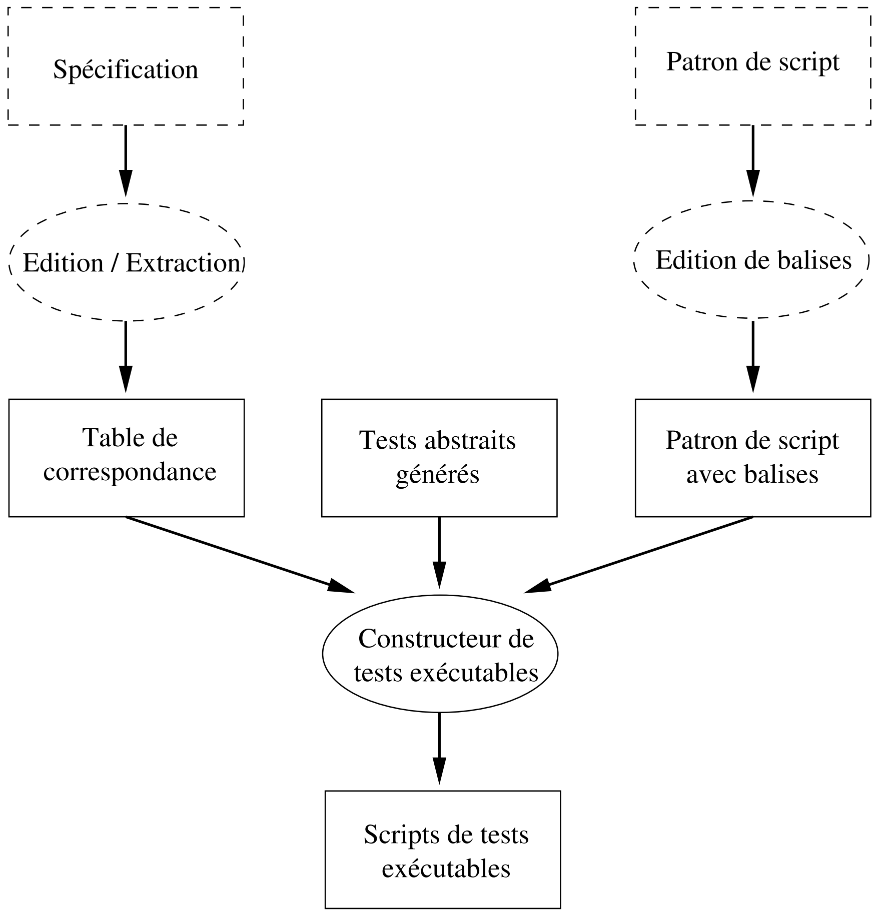


FIG. 2.2 – les étapes de la génération des scripts de test

Les contraintes qu'il exploite peuvent venir, soit de machines exprimées en langage B [Abr96], soit de spécifications OCL [WK98]. Casting s'appuie sur les prédicats associés aux opérations dans le cas du langage B ou aux contraintes exprimées dans un sous-ensemble d'OCL. L'utilisation de Casting est basée sur l'application de stratégies de décomposition des prédicats portant sur les transitions du diagramme d'états-transitions. Ces stratégies sont choisies par le testeur et ont pour but de dégager des comportements que l'ingénieur de test peut trouver intéressants. Par exemple, si un prédicat comporte le symbole "$\geq$", l'ingénieur de test peut vouloir différencier les cas "$>$" et "$=$". Casting opère en deux phases. La première consiste à appliquer les stratégies de décomposition à la spécification. La deuxième phase s'appuie sur un solveur de contraintes

permettant de trouver un ensemble de valeurs d'entrée qui activent un chemin donné dans la spécification.
Nous reviendrons sur Casting dans le chapitre 5.

2.3 Synthèse de tests à partir de spécifications comportementales

D'autres spécifications prennent la forme d'une description comportementale, notamment sous la forme d'un IOLTS. Pour effectuer de la synthèse de test à partir de ce type de spécification, certains outils utilisent des algorithmes de parcours de graphe qui permettent de définir des séquences de test. Après une présentation des IOLTS, nous présentons deux outils : Torx et TGV pour illustrer ces techniques.

2.3.1 Présentation des IOLTS

Les méthodes de synthèse de test fondées sur les systèmes de transitions (LTS pour « Labelled Transition System ») ont leur origine dans les travaux sur les équivalences comportementales et préordre [NH84, Abr87]. De façon simpliste, une instance sous test (nommée IUT par la suite) est équivalente à sa spécification si pour tout autre LTS observateur (le testeur), les observations (traces et blocages) que l'on peut faire par composition parallèle du testeur et de l'IUT sont incluses dans les observations possibles du testeur sur la spécification.

La différence entre entrées et sorties est importante dans le cadre du test, elle nécessite un modèle plus riche. Plusieurs de ces modèles ont été utilisés pour le test dont les Input Output Automata (IOA) de Lynch [Lyn88], les Input Output State Machines (IOSM) de Phalippou [Pha94], et les Input Output Transition Systems (IOTS) de Tretmans [Tre95]. Les transitions de ces modèles sont soit des entrées, soit des sorties, soit des actions internes. Ces modèles sont tout à fait similaires ainsi que les travaux qui en ont découlé.
Nous considérons dans la suite un modèle appelé système de transitions à entrée/sortie (IOLTS pour « Input Output LTS »). Ce modèle est une extension des LTS où on distingue trois types d'actions observables portant sur les transitions : les entrées, les sorties et les actions internes. D'un point de vue théorique, les IOLTS et des variantes de ce modèle sont utilisés pour modéliser la plupart des

objets intervenant dans la génération de test : la spécification, l'IUT, les objectifs de test et les cas de test. Sur un IOLTS les actions en entrée sont précédées par un !, les actions de sortie par un ? et les actions internes sont représentées par un τ. La figure 2.3 présente un IOLTS.

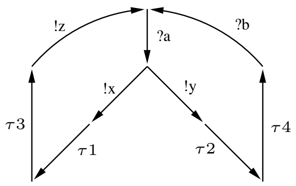


FIG. 2.3 – Représentation d'un IOLTS

Une spécification peut comporter des branchements menant à des blocages du système. Le blocage ou absence d'action (quiescence) peut être du à plusieurs causes facilement identifiables sur un IOLTS : le blocage complet (ou deadlock) : le système ne peut plus évoluer), le blocage de sortie (le système est bloqué en attente d'une entrée venant de l'environnement), ou le blocage vivant (ou livelock : le système diverge par une suite infinie d'actions internes). Il est intéressant de considérer les livelocks sur les systèmes finis qui correspondent simplement à des boucles d'actions internes. Les différents blocages sont illustrés dans la figure 2.4 suivante.

La définition 1 page 22 correspond à un IOLTS. Cette définition sera reprise et complétée d'un point de vue comportemental dans la partie 8.2.

2.3.2 L'outil Torx

L'outil Torx [FPdV01] a été développé dans le cadre du projet *Côte de Résyste* [BBF+01] dont le but était de développer des méthodes, techniques et outils pour le test de conformité de systèmes réactifs. Dans ce type de systèmes la communication entre les composants s'effectue sous la forme de messages. Un message est une communication effectuée entre l'environnement et l'implantation sous test. Torx génère des tests à partir d'une spécification décrite dans un langage tel que SDL [BH89], MSC [IT94], LOTOS [ISO89], ou Protocol Meta Language Promela [Hol91]. L'outil Torx travaille “à la volée”, c'est-à-dire qu'à chaque instant l'outil n'a connaissance que d'une petite partie de la spécification. Les

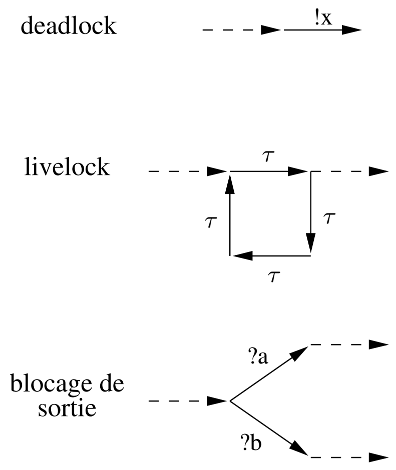


FIG. 2.4 – Les différents blocages visible dans un IOLTS

tests sont dérivés pas à pas et directement exécutés sur l'implémentation. Torx sert aussi d'oracle. En effet, pour chaque message émis par l'implantation, Torx le compare aux messages admis par la spécification. Une fois cette validation effectuée, Torx émet un nouveau message à l'attention de l'implantation en tirant au hasard l'un des messages autorisés dans l'état courant par la spécification.

2.3.3 L'outil TGV

L'outil TGV [FJJ$^+$96, Mor00, JJ02] nous intéresse plus particulièrement puisqu'il fait partie de la chaîne d'outils du projet COTE. Cet outil a été développé en collaboration par l'INRIA, le laboratoire Vérimag et l'équipe PAMPA de l'IRISA. TGV permet de produire de manière automatique des cas de test à partir d'une spécification comportementale et d'un objectif de test. L'objectif de test est un IOLTS donnant une description abstraite de comportements à tester, nous l'utilisons comme un critère de sélection des tests. Un test abstrait est un IOLTS contrôlable[3] qui ne possède que des actions observables et dont chaque état final est associé à un verdict. TGV effectue, à la volée, un produit synchrone entre la spécification et l'objectif de test.A partir de ce produit, TGV effectue la suppression des états internes, la déterminisation et la résolution des conflits de

[3]il peut à tout moment, soit faire une sortie, soit attendre toutes les entrées possibles

Définition 1 (IOLTS)

$$S \hat{=} (Q^s, A^s, T^s, q^s_{init})$$

Avec
– Q^s l'ensemble fini des états ;
– A^s l'alphabet des labels ;
– $T^s \subseteq Q^s \times A^s \times Q^s$ l'ensemble des transitions ;
– $q^s_{init} \in Q^s$ l'état initial.
L'alphabet A^s se partitionne en trois sous-ensembles : A^s_i (input) l'ensemble des messages d'entrée, A^s_o (output) l'ensemble des messages de sortie et A^s_τ (interne) l'ensemble des actions internes.

contrôlabilité.

Si l'objectif de test est réalisable, TGV renvoie un test abstrait correspondant à cet objectif de test et valide vis-à-vis de la spécification. Plusieurs cas de test peuvent correspondre à un même objectif de test et TGV peut être configuré de façon à obtenir un ensemble de cas de test.

2.4 Notre positionnement

Le projet COTE, présenté dans le chapitre suivant, se place dans le cadre du test fonctionnel de conformité. Le projet s'intéresse aux aspects comportementaux des spécifications. Il a pour objectif d'automatiser le processus de test, en effectuant de la synthèse de test.

Le projet COTE a adopté une approche de construction des tests basée sur une spécification comportementale en s'appuyant sur l'outil TGV. Les techniques de génération de test sont basées sur l'utilisation d'IOLTS.

Nous avons combiné l'outil de synthèse de test TGV avec une approche combinatoire pour permettre la génération de nombreux objectifs de test avec un effort moindre. Cette approche combinatoire est présentée dans le chapitre 4 consacré aux schémas de test et à l'outil TObiAs.

Le chapitre suivant présente le projet COTE. Ce projet, qui avait pour but la mise en place d'un environnement de synthèse et de génération de test centré autour d'UML, a servi de cadre à cette thèse durant les deux premières années.

Chapitre 3

Le projet COTE

Introduction

Ce chapitre est consacré au projet RNTL[1] COTE [COT]. Ce projet précompétitif réunissant les entreprises France Télécom Recherche et Développement, Gemplus, Softeam et les laboratoires IRISA et LSR-IMAG a duré deux ans (d'octobre 2000 à octobre 2002). Le but du projet était de créer une plate-forme de synthèse de test basée sur la notation UML.

Le projet COTE[2] est le support principal de cette thèse. Il est donc important de bien comprendre le contexte qui nous a amené à faire ce travail afin de justifier certains choix.

L'approche adoptée dans le projet COTE pour la génération des tests correspond à l'approche MDA[3] [BGMR03, OMG] qui consiste à définir un modèle de façon indépendante d'une plate-forme donnée. Ensuite pour chaque plate-forme visée, des outils permettent de transformer ce modèle abstrait en un modèle spécifique à la plate-forme. L'approche choisie dans le projet COTE repose sur le même processus en proposant différents niveaux de granularité pour décrire les tests, et ceci d'une façon indépendante d'une plate-forme cible.

Une autre particularité de l'environnement COTE est de permettre de décrire

[1] Réseaux National des Technologies Logicielles
[2] Ce projet est à l'origine du financement de cette thèse.
[3] Model Driven Architecture

les tests dans un cadre unifié, celui d'UML. Une partie du langage UML a été adaptée à la description de tests. Le projet COTE a ainsi contribué au groupe de travail de l'OMG chargé de définir un profil UML [OMG02a] pour le test par l'intermédiaire de Softeam et de l'IRISA.

La section 3.1 introduit la notation UML ainsi que la façon dont ce langage a été utilisée pour décrire les tests. La section 3.2 présente les différents niveaux d'abstraction qui ont été définis dans le projet COTE pour décrire des tests. Enfin la section 3.3 décrit l'architecture logicielle de l'environnement COTE et les différents outils.

3.1 La notation UML dans le projet COTE

Le développement d'une application nécessite une phase importante en amont de la programmation permettant de spécifier, documenter et poser les contraintes du système à réaliser. C'est durant cette phase qu'est définie entre autres la spécification de l'application. Celle-ci doit être validée par les différentes parties prenant part au projet, dont le client. Plusieurs niveaux de langages permettent d'exprimer la spécification :

- **Les langages informels.** Ce type de langage a l'avantage d'être compréhensible par tous mais le risque d'ambiguïté fait que les intervenants n'ont pas forcément la même interprétation de la spécification. On peut aussi noter que ces langages sont difficilement exploitables par des outils, leur sémantique étant complexe. Le langage naturel rentre dans cette catégorie.
- **Les langages semi formels.** Ils offrent généralement des notations graphiques avec une syntaxe clairement définie. Néanmoins ces langages sont dotés d'une sémantique qui peut donner lieu à différentes interprétations et d'un pouvoir d'expression limité. De telles notations ont l'avantage d'être assez intuitives et d'offrir une grande lisibilité. L'inconvénient est que ce type d'expression laisse encore une part assez importante à l'interprétation. Ainsi, suivant leur culture, deux personnes peuvent comprendre différemment le même diagramme. Le langage UML est un bon exemple de langage semi formel.
- **Les langages formels.** Ils offrent une sémantique très rigoureuse et une syntaxe qui permet d'exprimer des concepts de manière très précise sans aucune ambiguïté. Ces langages demandent un bon niveau d'expertise et

restent donc des outils de spécialistes. Les langage B [Abr96] et Z [Abr77] sont des langages formels.

La notation UML[4] est un **langage semi formel** dédié à la description de modèles. Ce langage se veut intuitif et lisible, et prend en compte différentes facettes du logiciel. La notation UML permet de représenter des aspects statiques, dynamiques ou bien encore matériels. Le nombre de diagrammes (ou de vues) que propose la notation UML 2.0 [OMG03] est de neuf, avec trois autres mécanismes d'organisation des modules. Ces diagrammes se répartissent en deux catégories : quatre représentent la structure statique de l'application (diagrammes de classes, d'objets, de composants et de déploiement) ; cinq représentent le comportement dynamique (diagrammes de cas d'utilisation, de séquences, d'activité, de collaboration et d'états-transitions). Les mécanismes pour organiser et gérer les modules qui composent le programme sont au nombre de trois, le mécanisme de packages, les sous-systèmes et les modèles. Les deux derniers mécanismes sont dérivés du mécanisme de package. La notation UML est renforcée par un langage de contraintes nommé OCL [WK98] qui permet de définir des assertions formelles comme des invariants, des pré et post conditions.

Le projet COTE utilise essentiellement trois types de diagrammes. Les diagrammes de classes et d'états-transitions permettent de définir les spécifications et les diagrammes de séquences sont utilisés pour décrire les tests. Nous avons aussi utilisé les **diagrammes d'objets** pour représenter le déploiement initial d'une application et des objets de test pour chaque campagne de test.

Les diagrammes de classes

Cette représentation offre une vue statique de l'application. Cette vue correspond à la structure interne du logiciel avec la décomposition en packages et par classes. Ce diagramme permet de décrire les notions d'héritage, d'agrégation, de composition ou d'association entre les classes. La figure 3.3 page 29 représente un diagramme de classes que nous détaillons en introduction de la section 3.2.

Les diagrammes d'états-transitions

Un diagramme d'états-transitions permet de modéliser le comportement dynamique d'une classe ou d'un système suivant l'élément auquel est associé ce diagramme. Ce type de diagramme est basé sur les statecharts de Harel [Har87].

[4] Unified Modeling Language

Les transitions correspondent à des messages envoyés ou reçus par les classes ou à des changements internes des valeurs des variables. La figure 3.1 présente le diagramme d'états-transitions d'un composant PANNEAU que nous utiliserons au chapitre 6. Suivant les actions recues dans l'état 1, le composant change d'état puis renouvele son affichage pour se retrouver à nouveau dans l'état 1.

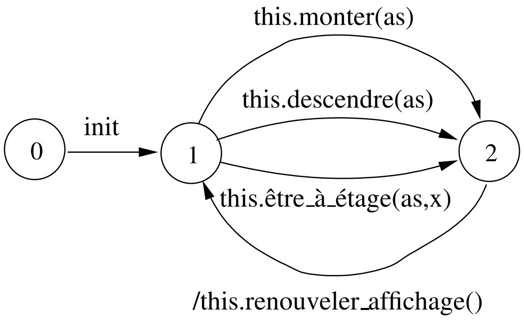


FIG. 3.1 – Exemple de diagramme d'états-transitions

Les diagrammes de séquences

Cette vue permet de décrire des interactions entre des instances des classes et ainsi d'exprimer des propriétés d'ordonnancement dans la succession d'appels de méthodes entre deux instances. Les messages sont ordonnés et l'axe vertical exprime la notion de temps. Les figures 3.5, 3.6 et 3.7 pages 31 à 32 décrivent des exemples de diagrammes de séquences.

3.2 Les niveaux d'abstraction des tests

Nous présentons ici les différents niveaux d'abstraction des tests adoptés dans le projet COTE. Pour chacun de ces niveaux, nous précisons ce qu'il apporte et sa syntaxe pour représenter les tests. Ces niveaux seront décrits plus formellement dans la troisième partie de ce mémoire.

Tout d'abord nous introduisons quelques notions utilisées par la suite dans cette thèse (figure 3.2). Nous appelons :

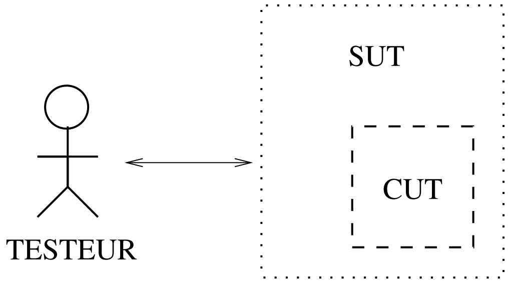


FIG. 3.2 – SUT, CUT et testeur

– **composant sous test** (CUT pour "Component Under Test") le composant, ou l'ensemble de composants, testé ;
– **système sous test** (SUT pour "System Under Test") les divers composants requis par le CUT, pour son exécution, ainsi que le CUT. Les composants du SUT ne faisant pas partie du CUT sont considérés comme exempts d'erreurs ;
– **testeur** le composant qui stimule le SUT, initialise, déclenche, coordonne les tests et fournit le verdict global.

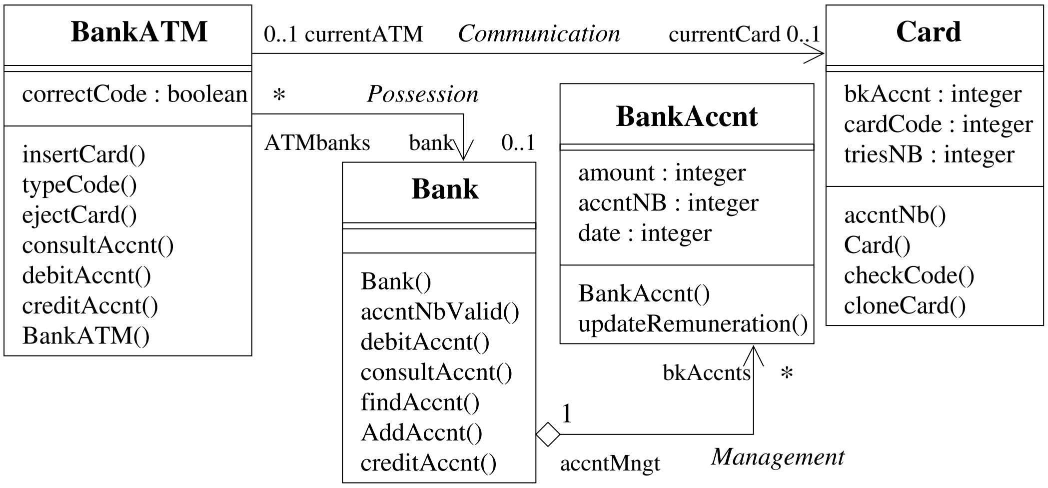


FIG. 3.3 – Diagramme de classes du distributeur bancaire

Nous présentons une étude de cas de distributeur bancaire afin d'illustrer les différentes formes de test considérées. Dans cet exemple, la banque *Bank* interagit avec un compte en banque (*BankAccnt*) par l'intermédiaire d'un distributeur bancaire (*BankATM*) et d'une carte à puces (*Card*). La figure 3.3 décrit le diagramme de classes de l'application. Les instances de la classe *BankATM* correspondent au **CUT**. Le **SUT** met en relation les classes *Card*, *BankATM*, *Bank* et *BankAccnt*. Au travers de cette étude de cas nous allons présenter le même test pour les différents niveaux d'abstraction du projet COTE. Le composant sous test considéré est la classe *BankATM*.

Test concret (ou test exécutable)

L'ingénieur de test peut écrire des tests dans le langage cible dans lequel est développée l'application.

```
public class Testeur {
  public BankATM ATM1 = new BanKATM() ;
  public CARD card01 = new CARD(3920) ;
  public Testeur(){}
  public Verdict sequence1(){
    if (ATM1.insertCard(card01) && ATM1.typecode(3920)
     && ATM1.consultAccnt() >= 0 && ATM1.ejectCard())
      return Pass ;
    else return Fail ; }
  public static void main(){
    ATM1 = new BankATM() ;
    Testeur1 = new Testeur() ;
    Testeur1.sequence1() ;}}
```

FIG. 3.4 – Un test exécutable en JAVA

La figure 3.4 décrit un test concret en JAVA. La séquence est la suivante : la carte **card01** est insérée dans le distributeur **ATM1** ; un code correct est tapé (**3920**) ; l'utilisateur consulte son compte puis demande l'éjection de sa carte. Le test renvoie **Pass** s'il s'exécute sans problème et **Fail** sinon. Le programme principal commence par créer les objets nécessaires pour ce test.

Comme nous venons de le voir, un test concret est lié à un langage de programmation. Ce lien peut parfois s'avérer contraignant. En effet si nous voulons tester

des composants ayant des fonctionnalités identiques mais écrits dans des langages différents ou pour des plates-formes d'exécution différentes, il faudra écrire des tests concrets pour chaque configuration. Les tests sont, de plus, encombrés par des détails propres à chaque langage.
L'environnement COTE a pour but de permettre d'écrire les tests sous un format UML indépendant du langage cible. Ces tests sont ensuite "compilés" automatiquement pour obtenir des tests exécutables sur une cible choisie à l'aide de drivers propres à chaque langage cible. De tels tests sont nommés *tests abstraits*, car ils font référence à la spécification.

Test abstrait (ou cas de test)

Dans le cadre du test fonctionnel, l'ingénieur doit s'appuyer sur la spécification du composant pour synthétiser des tests. L'idée est de permettre à l'utilisateur de décrire ses tests au même niveau d'abstraction que la spécification. Ainsi, si la spécification ne tient pas compte de la technologie cible, l'utilisateur pourra définir des tests "abstraits" indépendamment des contraintes technologiques. La figure 3.5 décrit un test exprimé dans le langage *TeLa* [PJH$^+$01] sous

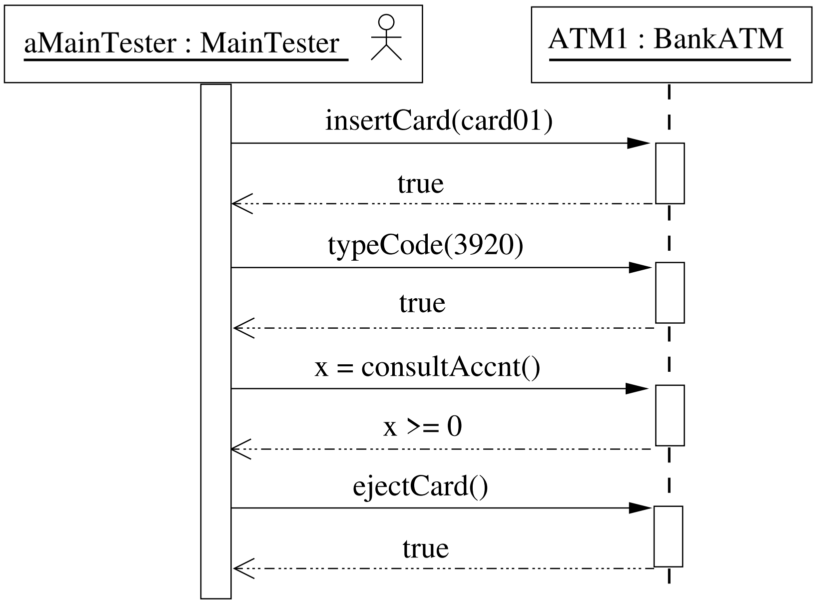


FIG. 3.5 – Un test abstrait

la forme d'un diagramme de séquences UML. Ce diagramme correspond à la version abstraite du test représenté dans la figure 3.4, il n'intègre pas les détails techniques associés à la création des instances et au traitement des échecs. Par défaut tout message ne figurant pas sur le diagramme est considéré comme menant à un échec.

Objectif de test

Un objectif de test permet de représenter un test sous une forme plus synthétique en n'utilisant que des transitions déterminantes du test, sans se préoccuper du chemin à suivre pour les atteindre. Nous représentons un objectif de test par une séquence de transitions de la spécification. La figure 3.6 décrit un objectif de test correspondant au test abstrait de la figure 3.5. Cet objectif de test est centré sur l'opération de consultation du solde **consultAccnt** dont le résultat doit vérifier une certaine propriété (positif ou nul).
Les objectifs de test sont écrits en UML dans le langage O-TeLa [BMdB+01b] qui a été développé et implanté par le laboratoire LSR. L'outil TGV permet de synthétiser des tests abstraits à partir d'objectifs de test.

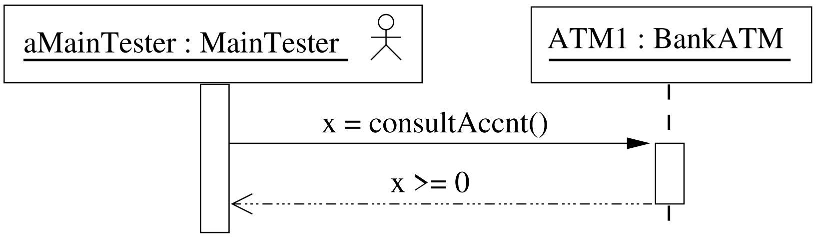


FIG. 3.6 – Un objectif de test exprimé en UML

Schéma de test

L'expérience industrielle [Mar01] montre que l'ingénieur de test peut être amené à décrire de nombreux objectifs de test qui ne diffèrent les uns des autres que par un appel de méthode, un paramètre ou une instance. C'est pourquoi un autre niveau d'abstraction, nommé *schéma de test*, a été introduit dans le projet COTE. Là où les objectifs de test décrivent partiellement une séquence de test, un schéma de test décrit un ensemble d'objectifs de test.
Le langage d'expression des schémas de test se base sur des abstractions des appels de méthodes. Quatre sortes d'abstractions ont été mises en place sur : les

valeurs des paramètres des méthodes, les instances, le nombre d'appels et les méthodes. L'utilisation de ces différentes abstractions permet de décrire en une seule expression un ensemble d'objectifs de test.
La figure 3.7 décrit six objectifs de test :

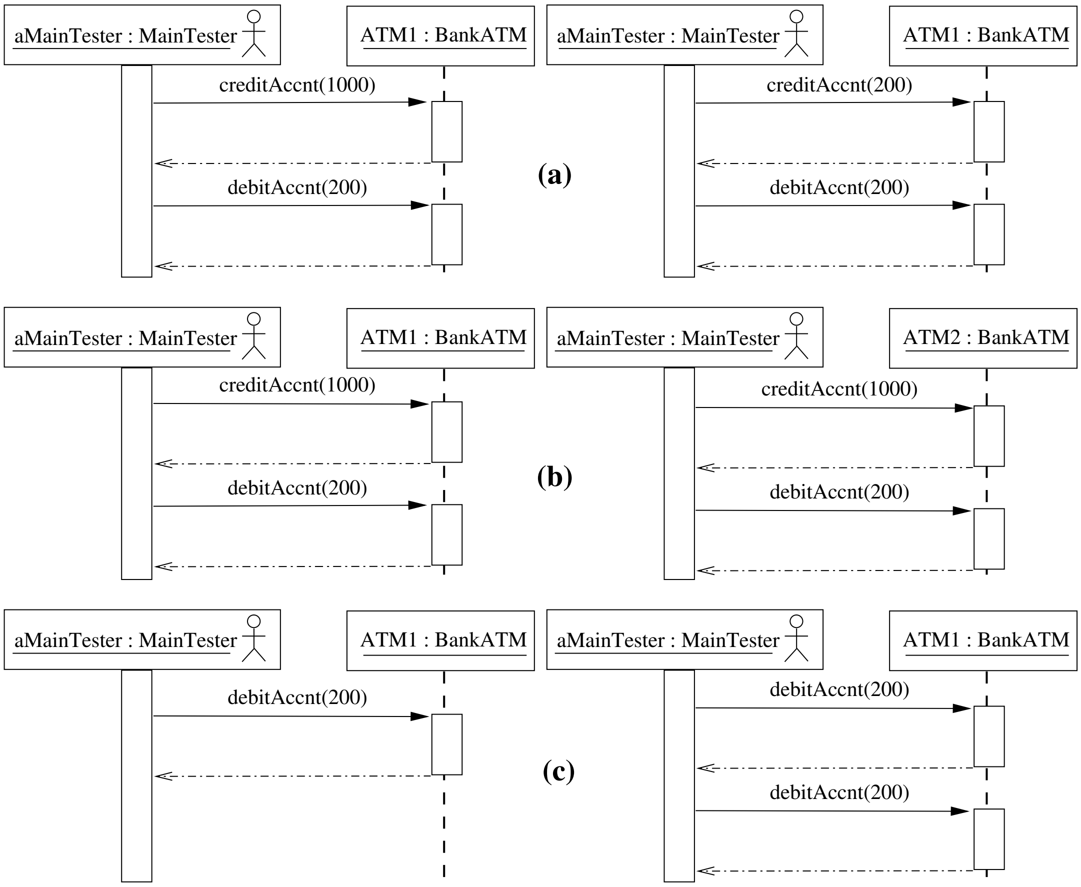


FIG. 3.7 – Des objectifs de test regroupés en un schéma de test

- Les deux objectifs de test de la ligne **(a)** ne diffèrent que par la valeur du paramètre de la méthode **creditAccnt()**. Un schéma de test peut donc faire abstraction de ces valeurs, en précisant pour chaque paramètre un ensemble de valeurs possibles. Dans notre exemple nous déclarons les associations suivantes :

creditAccnt(x), **x** $\in \{200, 1000\}$
debitAccnt(y), **y** $\in \{200\}$

– Les deux objectifs de test de la ligne **(b)** ne diffèrent que par l'instance de "BankATM" considérée (ATM1 ou ATM2). Une abstraction consiste à ne pas nommer l'instance référencée pour un appel de méthode. Nous pouvons représenter cela de la façon suivante :

o.creditAccnt(1000), **o** $\in \{ATM1, ATM2\}$

La notation * à la place du **o** signifie qu'on associe à la méthode toute instance définie dans le diagramme de déploiement initial, du moment que la méthode spécifiée (dans cet exemple : **creditAccnt**) existe bien dans la classe correspondant à l'instance.

– Les deux objectifs de test de la ligne **(c)** comportent les mêmes appels de méthode mais avec un nombre d'occurrences différent. Les schémas de test permettent de spécifier un nombre variable d'occurrences d'appel d'une méthode. Nous notons cela de la façon suivante :

ATM1.debitAccnt(200)ˆ 1..2

– Dans les six objectifs de test présentés, nous utilisons explicitement les opérations **creditAccnt()** et **debitAccnt()**. Ces opérations peuvent être assez semblables du point de vue de leurs signatures ou de leurs comportements et donc être testées de façon similaire. Les schémas de test offrent la possibilité de regrouper des méthodes sous un même label, c'est la notion de **groupe**. Au vu des objectifs de test de la figure 3.7 nous pouvons créer un groupe **operation** qui regroupe les opérations **creditAccnt()** et **debitAccnt()**. Nous notons cela :

ATM1.operation, **operation =** $\{$**creditAccnt,debitAccnt**$\}$

Les objectifs de test présentés dans la figure 3.7 (ainsi que d'autres) peuvent être regroupés sous le même schéma de test ***.operationˆ1..2**. Ce dernier schéma de test permet de générer 42 objectifs de test. Pour le groupe **operation**, nous pouvons avoir 3 combinaisons différentes : **creditAccnt(200), creditAccnt(1000)** et **debitAccnt(200)**. Chaque méthode peut être associée aux deux instances : ATM1 et ATM2. Chaque séquence peut contenir 1 ou 2 appels. Nous obtenons donc 42 objectifs de test[5].

Le langage d'expression des schémas de test n'est pas basé sur UML. En effet un schéma de test est textuel et nécessite des informations supplémentaires, comme l'ensemble des instances considérées, les groupes et les valeurs des paramètres. Une présentation plus complète des schémas de test et de leur grammaire est faite dans le chapitre 4.

3.3 L'architecture du projet COTE

Cette section détaille l'architecture de l'environnement COTE. Cet environnement correspond à une chaîne de traitement des tests à partir d'une spécification exprimée en UML. La figure 3.8 décrit l'architecture du projet COTE avec l'outil Objecteering de Softeam comme outil central. La communication avec les autres outils se fait en utilisant le format XMI [OMG02b] qui permet une description XML d'un modèle UML. Nous présentons maintenant plus en détail le rôle de chacun des outils concernés.

Objecteering

Objecteering est un environnement de modélisation UML. Dans le cadre du projet COTE il est utilisé pour exprimer la spécification à l'aide de diagrammes de classes, d'états-transitions et de déploiement. Un profil de test a été développé pour Objecteering afin de pouvoir exprimer des suites de tests avec des tests abstraits et des objectifs de test en UML sous la forme de diagrammes de séquences [PJH$^+$01].
Objecteering intègre aussi des drivers qui permettent la génération de tests exécutables sur différentes plates-formes.

[5](2+1)*2 = 6 possibilités pour le groupe **operation** et 1 ou 2 appels possibles à ce groupe. Soit 6+(6*6) = 42 objectifs de tests.

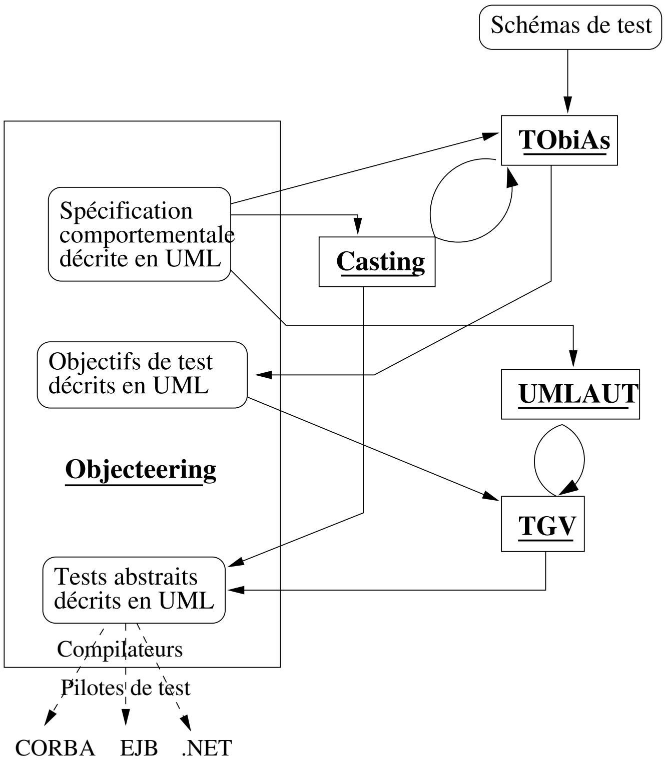


FIG. 3.8 – Le projet COTE

Les pilotes de test

Les pilotes de test, tel que Cactus[6] pour le test d'EJB, servent à exécuter une suite de tests concrets sur l'implémentation considérée et à évaluer si les tests se déroulent sans erreur. Dans le cas où une erreur survient, le pilote de test permet d'identifier la source de l'erreur.

Les compilateurs

Ces composants intégrés à Objecteering permettent de traduire un test abstrait sous la forme d'un test exécutable pour une cible donnée. Ces compilateurs ont été définis pour les langages Java et Corba, entre autres.

[6]http ://jakarta.apache.org/cactus/index.html

TGV

TGV effectue la génération de tests abstraits à partir d'objectifs de test et d'une spécification. Les objectifs de test sont fournis soit de manière manuelle soit via une interface d'Objecteering. TGV est présenté plus en détail dans le chapitre 2.

UMLAUT

L'outil UMLAUT développé par l'IRISA permet de simuler une spécification UML [JHGP99] en calculant le graphe d'accessibilité de la spécification. La spécification doit contenir un diagramme d'états-transitions étendu par du code Eiffel permettant l'animation de la spécification. Dans le cadre du projet COTE, UMLAUT construit le produit de l'ensemble des diagrammes d'états-transitions pour calculer la spécification comportementale du système [Gue01].

UMLAUT est couplé à TGV pour animer la spécification comportementale et fournir à TGV les transitions possibles. Ce travail conjoint permet à TGV de travailler sur la spécification comportementale à la volée.

TObiAs

L'outil TObiAs (Test OBjective desIgn ASsitant) permet de définir des schémas de test et de construire des objectifs de test associés à ces schémas. TObiAs interagit aussi avec l'outil Casting pour obtenir des jeux de valeurs pour les paramètres. TObiAs sera présenté de manière plus complète dans le chapitre 4.

Casting

L'outil Casting est en marge de la chaîne principale d'outils définie dans le projet. Dans COTE cet outil intervient de deux façons :

- en l'associant à TObiAs pour qu'il lui fournisse des valeurs de tests ou valide des séquences de test ;
- en synthétisant des tests abstraits à partir de la spécification UML [AJ03], et en offrant une mesure de couverture de la spécification en terme de couverture des branches du diagramme d'états-transitions.

Résumé

Le projet COTE avait comme objectif de mettre en place un environnement de test UML permettant d'automatiser, au moins partiellement, la phase d'élaboration des tests dans le cadre de spécifications comportementales exprimées en UML. Le projet COTE permet de réduire le temps de conception des tests en proposant différents niveaux de description de ceux-ci (tests abstraits, objectifs de test et schémas de test) et en permettant de passer d'un niveau à un autre automatiquement.

L'outil TObiAs permet de synthétiser des objectifs de test à partir des schémas de test tandis que les outils UMLAUT/TGV permettent d'obtenir des tests abstraits associés à ces objectifs de test. Les compilateurs intégrés à Objecteering et les pilotes de test spécifiques assurent la traduction des tests abstraits vers les langages cibles et leur exécution.

Un autre objectif de l'environnement COTE était de gérer la phase de décision d'arrêt des tests afin de couvrir tout le processus de test. Seul l'outil Casting intègre une notion de couverture de la spécification par une suite de tests. Mais cette spécificité n'est pas intégrée dans le processus principal de l'environnement COTE. La troisième partie de cette thèse présentera la notion de couverture de la spécification que nous avons mise en place au niveau des schémas de test.

Dans la partie suivante nous présentons plus en détail la notion de schéma de test ainsi que l'outil TObiAs. Nous décrivons des études de cas ayant permis d'évaluer l'apport de l'approche proposée.

Deuxième partie

Les schémas de test et leur outillage

Introduction aux schémas de test et à TObiAs

Le projet COTE présenté dans la partie précédente avait pour but de construire un environnement UML pour le test de conformité, notamment dans le cadre du test de composants. Cet environnement a la particularité de permettre la modélisation des tests à différents niveaux d'abstraction, ainsi que l'automatisation de la création de tests exécutables.

Nous avons effectué deux tâches principales, fortement liées, sur le projet COTE. La première a consisté à élaborer un nouveau niveau d'abstraction pour définir des tests, ce sont *les schémas de test*. La deuxième tâche consistait à outiller les schémas de test, afin de faciliter l'écriture de schémas de test et de générer automatiquement des tests à des niveaux d'abstraction moindre. Cet outil est l'outil TObiAs.

Dans la section 3.2 nous avons présenté les différents niveaux d'abstraction du projet, dont le niveau des objectifs de test. Ce niveau permet au testeur de décrire un test partiellement, l'outil TGV étant alors en charge de compléter ces objectifs de test pour obtenir des tests abstraits. L'inconvénient de TGV est qu'il ne produit qu'un test abstrait à partir d'un objectif de test à chaque exécution. Deux exécutions successives de TGV avec ce même objectif de test peuvent permettre de produire deux tests différents. De plus, il est souvent nécessaire que l'ingénieur de test décrive un objectif de test très détaillé afin de guider l'outil vers le cas de test souhaité [Mar01]. Sachant qu'une campagne de test industrielle peut comprendre plusieurs milliers de tests, l'ingénieur de test peut être amené à écrire autant d'objectifs de test.

Il a été constaté expérimentalement que les objectifs de test pouvaient présenter de nombreuses similarités [Mar01, dBMJ01]. C'est pourquoi nous avons proposé de définir la notion de schéma de test, qui permet de factoriser certains éléments des objectifs de test. Un schéma de test peut alors être déplié en plusieurs objectifs de tests, qui auraient sinon dû être écrits à la main (pour une partie). Ainsi, l'idée principale des schémas de test est d'exprimer les similarités entre les objectifs de tests via des expressions de plus haut niveau d'abstraction.

Cette deuxième partie de la thèse présente les schémas de test et les concepts qui leurs sont associés, ainsi que l'outil TObiAs. Le chapitre 4 sera consacré à la présentation de la syntaxe et de la sémantique des schémas de test puis à leur mise en œuvre dans TObiAs. Le chapitre 5 présentera les différentes orientations prises avec les schémas de test et TObiAs pour les exploiter dans d'autres cadres que celui de la génération d'objectifs de test. Enfin le chapitre 6 illustrera l'utilisation des schémas de test et de TObiAs sur deux études de cas. La première étude présentera une utilisation conjointe de TOBiAs et TGV sur un système d'ascenseur. La seconde étude, menée d'une part par le laboratoire de recherche de Gemplus et d'autre part par notre laboratoire, portera sur l'utilisation de TObiAs sur un système bancaire.

Chapitre 4

Schémas de test et TObiAs

Ce chapitre est composé de trois parties. La première partie présente les abstractions qu'apportent les schémas de test par rapport aux objectifs de test ainsi que la syntaxe des schémas de test. La deuxième partie décrit la sémantique associée aux schémas de test, en dépliant les schémas de test en des objectifs de test. La troisième partie est consacrée à l'outil TObiAs qui permet la création des schémas de test et leur dépliage en objectifs de test.

4.1 Les schémas de test

Nous avons vu précédemment que [Mar01] et [dBMJ01] mettaient en évidence la nécessité de générer de multiples objectifs de test semblables pour mener à bien une campagne de test. Comme nous avons pu le voir dans le chapitre 2, les objectifs de test mettent en jeu des instances ainsi que des appels de méthodes. Factoriser ces objectifs de test permet d'écrire une seule expression pour exprimer plusieurs objectifs de test. Cette expression se nomme **schéma de test**. Nous avons décidé d'effectuer cette factorisation par rapport aux méthodes, aux valeurs des paramètres, aux instances sur lesquelles s'appliquent les méthodes ainsi qu'au nombre de répétitions d'appels de méthodes. Un schéma de test se définit de la manière suivante :

> **Définition 2 (Schéma de test)** *chemin partiel dans la spécification comportementale qui fait abstraction des instances et factorise les répétitions d'appels de méthodes, les méthodes ainsi que les valeurs de leurs paramètres.*

4.1.1 L'environnement des schémas de test

Les schémas de test servent à exploiter les similarités entre plusieurs objectifs de test, tel que le fait que l'on puisse être amené à exécuter le même test sur des *instances* différentes.
De même, certains objectifs de test ont un même préambule suivi, soit d'un appel à une même méthode mais avec des valeurs de paramètres différentes, soit d'appels à des méthodes différentes. Les schémas de test comportent la notion de **groupe** qui permet d'exprimer de telles abstractions sur les méthodes et sur les paramètres.

> **Définition 3 (Groupe)** *représentation abstraite d'un ensemble de méthodes avec pour chaque paramètre de chaque méthode un jeu de valeurs possibles.*

A partir de ces informations sur les groupes et sur les instances mis en jeu, le testeur peut écrire des schémas de test suivant la grammaire définie dans le tableau 4.1.

Les trois notions de schéma de test, groupe et ensemble d'instances forment ce que nous appelons un **environnement de campagne de test**. Cet environnement constitue le cadre dans lequel est élaborée une campagne de test pour produire des suites de test.

> **Définition 4 (Environnement de campagne de test)** *correspond à un ensemble de paramètres constitutifs d'une campagne de test, à savoir les instances sous test et les groupes.*

4.1.2 La grammaire des schémas de test

Les schémas de test ayant pour but de décrire de façon abstraite des objectifs de test exploitables par TGV, nous avons choisi des constructions similaires à celles définies pour les langages O-TeLa et TeLa [PJH^{+}01] du projet COTE. Le tableau 4.1 décrit la grammaire des schémas de test. Les constructions offertes par la grammaire sont illustrées ci-dessous, en prenant l'exemple du diagramme de classes de la figure 4.1 page 46. Dans cet exemple l'*environnement de campagne de test* comporte le *groupe* g1 constitué des méthodes m1 et m2, le *groupe* g2

ST	: :=	**SEQ** [] **ST**
	|	**SEQ**
SEQ	: :=	**STRUCT** ; **SEQ**
	|	**STRUCT**
STRUCT	: :=	**BOUCLE**
	|	**COREG**
	|	**STRICT**
	|	−**APPEL**
	|	**APPEL**
	|	**ST**
APPEL	: :=	**INST** !**INST**.**GROUPE**
INST	: :=	identifiant
	|	*
GROUPE	: :=	identifiant
BOUCLE	: :=	**APPEL**^**NB**
	|	(**ST**)^**NB**
NB	: :=	entier..entier
	|	*
COREG	: :=	<**APPEL**>**COREG**
	|	<**APPEL**><**APPEL**>
STRICT	: :=	[**ST**]

TAB. 4.1 – Grammaire des schémas de test

constitué des méthodes m3 et m4 et les *instances* a1 de la classe A et b1 de la classe B.

FIG. 4.1 – Un exemple de diagramme de classes

APPEL

"*a1 !b1.g2*" est un appel qui signifie que dans l'instance a1 de la classe A, on fait appel à une méthode du *groupe* g2 de l'instance b1 de la classe B. Un appel doit respecter deux conditions : qu'il existe une relation entre les classes correspondant aux 2 instances mises en jeu (ici une relation entre A et B), et que le groupe considéré contienne une méthode définie dans la classe de la deuxième instance (ici g2 contient les méthodes m3 et m4, toutes deux définies dans la classe B). Dans un schéma de test, une instance peut être clairement identifiée par son nom ou remplacée par "*" qui désigne toutes les instances vérifiant les deux conditions ci-dessus.

BOUCLE FINIE

Les schémas de test permettent de décrire une répétition d'une même partie d'un schéma de test. Si nous avons par exemple : *(ST)*$^\wedge$ 2..4, cela signifie que nous avons deux à quatre appels successifs à *ST*. Il faut donc produire des objectifs de test avec 2, 3 et 4 fois la séquence représentée par ST. <u>entier</u> représente les bornes inférieure et supérieure de la répétition d'un appel. La borne inférieure ne peut pas prendre une valeur plus petite que 1 et doit être inférieure à la borne supérieure.

BOUCLE INFINIE

L'ingénieur de test a la possibilité d'exprimer le fait qu'on fasse un nombre d'itération non défini sur une séquence d'appel. On définit une boucle infinie non borné avec la notation $\boxed{\wedge\ *}$.

"*a1 !b1.g2*$^\wedge$ *" signifie que l'appel à une méthode du groupe g2 par l'instance a1 peut avoir lieu un nombre de fois d'affilé non borné.

SÉQUENCEMENT

Le séquencement est représenté par le symbole $\boxed{;}$. C'est l'opération de base des schémas de test. Cette opération permet d'ordonner les appels dans un schéma de test.

"*a1 !b1.g2 ; b1 !a1.g1*" signifie que la séquence est formée d'un appel à une méthode du groupe g2 suivi d'un appel à une méthode du groupe g1.

ALTERNATIVE

Les schémas de test peuvent contenir du non déterminisme, de la même façon que les objectifs de test gérés par TGV. Pour exprimer ce choix nous utilisons le symbole $\boxed{[]}$.

a1 !b1.m3() [] *a1 !b1.m4()* correspond au fait que pour ce schéma de test, l'instance a1 peut aussi bien procéder à un appel à la méthode m3 qu'à un appel à la méthode m4(). Le schéma de test "*a1 !b1.m4()* [] *a1 !b1.m3()*" a le même comportement.

CO-RÉGION

Le mécanisme de co-région permet de représenter le caractère parallèle ou non ordonné d'une exécution. Si nous avons par exemple *<app1><app2><app3>*, cela signifie que nous n'imposons pas d'ordre entre les occurrences des appels de méthode *app1*, *app2* et *app3*. Les séquences suivantes sont donc possibles :

- *app1 ; app2 ; app3*
- *app1 ; app3 ; app2*
- *app2 ; app1 ; app3*
- *app2 ; app3 ; app1*
- *app3 ; app1 ; app2*
- *app3 ; app2 ; app1*

NÉGATION

Dans un schéma de test on peut exprimer le fait qu'on ne veuille pas voir apparaître un appel. Le symbole $\boxed{-}$ est placé juste à gauche de l'appel non désiré.

"*a1 !b1.g2 ; −a1 !b1.g2 ; b1 !a1.g1*" signifie qu'après un premier appel à une méthode du groupe g2 il ne peut pas y avoir un autre appel à une méthode de ce même groupe tant qu'un appel à une méthode du groupe g1 n'a pas eu lieu.

SÉQUENCE STRICTE

Une séquence stricte est une séquence notée entre crochets. Cela correspond à une séquence qui reste insécable dans les tests abstraits.

"*[a1 !b1.g2 ; a1 !b1.g2 ; b1 !a1.g1]*" caractérise les objectifs de test qui comportent une suite d'appel stricte à des méthodes des groupes g2, g2, g1 et qui ne peut pas être interrompue par un autre appel de méthode.

4.2 La génération des objectifs de test

TObiAs permet de générer des objectifs de test correspondant à un schéma de test. Le format adopté pour les objectifs de test est un format propriétaire dans lequel les boucles finies sont dépliées et les groupes décomposés en méthodes instanciées. La fonction de traduction **T** décrite dans le tableau 4.2 permet de calculer, pour chaque forme de schéma de test, l'ensemble des objectifs de test qui lui sont associés.

Seul les APPELS et les BOUCLES FINIES sont interprétés à ce niveau. Les autres constructions sont interprétés dans un deuxième temps lors de la traduction vers un langage cible donné.

La fonction **T** nécessite un contexte qui décrit l'environnement de campagne de test et le diagramme de classes de la spécification.

- Comme décrit dans la définition 4, un environnement de campagne est constitué de l'ensemble "I" des instances sous test et des groupes. La fonction G prend en entrée un nom de groupe et renvoie l'ensemble des appels de méthodes de ce groupe.
- Le diagramme de classes permet de connaître les associations entre classes et la visibilité des méthodes (savoir si elles sont publiques ou privées). L'ensemble des associations entre classes, nommé "R", permet de savoir s'il existe une relation entre deux instances de classe et de déterminer pour chaque méthode la liste des instances qui peuvent les appeler[1].

La fonction $Classe$ prend comme paramètre un nom d'instance et renvoie le nom de la classe associée. Dans notre exemple, $Classe(a1) = A$. La fonction $MClasse$ renvoie, quant à elle, l'ensemble des méthodes de la classe associée à l'instance passée en paramètre.

[1]Dans le cas de la deuxième règle, l'existence d'une relation entre les classes des instances i1 et i2 est vérifiée à l'écriture du schéma. Il n'est pas utile d'effectuer à nouveau cette vérification.

$\mathbf{T}(\mathrm{E}\hat{\ }\mathrm{int1..int2})$	$= \{e_1 ;.. ; e_n \mid \forall i \in 1..n,\ e_i \in \mathbf{T}(\mathrm{E})$
	$\wedge\ \mathrm{int1} \leq \mathrm{n} \leq \mathrm{int2}\}$
$\mathbf{T}(\mathrm{i1\ !i2.groupe})$	$= \{\mathrm{i1\ !i2.m} \mid \mathrm{m} \in \mathrm{G(groupe)} \wedge m \in MClasse(\mathrm{i2})\}$
$\mathbf{T}(\mathrm{*\ !i2.groupe})$	$= \{\mathrm{i1\ !i2.m} \mid \mathrm{m} \in \mathrm{G(groupe)} \wedge \mathrm{i1} \in \mathrm{I}$
	$\wedge\ (Classe(i1), Classe(i2)) \in \mathrm{R} \wedge m \in MClasse(\mathrm{i2})\}$
$\mathbf{T}(\mathrm{i1\ !*.groupe})$	$= \{\mathrm{i1\ !i2.m} \mid \mathrm{m} \in \mathrm{G(groupe)} \wedge \mathrm{i2} \in \mathrm{I}$
	$\wedge\ (Classe(i1), Classe(i2)) \in \mathrm{R} \wedge m \in MClasse(\mathrm{i2})\}$
$\mathbf{T}(\mathrm{*\ !*.groupe})$	$= \{\mathrm{i1\ !i2.m} \mid \mathrm{m} \in \mathrm{G(groupe)} \wedge \mathrm{i1} \in \mathrm{I} \wedge \mathrm{i2} \in \mathrm{I}$
	$\wedge\ (Classe(i1), Classe(i2)) \in \mathrm{R} \wedge m \in MClasse(i2)\}$
$\mathbf{T}(\mathrm{E1}\ [\!]\ \mathrm{E2})$	$= \{\mathrm{e1}\ [\!]\ \mathrm{e2} \mid \mathrm{e1} \in \mathbf{T}(\mathrm{E1}) \wedge \mathrm{e2} \in \mathbf{T}(\mathrm{E2})\}$
$\mathbf{T}(\mathrm{E1} ; \mathrm{E2})$	$= \{\mathrm{e1} ; \mathrm{e2} \mid \mathrm{e1} \in \mathbf{T}(\mathrm{E1}) \wedge \mathrm{e2} \in \mathbf{T}(\mathrm{E2})\}$
$\mathbf{T}(\mathrm{E}\hat{\ }*)$	$= \{\mathrm{e}\hat{\ }* \mid \mathrm{e} \in \mathbf{T}(\mathrm{E})\}$
$\mathbf{T}(<\mathrm{E1}>\mathrm{E2})$	$= \{<\mathrm{e1}>\ \mathrm{e2} \mid \mathrm{e1} \in \mathbf{T}(\mathrm{E1}) \wedge \mathrm{e2} \in \mathbf{T}(\mathrm{E2})\}$
$\mathbf{T}(<\mathrm{E1}><\mathrm{E2}>)$	$= \{<\mathrm{e1}><\mathrm{e2}> \mid \mathrm{e1} \in \mathbf{T}(\mathrm{E1}) \wedge \mathrm{e2} \in \mathbf{T}(\mathrm{E2})\}$
$\mathbf{T}([\mathrm{E}])$	$= \{[\mathrm{e}] \mid \mathrm{e} \in \mathbf{T}(\mathrm{E})\}$
$\mathbf{T}(-\mathrm{E})$	$= \{-\mathrm{e} \mid \mathrm{e} \in \mathbf{T}(\mathrm{E})\}$
$\mathbf{T}((\mathrm{E}))$	$= \{(\mathrm{e}) \mid \mathrm{e} \in \mathbf{T}(\mathrm{E})\}$

TAB. 4.2 – Traduction des schémas de test

4.3 Mise en œuvre de TObiAs

Nous venons de présenter les schémas de test et les règles de traduction utilisées pour générer des objectifs de test à partir de ces schémas de test. Un schéma de test ne se compose pas d'une seule expression, il est soumis au contexte d'un diagramme de classes ainsi que d'un ensemble d'instances et d'un ensemble de groupes qui forment l'environnement de campagne de test.

Présentation

TObiAs se décompose en deux parties. La première est l'interface utilisateur qui permet au testeur de définir des environnements de campagne de test pour lesquels il définit des schémas de test. La seconde partie de TObiAs consiste en l'implantation d'un algorithme de génération d'objectifs de test, correspondant à la fonction de traduction **T** vue précédemment et leur traduction en IOLTS.

Interface utilisateur de TObiAs

1. Importation d'un diagramme de classes
TObiAs prend en entrée un diagramme de classes relatif au système à tester. Actuellement ce diagramme est sous la forme d'un fichier XMI généré par l'outil Objecteering. A partir du diagramme de classes, TObiAs affiche les différentes classes du système, avec leurs méthodes et les types de paramètres de ces méthodes. Ce diagramme permet également de construire les relations entre classes (la relation R de la section précédente) et les liens entre méthodes et classes (la fonction $MClasse$).

2. Définition de l'ensemble des instances
Chaque environnement de campagne de test s'appuie sur un ensemble d'instances. L'utilisateur peut définir autant d'instances différentes qu'il le souhaite par classe du diagramme de classe. Cette définition produit l'ensemble I et la fonction *Classe* de la section précédente.

3. Définition des groupes
Les groupes se définissent en sélectionnant des méthodes et en définissant pour chaque paramètre de ces méthodes un ensemble de valeurs. Les groupes peuvent être définis d'une façon globale au niveau du diagramme de classe. Dans ce cas ils sont utilisables dans tous les environnements de campagne de test. Ils peuvent aussi être rattachés à un environnement de campagne de test spécifique, dans ce cas les groupes ne peuvent être consitués que de méthodes appartenant à des classes représentées par des instances, et les groupes ne sont utilisables que dans cet environnement.

4. Définition des schémas de test
TObiAs possède une interface de définition des schémas de test (figure 4.2) qui permet toutes les constructions définies pour les schémas de test. Chaque schéma de test est défini dans le cadre d'un environnement de campagne de test, pour des ensembles d'instances et de groupes donnés.

Génération des objectifs de test

5. Dépliage et synthèse d'objectifs de test
La fonction **T** est implantée dans TObiAs et assure la génération automatique de l'ensemble des objectifs de test associés à un schéma de test.

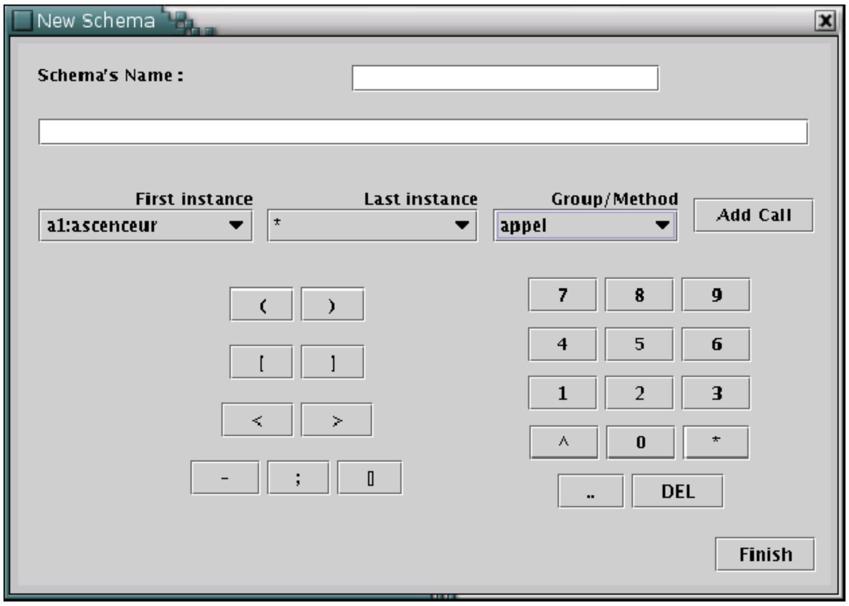


FIG. 4.2 – Interface de définition d'un schéma de test

6. Traduction en IOLTS
TObiAs intègre différents traducteurs du langage d'expression des objectifs de test dont un traducteur vers le langage d'expression Aldébaran [Fer89]. Ce langage sert à représenter les IOLTS de façon textuelle. Le traducteur de TObiAs interprète toutes les constructions qui ne l'ont pas été lors du dépliage pour produire un IOLTS sous le format Aldébran, c'est-à-dire le séquencement, l'alternative, la négation, la séquence stricte, la co-région et la boucle infinie.

Le diagramme de la figure 4.3 présente de quelle façon les notions de schéma de test, objectif de test, groupe, instance, classe, méthode et parametre sont associées dans une campagne de test dans TObiAs. On peut noter que les groupes sont associés à des instances de méthodes (IMéthode) et non pas aux méthodes directement. Cela permet d'associer une même méthode à différents groupes avec des jeux de valeur différents.

Aspects techniques de TObiAs

Nous avons développé TObiAs en JAVA. La conception de l'outil est modulaire afin de l'étendre facilement. TObiAs se compose de quatre packages (cf.

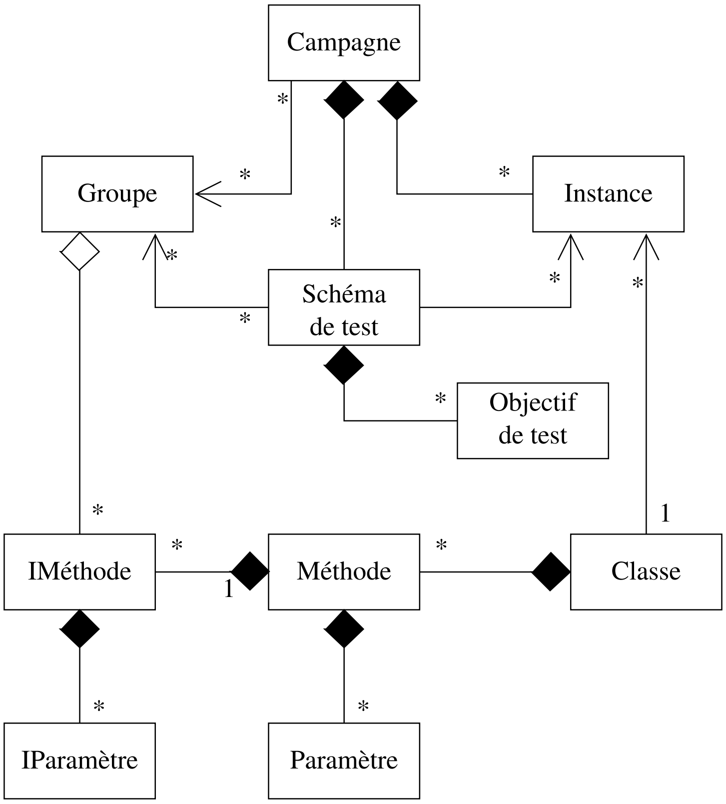


FIG. 4.3 – Modèle des concepts de TObiAs

figure 4.4), chacun d'eux comporte des ensembles de classes en relation les unes avec les autres.

TObiAs représente environ 14 500 lignes de code réparties entre 77 classes. La classe IHM permet de gérer l'interface avec l'utilisateur. Cette classe permet d'accéder aux fonctionnalités de TOBiAs présentes dans les différents packages qui composent l'outil (cf. figure 4.4).

– **PAJAX** (9 classes), pour **PArseur JAva Xml**, est un module qui permet la lecture et l'écriture de fichiers XML, et donc de gérer les entrées/sorties de TObiAs. Une des classes de PAJAX permet de traiter les informations des fichiers XMI pour les mettre au format interne de TObiAs. Pour la sauvegarde de l'état interne d'un projet, nous avons choisi un format XML afin de réutiliser le module PAJAX et permettre une exportation future du projet

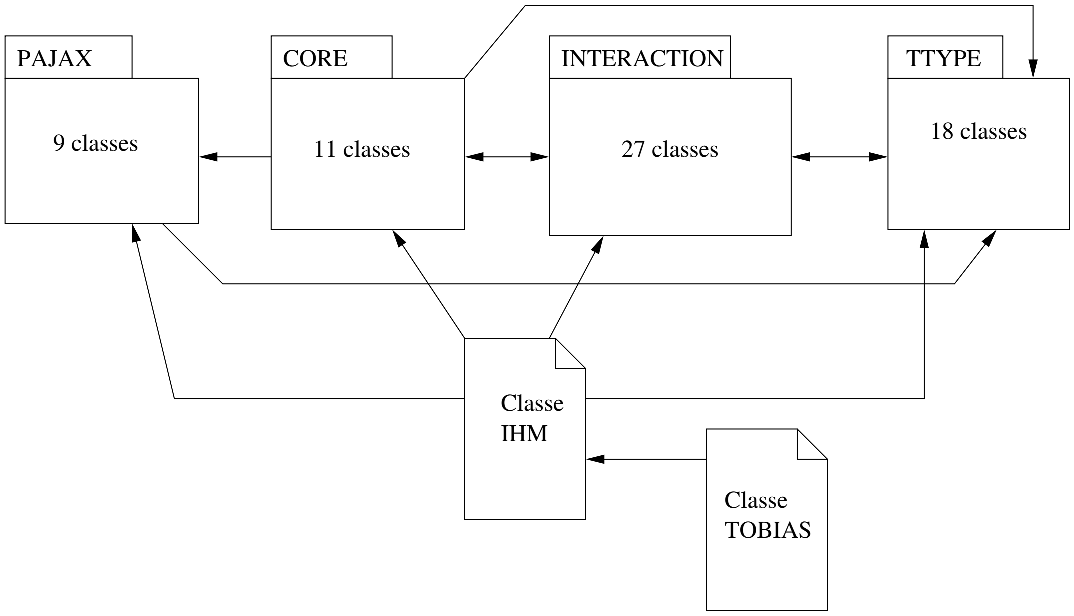


FIG. 4.4 – Architecture de TObiAs sous forme de packages

vers d'autres outils.

– **TTYPE** (18 classes) est le package qui définit tous les types de base spécifiques aux schémas de test et à TObiAs. Le modèle comprend les représentations des instances, des classes, des méthodes, des paramètres des méthodes, des groupes, des campagnes de test ainsi que des schémas de test et objectifs de test.
– **CORE** (11 classes) rassemble les sous-programmes qui permettent la transformation des données, comme la génération des séquences d'appel à partir des schémas de test. Ces séquences d'appel peuvent être interprétées comme des objectifs de test, mais aussi comme des tests abstraits tel que nous le verrons dans le chapitre 5. Il gère les routines de sauvegarde et chargement en relation avec le module PAJAX, ainsi que la transformation des séquences d'appel vers les langages JML, aldébaran, VDM et OTS. Ce dernier langage est un langage propre à TObiAs qui permet d'exprimer des objectifs de test.
– **INTERACTION** (27 classes) constitue le package associé à l'interface de TObiAs, il comporte toutes les fenêtres de dialogue et les systèmes d'interaction de TObiAs. C'est dans ce package qu'est géré, entre autres, l'arborescence des campagnes de test, groupes et schémas de test. Il fournit

aussi l'interface de toutes les commandes disponibles dans TObiAs, que ce soit pour gérer les sauvegardes et chargements ou bien pour la création des groupes.

Résumé

Dans ce chapitre nous avons présenté la notion de *schéma de test* qui permet de décrire plusieurs objectifs de test en une seule expression. Les schémas de test utilisent la notion d'*environnement de campagne de test*, qui représente l'ensemble des instances sous test, ainsi que les *groupes* qui permettent de factoriser des méthodes avec différentes valeurs d'appel de leurs paramètres.

La notion de schéma de test associé à une campagne de test a été mis en place dans l'outil TObiAs. Cet outil permet de synthétiser des objectifs de test à partir de schémas de test, vers le format interne de TObiAs (format OTS).

Dans le chapitre suivant nous présentons d'autres approches liées à l'utilisation de TObiAs ainsi que différentes solutions au problème de l'explosion combinatoire liée au mode de génération des objectifs de test.

Chapitre 5

Evolutions de TObiAs

TObiAS génère des séquences d'appel à partir d'un shéma de test et d'un environnement de campagne de test. Dans le chapitre précédent nous avions associé une sémantique à ces séquences afin d'en faire des objectifs de test. Nous pouvons leur associer une nouvelle sémantique pour en faire des tests abstraits et élargir le champ d'action de TObiAs.

L'utilisation de TObiAs avec TGV (en générant des objectifs de test) permet d'obtenir un oracle pour vérifier les résultats attendus. Néanmoins il existe d'autres façons d'obtenir un oracle pour les séquences générées par TObiAs. D'autres travaux ont été développés dans l'équipe pour interfacer TObiAs avec d'autres outils, améliorer le moteur de synthèse de tests et diversifier les formats de sortie de TObiAs.

Ce chapitre résume ces travaux. La section 5.1 s'intéresse au problème de l'oracle lors de l'exécution des tests. Dans ce cadre deux langages permettant d'exprimer des spécifications exécutables ont été étudiés : JML et VDM. La section 5.2 s'intéresse au problème de la sélection automatique des valeurs des paramètres. Enfin la section 5.3 s'intéresse au problème de l'explosion combinatoire. Ces travaux ont principalement été développés par Olivier Maury [ML02, LdBMB04] ainsi que par des étudiants de DEA [Bou02, Beg02, Kes03].

5.1 Le problème de l'oracle dans TObiAs

Lorsque nous exécutons un programme en vue de le tester, dans le cadre du test de conformité, nous devons nous assurer que le comportement observé est bien celui attendu. Par exemple, la fonction *cube()* calcule le cube d'un entier. Pour *cube(2)* la valeur attendue en retour est 8. Décider (automatiquement ou non) si le comportement du programme est correct au cours d'un test s'appelle le "problème de l'oracle".

TObiAs n'intègre pas de notion d'oracle : le testeur ne peut pas décrire explicitement les résultats attendus et TObiAs ne calcule pas cet oracle. Lorsque TObiAs est intégré au sein de l'environnement COTE, il produit les objectifs de test et TGV se charge de les compléter pour produire des tests abstraits. Les verdicts associés aux tests abstraits (i.e. l'oracle) sont calculés à partir de la spécification comportementale donnée en entrée à TGV.

Il existe plusieurs façons de modéliser la spécification d'un programme. TGV utilise des IOLTS, bien adaptés à la spécification des protocoles. Une autre approche (utilisée par exemple dans le domaine des systèmes d'information) consiste à modéliser la spécification en termes d'assertions (invariants, pré/post conditions). Le choix d'une modélisation par rapport à l'autre dépend souvent des informations comprises dans la spécification. Dans les deux cas il faut trouver le moyen d'obtenir un oracle. Deux façons de faire sont envisageables : soit en prévoyant un mécanisme permettant à TObiAs de calculer les oracles associés aux tests produits ; soit en exploitant un mécanisme extérieur qui évalue l'oracle à l'exécution.

La première solution nécessite de prendre en compte dans TOBiAs la spécification au delà du diagramme de classes. Il faudrait pour cela effectuer de profonds changements en calculant les résultats des tests ou en introduisant les valeurs de retours d'appels dans le langage de schéma de tests. Nous avons donc exploré la deuxième solution. Pour cela, l'ingénieur de test doit utiliser une spécification et vérifier que les résultats satisfont à certaines propriétés. Nous avons associé à l'utilisation de TObiAs deux langages offrant cette possibilité : VDM (section 5.1.1) et JML (section 5.1.2).

Dans le cadre de l'utilisation de TObiAs pour générer des tests en VDM et en Java, l'outil ne génère plus des objectifs de test mais des tests abstraits. Pour cela, nous utilisons le moteur de génération des objectifs de test de TObiAs mais nous faisons certaines restrictions. Ainsi les constructions d'alternative, co-région, séquences strictes, négations et boucles non bornées (avec le symbole "*") ne sont plus admises. Les séquences produites sont considérées comme étant des séquences strictes et sont des tests abstraits.

5.1.1 Extension VDM

VDM [FL98, Jon90] est un langage où les spécifications sont exprimées sous la forme d'invariants et de pré/post conditions sur les opérations. Les prédicats portent sur les variables du programme.

Les travaux exposés dans [ML02] ont nécessité l'utilisation de l'environnement VDMTools [Gro00]. VDMTools offre une infrastructure de test où une implantation peut être comparée à une spécification lors de l'exécution des tests. Dans VDMtools, les invariants sont vérifiés après chaque instruction d'une opération, en plus des vérifications en entrée et sortie des opérations. L'outil offre aussi une mesure de couverture des suites de test basée sur le pourcentage d'instructions exécutées.

TObiAs peut générer des fichiers de test au format de VDMTools. Dans ce cas les schémas de test produisent des tests abstraits traduits sous la forme de tests exécutables pour VDM, et non pas des objectifs de test comme nous l'avons vu dans l'introduction au problème de l'oracle. TObiAs crée un fichier regroupant l'ensemble des tests générés pour un schéma de test, ce fichier sert d'entrée à VDMTools en plus de l'implémentation au format VDM. VDMTools exécute ensuite l'ensemble des tests de manière séquentielle et disjointe[1]. Si des erreurs sont trouvées lors de l'exécution du jeu de test, VDMTools le notifie à l'utilisateur.

Etude de cas de gestion de groupes d'étudiants

En permettant à TObiAs d'exprimer des schémas de test adaptés à VDM, nous pouvons comparer l'effort requis pour obtenir une suite de test via la création des schémas de test avec celui requis pour écrire manuellement une suite de test pour VDMTools. Une étude de cas a été développée dans [ML02]. Il s'agit d'une gestion de groupes d'étudiants. Une classe d'étudiants est divisée en groupes qui doivent satisfaire les contraintes suivantes correspondant à l'invariant de la spécification :

- un groupe est composé de 3 à 5 étudiants, si une classe comporte moins de 3 étudiants, ils doivent appartenir au même groupe ;
- le groupe le plus grand doit comporter au plus 1 étudiant de plus que le groupe le plus petit ;

[1] Chaque cas de test est indépendant des autres cas de test.

– chaque étudiant appartient à un groupe et un seul.

Cette application de gestion de groupes d'étudiants comporte plusieurs opérations dont nous donnons ici une spécification informelle. Cette spécification est non déterministe :

– `load_table` charge un état initial du système avec des groupes d'étudiants conformes ;
– `swap_two_students` intervertit deux étudiants ;
– `add_group` ajoute un nouveau groupe au système et répartit les étudiants en conséquence ;
– `add_student` ajoute un nouvel étudiant au système et répartit les étudiants en conséquence ;
– `add_set_students` appelle `add_student` pour chaque étudiant d'un ensemble d'étudiants donné en paramètre ;
– `delete_student` retire un étudiant du système et répartit les étudiants restants en conséquence ;
– `change_student` déplace un étudiant donné vers un groupe donné ;
– `mix` et `mix_better` modifient de manière arbitraire la composition des groupes sans ajouter ou retirer d'étudiants ;
– `print_groups` affiche les étudiants groupe par groupe.

Deux suites de test relatives à cette spécification on tété construites, l'une élaborée de façon manuelle dans VDMTools et la seconde construite à partir de schémas de test. Deux expériences ont été menées avec ces suites de test. La première expérience a consisté à exécuter ces suites de test sur une implantation réalisée au préalable. La deuxième expérience a consisté à les exécuter sur 14 implantations développées par 14 groupes d'étudiants de DESS.

La première expérience a permis de voir que les deux suites de test couvraient la spécification suivant les mêmes proportions, soit 85% des instructions. Néanmoins le nombre d'appels est nettement plus important avec les cas de test produits par TObiAs qu'avec la campagne effectuée manuellement (tableau 5.1)[2]. Alors que la suite manuelle teste entre une dizaine de fois et quelques centaines de fois chaque méthode, la suite générée par TObiAs teste les méthodes plusieurs milliers de fois. Cette intensification des tests a permis de détecter deux erreurs non décelées par la suite manuelle. Ces deux erreurs interviennent dans certains

[2] $\surd$ signifie 100% de couverture

	Suite manuelle		TObiAs	
	#Appels	couverture	#Appels	couverture
print_groups	1	$\surd$	0	0%
add_set_students	1	$\surd$	3600	$\surd$
add_student	9	$\surd$	8208	$\surd$
add_group	2	$\surd$	1872	$\surd$
change_student	3	$\surd$	1080	$\surd$
load_table	16	$\surd$	4320	$\surd$
swap_two_students	12	$\surd$	4662	$\surd$
swap_two_studentsF	4	$\surd$	1620	$\surd$
delete_student	5	$\surd$	4320	$\surd$
maximum	316	95%	80515	95%
mix	1	$\surd$	1440	$\surd$
mix_better	0	0%	1440	48%
minimum	302	95%	75366	95%
max_nb_of_gr	2	$\surd$	1972	$\surd$
min_nb_of_gr	0	0%	0	0%
table2partition	379	$\surd$	123974	$\surd$
table2table-inverse	0	0%	0	0%
size-max-gr	194	94%	65375	94%
size-min-gr	183	94%	58599	94%
Couverture Totale		**85%**		**85%**

TAB. 5.1 – Une comparaison des mesures de couvertures effectuées par l'outil VDMTools entre une suite de test construite manuellement et une suite de test générée à partir de schémas de test avec TObiAs

cas d'initialisation du système. En testant systématiquement chaque méthode pour toutes les instanciations identifiées par l'ingénieur de test, la suite de test générée par TObiAs a permis de détecter ces erreurs.

Pour la deuxième expérience, la campagne manuelle a détecté 9 implantations erronées de l'opération `add_student` pour 13 détectées par la campagne générée avec TObiAs. De plus la campagne générée par TObiAs est la seule à avoir détectée une version erronée de `add_group`.

Ces expériences permettent de mettre en avant l'intérêt du caractère combinatoire de TObiAs. La création des deux campagnes a requis le même effort de la part des concepteurs des tests (une cinquantaine de lignes), mais celle obtenue à partir des schémas de test permet de détecter plus d'erreurs, grâce à son caractère systématique et au plus grand nombre de tests générés.

5.1.2 Extension JML

JML (Java Modelling Language) est un langage de spécification pour les programmes JAVA à base d'assertions : invariants, pré et post conditions. Ce principe d'assertion est utilisé dans de nombreux langages de spécification, tels que VDM que nous venons de voir ou encore les langages B [Abr96] et Z [Abr77]. Le langage JML est syntaxiquement proche du langage JAVA, cela offre au programmeur une plus grande aisance pour exprimer les spécifications.

Les assertions JML apparaissent dans le code JAVA comme des commentaires portant des étiquettes distinctives pour définir les invariants, pré et post-conditions. Les assertions JML ne sont donc pas compilées par le compilateur JAVA. Il faut utiliser un compilateur spécifique, nommé JMLC, pour compiler les assertions JML[3]. Les assertions JML peuvent devenir l'oracle des tests lors de leur exécution.

JUNIT[4] est un framework de test unitaire pour programmes JAVA. Il permet d'exécuter une suite de tests JAVA en utilisant la spécification JML comme oracle. Cet outil est la plate-forme d'exécution et d'interpretation des tests que nous uti-

[3]Une petite partie de la syntaxe du langage JML reste toutefois non exécutable. C'est le cas de certaines formes de quantificateurs universels.

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

lisons avec TObiAs.

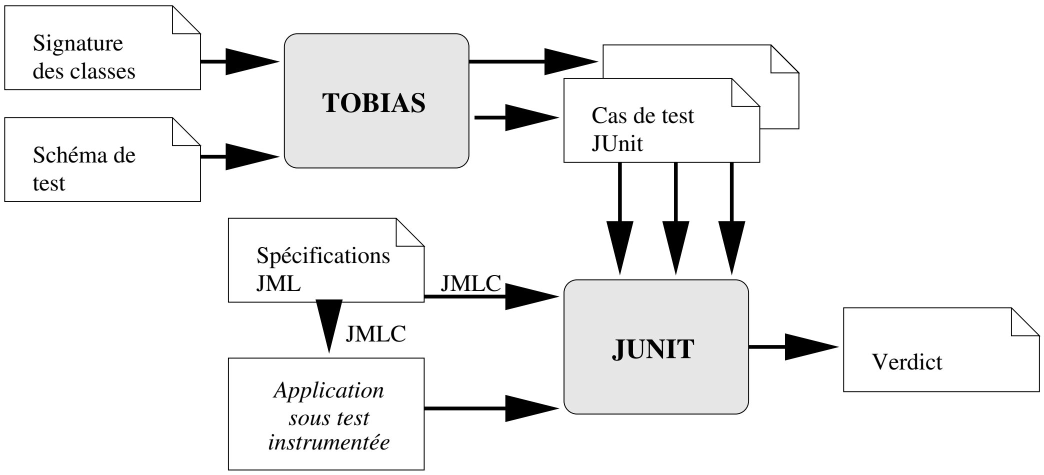


FIG. 5.1 – Utilisation de TObiAs avec des spécifications JML et un pilote JUNIT

La figure 5.1 présente la façon dont nous avons utilisé TObiAs avec JML et JUNIT dans [LdBMB04]. Le programme est spécifié en JML et compilé à l'aide du compilateur JMLC. L'ingénieur de test n'a plus à se préoccuper du fait d'associer un oracle à ses tests, celui-ci fait déjà partie de la spécification du programme à tester. Tous les tests abstraits générés pour un schéma de test sont réunis dans un seul fichier généré par TObiAs sous la forme de cas de test JAVA après traduction de la part de TObiAs. Les cas de test sont représentés par des appels de méthode dont le nom commence nécessairement par "`test`", afin d'être reconnus comme étant des cas de test par JUNIT. L'application JAVA et sa spécification JML sont compilées par le compilateur JMLC. Enfin JUNIT exécute de manière automatique chacun des cas de test en utilisant les assertions JML comme oracle. JUNIT établit pour chaque test un verdict indiquant s'il a réussi ou échoué, et dans le deuxième cas il permet de localiser l'erreur. Dans la section 6.4 nous présentons une étude de cas utilisant TObiAs et JML.

D'autres outils ont une approche semblable à la nôtre, à savoir générer de manière automatique ou semi-automatique des cas de test pour des programmes JAVA en utilisant des assertions JML comme oracle. JML-UNIT [CL02] est une évolution de JUNIT puisqu'il intègre directement les assertions JML associées au programme JAVA. Il permet d'effectuer du test combinatoire en faisant varier les paramètres des constructeurs des classes, ainsi que les paramètres des méthodes.

Par contre chaque cas de test se limite à un appel de constructeur suivi d'un seul appel de méthode, ce qui restreint les possibilités pour tester des comportements complexes. L'outil Korat [BKM02] est un outil de génération automatique de cas de test qui utilise lui aussi des assertions JML comme oracle. Nous présenterons plus en détail cet outil dans la section 5.4.

5.2 La sélection des valeurs avec UCasting

La sélection des valeurs de test pour les groupes associés aux schémas de test est de la responsabilité du testeur. Néanmoins, choisir un ensemble de valeurs pertinentes pour chaque paramètre de chaque méthode peut s'avérer être une tâche longue et complexe.

Pour nous affranchir en partie de cette tâche, nous avons étudié le couplage de TObiAs avec un outil de génération de valeurs de test : UCasting [AJ03]. UCasting est la version dédiée aux spécifications décrites en UML de l'outil Casting [ABM97b, ABM97a, Aer98]. L'utilisation de UCasting nécessite un diagramme de classes et un diagramme d'états-transitions pour chaque classe du système. Ces diagrammes sont annotés en OCL pour exprimer des invariants, des pré et post conditions ou des gardes associées aux transitions. UCasting peut employer ces informations de deux façons différentes pour générer des cas de test :

- UCasting procède à la décomposition des contraintes OCL suivant des stratégies prédéfinies, ou définies par l'utilisateur. Ces stratégies permettent d'obtenir des valeurs qui, par exemple, testent une même transition de plusieurs façons. Par exemple une transition ayant comme garde une contrainte OCL de la forme $A \geq B$ sera décomposée en deux transitions gardées respectivement par $A > B$ et $A = B$. Une fois toutes les transitions décomposées, c'est le nouveau système de transition que UCasting prend en compte.
- UCasting peut, à partir d'une séquence d'appel, définir la contrainte correspondant à l'enchaînement des pré et post-conditions de cette séquence. Le solveur de contraintes associé à UCasting peut alors évaluer cette contrainte suivant deux cas de figure. Soit toutes les valeurs associées à la contrainte sont définies et le solveur l'évalue à vrai ou à faux. Soit certaines valeurs ne sont pas définies et le solveur propose des valeurs permettant d'évaluer

la contrainte à vrai.

La génération des cas de test avec UCasting est décrite dans [AJ03]. Dans ce cas là UCasting travaille seul à partir de la spécification UML. La définition et le traitement d'une contrainte pour une séquence d'appels nécessite d'avoir cette séquence en entrée de UCasting. TObiAs peut être utilisé pour produire de telles séquences.

L'apport du couplage UCasting TObiAs

Le couplage de ces deux outils exploite principalement le solveur de contraintes de UCasting pour valider ou compléter des tests abstraits produits par TObiAs. Une option de TObiAs permet en effet de générer, à partir des schémas de test, des tests abstraits sans définir de valeurs pour les paramètres des méthodes. UCasting génère des contraintes correspondant aux chemins des tests abstraits (vus comme des séquences) puis vérifie si ces contraintes sont satisfiables. Si c'est le cas, cela permet de valider les séquences et de donner des solutions pour les valeurs des paramètres en accord avec les invariants et les pré et post conditions. UCasting peut donc soulager le testeur de la recherche d'une partie des valeurs intéressantes pour ses tests.

Les limitations du couplage UCasting-TObiAs

L'utilisation de UCasting est cependant limitée par de plusieurs points qui restreignent la portée de cette combinaison d'outils.

1. Quand nous avons fait cette expérimentation, UCasting ne gérait que les paramètres de type `entier` et ne disposait pas de structures de donnée ni de quantificateurs. Ceci limite grandement les méthodes qui peuvent être gérées dans les séquences d'appel.
2. Seuls des systèmes simples sont pris en compte, à savoir des systèmes ne comportant qu'une seule instance d'une seule classe.
3. UCasting ne prend en compte que des séquences strictes. Ceci n'est pas vraiment une restriction pour UCasting qui est dédié à la génération de cas de test, mais c'est une restriction pour TObiAs. Ceci signifie que le couplage ne peut être utilisé que dans le cas où nous souhaitons générer des tests abstraits à partir des schémas de test, et non pas des objectifs de test dans une combinaison TObiAs - UCasting - TGV.

Les limitations 1 et 2, qui sont pour nous les plus contraignantes, devraient être levées dans les versions à venir de UCasting. La restriction 3 quant à elle nécessite que nous utilisions le couplage TObiAs - UCasting dans le seul cadre de la génération de tests abstraits, comme nous le faisons pour les langages VDM et JML.

5.3 L'explosion combinatoire

TObiAs permet de générer un grand nombre de séquences de test. Si cela est sa force, c'est aussi sa faiblesse si nous n'arrivons pas à contrôler ce nombre. En effet les capacités mémoire des machines sur lesquelles TObiAs est exécuté sont limitées. Même s'il est possible de générer un grand nombre de séquences, leur temps d'exécution total peut s'avérer long, voire coûteux. Enfin, le dépouillement des résultats des tests peut s'avérer fastidieux.

Pour résoudre en partie le problème de l'explosion combinatoire, nous avons exploré plusieurs pistes avec des étudiants de DEA [Bou02, Beg02, Kes03]. Nous présentons ces différentes pistes avant de décrire des outils proches de TObiAs, tels que AETG et Korat, qui proposent aussi des solutions pour maîtriser l'explosion combinatoire.

5.3.1 Contraindre les séquences

Le caractère combinatoire du dépliage des schémas de test mène souvent à un trop grand nombre de tests abstraits ou d'objectifs de test. Une façon de contrôler le nombre de tests produits est d'associer des contraintes aux schémas de test. Nous avons développé un langage de contraintes pour les schémas de test [Beg02]. Ce langage permet d'exprimer des contraintes sur les paramètres des méthodes à l'aide de prédicats du premier ordre.

Ce langage, tel qu'il a été défini, permet de définir des contraintes servant à filtrer les objectifs de test, à la génération, pour ne garder que ceux souhaités. TObiAs utilise actuellement VDM pour interpréter des prédicats de filtrage associés aux schémas de test.

5.3.2 Générer des séquences pertinentes

Les "séquences[5] pertinentes" correspondent à des suites d'appel telles que chaque appel de ces suites d'appel soit autorisé par la spécification. TObiAs prenant en compte peu d'aspects de la spécification, il est possible de définir des séquences dites "non pertinentes" car elles comportent des appels qui ne sont pas permis par la spécification. Le travail d'Hocine Bouldjedri [Bou02] consiste à accroître le nombre d'informations que TObiAs retire des diagrammes UML afin de mieux contraindre les séquences produites, et ainsi de limiter le nombre de séquences non pertinentes.

La notion de séquence pertinente consiste à prendre en compte le sens des associations entre les classes du diagramme de classes. En UML, deux classes ne peuvent échanger un message que si elles sont liées par une association, ces associations comportent des sens de navigation comme on peut le voir sur la figure 5.2. Dans cet exemple la classe A peut faire appel à la classe B, la classe B à la classe A et la classe C à la classe A. Par contre la classe A ne peut pas faire appel à la classe C car la relation liant ces deux classes n'est navigable que dans le sens de C vers A.

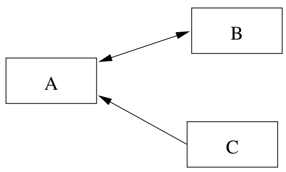


FIG. 5.2 – Exemple de diagramme de classes pris en compte par TOBiAs

Actuellement TObiAs prend en compte l'existence d'une association entre deux classes, mais ne considère pas le sens de navigation de celle-ci. Une première amélioration de la prise en compte des informations fournies par la notation UML consiste à prendre en compte le sens de navigation des relations. Le tableau 5.2 présente les associations possibles dans TObiAs en prenant ou non en compte le sens de navigation des relations pour l'exemple de la figure 5.2.

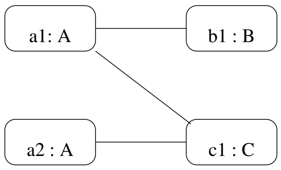
Un diagramme de classes UML ne nous permet pas d'obtenir d'informations sur les relations entre instances. Par contre l'environnement d'une campagne de

[5] Le terme séquence représente des tests abstraits et des objectifs de test.

Les appels possibles entre les classes		sans prise en compte des relations	avec prise en compte des relations
A !B	A fait appel à B	√	√
A !C	A fait appel à C	√	×
B !A	B fait appel à A	√	√
B !C	B fait appel à C	×	×
C !A	C fait appel à A	√	√
C !B	C fait appel à B	×	×

TAB. 5.2 – Les appels considérés valides par TOBiAs suivant la prise en compte ou non du sens des relations

test fournit de l'information sur un déploiement. Ce déploiement fournit des informations sur les associations existantes, non plus au niveau des classes, mais au niveau des instances. Il faut pour cela accéder à un diagramme d'objets comme cela était suggéré dans la section 3.1. La figure 5.3 présente les optimisations qui peuvent être obtenues en prenant en compte un diagramme d'objets pour un déploiement donné, en plus d'une prise en compte du diagramme de classes avec le sens de navigation. Dans ce diagramme, on exploite par exemple l'absence de lien entre les instances a2 et b1 pour interdire l'envoi de messages entre ces objets (alors qu'une association existe entre les classes correspondantes).



a1!b1 √	a2!c1 ×	b1!c1 ×	c1!b1 ×
a1!c1 ×	b1!a1 √	c1!a1 √	
a2!b1 ×	b1!a2 ×	c1!a2 √	

FIG. 5.3 – Prise en compte d'un diagramme d'objets

5.3.3 Optimiser le traitement des séquences

Un troisième étudiant de DEA, Medhi Kessis, a proposé des solutions pour réduire à l'exécution l'explosion combinatoire des tests, dans le cadre de JML [Kes03][6].

[6]Ce travail effectué avec JML s'applique aussi à VDM.

Jusqu'à présent nous utilisions JUNIT pour exécuter les cas de test produits par TObiAs dans le cadre d'une génération de tests JAVA en utilisant une spécification JML composée de pré et post conditions ainsi que d'invariants. L'un des problèmes du pilote JUNIT vient du fait qu'il exécute chacun des tests sans prendre en compte le résultat des tests précédents. Si nous prenons l'ensemble de cas de test suivant :

1. init() ; debit(75)
2. init() ; debit(75) ; debit(75)
3. init() ; debit(75) ; credit(50)
4. init() ; debit(75) ; credit(100)
5. init() ; credit(50)
6. init() ; credit(50) ; debit(75)
7. init() ; credit(50) ; credit(50)
8. init() ; credit(50) ; credit(100)
9. init() ; credit(100)
10. init() ; credit(100) ; debit(75)
11. init() ; credit(100) ; credit(50)
12. init() ; credit(100) ; credit(100)

JUNIT rejette le cas de test **1** car la pré-condition de debit n'est pas respectée lors de l'exécution. Par conséquent, on sait que les cas de test **2**, **3** et **4** seront eux aussi rejetés. Malgré cela JUNIT exécute les 4 cas de test car ils sont traités de manière indépendante les uns des autres. L'approche choisie pour optimiser l'exécution des cas de test consiste à organiser les cas de test sous la forme d'arbres (Figure 5.4). Le pilote de test "JML-Tobias" a été développé pour traiter une liste de tests et se substitue à JUNIT pour exécuter et évaluer si ces tests détectent ou pas des erreurs liées à la violation d'une contrainte. Ce nouveau pilote de tests évite d'exécuter les fils d'un nœud dont l'exécution a mené à une erreur.

Avec le pilote JML-Tobias, seuls les cas de test **1**, **6**, **7**, **8**, **10**, **11** et **12** seront exécutés, comme nous pouvons le voir sur la figure 5.5. En effet le cas de test **1** est préfixe des cas de test **2**, **3** et **4**. Si une erreur survient sur l'opération debit(75), cela revient à avoir une erreur lors de l'exécution du cas de test **1**. Il n'est donc pas nécessaire d'exécuter les cas de test dont il est le préfixe. Dans le cas du cas de test **5**, si aucune erreur ne survient au terme de son exécution, on poursuit cette exécution avec les branches correspondant aux tests **6**, **7** et **8**. De cette façon le cas de test **5** n'est pas joué en tant que tel mais en tant que préfixe d'autres cas de test. Des deux pilotes de test, seul JML-TObiAs possède cette optimisation.

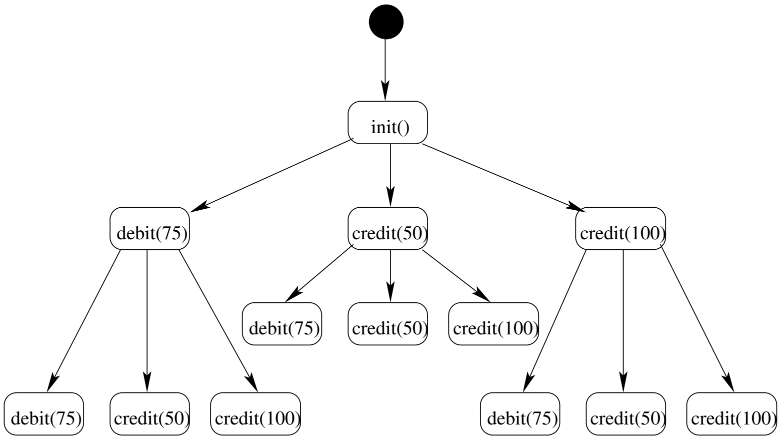


FIG. 5.4 – Représentation des cas de tests sous forme d'arbre

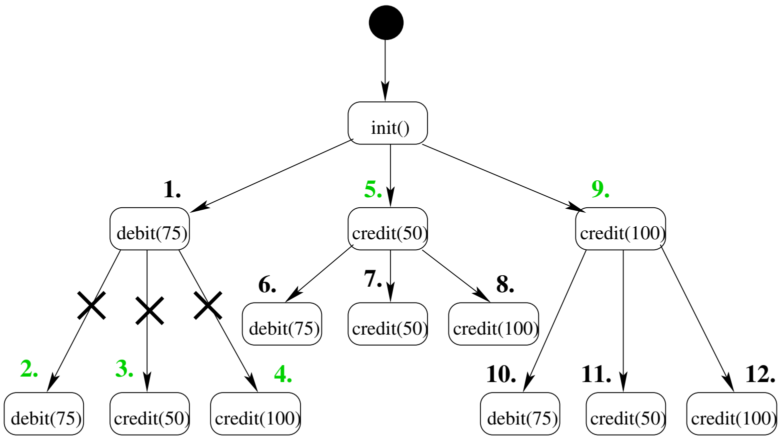


FIG. 5.5 – Illustration du mode d'exécution des cas de test avec le pilote JML-Tobias

Dans le cadre d'une étude portant sur un système de gestion de tampons [LdBMB04], nous avons constaté une réduction d'un facteur 6 du nombre de tests exécutés et le pilote JML-TObiAs s'est montré plus rapide que le pilote JUNIT et a relevé les mêmes erreurs. Cette optimisation se fait sans intervention de l'utilisateur et n'élimine que des tests inutiles.

5.4 Autres outils combinatoires

TObiAs se base sur des techniques simples de test combinatoire sur les paramètres, les méthodes et les instances. Il existe d'autres outils basés sur des hypothèses de test portant sur les aspects combinatoires des données de test. Nous présentons plus en détail deux méthodes AETG et Korat. AETG se base sur des hypothèses de test particulières afin de réduire l'explosion combinatoire, et Korat effectue du test unitaire de méthodes JAVA de manière exhaustive pour des structures de données restreintes.

5.4.1 AETG

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

Le tableau 5.3 décrit trois paramètres **A**, **B** et **C** qui peuvent chacun prendre les valeurs 0 ou 1. Il existe huit combinaisons possibles de valeurs entre ces trois paramètres, si nous souhaitons couvrir seulement les combinaisons deux à deux des paramètres. Nous pouvons dans ce cas nous contenter des lignes 2, 3, 4 et 5. Nous aurions aussi bien pu considérer les lignes 1, 4 , 6 et 7.

K. Burr et W. Young [BY98] présentent l'exemple d'une méthode avec 13 paramètres pouvant tous prendre 3 valeurs différentes. L'ensemble des combinai-

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

TAB. 5.3 – Combinatoire sur trois paramètres binaires

sons représente 1 594 323 cas de test. La couverture par paires d'AETG permet de réduire le nombre de cas de test nécessaires à seulement 19 cas de test.

5.4.2 Korat

Korat [BKM02] est un outil basé sur la même approche que JML-Junit. Cette approche consiste à tester de manière exhaustive une méthode prenant comme paramètre des structures d'objets. Korat permet de générer des ensembles de structures d'objets valides et non isomorphes pour la méthode testée. Des structures d'objets non isomorphes sont des structures qui présentent toutes une architecture différente. C'est le cas, par exemple, des structures d'arbre de la figure 5.6.

La génération de structures non isomorphes est la méthode utilisée par Korat pour limiter l'explosion combinatoire. Korat repose sur la "*small scope hypothesis*" [ADKM02] qui constate de manière empirique qu'un grand nombre d'erreurs se manifestent déjà avec des structures de données de petites tailles. Korat permet ainsi de poser des contraintes sur la taille, qui doit être bornée, des structures d'objets. L'outil construit automatiquement tous les cas de test non isomorphes, pour une taille donnée, exécute la méthode pour chaque cas de test et évalue si l'exécution a détecté, ou non, une erreur de la même façon que JML-Junit.

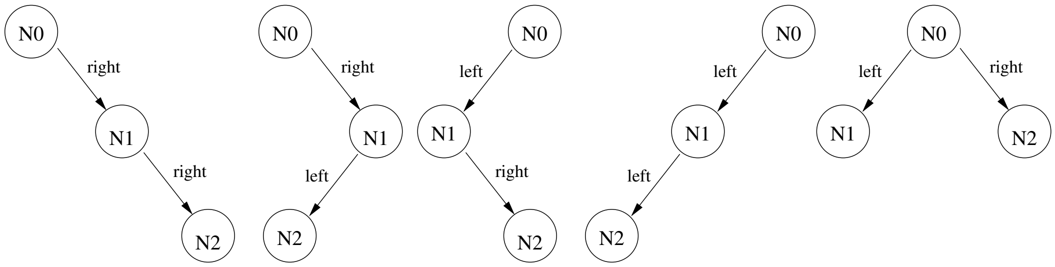


FIG. 5.6 – Arbres générés dans le cas d'arbres binaires non isomorphes de taille 3

Pour ne prendre en compte que des structures d'objets considérés comme correctes par l'ingénieur de test, Korat utilise un prédicat nommé `repOk` défini par le testeur. Ce prédicat est associé à une classe nommée *finitization* qui génère chaque objet tandis que le prédicat *repOk* s'assure que les objets ainsi générés sont valides. Au delà de ce filtre, Korat s'assure que les structures d'objets retenues sont non isomorphes.

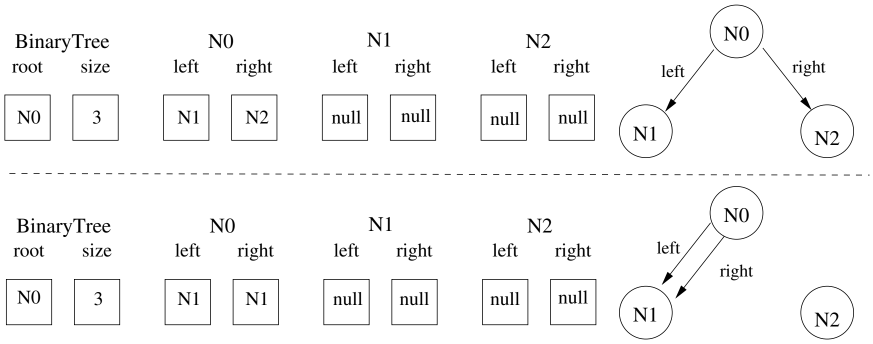


FIG. 5.7 – Un arbre binaire correct (au dessus) et un arbre binaire non correct rejeté (au dessous)

Prenons comme exemple une méthode ayant comme paramètre un arbre binaire de taille trois. Cet arbre comprend trois nœuds (N0, N1 et N2) ainsi qu'une racine (BinaryTree) associée à l'un des trois nœuds. Chaque nœud comporte 2 fils qui peuvent prendre 4 valeurs (null, N0, N1 et N2). Le nombre de configurations possibles est de $(4 * 4)^3 * 4 = 2^{14}$ candidats correspondant aux différentes combinaisons de ces valeurs. La figure 5.7 décrit un candidat qui est un arbre binaire défini comme valide, ainsi qu'un candidat qui n'est pas un arbre binaire valide d'après les contraintes définies dans le prédicat *repOk*[7]. Pour cet exemple Korat génère cinq arbres non isomorphes (figure 5.6), tous représentatifs d'une partition de six arbres distincts correspondant aux 3 ! permutations des nœuds.

L'outil Korat a comme avantage d'être capable de construire de façon combinatoire des objets invoquant la méthode sous test. Mais les cas de test que construit Korat se réduisent à l'exécution d'une seule méthode. Cet outil ne permet pas de tester des séquences de plusieurs méthodes. Néanmoins Korat présente une méthode intéressante pour limiter l'explosion combinatoire dans le cadre de structures d'objets, alors que TObiAs n'est actuellement utilisé qu'avec des données simples.

Résumé et conclusion

Résumé

Comme nous venons de le voir, TObiAs permet de générer aussi bien des objectifs de test pour s'interfacer par exemple avec un outil tel que TGV, que des cas de test. Nous avons expérimenté la génération de cas de test avec deux langages cibles : VDM et JAVA. Ces langages permettent d'utiliser une spécification basée sur l'utilisation de pré et post conditions ainsi que d'invariants (en JML pour le langage JAVA) afin d'obtenir un oracle utilisable à travers des outils tel que VDMTools et JUNIT.

Nous avons aussi exploré différentes pistes pour apporter des améliorations à TObiAs, en déchargeant l'ingénieur de test du choix des valeurs de test avec un couplage des outils TObiAs et UCasting, et en cherchant des réponses au problème de l'explosion combinatoire auquel TObiAs doit faire face.

Pour limiter l'explosion combinatoire nous pouvons exploiter au mieux la spécification et l'état initial du système du point de vue des instances. Nous pouvons aussi filtrer la génération des cas de test et objectifs de test en posant des contraintes logiques, sur les schémas de test. Nous avons aussi cherché à réduire le temps d'exécution des cas de test pour JAVA. Pour cela nous avons créé un pilote de test optimisé pour les cas de test qui, s'ils sont issus d'un même schéma

[7]Un nœud ne peut pas être fils plus d'une fois.

de test, ont de nombreuses similitudes. Ces similitudes portent sur des préfixes commun qui permettent de représenter l'ensemble des cas de test comme une arborescence, et d'utiliser cette arborescence pour exécuter l'ensemble des tests de façon optimisée.

Nous avons aussi étudié d'autres outils de test combinatoire : AETG et Korat. Korat permet de tester des méthodes avec des structures de données, mais les cas de test se limitent à un seul appel de méthode. AETG se base sur une hypothèse d'indépendance entre les variables des méthodes pour limiter le nombre de tests nécessaires. Cette méthode a été reprise dans TObiAs.

Conclusions

Ce chapitre permet de faire ressortir trois points importants sur les possibilités de TObiAs :

1. TObiAs peut être mis en œuvre dans des contextes technologiques divers (TGV, VDM, JML) basés sur la disponibilité d'une spécification exécutable. Cette flexibilité vient de la notion de test abstrait, indépendante d'une technologie particulière.
2. Plusieurs solutions peuvent être mises en œuvre pour faire face à l'explosion combinatoire. Nous distinguons deux types de solutions : a priori par analyse de la spécification (filtrage à la génération), et a posteriori par analyse des tests (filtrage à l'exécution et génération par paires). Même si elles ne résolvent pas complètement le problème, elles complètent de manière intéressante l'outil.
3. TObiAs peut être combiné avec d'autres outils, notamment pour la génération de valeurs de test. Il apparaît comme complémentaire d'outils comme UCasting ou BZTT, même si le couplage n'a pas encore été tenté avec ce dernier outil.

Le prochain chapitre présente des illustrations de l'utilisation de TObiAs.

Chapitre 6

Etudes de cas effectuées avec TObiAs

Ce chapitre comporte deux parties distinctes illustrant l'utilisation de TObiAs.

La première partie correspond à la section 6.1 et présente une illustration de l'utilisation de TObiAs et des schémas de test dans le cadre d'une étude de cas d'un système d'ascenseurs. Cette étude de cas sera poursuivie dans la partie 3 de cette thèse.

La deuxième partie présente deux expérimentations menées sur une étude de cas développée par Gemplus dans le cadre du projet COTE pour valider les outils et la chaîne d'outils du projet.

Après une présentation de l'étude de cas considérée dans la section 6.2, la section 6.3 présente le travail mené par les chercheurs de Gemplus sur la façon d'utiliser TObiAs à partir d'un plan de test pour produire des objectifs de test pour TGV. Cette expérimentation nous a permis d'évaluer l'utilité et les apports de TObiAs dans un cadre industriel.

La section 6.4 correspond aux expérimentations que nous avons faites sur cette étude avec différentes approches pour tester une implantation Java du système bancaire. Cette implantation étant dotée d'une spécification JML, nous avons pu utiliser TObiAs avec le pilote de test JML-Tobias que nous avons présenté dans la section 5.1.2.

6.1 Etude de cas d'un système d'ascenseurs

Dans cette section nous allons nous appuyer sur une étude de cas pour illustrer les notions de schémas de test, montrer de quelle façon nous pouvons produire des schémas de test avec TObiAs et comment ceux-ci sont utilisables pour générer rapidement de nombreux objectifs de test.

La première partie de cette section présente la spécification de cette étude de cas avec un cahier des charges associé. Nous présenterons ensuite le déploiement sur lequel nous avons travaillé. La troisième partie est consacrée à la manière dont nous avons utilisé les outils TObiAs et TGV à notre disposition ainsi que les résultats obtenus. Enfin nous tirons les conclusions de ce que nous a apporté cette étude de cas dans la dernière section de ce chapitre. La suite de cette étude de cas se trouve dans la section 10 qui présente la mise en œuvre d'une mesure de couverture sur les schémas de test.

6.1.1 La spécification du système étudié

Le système pris en compte est un ensemble d'ascenseurs évoluant tous sur les mêmes étages avec de multiples utilisateurs qui peuvent prendre les ascenseurs ou utiliser les escaliers pour changer d'étage. L'état de chaque ascenseur est constamment visible grâce à un panneau d'affichage qui retransmet chaque mouvement et position de l'ascenseur. Il n'y a qu'un seul panneau d'affichage pour l'ensemble des ascenseurs. La figure 6.1 correspond au diagramme de classes de ce système. La figure 6.5 de la page 81 présente le déploiement de l'application et l'état initial par défaut de chaque instance.

Le système comporte trois classes distinctes que nous détaillons :

– **UTILISATEUR :** chaque instance de cette classe représente une personne physique qui manipulerait les ascenseurs du système. Un utilisateur peut se rendre d'un étage à un autre par deux moyens, soit en utilisant un ascenseur (méthode *requête_étage(as,et)*), soit en prenant les escaliers (méthode *aller_à(et)*). Si l'utilisateur décide de prendre un ascenseur, l'ascenseur doit être présent au même étage que l'utilisateur. L'utilisateur peut appeler un ascenseur situé à un autre étage (méthode *appeler(as)*) pour le faire venir. A tout moment l'utilisateur peut nous renseigner sur sa position (méthode *ma_position(et)*). La classe UTILISATEUR comporte deux variables d'état, *courant* et *destination* correspondant respectivement à l'étage où il se situe

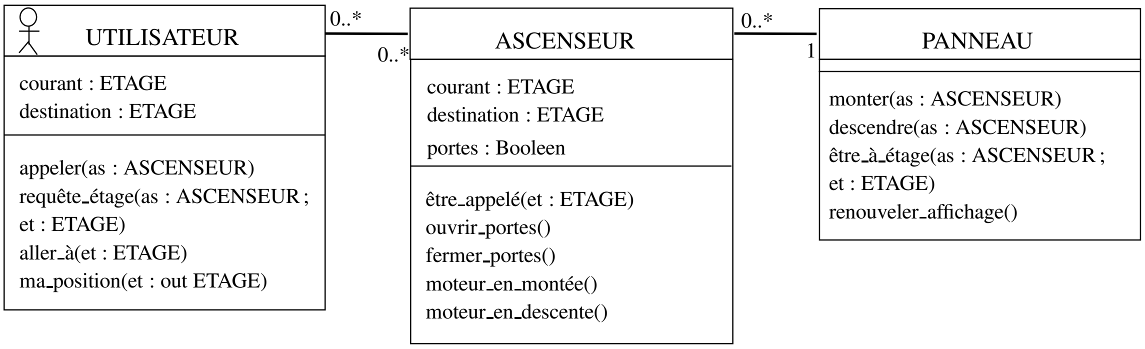


FIG. 6.1 – Diagramme de classes du système

et l'étage où il compte se rendre.

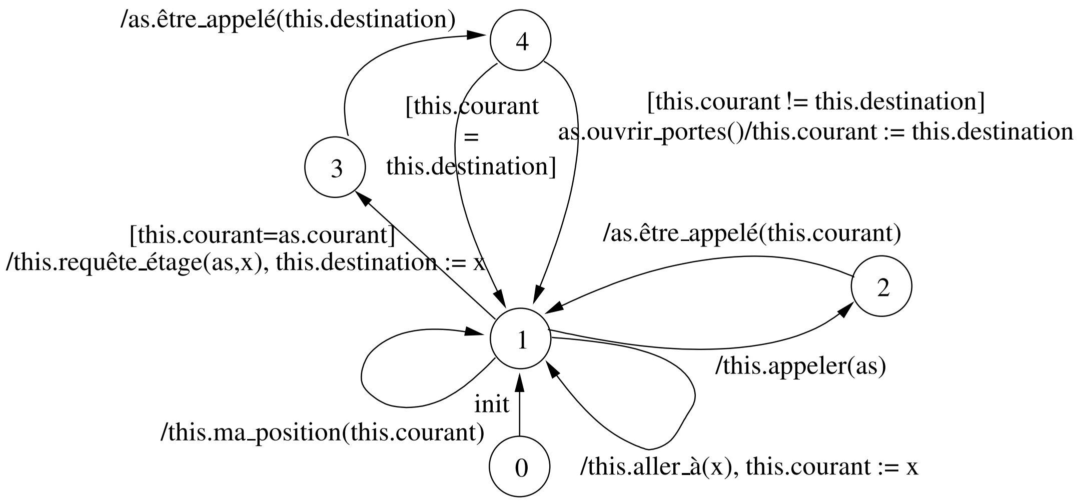


FIG. 6.2 – Diagramme d'états-transitions de la classe UTILISATEUR

Diagramme d'états-transitions : (figure 6.2) Soit une instance utilisateur de la classe UTILISATEUR. Dans l'état initial 1, l'utilisateur peut appeler un ascenseur as pour que celui-ci se rende à l'étage $courant$ de l'utilisateur (boucle menant à l'état 2). L'utilisateur peut changer son état courant en se rendant à l'étage x avec la méthode *aller_à(x)* ou il peut dire quel est son état courant avec la méthode *ma_position(this.courant)*. Si l'ascenseur as est à l'étage $courant$ de l'utilisateur, l'utilisateur peut demander à cet ascenseur de se rendre à l'étage x qui devient l'étage $destination$ de l'utilisateur. Si l'étage de destination est l'étage courant, l'ascenseur ne fait rien et l'état de l'utilisateur passe de 4 à 1. Si l'étage de destination est différent de l'étage courant, l'ascenseur effectue plusieurs actions qui se terminent par l'ouverture des portes. A ce moment l'étage destination de l'utilisateur devient l'étage courant et il repasse dans l'état 1.

– **ASCENSEUR :** une instance de cette classe correspond au comportement d'un ascenseur. Un ascenseur répond à une demande (méthode *être_appelé(et)*) pour se rendre à un étage donné. Pour cela il doit monter (méthode *moteur_en_montée()*) ou descendre (méthode *moteur_en_descente()*). Un préalable à tout mouvement de l'ascenseur est la fermeture des portes (méthode *fermer_portes*). Arrivé à l'étage désiré l'ascenseur peut rouvrir ses portes (méthode *ouvrir_portes()*). Chaque instance de cette classe comporte 3 variables d'état : $courant$ définit l'étage où est stationné l'ascenseur, $destination$ est l'étage vers lequel se dirige l'ascenseur, et $portes$ prend pour valeur $true$ si les portes sont ouvertes, $false$ sinon.

  Diagramme d'états-transitions : (figure 6.3) Soit une instance ascenseur de la classe ASCENSEUR. A l'état initial 1, l'ascenseur est portes ouvertes (variable $portes$ à $true$) à l'étage $courant$ comme spécifié dans le diagramme de déploiement page 81. Il est en attente d'une activation de sa méthode *être_appelé(x)*. La $destination$ de l'ascenseur devient l'étage x. Si cet étage est l'étage $courant$, l'ascenseur revient dans l'état 1. Sinon l'ascenseur ferme ses portes avec la méthode *fermer_portes()* et rentre en communication avec le panneau de contrôle pour l'ensemble de sa phase de mouvement. Il attend que le panneau ait bien renouvelé l'affichage pour effectuer la manœuvre correspondante. Une fois arrivé à l'étage $destination$, le panneau est mis à jour et l'étage $destination$ devient l'étage $courant$. Puis les portes s'ouvrent (méthode *ouvrir_portes()*) une fois que le panneau d'affichage a renouvelé l'affichage. L'ascenseur se retrouve à nouveau dans l'état 1 en attente d'un appel.

– **PANNEAU :** il n'existe qu'une seule instance de panneau par système. Le panneau est relié à tous les ascenseurs et affiche en permanence leur état communiqué par les ascenseurs (méthodes *monter(as), descendre(as)* et *être_à_étage(as)*). Après chaque communication de la part d'un ascenseur

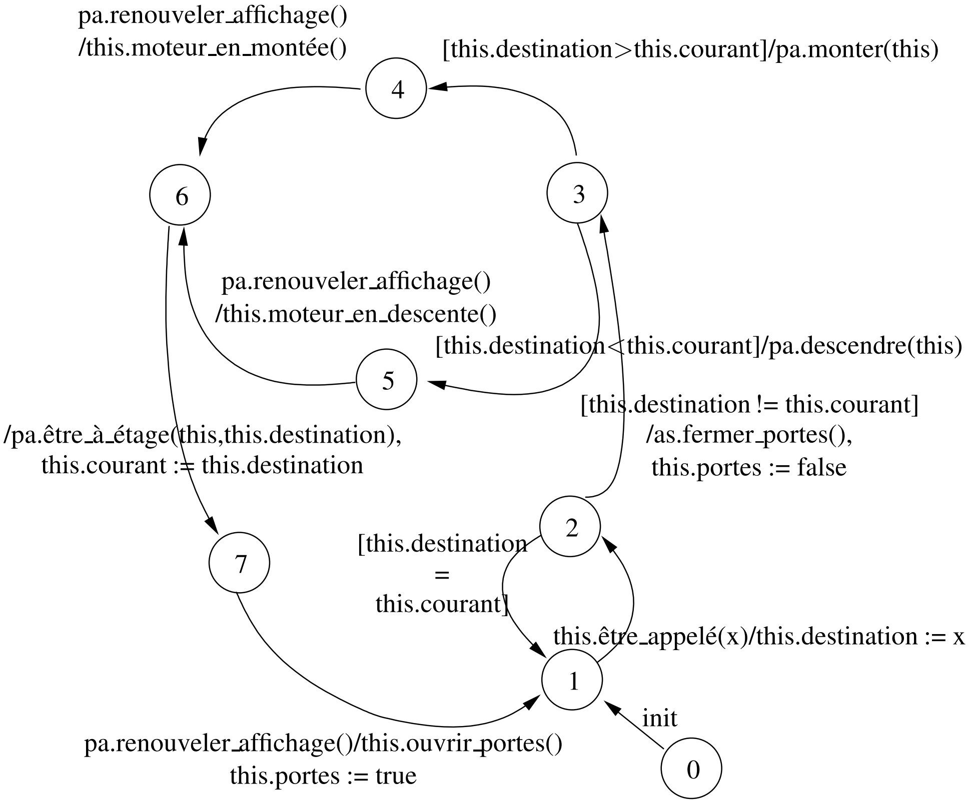


FIG. 6.3 – Diagramme d'états-transitions de la classe ASCENSEUR

le panneau met ses informations à jour (méthode *renouveler_affichage()*).

Diagramme d'états-transitions : (figure 6.4) Soit une instance panneau de la classe PANNEAU. A l'état initial 1, le panneau attend qu'un ascenseur as lui fasse part d'un mouvement prévu : soit en montée avec la méthode *monter(as)* ; soit en descente avec la méthode *descendre(as)*. Le panneau peut aussi attendre qu'un ascenseur as dise être arrivé à l'étage x avec la méthode *être_à_étage(as,x)*. La réception de l'un de ces trois messages amène le panneau dans l'état 2 du diagramme, alors le panneau renouvelle l'affichage en conséquence (ce message permet à l'ascenseur de valider son action) avec la méthode *renouveler_affichage()*. Le panneau se retrouve à nouveau dans l'état 1 et attend un nouveau message de la part d'un ascenseur.

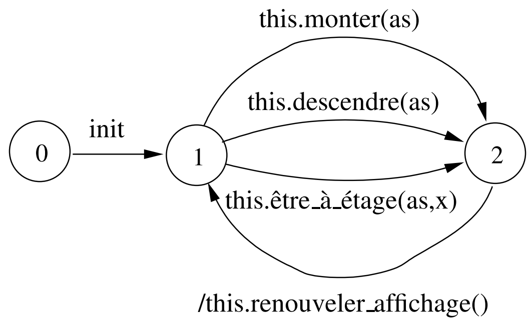


FIG. 6.4 – Diagramme d'états-transitions de la classe PANNEAU

Exigences à tester

Nous considérons six exigences qui doivent être respectées par la spécifications et les différentes implémentations possibles du système.

1. un ascenseur répond correctement aux demandes qui lui sont faites ;
2. lorsqu'un ascenseur monte, cette information est affichée ;
3. lorsqu'un ascenseur descend, cette information est affichée ;
4. lorsqu'un ascenseur est à un étage, cette information est affichée ;
5. un utilisateur peut toujours se rendre à un étage donné ;
6. un ascenseur doit être portes fermées lorsqu'il est en mouvement ;

Dans la section 6.1.3 nous allons formaliser dans l'environnement TObiAs des schémas de test qui correspondent chacun à une ou plusieurs exigences.

6.1.2 Le déploiement pris en compte

Nous considérons le système correspondant à la figure 6.5. Ce système comporte une instance d'UTILISATEUR, deux instances d'ASCENSEUR et une instance de PANNEAU. A l'état initial les instances de la classe ASCENSEUR sont toutes à l'étage le plus bas avec leur variable *courant* désignant cet étage et leurs portes ouvertes. L'instance d'UTILISATEUR se trouve au même étage avec sa variable *courant* correspondant au même étage. Le nombre d'étages considérés est ici de deux.

La spécification comportementale associée à ce déploiement correspond à la mise en parallèle des diagrammes d'états-transitions instanciés de ces quatre

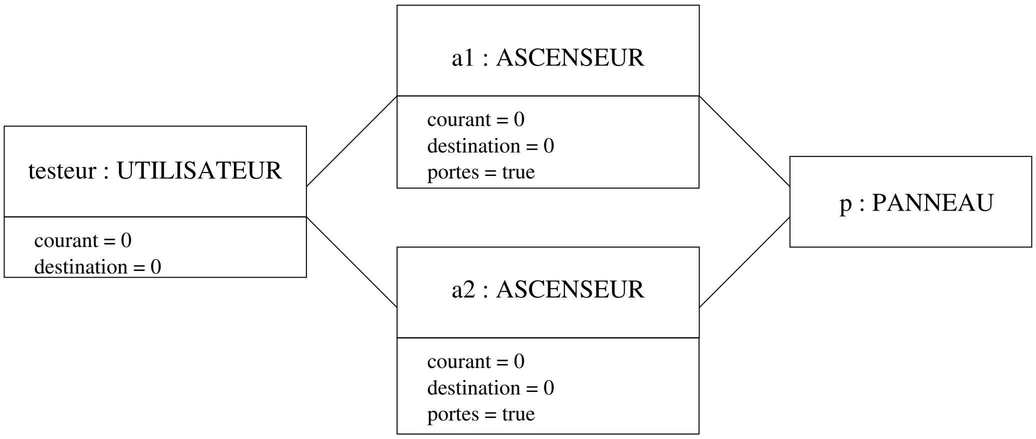


FIG. 6.5 – Diagramme de déploiement pour 1 utilisateur et 2 ascenseurs

instances. Cette spécification a été traduite "à la main" en un IOLTS pour servir d'entrée à TGV. La configuration testée, avec l'utilisation de deux étages, comprend 153 états et 201 transitions. La taille du diagramme correspondant à cette spécification comportementale est trop grande pour que nous présentions ce diagramme dans ce mémoire. L'IOLTS correspondant à la spécification comportementale a été calculé de manière manuelle.

Ce déploiement n'est sans doute pas le plus adapté pour tester les ascenseurs de façon satisfaisante. En effet il ne fait intervenir qu'un seul utilisateur alors que le système prévoit de multiples utilisateurs. De même le nombre d'étages est assez restreint. Comme certaines opérations (traduction des diagrammes détats transitions, certaines mesures de couverture) se sont faites sans outils, il était nécessaire de garder une configuration de petite taille.

En fait, ce déploiement ne sert qu'à illustrer l'utilisation des outils TObiAs et TGV, sa petite taille permet d'illustrer facilement les différents concepts. De plus le diagramme IOLTS croît de manière importante si on modifie le nombre d'ascenseurs ou le nombre d'étages pris en compte.

6.1.3 L'utilisation des outils TObiAs et TGV

Ecriture des schémas de test dans TObiAs

TObiAs utilise le diagramme de classes (figure 6.1) de l'application. A partir des informations relatives aux classes, aux méthodes et aux paramètres des méthodes, TObiAs permet à l'utilisateur de créer un environnement de campagne de test mettant en jeu les instances définies dans un déploiement initial. La figure 6.6 présente l'interface de TObiAs avec la définition d'une campagne de test nommée "*2 ascenseurs 2 etages*" qui comporte une instance *u* d'utilisateur, deux instances *a1* et *a2* d'ascenseur et une instance *p* de panneau telles qu'elles ont été définies dans la figure 6.5.

Dans le cadre de cette étude, nous avons défini 5 schémas de test. Tous les groupes qui vont être présentés dans ces schémas de test sont définis dans le tableau 6.1 page 85. Ces schémas de test sont fortement inspirés de la structure de la spécification comportementale. l'objectif de chacun des schémas de test est de générer des objectifs de test activant certaines transitions de la spécification qui correspondent aux exigences définies.

1. Les exigences 2, 3 et 4 portent sur la relation entre les mouvements des ascenseurs et le panneau d'affichage. Ces trois exigences peuvent être évaluées dans un même schéma de test.

   *** !p.ETAT_AS ;* !p.renouveler_affichage**

   Nous créons un groupe ETAT_AS comportant les méthodes *monter, descendre* et *être_à_étage* pour chacun des 2 ascenseurs. Ce groupe comporte toutes les actions de mouvement possibles des différents ascenseurs.

   Dans la spécification comportementale, les méthodes *monter, descendre* et *être_à_étage* sont toujours suivies d'un renouvellement d'affichage. Ce schéma de test permet d'activer ces paires de transitions de la spécification avec tous les paramètres de ces méthodes.

2. L'exigence 5 précise qu'en tout lieu (sauf dans un ascenseur) les utilisateurs ont le moyen de se rendre à l'étage désiré. Nous définissons le schéma :

   u !u.MOUVEMENT_UT^ 2..3

   Ce schéma de test correspond à l'ensemble des combinaisons de deux et trois appels successifs à des méthodes du groupe MOUVEMENT_UT. Ce choix de valeurs (2 et 3) nous permet, dans un même cas de test d'observer des mouvements de l'utilisateur sur l'ensemble des étages définis. Le groupe MOUVEMENT_UT se compose des méthodes *requête_étage* et *aller_à* pour tous les étages et tous les ascenseurs. Ce schéma permet d'avoir de

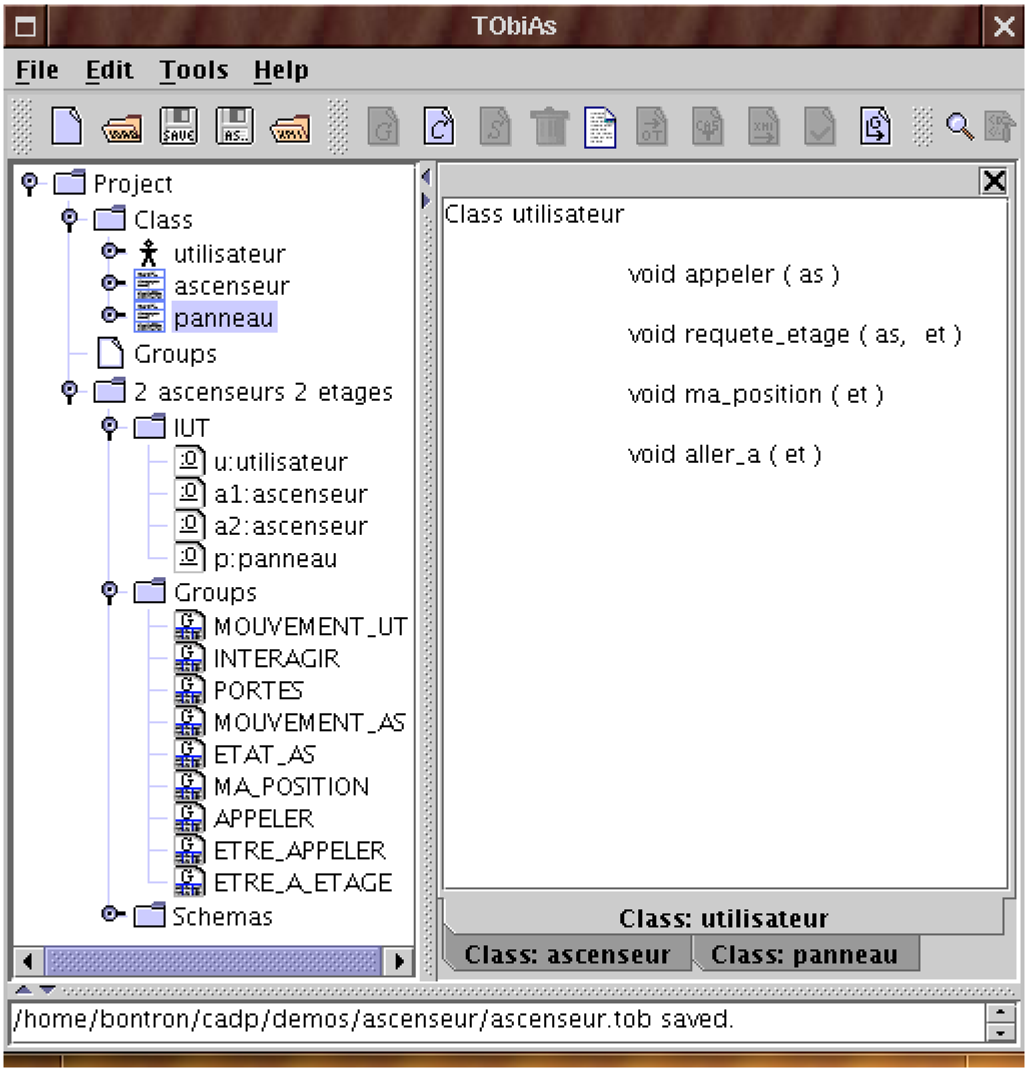


FIG. 6.6 – Interface de TObiAs

nombreux comportements de l'utilisateur (déplacement à pied et par ascenseur). Ces comportements comprennent le libre accès aux différents étages à tout moment. TObiAs génère 252 objectifs de test pour ce schéma de test.

Dans ce schéma nous précisons que l'instance qui utilise les méthodes du groupe MOUVEMENT_UT est l'instance utilisateur. L'ascenseur associé à l'utilisateur n'a pas accès en fait aux méthodes de l'utilisateur, celles-ci sont privées.

3. L'exigence 6 est une exigence de sûreté. Pour nous assurer que les portes sont bien fermées durant tout déplacement d'un ascenseur, nous créons un schéma de test mettant en jeu les méthodes d'ouverture et de fermeture des portes, ainsi que les mouvements des ascenseurs. Nous créons le groupe PORTES comprenant les méthodes *ouvrir_portes* et *fermer_portes* ainsi que le groupe MOUVEMENT_AS qui regroupe *moteur_en_montée* et *moteur_en_descente*. Nous modélisons le schéma de test suivant :

   *** !*.PORTES ;* !*.MOUVEMENT_AS ;* !*.PORTES**

   Ce schéma de test génère 64 objectifs de test.

4. Pour l'exigence 1 nous avons défini deux schémas de test. Le premier repose sur le fait que si un utilisateur souhaite se rendre à un étage et utilise un ascenseur. Une fois le déplacement effectué, l'utilisateur annonce sa position grâce à la méthode *ma_position*. On doit alors observer que l'ascenseur et l'utilisateur sont bien arrivés au même étage. Le schéma est le suivant :

   u !u.MOUVEMENT_UT ;u !*.MOUVEMENT_AS ;u !u.MA_POSITION

   Le groupe MOUVEMENT_AS est constitué des méthodes *moteur_en_montée* et *moteur_en_descente*. Ce schéma de test correspond à 48 objectifs de test.

5. L'autre schéma de test associé à l'exigence 1 utilise une négation :

   u !*.ETRE_APPELE ;−u !*.ETRE_APPELE ;u !p.ETRE_A_ETAGE

   La négation signifie qu'entre l'appel correspondant au groupe ETRE_APPELE et l'appel correspondant à ETRE_A_ETAGE l'utilisateur ne peut pas faire appel une nouvelle fois à une méthode du groupe ETRE_APPELE. Ce schéma de test permet d'observer si l'ascenseur se rend bien à l'étage où il a été appelé dans le cas où c'est la même instance d'ascenseur qui est appelé pour les différentes méthodes du cas de test. Ce schéma de test produit 128 objectifs de test.

Le tableau 6.1 rappelle les différents groupes de méthodes définis, avec les valeurs choisies pour chacun des paramètres des méthodes. Ces groupes peuvent être définis dans la campagne de test ou de manière globale dans TObiAs.

MOUVEMENT_UT	requête_étage(as∈{a1,a2} ; et∈{0,1}) aller_à(et∈{0,1})
INTERAGIR	appeler(as∈{a1,a2}) requête_étage(as∈{a1,a2} ; et∈{0,1})
PORTES	ouvrir_portes() fermer_portes()
MOUVEMENT_AS	moteur_en_montée() moteur_en_descente()
ETAT_AS	monter(as∈{a1,a2}) descendre(as∈{a1,a2}) être_à_étage(as∈{a1,a2} ; et∈{0,1})
MA_POSITION	ma_position(et∈{0,1})
APPELER	appeler(as∈{a1,a2})
ETRE_APPELE	être_appelé(et∈{0,1})
ETRE_A_ETAGE	être_à_étage(as∈{a1,a2} ; et∈{0,1})

TAB. 6.1 – Les groupes définis

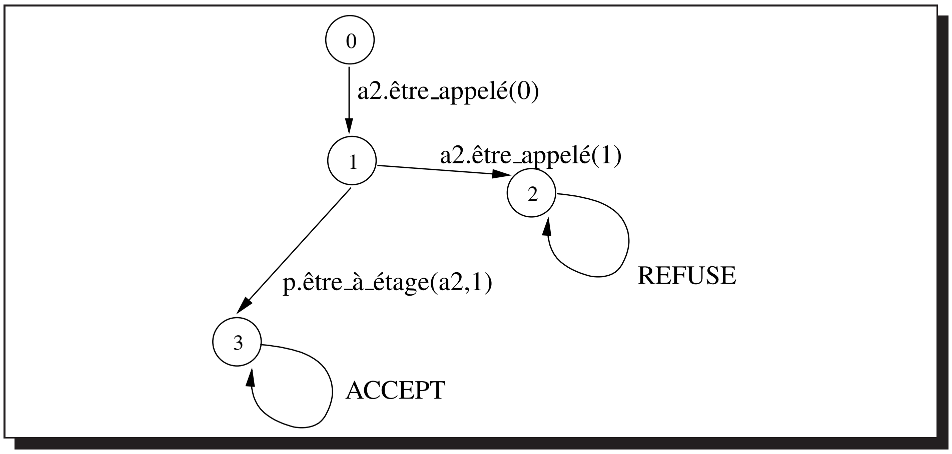
Une même méthode peut appartenir à plusieurs groupes, comme c'est le cas avec la méthode *être_à_étage* qui appartient aux groupes ETAT_AS et ETRE_A_ETAGE. Cela ne pose aucun problème pour la génération d'objectifs de test et permet d'être plus flexible dans la description des schémas de test. Cela permet aussi de mieux cibler ce que nous souhaitons représenter avec un schéma de test. Certains groupes n'ont qu'une seule méthode afin de permettre de définir des valeurs pour les paramètres de ces méthodes. Car les schémas de test ne font appel qu'à des groupes, pas aux méthodes de manière directe.

TObiAS génère pour ces 5 schémas de test un total de 620 objectifs de test en moins de 2 minutes[1].

Génération des tests abstraits avec TGV

Ces 620 objectifs de test sont transmis un par un à TGV afin qu'il associe à chacun un test abstrait conforme à la spécification comportementale. Sur la même machine que pour la génération des objectifs de test avec TObiAs, l'exécution de TGV pour ces 620 objectifs de test prend moins de 10 minutes.

A partir des 620 objectifs de test, TGV a pu créer 608 tests abstraits à raison d'un par objectif de test. 12 objectifs de test n'ont pas donné lieu à un cas de test car ils sont irréalisables dans le contexte de la spécification. Ces 12 objectifs de test sont tous issus du schéma de test numéro 5 qui comporte une négation. En effet, le fait de faire un rejet sur une action peut empêcher le déclenchement d'une autre dans le futur. L'un des objectifs de test en cause est le suivant :



Dans cet objectif de test, l'ascenseur $a2$ doit être appelé à l'étage 0, puis se retrouver plus tard à l'étage 1 sans avoir été à cet étage entre temps. Or le diagramme d'états-transitions d'un ascenseur (figure 6.3) montre bien que pour avoir l'appel à la méthode *être_à_étage* de l'instance panneau avec les valeurs *a2* et 1, il faut nécessairement qu'il y ait eu un appel à la méthode *être_appelé* de l'instance *a2* d'ascenseur avec la valeur 1. Il est donc tout à fait normal que cet objectif de test ne puisse pas produire de test abstrait. Il en est de même pour les 11 autres objectifs de test n'ayant pas permis à TGV de générer un cas de test.

[1]Sur un Pentium 3 cadencé à 700 MHz avec 512 Mo de mémoire vive.

6.1.4 Retours sur expérience

Cette étude de cas nous permet de faire le point sur les techniques que nous employons pour la génération de tests. Nous pouvons constater qu'une fois la phase d'écriture des schémas de test accomplie, la génération jusqu'aux tests abstraits se fait automatiquement avec les outils TObiAS et TGV, sauf dans le cas du schéma de test 5 qui est partiellement irréalisable. Le testeur peut donc consacrer tout son temps à la création d'une campagne de test avec les schémas de test, instances et groupes qu'il souhaite manipuler. Dans le cas de cette étude la conception de la campagne de test a pris un homme/jour.

TObiAs peut être amené à générer des objectifs de test non valides, cela est dû au fait que TObiAs ne confronte pas ses objectifs de test à la spécification pour s'assurer qu'il sont bien valides pour cette spécification. Dans ce cas là, TGV ne génère pas de tests abstraits, ceci permet de trier a posteriori les objectifs de tests valides et les objectifs de test non valides. Seules les constructions *négation* et *séquences strictes* des schémas de test peuvent amener à des objectifs de test non valides à cause des contraintes qu'apportent ces constructions. Ceci est vrai dans le cas où la spécification comportementale ne comporte pas d'états puits[2]. Quand les systèmes ont un caractère cyclique les constructions des schémas de test peuvent toujours se dériver en un ensemble d'objectifs de test valides vis-à-vis de la spécification.

6.1.5 Conclusion

Dans cette illustration nous avons mené à bien toute une phase de synthèse de tests avec les outils TObiAs et TGV. A partir de six exigences inspirées de la spécification nous avons défini 9 groupes pour 13 méthodes et 5 schémas de test. TObias nous a permis de générer 620 objectifs de test à partir de ces schémas de test. 608 d'entre eux ont permis de générer des cas de test valides avec TGV, les 12 autres objectifs de test comportaient des incohérences vis-à-vis de la spécification comportementale. Cela ne permet pas d'en déduire que le schéma de test 5 a été mal écrit, car il a tout de même généré 116 objectifs de test valides. Le rapport du nombre d'objectifs de test incohérents sur le nombre d'objectifs de test valides reste parfaitement acceptable, surtout que l'exécution de l'ensemble

[2]Un état puit est un état dont toutes les transitions sortantes de cet état sont dirigées vers lui même.

de tous les objectifs de test avec TGV a pris moins de dix minutes.

L'étude n'a pas été poussée jusqu'à la création de tests exécutables et à leur mise en œuvre sur une implémentation du système. Cette phase de test sort du domaine de cette thèse et nous ne possédions pas d'implémentation de ce système.

Cette illustration permet de mettre en avant les concepts que nous voulons mettre en évidence. L'ensemble des mécanismes d'abstraction des schémas de test peut être mis en œuvre, que ce soit sur les instances, les méthodes, leurs paramètres ou le nombre d'occurrences des méthodes. Cette illustration ne présente pas toutes les constructions possibles des schémas de test comme l'alternative, la co-région ou encore les séquences strictes. Ces différentes constructions sont néanmoins utilisable dans la perspective de traduites en IOLTS donnée en entrée de TGV les objectifs de test générés par TObiAs.

6.2 L'application bancaire de Gemplus

L'étude de cas proposée par Gemplus pour le projet COTE comporte une spécification UML et une implantation en Java. Cette dernière a été spécifiée en JML a posteriori. L'étude de cas est une application bancaire qui gère divers comptes en banque de diverses banques avec la possibilité d'effectuer des transactions entre ces comptes. L'environnement comporte trois acteurs : *BankOfficer*, *Customer* et *Timer*. Ce système est représentatif des services en ligne des banques proposés aux clients pour gérer leurs comptes. Le système sous test comporte huit classes ainsi que les trois acteurs représentant l'environnement. Sous sa forme Java ce système représente plus de 500 lignes de code (TAB. 6.2, page 96). La figure 6.7 représente le diagramme de classes de l'application. Une description UML plus détaillée de cette étude de cas est faite dans le rapport d'activité [Lan02b] du projet COTE.

Les acteurs du système

Les acteurs représentent l'environnement du système considéré. C'est par leur biais que le testeur peut influer sur le système. Nous en comptons trois différents qui correspondent aux actions du *client*, à celles de la *banque* et au *temps*. La représentation du temps comme un acteur est nécessaire pour UMLAUT/TGV.

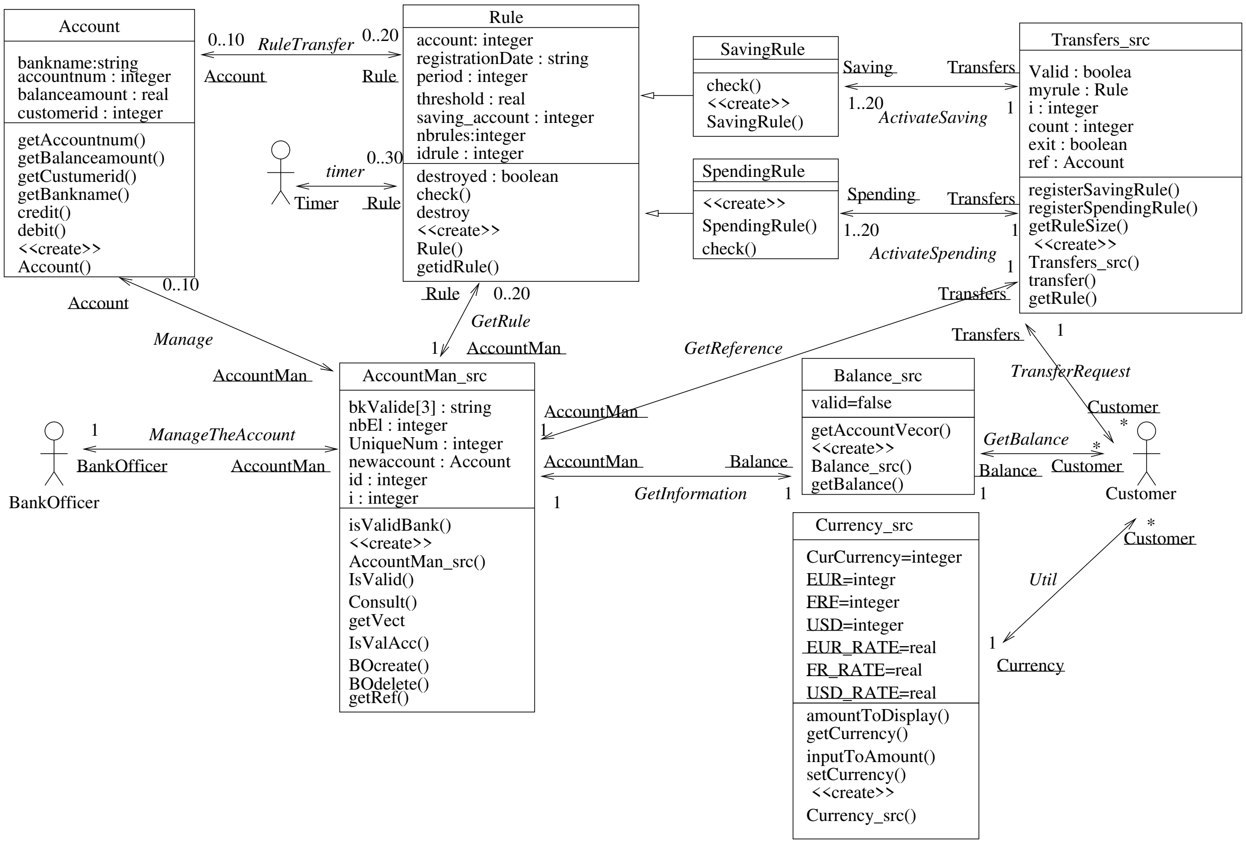


FIG. 6.7 – Diagramme de classe de l'application bancaire

– **Customer** consulte les soldes des comptes qu'il détient, crée des règles de transfert de fonds de manière ponctuelle ou permanente. Il peut aussi visualiser ses soldes dans différentes devises (Dollar US, Euro ou Franc français).
– **BankOfficer** crée et détruit les comptes et attribue les identifiants des comptes.
– **Timer** simule un mécanisme d'horloge. Toutes les secondes le système vérifie quelles sont les règles de transfert à activer et procède à leur activation si nécessaire.

Les classes du système sous test

- **AccountMan_src** : cette classe gère les comptes des utilisateurs. Elle peut les créer et les détruire suite aux requêtes de l'acteur *BankOfficer*. C'est cette classe qui gère l'unicité de l'identifiant de compte ainsi que la liste des banques autorisées. Par simplification cette classe gère un maximum de 10 comptes, d'où une association de 0..10 sur le diagramme de classes.
- **Transfers_src** : cette classe gère les règles de transfert de fonds d'un compte à l'autre ainsi que les demandes de transfert immédiat. Il s'assure que les règles sont valides avant de les accepter. Les transferts ne s'effectuent qu'entre les comptes d'un même client, 2 clients différents ne peuvent pas transférer d'argent entre leurs comptes.
- **Rule** : la classe Rule permet de définir des règles avec des conditions d'activation plus ou moins évoluées. Les classes *SpendingRule* et *SavingRule* héritent de la classe *Rule* et surchargent la méthode check.
- **Balances_src** : cette classe permet la consultation des soldes des comptes détenus par un utilisateur.
- **Currency_src** : cette classe fonctionne comme un convertisseur de monnaie, elle convertit des sommes d'une devise (Dollar ou Euro) vers le Franc ou inversement.
- **Account** : cette classe définit un compte bancaire. Les comptes sont gérés par un gestionnaire de compte, la classe **AccountMan_src**.
- **SpendingRule** : cette classe hérite de *Rule* et définit une règle de dépense. Elle permet sous certaines conditions de transférer un montant d'un compte vers un autre.
- **SavingRule** : cette classe hérite de *Rule* et définit le comportement d'une règle d'épargne d'un compte courant vers un compte d'épargne.

6.3 Expérimentations menées par Gemplus

Dans le cadre du projet COTE, les chercheurs de Gemplus devaient entre autre valider la chaîne d'outils mise en place. Nous leur avons fourni l'outil TObiAs afin qu'ils puissent l'évaluer sur leur étude de cas.

La démarche qui a été prise pour cette étude consiste à exprimer les exigences listées dans le cahier des charges (Annexe A.1) à l'aide de schémas de test. Les schémas de test sont définis dans l'outil TObiAs afin de générer automatiquement les cas de test correspondants[3]. Les tests sont exécutés sur une implantation Java/EJB, ce choix d'implantation a des répercussions sur le choix des exigences testées comme nous le verrons par la suite.

La synthèse des schémas de test s'est déroulée en deux étapes correspondant à des abstractions successives des exigences du cahier des charges. Leur mise en œuvre dans TObiAs a permis d'observer certaines lacunes de TObiAs.

Déploiement initial du système

Pour l'ensemble des tests qui sont exécutés dans le cadre de cette étude de cas par Gemplus, les objets suivants (FIG. 6.8) ont été instanciés afin de former le système sous test :

- un objet de type **AccountMan_src**, nommé *account_manager*, représente l'interface de gestion des comptes du client.
- trois objets de type **Account_src**, créés grâce à l'instance *account_manager*, représentent les comptes du client :
  1. *account1* est un compte dans la banque “BNP” avec un solde de 1000. Le numéro de compte est 12345678 et il appartient au client ayant comme identifiant 1.
  2. *account2* est un compte dans la banque “CA” avec un solde de 5000. Le numéro de compte est 87654321 et il appartient au client ayant comme identifiant 1.
  3. *account3* est un compte dans la banque “CE” avec un solde de 8000. Le numéro de compte est 87651234 et il appartient au client ayant comme identifiant 1.
- un objet de type **Transfers_src** gère les virements ente les comptes. Cet objet se nomme *transfer*.
- un objet de type **Balances_src**, nommé *balance*, gère la consultation des soldes des comptes du client.
- un objet de type **Currency_src**, nommé *currency* gère la conversion des sommes sur les comptes en fonction des devises.

Ce déploiement présente le cas de figure d'un seul client ayant 3 comptes dans trois banques différentes. On prend comme hypothèse qu'il n'existe pas d'effets de bord dans le traitement de plusieurs clients, comme ceux-ci ne peuvent pas

[3]Dans cette étude TObiAs n'est pas utilisé pour générer des objectifs comme dans l'étude précédente.

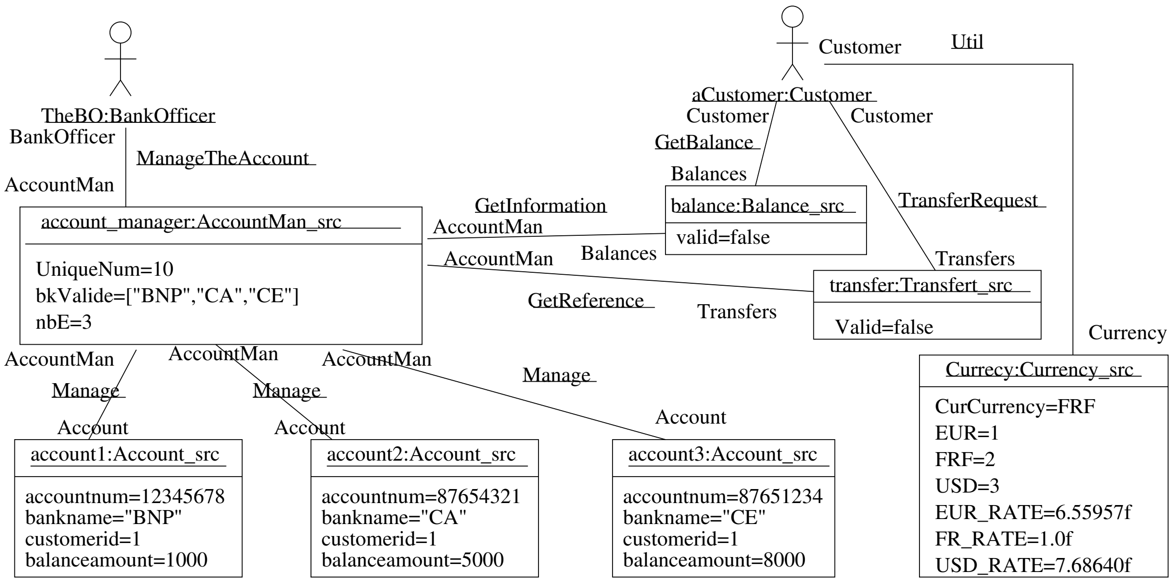


FIG. 6.8 – Diagramme d'objets du déploiement pris en compte

interagir entre eux (l'application ne permet pas d'effectuer de transfert entre 2 clients), on peut se contenter d'effectuer une étude de cas avec un seul client.

Des exigences aux schémas de test

Le cahier des charges correspondant à la spécification de ce service bancaire [Com02] comporte 40 exigences (Annexe A.1). Les chercheurs de Gemplus ont tenté dans la mesure du possible d'exprimer des schémas de test à partir de ces exigences.
Une première analyse des exigences montre qu'elles ne sont pas toutes testables dans l'environnement COTE de par leur nature ou qu'elles ne nécessitent pas d'être testées du fait de la plateforme d'implantation considérée. En effet l'application est portée sur un serveur de type Java/EJB qui fournit des services de base tels que la possibilité de protéger les connexions par mot de passe ou de spécifier les types de transaction entre client et serveur. De ce fait de nombreuses exigences sont prises en charge par la technologie sous-jacente. Ces exigences, au nombre de 9, sont regroupées en catégories et listées ci-dessous :

Exigence ne pouvant pas être testée :

– le fait que l'application soit testable.

On ne peut pas définir un test montrant la testabilité de l'application, mais le fait qu'une suite de test est produite et qu'elle ait certaines qualités démontre que l'application est testable. Au delà de cela, il reste à savoir dans quelle mesure l'application est plus ou moins testable, c'est-à-dire si l'application offre des interfaces suffisantes pour pouvoir solliciter l'application puis interpréter les résultats des tests.

Exigences dont les fonctionnalités sont prises en charge par le modèle à composants Java/EJB et qui sont considérées comme exemptes de défaut de développement :

– l'utilisateur doit être authentifié par code secret,
– l'authentification doit s'opérer avant la connexion,
– le mot de passe n'est pas modifiable par l'utilisateur,
– le nombre de saisies de mot de passe est illimité,
– toutes les transactions terminal/serveur s'opèrent par “secure-messaging”.

Exigence ne pouvant pas être testée par l'utilisateur car la possibilité ne lui en est pas offerte :

– le solde d'un compte n'est pas plafonné. Dans ce cas seule la vérification du codage du solde peut nous renseigner. En effet l'utilisateur ne peut qu'effectuer des virements entre ses comptes. A un moment donné, la somme maximum contenue dans un compte ne peut donc pas excéder l'ensemble des sommes présentes sur les comptes à l'état initial. On ne peut donc pas faire croître cette somme vers l'infini pour vérifier par le test fonctionnel si oui ou non le solde est plafonné.

Les exigences testables ont été reprises [Lan02a] pour servir de base à une campagne de test. Les testeurs de Gemplus ont produit à partir de ces exigences des cas d'utilisation correspondant à une *exécution normale* et à une *exécution en erreur* menant à un code d'erreur défini dans la spécification[4]. Les divers tests abstraits ainsi définis sont au nombre de 36. C'est en partant de ces tests abstraits que les chercheurs de Gemplus ont défini des schémas de test.

Pour couvrir les 36 cas d'exécution, les chercheurs de Gemplus ont créé 14 groupes pour 14 schémas de test (Annexe A.2). Ces 14 schémas de test permettent de générer 1900 cas de test. Certains groupes sont inclus dans d'autres ou peuvent

[4] Cela reste donc du test de conformité.

être fusionnés. En discutant avec les chercheurs de Gemplus, nous avons ainsi pu passer de 14 à 5 schémas de test après avoir réduit le nombre de groupes à 6 (Annexe A.3). Cette réduction du nombre de schémas de test ne réduit pas la variété des cas de test exprimés, mais offre au contraire de nouveaux cas de test auxquels les testeurs de Gemplus n'avaient pas pensé.

Apports de cette étude pour TObiAs

Cette étude est la première étude de cas assez conséquente sur laquelle a été utilisé TObiAs. C'est aussi la première fois que TObiAs a été utilisé hors du laboratoire par des personnes n'ayant pas participé à sa conception. Ceci nous a permis de tirer des enseignements sur la façon dont pouvait se présenter une vraie spécification, de savoir de quelle manière les utilisateurs pouvaient aborder TObiAs et quelles étaient les lacunes de TOBiAs à leurs yeux.

D'un point de vue méthodologique, les chercheurs de Gemplus ont décidé de définir un plan de test à partir des exigences du cahier des charges. De définir des exécutions types pour chaque exigence testable et de modéliser une première série de schémas de test. Enfin ils sont revenus sur ces schémas de test pour réduire leur nombre en optimisant le nombre de groupes et de schémas de test définis. Ce regroupement permet en outre d'obtenir des tests encore plus variés.

TObiAs offre aux ingénieurs de test une structuration des cas de test. Ce sont en effet plus de 1900 cas de test qui ont été ainsi produits. Chaque test est issu d'un schéma de test. Chaque schéma de test correspond à une propriété du cahier des charges ou à une famille de propriétés proches.

Du point de vue de TObiAs, nous avons pu prendre connaissance des lacunes ressenties par les chercheurs de Gemplus sur la version de TObiAs disponible à cette époque (mai 2002). Ces lacunes étaient au nombre de trois :

- *TObiAs ne gérait pas le fait d'avoir plusieurs testeurs.* En effet les acteurs du système (*BankOfficer*, *Timer* et *Customer*) font partie de l'environnement du point de vue de l'application. Les acteurs sont donc tous des testeurs avec autant d'interfaces de test différentes. Or TObiAs (dans la version de l'époque) ne gérait qu'un seul testeur nommé MTC[5], les acteurs définis dans l'application n'étaient pas pris en compte par TObiAs.

[5]Main Test Component.

Il est donc nécessaire de renommer le MTC dans les tests produits à la main puisque tous les testeurs sont regroupés sous cette même appellation.

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

Depuis cette expérimentation TObiAs gère les acteurs. Les notions d'hyper-groupe et de gestion de retours de méthode n'ont pas encore été intégrées. En effet la mise en place de ces deux notions nécessiterait de profonds changements aussi bien dans TObiAs, pour la création d'hyper-groupe, que dans la notion même des schémas de test et tests abstraits pour la prise en compte des retours de méthodes.

6.4 Expérimentations menées par le LSR

Gemplus nous a fourni les sources Java et la spécification JML de cette étude de cas. La portion de spécification JML compte 615 lignes pour 500 lignes de Java (tableau 6.2).

Le but de cette expérimentation était de trouver les erreurs que n'avait pas pu détecter un processus de preuve partielle de l'application avec l'outil Jack [BR02]. Plus exactement, nous essayons de trouver des incohérences entre le code, les assertions JML et le cahier des charges. On peut définir trois cas de figure :

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

TAB. 6.2 – Informations sur l'application bancaire en Java

– Le code et les assertions JML sont incohérents. Cela est détecté lors des tests par le viol d'une assertion JML, ou quand une exception Java inattendue est levée. Des trois types d'erreurs, seules celles là peuvent être détectées par un oracle automatique.
– Les assertions JML sont incohérentes avec le cahier des charges mais pas avec le code. Une telle erreur ne peut être détectée que par une analyse humaine des assertions JML ou de l'exécution du code.
– Les assertions JML, le code et le cahier des charges sont tous cohérents, mais l'observation du comportement de l'application laisse apparaître que des exigences nécessaires, ou de bon sens, n'ont pas été précisées.

L'application a été testée de deux manières différentes dans un temps limité de 3 jours pour confronter les apports respectifs des deux méthodes choisies. La première façon de tester l'application a été, dans un premier temps, une revue de code, puis l'utilisation d'un outil de test aléatoire. La deuxième façon de tester l'application a été l'utilisation de TObiAs en définissant des schémas de test avec la méthode "categories and partitions".
L'ensemble de ce travail a été effectué par Lydie du-Bousquet, Olivier Maury et Catherine Oriat [dBLM+04].

Après une présentation des approches prises, nous présentons les résultats obtenus puis nous porterons un regard critique sur ces résultats.

6.4.1 Inspection de code et test aléatoire

La première approche expérimentée par Catherine Oriat s'est déroulée en deux temps : une inspection de code puis un processus de test aléatoire [DN81]. Le processus de test aléatoire a été effectué avec un outil développé dans notre laboratoire permettant de générer des tests aléatoires à partir d'une application Java et de son code.

Inspection de code : cette technique consiste à lire “consciencieusement” le code source pour trouver des erreurs. Dans le cas présent on effectue la lecture du code source Java ainsi que de la spécification JML. En plus de permettre de mettre à jour des erreurs confirmées par des tests écrits à la main, cette méthode permet d'identifier des parties de code suspectes qui feront l'objet de tests plus poussés. Cette pratique a permis de déceler 4 erreurs (tableau 6.3 page 100) et a duré une journée.

Test aléatoire : l'outil Jartege (Java Random Test Generator) développé au LSR par Catherine Oriat [Ori02] permet de générer dynamiquement des tests unitaires pour des classes Java comportant une spécification JML. Jartege interagit avec le programme sous test et sélectionne aléatoirement et de manière dynamique des méthodes dont la pré-condition est évaluée à vrai. Ces sélections successives permettent de former une suite d'appels de méthodes correspondant à un test. Cet outil peut être paramétré suivant plusieurs aspects. On peut associer des poids aux classes et aux méthodes pour favoriser l'apparition plus fréquente de certaines d'entre elles dans les tests. On peut définir le nombre d'occurrences d'une instance dans les séquences de test générées. Cette deuxième phase de l'expérimentation a permis de déceler 5 nouvelles erreurs ou situations suspectes (tableau 6.3).

6.4.2 Utilisation de TObiAs

Lydie du-Bousquet et Olivier Maury sont partis de la spécification informelle de la spécification pour identifier 7 propriétés du système. Pour chaque propriété, des tests d'exécution ont été définis. Ces tests d'exécution correspondent à des éléments de la spécification en rapport avec la propriété et qu'il pourrait être intéressant de tester à l'aide des schémas de test et de TObiAs.

1. **Propriété :** *les opérations d'ouverture et de clôture des comptes satisfont la spécification.*

   5 situations ont été identifiés pour cette propriété :

   – l'opération d'ouverture de compte (considérée de façon isolée) satisfait sa spécification ;

- la création consécutive de plusieurs comptes s'effectue sans erreurs ni effets de bord ;
- on tente de créer 501 clients, la spécification limite normalement le nombre de clients à 500 ;
- la suppression d'un compte s'effectue sans erreurs ni effets de bord ;
- une suite de créations et de suppressions de comptes s'effectue sans erreurs ni effets de bord.

2. **Propriété :** *les opérations de transfert satisfont la spécification.*

   3 situations ont été identifiés pour cette propriété :
   - une opération de transfert d'un compte valide vers un autre compte valide s'effectue sans erreurs ;
   - une suite de transfert conserve les sommes d'argent mises en jeu ;
   - le transfert d'une somme proche de zéro s'effectue correctement.

3. **Propriété :** *l'établissement des règles d'épargne et de dépense se font sous de bonnes conditions.*

   On définit trois types de test pour tenter de créer des règles dans des conditions non valides et s'assurer que cette création ne s'effectue pas :
   - une règle n'est pas enregistrée si le seuil indiqué est incorrect ;
   - une règle n'est pas enregistrée si le numéro de compte épargne indiqué est incorrect ;
   - une règle n'est pas enregistrée si la période indiquée est incorrecte.

4. **Propriété :** *les règles d'épargne et de dépenses sont activées de façon périodique.*

   On teste successivement les règles d'épargne et de dépense pour s'assurer de la conformité de leur activation :
   - les règles de dépense sont activées régulièrement ;
   - les règles d'épargne sont activées régulièrement.

5. **Propriété :** *deux règles ne peuvent pas être exécutées en même temps.*

   On s'assure qu'une section critique est mise en place afin qu'une règle n'interfère pas avec une autre règle déjà en cours. On reprend les tests de la propriété 4.

6. **Propriété :** *l'opération de destruction d'un compte et l'utilisation des règles de transfert ne sont pas incompatibles.*

   2 situations ont été identifiés pour cette propriété :
   - la destruction de compte et les règles de transfert ne sont pas incompatibles ;

– la destruction de compte et les règles de dépense ne sont pas incompatibles.

7. **Propriété :** *les convertisseurs fonctionnent correctement.*
Ici on définit 2 situations pour cette propriété :
– les conversions sont correctes par rapport à la monnaie définie dans le convertisseur à la création ;
– les conversions sont correctes par rapport à la monnaie définie dans le convertisseur après une modification de la monnaie de référence.

Gr_BOcreate1	=	{BOcreate(c,bn,b} : c∈{1} bn∈{"bnp"} b∈{100}
Gr_BOcreate2	=	{BOcreate(c,bn,b} : c ∈ {1} bn ∈ {"ce"} b∈{100,POSITIVE_INFINITY}
Gr_transfert1	=	{transfert(s,d,a} : s∈{11} d∈{12} a∈{99.9f}
Gr_transfert2	=	{transfert(s,d,a} : s∈{11} d∈{12} a∈{99.9f}
Gr_transfert3	=	{transfert(s,d,a} : s∈{12} d∈{11} a∈{100, 99.9999999999998f, 99.9999998f, 0, 0.0000000000001f, 0.0000001f}
Print1	=	{PrettyPrintBalance(c)} : c ∈ {1}
Print2	=	{PrettyPrintBalance(c)} : c ∈ {2}

FIG. 6.9 – Les groupes définis pour les schémas de test de la propriété 2.

17 schémas de test on été définis pour ces 17 éléments que nous souhaitons tester au vue des 7 propriétés qui ont été définies. A titre d'exemple nous présentons les deux schémas de test et leurs groupes (figure 6.9) correspondant aux situations 2 et 3 de la propriété 2 :

– le premier schéma de test correspond à la situation 2, il cherche à déterminer si une suite d'opérations de transfert peut accroître les erreurs de calcul que le premier schéma de test de la propriété 2 a décelé. Le schéma retenu est "*Gr_BOcreate1 ; Gr_BOcreate2 ; (Gr_transfert1 ; Gr_transfert2)∧200..200 ; Print1 ; Print2*"
Les résultats observés montrent que l'on peut "créer" de l'argent en effectuant moins de 400 transferts ;
– le second schéma de test correspondant à la troisième situation cherche à illustrer le comportement de l'application lors d'un retrait d'une somme presque égale à zéro. Le schéma retenu est "*Gr_BOcreate1 ; Gr_BOcreate2 ; Gr_transfert3 ; Print1 ; Print2*". En effet le groupe *Gr_transfert3* comporte

	équipe 1		équipe 2		
Err.	Inspection de code	Test aléatoire	TObiAs	type d'erreur	méthode de détection
1			X	limite	oracle humain
2			X	limite	oracle humain
3	X	/////////////	X	calcul flottant	Insp. code + or. JML
4		X	X	calcul flottant	oracle JML
5		X	X	calcul flottant	oracle JML
6		X	X	calcul flottant	oracle JML
7			X	post-condition	oracle JML
8			X	post-condition	oracle JML
9		X	X	design	oracle JML
10	X	/////////////		design	insp.code
11			X	limite	oracle humain
12			X	limite	oracle humain
13	X	/////////////	X	design	Insp. code + exec Java
14	X	/////////////		post-condition	insp.code
15		X	X	nombreux	Java exception
16			X	sous-spécifié	oracle humain
17			X	sous-spécifié	oracle humain
18			X	calcul flottant	oracle humain

TAB. 6.3 – Les "erreurs" détectés

une règle de transfert d'une somme de 0.0000000000001f.

Les 17 schémas de test correspondent à 1241 tests abstraits, soit 40000 lignes de code Java pour JUnit. Ce travail a pris 6 hommes/jour et a permis de découvrir 16 erreurs ou situations suspectes (tableau 6.3).

6.4.3 Les résultats obtenus

Par situation suspecte nous considérons que la spécification formelle et le code sont cohérents mais celui-ci ne correspond pas aux exigences informelles ou au "bon sens" car l'implantation est sous-spécifiée.

Le tableau 6.3 présente les erreurs détectées et les méthodes qui l'ont permis.

Les différentes erreurs peuvent être classées en 6 catégories différentes :

- approximation sur les calculs des flottants. On retrouve ce type d'erreur dans 5 cas (Err. 3, 4, 5, 6 et 18). Le type flottant sert à représenter la balance d'un compte. L'erreur survient lorsque la post-condition et le code calculent la même valeur de façon différente. $(x + y) - z$ est différent de $x + (y - z)$ si x, y, z sont des nombres flottants. Les opérations $+$ et $-$ ne sont pas commutatives à cause des limites de précision dans le cadre du calcul de flottants ;
- erreurs aux limites. Dans 4 cas (Err. 1, 2, 11 et 12) des tests avec des valeurs limites ont fait apparaître une erreur. Par exemple il est possible de créer une règle d'épargne avec une période égale à 0. La spécification informelle indique explicitement que les règles d'épargne ne doivent pas être enregistrées si la période est nulle ;
- mauvaise post-condition JML. Les erreurs 7, 8 et 14 sont dûes à des erreurs syntaxiques dans la post-condition où l'argument "$\backslash old$" a été oublié ;
- erreur de design. Nous avons mis en évidence 3 erreurs (Err. 9, 10 et 13) associées au fait qu'une variable est déclarée "*public*" alors qu'elle aurait dû être définie comme "*private*", ou que les accès aux comptes ne se font pas dans une section critique alors qu'ils sont gérés comme des threads ;
- sous-spécification. Les erreurs 16 et 17 proviennent d'un manque de spécification de l'application, il est en effet possible d'effacer un compte sur lequel portent des règles de transfert dont ce compte est le destinataire. Intuitivement on peut imaginer que de telles règles de transferts doivent être supprimées si le compte est effacé, or ce n'est pas le cas ;
- plusieurs classifications pour une même erreur. Dans le cas de l'erreur 15, nous avons une erreur qui porte sur un paramètre de type *string* représentant un *float*. L'erreur peut alors être considérée comme une erreur de conception avec un mauvais typage, ou comme une erreur de pré-condition inadéquate. L'erreur peut aussi être considérée comme un manque au niveau de la spécification qui n'indique pas quel doit être le format du paramètre.

6.4.4 Bilan de cette étude

L'étude effectuée avec TObiAs nous a permis de trouver plus d'erreurs qu'avec l'approche qui a consisté en une étude de code et l'utilisation de Jartege. On peut néanmoins noter que ces deux méthodes ont permis de révéler une erreur que TObiAs n'a pas pu déceler (l'erreur n° 14 n'était pas présente dans la spécification utilisée avec TObiAs). Cette erreur ne pouvait toutefois pas être décelée par du test de conformité : l'erreur n° 10 porte sur l'attribut *UniqueNum* de la classe *AccountMan_src* qui est défini comme *public*. Cet attribut peut donc être modifié de façon externe (par exemple, par une méthode de test, ce qui peut entraîner une erreur sur un invariant). Du test de robutesse aurait permis de détecter cette erreur.

Le bilan pour TOBiAs est positif. D'une part il a permis de détecter un grand nombre d'erreurs. Et d'autre part il a offert un élément de structuration pour les 1241 cas de test générés en les plaçant sous 17 schémas de test. TObiAs perd néanmoins l'information correspondant au lien entre ces 17 schémas de test et les 7 propriétés du système qui avaient été définies. Cette information pourrait s'exprimer sous la forme de commentaires associés aux schémas de test.

6.5 Conclusion

L'illustration en première partie du chapitre présente de quelle façon l'outil TObiAs peut facilement s'utiliser avec l'outil TGV de l'IRISA dans le cadre de la génération d'objectifs de test.

Les différentes études menées sur l'étude de cas de Gemplus avec TObiAs nous ont permis de voir que cette approche de test combinatoire en exprimant les tests sous la forme de schémas de test pouvait s'avérer être une bonne approche car le nombre d'erreurs qu'elle trouve est significatif et le temps de conception et de codage des tests avec l'outil TObiAs est réduit.

Nous avons pu noter certaines lacunes de TObiAs qui ont pu gêner la conception de certains tests. Une partie de ces lacunes (gestion des acteurs et système de contraintes) ont été comblées depuis ces expérimentations.

Les deux études permettent aussi de mettre en avant l'intéret méthodologique de l'utilisation de TObiAs avec la structuration qui est mise en place autour des cas de test générés grâce à la notion de schéma de test.

Dans l'illustration de l'utilisation de TObiAs avec TGV dans la section 6.1, nous n'avons pas de notion de couverture formelle pour les 608 tests abstraits créés pour évaluer la pertinence des tests. Nous pouvons juste considérer que d'un point de vue informel nous avons une bonne couverture des exigences liées au système puisque les schémas de test ont été définis pour satisfaire ces exigences.

L'ajout d'une mesure formelle de couverture des tests est un élément majeur qui manque à cette démarche pour juger de la pertinence de notre campagne de test, et donc de la suite de test qui en découle. Dans la troisième partie de ce mémoire nous verrons comment mettre en place une mesure de couverture associée aux schémas de test et de quelle façon cette mesure de couverture peut être mise en relation avec une mesure de couverture associée aux cas de test.

Synthèse sur TObiAs

TObiAs était dans un premier temps un outil qui permettait la génération d'objectifs de test à partir d'une spécification sous la forme d'un diagramme de classes et de schémas de test qui sont des objectifs de test sous une forme plus abstraite (au niveau des paramètres, méthodes et instances). Très vite nous avons étendu le champ d'utilisation de TObiAs à la génération de cas de test pour des langages tels que VDM ou JAVA afin d'utiliser TObiAs de la façon la plus large possible. Le chapitre 6 illustre différentes utilisations de TObiAs. Avec notamment celle faite par les chercheurs de Gemplus d'un point de vue méthodologique.

Cet outil ne devrait pas en rester là, dans le chapitre 5 nous avons montré les différentes pistes que nous avions déjà explorées, telles que :

– un langage de contraintes évolué pour accroître le pouvoir d'expressivité des schémas de test ;
– la prise en compte d'une spécification plus complète pour s'assurer d'une plus grande conformité des schémas de test et des objectifs de test générés ;
– les travaux menés pour réduire l'explosion combinatoire en gérant sous forme d'arbre les objectifs de test/cas de test générés par TObiAs. Ou encore en prenant en compte la couverture des paires de paramètres.

Jusqu'à présent TObiAs n'aborde pas le thème de la couverture de test. Dans le projet COTE seul l'outil Casting offre une mesure de couverture [AJ03]. La mesure offerte par Casting n'est accessible que dans le cadre de l'utilisation de Casting hors de la chaîne principale de traitement des tests du projet COTE.

Nous exploitons l'introduction des schémas de test pour définir un critère de couverture à ce niveau d'abstraction. Nous abordons dans la troisième partie de ce mémoire les différentes notions de critères de couvertures et la façon dont nous appliquons un critère de couverture aux schémas de test.

Troisième partie

La couverture a priori des schémas de test

Chapitre 7

Couverture de test : état de l'art

7.1 Introduction

Ce chapitre a pour but de présenter les différentes formes de couvertures de test qui peuvent être trouvées dans la littérature. La notion de couverture de test se rapporte à la notion de *critère d'arrêt* des tests. Cette notion de critère d'arrêt est une notion importante du processus de développement d'une application et de la phase se rapportant à la validation de cette application. Le nombre de tests pouvant être définis est bien souvent infini, or il faut décider à un moment d'arrêter cette campagne de test. On peut distinguer plusieurs types de critères d'arrêt correspondant à des priorités différentes. Ce critère peut être de nature financière quand les crédits alloués à la phase de test sont épuisés, ou de nature temporelle si les développeurs sont tenus par des délais précis. Nous nous intéressons à un troisième critère reposant sur des mesures associées aux tests produits. Ce critère d'arrêt porte le nom de *couverture de test*. Nous définissons la couverture de test dans le cadre d'une spécification :

Définition 5 (Couverture de test) *La couverture de test d'une suite de test pour une spécification donnée correspond au pourcentage de la spécification exercée par la suite de test suivant un critère donné.*

La couverture obtenue pour un jeu de test donné correspond donc au rapport du nombre d'éléments, satisfaisant le critère de couverture, de la spécification couverts sur le nombre d'éléments, satisfaisant le critère de couverture, constituant la spécification. Obtenir une couverture de 100% est parfois irréalisable pour certaines spécifications où il existe des états non atteignables.

Pour une spécification donnée, la couverture de test d'une même suite de test peut varier suivant le critère de couverture choisi. Au cours de ce chapitre nous présenterons successivement les différents types de couvertures présents dans la littérature pour positionner notre approche. Nous ne ferons référence qu'à des couvertures applicables à du test fonctionnel puisque c'est ce type de test que nous pratiquons. Nous présenterons en premier lieu le cas des spécifications informelles (section 7.2) puis nous présenterons divers critères de couverture de spécifications formelles dans la section 7.3. Enfin dans la section 7.4 nous préciserons quel critère de couverture nous avons décidé de prendre pour la suite de ce travail.

7.2 Couverture de spécifications informelles

Dans le cas des spécifications informelles il existe des techniques de couverture de test pour décider de l'arrêt d'une campagne de test. Ces techniques ne peuvent pas être automatisées à cause de l'imprécision des spécifications et du manque d'outils de traitement automatique. Cette même imprécision de ces langages fait que la portée d'un test par rapport à cette spécification informelle est difficile à identifier précisément. Ce choix repose sur la compréhension que le testeur a de la spécification et de l'interprétation qu'il en fait.

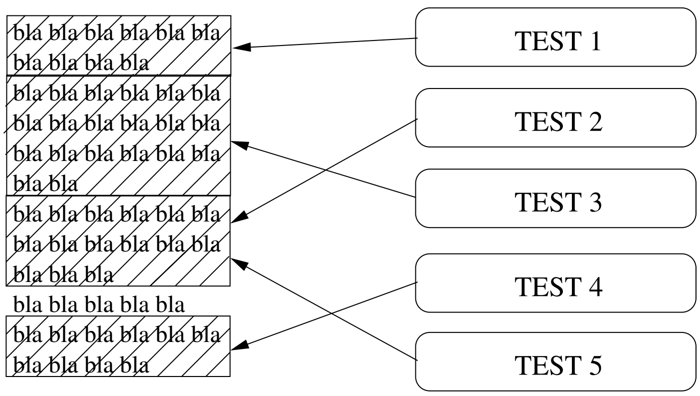


FIG. 7.1 – Couverture de spécification informelle

Il est néanmoins possible d'associer les tests à des exigences de la spécification. On peut imaginer que pour chaque test nous hachurons le texte correspondant dans la spécification (FIG. 7.1). La couverture de test peut alors être vue comme la partie hachurée de la spécification, ou comme le pourcentage de cette partie par rapport au texte global. Cette approche est celle utilisée dans la norme [ISO91]. Un test peut par ailleurs se rapporter à plusieurs exigences. Le choix de la granularité des exigences est lui aussi difficile à définir : mot, phrase, paragraphe, chapitre ...

Dans l'expérimentation menée par Gemplus (chapitre 6.3), c'est cette technique qui est utilisée pour définir les tests à exécuter en ne prenant en compte que la partie "exigences" de la spécification informelle. Dans ce cas là, le choix de granularité ne se pose pas puisque chaque exigence reflétait un point précis à tester, la spécification avait été pensée dans une optique de testabilité.

Cette technique permet de s'assurer qu'on teste tel et tel point de la spécification, mais en contrepartie les tests se contentent de vérifier les différents points du cahier des charges sans réellement être tournés vers la recherche d'erreurs.

7.3 Couverture de spécifications formelles

Avant de présenter différents critères de couverture, nous précisons que nous nous intéresserons uniquement aux spécifications formelles de type machines d'états finies (sans aborder d'autres styles de spécifications comme ceux de JML ou VDM).

Dans le cadre des spécifications formelles, nous reprenons les principes de la couverture de test de logiciel [Bei90] en l'appliquant au test boîte noire pour des systèmes de transitions finis. Pour les systèmes de transitions finis il existe différentes sortes de critères de couverture correspondant à des critères structurels [Cho78] : la couverture des états, la couverture des branches ou encore la couverture des paires de branches. Nous présentons chacun de ces critères de couverture. ainsi que d'autres critères de couverture moins classiques.

7.3.1 Couverture des états

Cette forme de couverture consiste à couvrir tous les états du système de transition au moins une fois. Dans le cas du diagramme de la figure 7.2. Pour obtenir une couverture totale des états, on peut se contenter de l'exécution suivante :

- **b , c , f**

Cette séquence d'appels nous assure que les états 1, 2, 3 et 4 sont couverts.

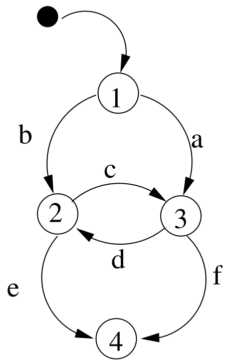


FIG. 7.2 – Exemple de système de transitions fini

7.3.2 Couverture des branches

La couverture des branches consiste à parcourir toutes les transitions du système considéré à partir de l'état initial. Dans le cas de la figure 7.2 nous prenons les tests suivants :

- **a , d , e**
- **b , c , f**

Ces deux tests nous assurent une couverture totale des branches.

7.3.3 Couverture des paires de branches

Cette notion de couverture est proche de la notion de couverture des branches. Dans ce cas, on ne regarde pas chaque branche comme une seule entité, mais les branches deux à deux lorsqu'elles sont successives. Dans notre exemple les paires de branches qui peuvent être considérées sont : **ad ; af ; bc ; be ; cd ; cf ; dc ; de**. Les tests suivants nous assurent une couverture totale de ce système dans le cadre d'une couverture des paires de branches :

- **a , f**
- **a , d , c , f**
- **b , c , d , e**
- **b , e**

Les deux notions de couverture des branches et de couverture des paires de branches peuvent se généraliser à la couverture de *n-branches* où le critère de couverture consiste à couvrir toutes les séquences possibles de n branches successives. La couverture des branches correspond à une couverture 1-branche et la couverture de paires de branches correspond à une couverture 2-branches.

7.3.4 Couverture des chemins

La couverture des chemins consiste à couvrir la spécification avec tous les chemins d'exécution possibles. Un chemin d'exécution est une suite de transitions partant d'un état initial et finissant sur un état final. Ici nous considérons l'état 4 comme étant un état final. C'est une notion de couverture bien plus forte que toutes celles vues précédemment et chaque test proposé dans notre exemple pour les autres formes de couverture va être présent pour satisfaire pleinement cette notion de couverture.

Cette notion de couverture peut nécessiter certaines hypothèses. En effet dans la figure 7.2 il existe une boucle entre les états 2 et 3. Sans hypothèses de notre part sur le nombre, jugé suffisant, de passages dans cette boucle nous aurions un ensemble infini de tests pour couvrir totalement la spécification suivant le critère de couverture des chemins. Une autre pratique consiste à borner la longueur des chemins acceptables comme séquence de test. Dans notre exemple nous considérons qu'un seul passage dans cette boucle est suffisant. L'ensemble de tests satisfaisant une couverture totale de la spécification suivant le critère de couverture des chemins est :

- **b , e**
- **b , c , f**
- **b , c , d , e**
- **a , f**
- **a, d , e**
- **a , d , c , f**
- **b , c , d, c , f**
- **a , d , c, d , e**

7.3.5 Couverture basée sur les distances et H-couverture

Il existe d'autres mesures de couverture de spécifications formelles moins "conventionnelles" que celles qui viennent d'être présentées. Nous présentons deux mesures de couverture proposées dans le cadre du test de protocoles.

Couverture basée sur les distances

La notion de distance entre les tests [VAC92, ACV93, MV95] repose sur deux points : la constatation de la similitude, du point de vue du test, qui peut exister entre des séquences de message et l'hypothèse que si deux tests sont assez semblables, l'exécution d'un seul est suffisante pour une campagne de test. Les similitudes entre deux tests sont exprimées par une distance. Plus cette distance est petite, plus ces tests sont considérés comme étant “proches”.

Soit S l'ensemble des comportements décrits par la spécification, la distance dont on munit S est choisie de manière à tenir compte du comportement récursif des protocoles. Pour cela on écrit les séquences d'exécution comme des couples de la forme (m_i,n_i) où m_i est le nom d'un message et n_i son nombre d'occurrences consécutives. Une séquence de la forme aaabbbabcc s'écrit de la façon suivante : (a,3) (b,3) (a,1) (b,1) (c,2).

Pour calculer la distance entre 2 séquences on définit au préalable 2 suites :

- $\{P_k\}_{k\geq 1}$ une suite réelle positive telle que $\sum_{k=1}^{+\infty} P_k = P < +\infty$
- $\{r_j\}_{j\geq 0}$ une suite croissante dans $[0,1]$ avec $\lim_{j\to+\infty} r_j = 1$

Soient deux séquences s et t telles que $s = \{(a_i,\alpha_i)\}_{i=1}^{K}$ et $t = \{(b_i,\beta_i)\}_{i=1}^{L}$ où $\alpha_i et \beta_i$ son le nombre d'occurence consécutive des messages a_i et b_i, et avec $K, L \in \mathbb{N} \cup \{\infty\}$. On définit la distance $dt(s,t)$ entre s et t par :

$$dt(s,t) = \sum_{k=1}^{max\{K,L\}} P_k \times r_{\omega_k(s,t)}$$

avec : $\omega_k(s,t) = \|\alpha_k - \beta_k\| \; si \; a_k = b_k, \; \infty \; sinon$

La suite $\{P_k\}_{k\geq 1}$ permet de tenir compte du rang des messages dans la séquence de test. Par exemple, la suite $P_k = 2^{-k}$ fait décroître exponentiellement le poids d'un message en fonction de son ordre d'arrivée. La suite $\{r_j\}_{j\geq 0}$ permet de pondérer en fonction du nombre d'occurrences de chaque message.

On peut définir la distance d'une séquence s à un ensemble de séquences T comme : $dt(s,T) = inf\{dt(s,t) \| t \in T\}$ où $inf\{E\}$ renvoie la plus petite valeur de l'ensemble E.

On définit une distance maximale entre l'ensemble S de toutes les exécutions possibles et l'ensemble T des séquences d'exécutions d'une campagne de test comme étant $m_S(T)$:

$m_S(T) = \frac{sup\{dt(s,T) | s \in S \setminus T\}}{\sum_{k=1}^{+\infty} P_k}$, le dénominateur $\sum_{k=1}^{+\infty} P_k$ servant à normer $m_S(T)$

entre 0 et 1. $sup\{E\}$ renvoie la plus grande valeur de l'ensemble E.

Intuitivement, évaluer $m_S(T)$ revient à calculer la distance entre T et le point de S le plus éloigné de T. La couverture basée sur les distances revient à définir la répartition d'un ensemble de tests T sur une spécification S (FIG. 7.3). $m_S(T)$ représente le rayon d'une boule centrée sur un test de T, c'est l'ensemble des comportements considérés comme analogues. L'interprétation de cette mesure est liée au choix de la distance qu'on a fait. Plus cette distance est petite, plus les tests peuvent être semblables entre eux, la couverture est ainsi meilleure.

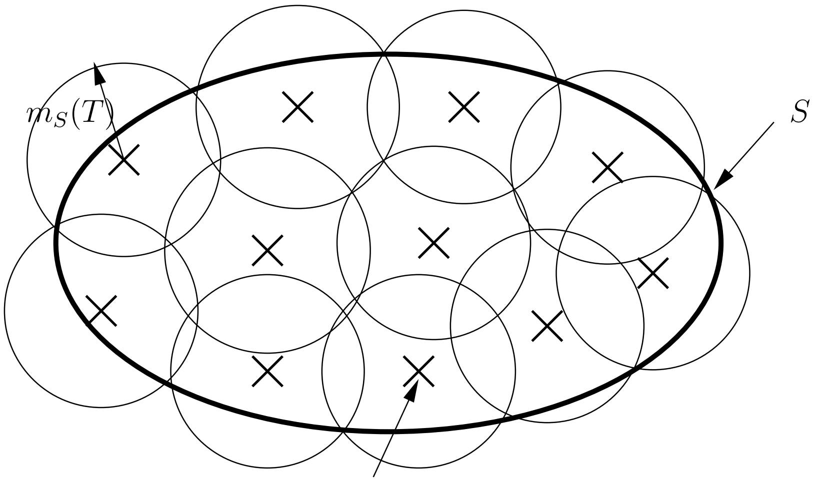


FIG. 7.3 – Couverture de S par un ensemble de boules centrées sur les tests de T

Dans la logique de notre approche avec des schémas de test et des objectifs de test, nous cherchons à produire de nombreux tests dont certains sont très proches car issus d'une même racine (objectif de test ou schéma de test.) Les hypothèses associées aux métriques ne sont pas celles qui correspondent à notre approche puisque nous nous intéressons à des séquences ayant de fortes similitudes alors que cette méthode tend à réduire ces similitudes. Il est à noter que l'aspect récursif pris en compte ne se base que sur la répétition d'un message, alors qu'il serait intéressant de noter la récursion de “cycles de messages”. On peut aussi noter que la base du calcul de la distance donne une importance bien plus grande aux récursions en début de test qu'aux récursions en fin de test. Enfin notre approche se veut la plus automatisée possible, la couverture par les distances est d'une certaine complexité qui rend difficile l'automatisation de la technique de couverture. On peut toujours définir P_k et r_j de façon générique pour toutes les applications, mais une mesure mieux adaptée à l'application testée nécessiterait de définir manuellement ces deux suites pour obtenir une distance dédiée au test de l'application considérée.

Couverture par hypothèses de test

Cette couverture, aussi nommée H-couverture, a été définie par O. Charles [Cha97] au cours de sa thèse. Les hypothèses de test supposent que l'implantation sous test a de bonnes propriétés qui restent à définir par le testeur (telles que les hypothèses d'uniformité, de régularité ou encore d'équité). Les hypothèses de test peuvent aussi servir d'outils de sélection des tests [BGM91, Gau93]. On peut envisager différentes hypothèses de test qui nous aident à définir une bonne suite de test. Les hypothèses servent à limiter la taille d'une suite de test, mais pas à créer une suite de test depuis rien.

Les hypothèses de test permettent de rendre fini un jeu de test couvrant un ensemble infini de comportements correspondant à un test exhaustif de l'application. Sous les hypothèses de test on peut donc effectuer du test exhaustif à partir d'un ensemble de tests fini. La notion de H-couverture d'une spécification *S* par une suite de tests *T* sous l'hypothèse *H* se définit de la façon suivante :

Définition 6 (H-couverture) *L'hypothèse H est une H-couverture de T si et seulement si T est exhaustif pour S sous l'hypothèse H. On dit aussi que l'hypothèse H H-couvre T.*

Propriété 1

1. *T a une H-couverture complète ssi $vrai$ est une H-couverture de T. T n'a pas besoin d'hypothèse pour être exhaustif.*
2. *$Faux$ est une H-couverture de n'importe quelle suite de tests T.*

Dans le cadre de la H-couverture, les tests sont ramenés à des traces d'exécution sur des IOLTS et les hypothèses sont traduites sous forme de fonctions sur des automates partiels correspondant aux exécutions possibles. Cette forme de couverture est élégante et permet de bien exprimer sous quelles conditions le test exhaustif est obtenu. Mais cette méthode ne permet pas une génération automatique d'une suite de test à partir des hypothèses, les hypothèses de test ne permettent pas de faire une structuration satisfaisante de la suite de test. Et pour chaque hypothèse de test nouvelle il faut pouvoir l'exprimer dans le formalisme adéquat.

7.4 Critère de couverture prise en compte

Nous utilisons dans notre approche la couverture des branches qui s'avère dans la pratique bien adaptée, car elle apporte un bon compromis entre le nombre de tests nécessaires et leur pertinence. Dans [GCR96] les auteurs montrent que cette couverture s'avère suffisante dans le cadre du test de conformité de protocoles de communication. On peut aussi apprécier le fait que cette notion de couverture s'attache aux transitions car ce sont les principaux éléments des spécifications sous forme de machines d'états finies.

Cet argument est repris dans la norme [ISO91] qui précise que le but d'un test revient à tirer la transition (ou l'ensemble des transitions) permettant de satisfaire une exigence de conformité formulée dans la spécification. Ceci lorsque les propriétés formulées dans le cahier des charges portent sur les transitions de la spécification.

Les approches basées sur les distances ou les hypothèses de test sont intéressantes mais elles sont aussi d'une plus grande complexité car il faut soit définir une distance, soit des fonctions correspondant aux hypothèses de test. Ces approches sont encore très marginales et s'intègrent difficilement dans le cadre des schémas de test et dans notre recherche d'automatisation.

Chapitre 8

Couverture et formalisation

Proposer, pour chaque niveau d'abstraction, une notion de couverture permet au testeur de rester au même niveau de formalisation tout au long de la campagne de test. Au cours de ce chapitre nous allons associer à chaque niveau d'abstraction une mesure de couverture pour déterminer quand l'activité de test se termine. Les schémas de test constituent le niveau le plus abstrait de description des tests, notre souhait est donc que le testeur puisse le plus aisément possible utiliser ce niveau d'abstraction.

L'intérêt d'effectuer une mesure de couverture au niveau des schémas de tests est d'obtenir rapidement des informations sur l'impact de la création d'un nouveau schéma de test sur une campagne de test sans avoir à changer de niveau d'abstraction. Pour exprimer une couverture au niveau des schémas de test et que cette couverture ait une pertinence, il faut pouvoir la mettre en relation avec les couvertures de niveau d'abstraction moindre (les couvertures au niveau des tests abstraits sont des couvertures connues). C'est pourquoi à chaque niveau d'abstraction des tests nous allons associer une couverture et la lier aux couvertures de niveau d'abstraction moindre.

Le but de ce chapitre est de présenter comment les schémas de test peuvent être mis en relation avec les autres niveaux d'abstraction. Nous allons successivement présenter de manière formelle la spécification comportementale sur laquelle s'appliquent les couvertures de test (section 8.2), les tests abstraits et la notion de couverture associée (section 8.3), les objectifs de test et la notion de couverture associée (section 8.4) et pour finir les schémas de test et la notion de couverture associée.

Pour formaliser ces différentes notions nous prendrons comme hypothèse que les schémas de test et les objectifs de test sont réduits à des séquences, nous discuterons des autres opérateurs disponibles pour écrire les schémas de test dans la section 11.

8.1 Les restrictions prises

Pour définir notre couverture nous formalisons les différents niveaux d'abstraction des tests. Lors de cette formalisation nous procédons à certaines restrictions pour faciliter notre étude. Ces restrictions font que la notion de schéma de test manipulée par la suite n'est plus tout à fait celle présentée dans le chapitre 4.

La première restriction porte sur le caractère des tests définis. Quelque soit leur niveau de description (tests abstraits, objectifs de test ou schémas de test), les tests sont non-branchus. Nous écartons donc les opérateurs d'alternative, de négation ainsi que de co-région. Nous n'effectuons en fait que des restrictions sur les schémas de test. Les restrictions sur les objectifs de test ainsi que sur les tests abstraits découlent de celles-ci.

Nous ne considérons comme opérateur que l'opérateur de séquencement, les schémas de test sont donc des suites d'appels. Nous ne considérons pas non plus les séquences strictes. Ces restrictions nous permettent de ne gérer que des chemins dans la spécification, et cela pour tous les niveaux d'abstraction.

Une autre restriction porte sur la notion de groupe. Comme nous le verrons dans la section 8.5.2, une méthode ne peut appartenir qu'à un groupe, alors que dans le cadre général des schémas de test nous pouvons associer une même méthode à différents groupes. Cette restriction nous permet de simplifier l'élaboration de l'abstraction de la spécification que nous aborderons dans le chapitre 9.

8.2 La spécification comportementale

Une spécification comportementale, telle que nous la prenons et telle qu'elle est définie dans le cadre de TGV [FJJ$^+$96], décrit l'ensemble des comportements autorisés par la spécification en matière d'envoi et de réception de messages. Nous faisons ici abstraction des actions internes, les messages sont respectivement les *inputs* et les *outputs* du système sous test. La spécification est décrite sous la forme d'un Input-Output Labelled Transition System (IOLTS) fini. Soit S une spécification comportementale :

Définition 7 (spécification comportementale)

$$S \hat{=} (Q^s, A^s, T^s, q^s_{init})$$

Avec Q^s l'ensemble fini des états, A^s l'alphabet des labels, $T^s \subseteq Q^s \times A^s \times Q^s$ l'ensemble des transitions et $q^s_{init} \in Q^s$ l'état initial. L'alphabet A^s se partitionne en deux sous-ensembles : A^s_i (input) l'ensemble des messages d'entrée et A^s_o (output) l'ensemble des messages de sortie. Un label de l'alphabet A^s est de la forme $i/o\ inst.methode(val^*)$ où i/o permet de savoir si le message est de type input ou output, $inst$ est le nom de l'instance réceptionnant le message, $methode$ celui d'une méthode associée à l'instance et val^* la liste instanciée des valeurs de ses paramètres.

Nous imposons que la spécification soit déterministe et contrôlable au sens des systèmes d'état-transition [Mor00], il ne peut pas y avoir deux transitions portant un même label depuis un même état, et si il existe une transition ayant un label de sortie (output), il ne peut pas y avoir d'autres transitions depuis cet état (ni input, ni output). Par contre un état peut comporter plusieurs transitions portant des labels d'entrée (input). Nous définissons différentes notions sur les IOLTS :

① $p \xrightarrow{\alpha} q$ si $(p, \alpha, q) \in T^s$

② $p \overset{\sigma}{\Rightarrow} q$ si $\exists \alpha_1, \alpha_2 \ldots \alpha_n \in A^s,\ p_0, \ldots, p_n \in Q^s . \sigma = \alpha_1.\alpha_2 \ldots \alpha_n \ \wedge\ p_0 = p \wedge p_n = q \wedge\ p_i \xrightarrow{\alpha_{i+1}} p_{i+1}\ \forall i \in 0..n-1$

③ $p\ after\ \sigma = \{q \in Q^s \mid p \overset{\sigma}{\Rightarrow} q\}$

④ $traces(p) = \{\sigma \in A^{s^+} \mid p\ after\ \sigma \neq \varnothing\}$

⑤ $L(q) = \{\alpha \in A^s | \exists q' \in Q^s \,.\, q \xrightarrow{\alpha} q'\}$

⑥ $O(q) = L(q) \cap A^s_o$

⑦ $I(q) = L(q) \cap A_i^s$

① définit une transition de S, p étant l'état initial de la transition, q son état final et α son label. ② signifie qu'il existe une suite de transitions qui permet de se rendre de l'état p à l'état q. Cette suite de transitions correspond à la suite de labels formant le mot σ. ③ donne l'ensemble des états atteignables depuis l'état p par le mot σ. Dans le cas d'un IOLTS déterministe, cet ensemble est réduit à au plus un état. ④ est l'ensemble des séquences finies possibles depuis l'état p. On définit aussi $traces_S = traces(q_{init}^s)$ comme étant l'ensemble des traces de la spécification S. ⑤ représente l'ensemble des labels des transitions issues de l'état q. ⑥ et ⑦ représentent l'ensemble des messages qui peuvent être émis depuis l'état q, respectivement les messages en direction de l'environnement et les messages en direction du système sous test. Nous pouvons maintenant définir formellement la contrôlabilité d'un IOLTS, ainsi que le fait qu'il soit déterministe :

- contrôlable(S) $\equiv \forall p \in Q^s \,.\, |O(p)| = 0 \vee (|O(p)| = 1 \wedge I(p) = O)$;
- déterministe(S) $\equiv \forall p \in Q^s \,.\, \forall \sigma \in A^{s^+} \,.\; |p\, after\, \sigma| \leq 1$.

8.3 Les tests abstraits

Les tests abstraits sont représentés syntaxiquement et sémantiquement dans le projet COTE dans le langage TeLa [PJH+01]. Ce langage s'écrit sous forme de diagrammes de séquences UML, ce qui permet d'intégrer les tests abstraits à l'environnement Objecteering. D'un point de vue sémantique, les tests abstraits sont des IOLTS déterministes, respectant la condition de contrôlabilité. Les tests abstraits sont des chemins dans S si ils sont cohérents avec S. Nous définissons l'ensemble E_{ta}^s des tests abstraits d'une spécification S :

Définition 8 (Ensemble des tests abstraits d'une spécification S)

$$E_{ta}^s = \{ta_S | ta_S = < (q_{init}^s, \alpha_1, q_1), (q_1, \alpha_2, q_2), \ldots (q_{k-1}, \alpha_k, q_k) > \wedge$$

$$\alpha_1\alpha_2 \ldots \alpha_k \in traces_S \wedge \forall i \in 1..k-1 \,.\, q_i \xrightarrow{\alpha_{i+1}} q_{i+1} \wedge q_i \in Q^s \wedge |O(q_k)| = 0\}$$

La condition sur les transitions issues de q_k signifie que q_k est un état stable, c'est-à-dire que c'est un état dans lequel la spécification ne peut plus progresser sans l'aide de l'environnement (le testeur). Un test abstrait ta_S est un test abstrait cohérent avec la spécification S s'il appartient à l'ensemble E_{ta}^s :

Définition 9 (Test abstrait sur la spécification S)

$$ta_S \in E^s_{ta}$$

Puisque la spécification est déterministe, à une suite de labels nous ne pouvons associer qu'un chemin dans la spécification, et inversement. Nous définissons la fonction $LLabel$ correspondant à la suite de labels formant un test abstrait ta_S telle que :

$$LLabel(ta_S) \;=\; \alpha_1\alpha_2\ldots\alpha_k$$

Nous pouvons associer des notions de couverture aux tests abstraits. Dans ce cas, la couverture porte sur la spécification. Soit $L_t(ta_S)$ l'ensemble des transitions de ta_S : $L_t(ta_S) = \{(q_i, a, q_j) \,|\, (q_i, a, q_j) \in ta_S\}$ ($\in$ désigne ici le prédicat est-élément d'une séquence). On peut maintenant définir la couverture des transitions (ou 1-branche) comme l'ensemble des transitions qu'il exerce et le taux de couverture d'un test abstrait comme le pourcentage de transitions de l'IOLTS qu'il exerce :

Définition 10 (couverture et taux de couverture d'un test abstrait pour une spécification S)

$$Couverture_s(ta_S) = L_t(ta_S)$$

$$TauxCouverture_s(ta_S) = \frac{|Couverture_s(ta_S)|}{|T_s|}$$

Note : on voit bien ici que cette notion de couverture n'a de sens que si l'ensemble des transitions est fini. Sinon la couverture serait $\frac{|n|}{\infty} = 0$

8.4 Les objectifs de test

Dans le projet COTE, les objectifs de test sont décrits dans le langage O-TeLa [BMdB+01b] qui reprend la syntaxe du langage TeLa. Sémantiquement, un objectif de test est un IOLTS. Dans cette étude nous réduisons l'objectif de test à une suite de labels ordonnés mais non forcément consécutifs dans la spécification. Nous considérons simplement une suite car les restrictions apportées aux schémas de test dans la section 8.1 nous assurent que les objectifs de test qui seront déduits des schémas de test seront non-branchus. La définition 11 est celle d'un objectif de test ot pour la spécification S.

Définition 11 (objectif de test)

$$ot \hat{=} a_1 a_2 \dots a_n \text{ où } a_i \in A^s \text{ avec } i = 1..n$$

Nous présentons une définition formelle de l'ensemble des tests abstraits associés à un objectif de test. Un test abstrait ta_S tel que $LLabel(ta_S) = \alpha_1 \dots \alpha_k$ appartient à $TA_S(ot)$ si la fonction $Realise_{OT}$ est vérifiée[1], s'il se termine sur le premier état stable rencontré après reconnaissance de l'objectif de test et s'il ne comporte pas de boucles improductives du point de vue de l'objectif de test.

Définition 12 ($\mathbf{Realise_{OT}}$)

$$Realise_{OT}(ot, LLabel(ta_S)) \equiv \exists h \in (1..n \rightarrowtail 1..k) \,.\, (\forall i \in 1..n,$$
$$j \in 1..n \,.(\, a_i \equiv \alpha_{h(i)} \wedge \, i < j \Rightarrow h(i) < h(j)))$$

où n est le nombre de labels de ot et k le nombre de labels de ta_S.

Cette définition met en relation les éléments a_i d'un objectif de test et les éléments α_j d'un test abstrait en respectant l'ordre d'apparition. La fonction h est une fonction injective et totale sur $1..n$. On peut définir la fonction $H_h(LLabel(ta_S)) = ot$ qui donne l'objectif de test associé à un test abstrait, pour une fonction h donnée.

Condition 1 (les labels éligibles ne sont pas négligés) $\forall i \in 1..n, \; a_i \notin \alpha_k..\alpha_{l-1}$ *avec* $l = h(i)$ *et, si* $i = 1$ *alors* $k = 1$ *sinon* $k = h(i-1)$.

Cette condition nous assure que pour tout chemin donné dans la spécification correspondant à un objectif de test, il n'y a qu'un seul test abstrait généré, et c'est

[1] ot est une sous-séquence de ta_S.

le plus court de tous.

Condition 2 (termine sur le premier état stable) $\forall l \in h(n)..k-1, |I(q_l)| = 0$

Nous nous assurons ainsi que le test abstrait, après reconnaissance de la chaîne de label correspondant à l'objectif de test ot, ne comporte qu'un état stable et que celui-ci est le dernier état du test abstrait[2].

Condition 3 (pas de cycle improductif) *Si ta_S contient une sous-séquence de la forme* $< (q_j, \alpha_l, q_{j+1}), \ldots (q_m, \alpha_{l+m}, q_j) >$ *alors (l..l+m)* $\cap\, ran(h) \neq \varnothing$

Cette condition signifie que si un test abstrait contient une boucle de taille m sur l'état q_j, alors cette boucle "fait avancer" la lecture de l'objectif de test correspondant[3]. Il n'existe donc pas de boucle dite improductive dans un test abstrait, pour un objectif de test donné.

La définition 12 et les conditions 1, 2 et 3 permettent d'associer à un objectif de test ot un ensemble $TA_S(ot)$ fini de tests abstraits. Ceux-ci peuvent être calculés en énumérant les chemins vérifiant les conditions 1, 2 et 3.

Définition 13 ($\mathbf{TA_S(ot)}$)

$$TA_S(ot) = \{ta_S | ta_S \in E^s_{TA} \wedge Realise_{OT}(ot, LLabel(ta_S))$$

$$\wedge ta_S \textit{ vérifie les conditions 1, 2 et 3}\}$$

On peut définir la couverture des objectifs de test vis-à-vis de la spécification S de la façon suivante :

Il est à noter que la notion de relation entre objectifs de test et tests abstraits définie pour la couverture des objectifs de test dans le cadre présent considère l'ensemble des tests abstraits possibles que l'on peut obtenir à partir d'un objectif de test. Cela est difficile à réaliser avec TGV puisque cet outil calcule un test abstrait pour chaque objectif de test à chaque exécution, il faudrait plusieurs exécutions pour un même objectif de test pour obtenir possiblement plusieurs tests abstraits.

[2]Pour rappel, par définition le test abstrait se termine par un état stable.

[3]ran(h) désigne le co-domaine de la fonction h.

Définition 14 (couverture et taux de couverture d'un objectif de test pour une spécification S)

$$Couverture_s(ot) = \bigcup_{ta_S \in TA_S(ot)} Couverture_s(ta_S)$$

$$TauxCouverture_s(ot) = \frac{|Couverture_s(ot)|}{|T_s|}$$

8.5 Les schémas de test

En introduisant des abstractions supplémentaires, nous pouvons synthétiser l'écriture de plusieurs objectifs de test en un schéma de test. Nous présentons formellement les différentes abstractions proposées avant d'étudier comment nous pouvons exprimer la couverture des schémas de test vis-à-vis de la spécification.

Avant de présenter les différentes abstractions, nous posons quelques opérations sur les labels. Un label $a \in A^s$ est de la forme "$i/o\ inst.methode(val^*)$", comme présenté dans la définition 7 de la section 8.2. Les opérations associées à un label a sont :

- $\boxed{M_L(a) = methode}$ cette fonction renvoie la méthode associée à un label a ;
- $\boxed{Inst_L(a) = inst}$ cette fontion renvoie l'instance associée à un label a ;
- $\boxed{IO_L(a) = i/o}$ cette fonction renvoie la notation du label a qui indique si il s'agit d'un message de type "input" ou "output".

Pour chaque abstraction nous présentons les différences entre la définition des schémas de test faite ici et celle faite dans le chapitre 4.

8.5.1 Abstraction sur les paramètres

Dans la spécification S, les méthodes portées sur les labels ont toutes leurs paramètres instanciés. Certains objectifs de test peuvent varier d'une simple valeur de paramètre. Nous proposons d'effectuer un changement d'alphabet des labels, afin de faire abstraction des paramètres des méthodes. Soit A_p le nouvel alphabet, un label est alors de la forme $a' = i/o\ inst.methode$. On a une fonction totale

$F_p \in A^s \rightarrow A_p$ telle que :

Définition 15 ($\mathbf{F_p}$)

$$\forall a \in A^s.\, IO_L(a) = IO_L(F_p(a)) \wedge Inst_L(a) = Inst_L(F_p(a)) \wedge$$
$$M_L(a) = M_L(F_p(a))$$

Soit : $F_p(i/o\ inst.methode(val^*)) = i/o\ inst.methode$. Pour la suite nous étendons la fonction F_p aux chaînes de labels : $F_p(a_1, \ldots, a_n) = F_p(a_1), \ldots, F_p(a_n)$.

Les valeurs ne sont plus accessibles pour les schémas de test. Dans le chapitre 4 les valeurs étaient associées aux méthodes dans les groupes, tandis qu'ici elles sont inexistantes.

8.5.2 Abstraction sur les méthodes

Une autre abstraction consiste à regrouper des méthodes différentes sous un même label. Au lieu d'écrire plusieurs fois les mêmes tests avec des noms d'opérations différents, nous pouvons créer des **groupes**, qui regroupent certaines méthodes. On définit une fonction totale G. $G : m \rightarrow g$ où m est l'ensemble des méthodes et g est l'ensemble des groupes. Toutes les méthodes de l'alphabet sont associées à un groupe et un seul. Un groupe peut être réduit à une seule méthode. Soit A_g l'alphabet prenant en compte les groupes, un label est alors de la forme : $a'' = i/o\ inst.groupe$. On surcharge la fonction M_L tel que $M_L(a'') = groupe$. On a une fonction $F_g \in A_p \rightarrow A_g$ telle que :

Définition 16 ($\mathbf{F_g}$)

$$\forall a' \in A_p.\, IO_L(a') = IO_L(F_g(a')) \wedge Inst_L(a') = Inst_L(F_g(a')) \wedge$$
$$G(M_L(a')) = M_L(F_g(a'))$$

Soit : $F_g(i/o\ inst.methode) = i/o\ inst.G(methode) = i/o\ inst.groupe$. Pour la suite nous étendons la fonction F_g aux chaînes de labels :

$F_g(a_1, \ldots, a_n) = F_g(a_1), \ldots, F_g(a_n)$.

Dans le chapitre 4 nous ne définissions pas de restriction sur les méthodes, elles pouvaient apparaître dans différents groupes avec différentes valeurs de paramètre. Dans le cas présent, une méthode appartient à un et un seul groupe, nous établissons une fonction totale entre l'ensemble des méthodes et l'ensemble des groupes.

8.5.3 Abstraction sur les instances

Lorsque nous avons plusieurs instances d'une même classe, nous souhaitons souvent exécuter les mêmes tests sur ces différents objets. Nous offrons la possibilité de faire une abstraction sur les instances, afin de factoriser des schémas de test. L'alphabet des schémas de test ne comporte plus d'instances. Soit A' l'alphabet des abstractions sans instance, un label est alors de la forme : $A' = i/o\ groupe$. Nous avons une fonction $F_i \in A_g \rightarrow A'$ telle que :

Définition 17 ($\mathbf{F_i}$)

$$\forall a'' \in A_g.\ IO_L(a'') = IO_L(F_i(a'')) \wedge M_L(a'') = M_L(F_i(a''))$$

Soit : $F_i(i/o\ inst.groupe) = i/o\ groupe$. Pour la suite nous étendons la fonction F_i aux chaînes de labels : $F_i(a_1, \ldots, a_n) = F_i(a_1), \ldots, F_i(a_n)$.

Les instances sont là encore totalement ignorées, alors que dans le chapitre 4 on pouvait les exprimer dans les schémas de test.

8.5.4 Schémas de test et couverture

Nous venons de présenter les différentes abstractions mises en place à l'aide des fonctions $F_p,\ F_g$ et F_i. Nous définissons maintenant une fonction $F_A \in A^s \rightarrow A'$ comme étant la composition de ces trois fonctions. Nous avons alors :

Définition 18 ($\mathbf{F_A}$)

$$\forall a \in A^s.\ IO_L(a) = IO_L(F_A(a)) \wedge G(M_L(a)) = M_L(F_A(a))$$

Soit : $F_A(i/o\ inst.methode(val^*)) = i/o\ G(methode) = i/o\ groupe$. Pour la suite nous étendons la fonction F_A aux chaînes de labels : $F_A(a_1, \ldots, a_n) = F_A(a_1), \ldots, F_A(a_n)$.

Soit st un schéma de test décrit par la suite de labels $a'_1 a'_2 \ldots a'_n$ avec $i \in 1..n$ et $a'_i \in A'$. Un objectif de test ot associé à st vérifie la condition $F_A(ot) = st$. On note alors $OT_S(st)$ l'ensemble des objectifs de test de la spécification S dont l'abstration par F_A est st :

Définition 19 ($\mathbf{OT_S(st)}$)

$$OT_S(st) = \{ot | F_A(ot) = st\}$$

Comme pour les autres niveaux d'abstraction, nous pouvons définir une notion de couverture des branches de la spécification S pour les schémas de test en se basant sur les notions de couverture des niveaux d'abstraction moins élevés. Ainsi on peut définir la couverture par st de la spécification S de la façon suivante :

Définition 20 (couverture et taux de couverture d'un schéma de test pour une spécification S)

$$Couverture_s(st) = \bigcup_{ot \in OT_S(st)} Couverture_s(ot)$$

$$TauxCouverture_s(st) = \frac{|Couverture_s(st)|}{|T_s|}$$

La notion de schéma de test que nous définissons ici est donc bien différente de celle définie dans le chapitre 4. Elle ne comporte que le constructeur de séquences, les labels pris en compte ne reconnaissent plus les instances, une méthode n'est associée qu'à un seul groupe et les valeurs des méthodes ne sont plus prises en compte.

Nous venons de définir formellement les différents niveaux d'abstraction pour définir des tests. A chaque niveau nous avons associé une notion de couverture de test qui reste cohérente entre chaque niveau d'abstraction. Dans le chapitre suivant, nous allons montrer de quelle autre façon nous pouvons exprimer une couverture de test au niveau des schémas de test sans avoir recours aux objectifs de test et aux tests abstraits. Nous exprimerons aussi la relation qui existe entre ces deux couvertures de test.

Chapitre 9

Le critère de couverture proposé

La notion de couverture que nous définissons par les schémas de test s'appuie sur la couverture des objectifs de test, qui elle-même s'appuie sur les tests abstraits, donc la couverture par un schéma de test peut s'exprimer par la couverture par les tests abstraits qui lui sont associés via des objectifs de test. Nous pouvons définir une relation entre un schéma de test et ses tests abstraits associés. On définit la fonction EM_{TA}, qui associe un ensemble de tests abstraits à un ensemble d'objectifs de tests de la façon suivante : $EM_{TA}(\{ot_1, \ldots, ot_n\}) = \bigcup_{ot \in \{ot_1,\ldots,ot_n\}} TA_S(ot)$.

On peut ainsi calculer l'ensemble des tests abstraits associés à un schéma de test par composition de EM_{TA} et de OT_S (définition 19). On déplie un schéma de test en un ensemble d'objectifs de test en tenant compte de l'alphabet A^s du système. On a ainsi la fonction $ST_{TA}(st)$ suivante :

Définition 21 ($\mathbf{ST_{TA}(st)}$)

$$ST_{TA}(st) = EM_{TA} \circ OT_S(st) \equiv \bigcup_{ot \in OT_S(st)} TA_S(ot)$$

Nous allons maintenant chercher à obtenir de manière plus effective la couverture potentielle de la spécification par un schéma de test ou un ensemble de schémas de test. Pour cela, nous n'exécutons plus la chaîne *schéma de test → ensemble d'objectifs de test → ensemble de tests abstraits → spécification* illustrée dans la figure 9.1. Une façon plus directe consiste à construire une spécification abstraite sur l'alphabet A' des schémas de test et à mesurer la couverture des

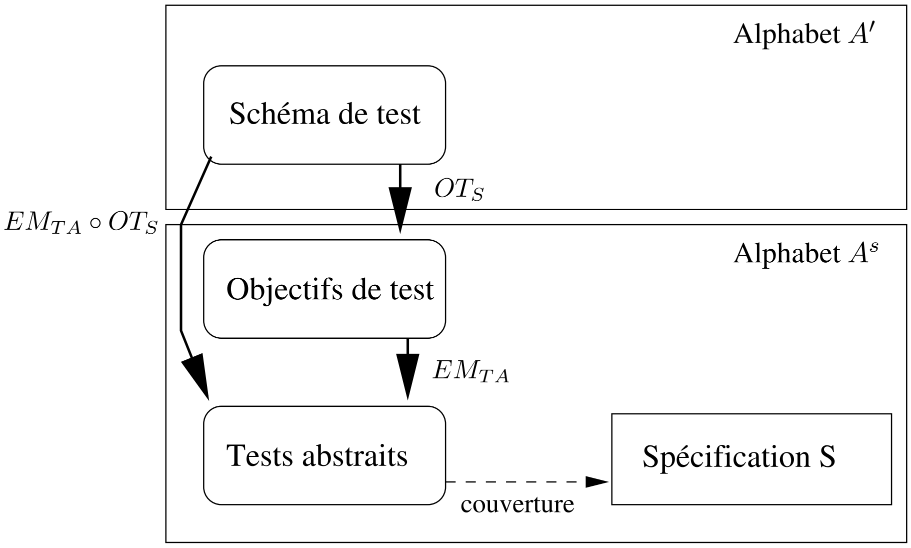


FIG. 9.1 – Calcul de couverture à partir d'un schéma de test

schémas de test sur cette spécification abstraite. Ce processus est décrit dans la section 9.1, ainsi que la définition de la couverture des schémas de test pour cette abstraction. La section 9.2 met en relation cette abstraction de la spécification avec la spécification comportementale initiale. Cette mise en relation des spécifications nous permet de mettre aussi en relation les deux notions de couverture associées aux schémas de test. Enfin la section 9.3 présentera comment cette mesure de couverture est applicable sur un exemple simple.

9.1 Abstraction S' de la spécification

Nous avons vu qu'il était possible de calculer la couverture des schémas de test sur la spécification S de façon indirecte, en calculant les objectifs de test puis les tests abstraits. Cette méthode oblige à dérouler toute la chaîne de génération des tests. Au lieu de chercher à déplier chaque schéma de test pour le mettre au niveau de la spécification, nous allons construire une abstraction de la spécification sur laquelle nous allons exprimer la couverture des schémas de test.

L'abstraction de la spécification S porte sur les labels des transitions. Les labels de S appartiennent à l'alphabet A^s, nous allons les transformer en des labels appartenant à l'alphabet A', tel que nous l'avons introduit pour les schémas de test (section 8.5), en utilisant la fonction F_A (définition 18). Soit une fonction d'abstraction $Lab_A \in IOLTS \to IOLTS$ qui abstrait les labels d'un IOLTS par la fonction F_A. Nous avons $S' = Lab_A(S)$.

Définition 22 (abstraction de la spécification) $Lab_A(S)$

$$S' \hat{=} (Q^s, A', T', q^s_{init}) = Lab_A(S)$$

avec $A' = F_A(A^s) \wedge T' = \{(p, F_A(a), q) | (p, a, q) \in T^s\}$.

S' est un IOLTS éventuellement non-déterministe qui reconnaît un langage sur l'alphabet A'. Par construction nous avons les propriétés suivantes entre les spécifications S et S' :

Propriété 2 $w \in traces(q^s_{init}) \Rightarrow F_A(w) \in traces'(q^s_{init})$ *où* $traces'$ *est la fonction* $traces$ *dans* S'. *Toute trace de* S *a une abstraction dans* S'.

Propriété 3 $w' \in traces'(q^s_{init}) \Rightarrow \exists w \,.\, w \in traces(q^s_{init}) \wedge F_A(w) = w'$. *Toute trace de* S' *a au moins une concrétisation dans* S.

Propriété 4 *La fonction* F_A *préserve les ensembles input et output des labels, la notion de contrôlabilité est donc respectée sur* S'. *Nous conservons la même contrôlabilité pour un état* q_i *de* S *et de* S'.

Comme décrit dans la section 8.5.4, un schéma de test st est défini comme une suite de labels $a'_1 a'_2 \dots a'_n$ avec $a'_i \in A'$. Les schémas de test peuvent être vus comme des objectifs de test pour la spécification S'. On peut donc calculer l'ensemble des tests abstraits $TA_{S'}(st)$ sur S', comme décrit dans la section 8.4, et appliquer les calculs de couverture des objectifs de test (définition 14) aux schémas de test sur la spécification S'.

9.2 Relation de couverture entre S et S'

Nous voulons que les informations de couverture obtenues pour la spécification S', par un ensemble de schémas de test, soient interprétables sur la spécification S telle que $Lab_A(S) = S'$. Nous allons montrer les deux propriétés suivantes :

Propriété 5 *Si, pour un schéma de test st donné, une transition (q_i, a', q_j) de S' est couverte par un test abstrait issu de $TA_{S'}(st)$, alors toutes les transitions de S de la forme (q_i, a, q_j), avec $F_A(a) = a'$, sont couvertes.*

Propriété 6 *Si pour un ensemble de schémas de test nous obtenons $TauxCouverture_{S'} = 1$ alors nous obtenons aussi $TauxCouverture_S = 1$.*

La propriété 6 découle directement de la propriété 5 puisque la fonction d'abstraction Lab_A est une fonction totale qui associe à toute transition de S une transition de S'. Donc si toutes les transitions de S' sont couvertes, toutes celles de S le sont aussi si la propriété 5 est vraie.
Pour établir la propriété 5, nous allons montrer que, pour un schéma de test st, les tests abstraits sur S obtenus par $ST_{TA}(st)$ sont liés aux tests abstraits sur S' obtenus par $TA_{S'}(st)$, selon[1] :

$$(1) InvLab^s_A[TA_{S'}(st)] \subseteq ST_{TA}(st)$$

Avec $InvLab^s_A(ta') = \{ta_S | ta_S \in E^s_{TA} \wedge Lab_A(ta_S) = ta'\}$ où ta_S est un test abstrait sur S (définition 9) et ta' un test abstrait sur S'.

La figure 9.2 présente les différentes fonctions qui lient un schéma de test st à des objectifs de test, ainsi que des tests abstraits sur S à des tests abstraits sur S'.

Un schéma de test st peut donc se dériver en un ensemble $TA_{S'}(st)$ de tests abstraits pour l'abstraction S' et en un ensemble $ST_{TA}(st)$ de tests abstraits pour la spécification S. La relation (1) entre ces deux ensembles revient à montrer la formule (2) suivante :

$$(2)\ ta \in InvLab^s_A[TA_{S'}(st)] \Rightarrow ta \in ST_{TA}(st)$$

Nous réécrivons cette formule afin de mieux l'exploiter. La partie gauche de l'implication peut se réécrire de la façon suivante :

$$\begin{aligned} ta \in InvLab^s_A[TA_{S'}(st)] &= \exists ta'.ta' \in TA_{S'}(st) \wedge ta \in InvLab^s_A(ta') \\ &= \exists ta'.ta' \in TA_{S'}(st) \wedge \\ &\quad\ ta \in \{ta_S | ta_S \in E^s_{TA} \wedge Lab_A(ta_S) = ta'\} \\ &= ta \in E^s_{ta} \wedge Lab_A(ta) \in TA_{S'}(st) \end{aligned}$$

[1]Nous utilisons la notation B "$[E]$" pour représenter l'image d'un ensemble E par une fonction.

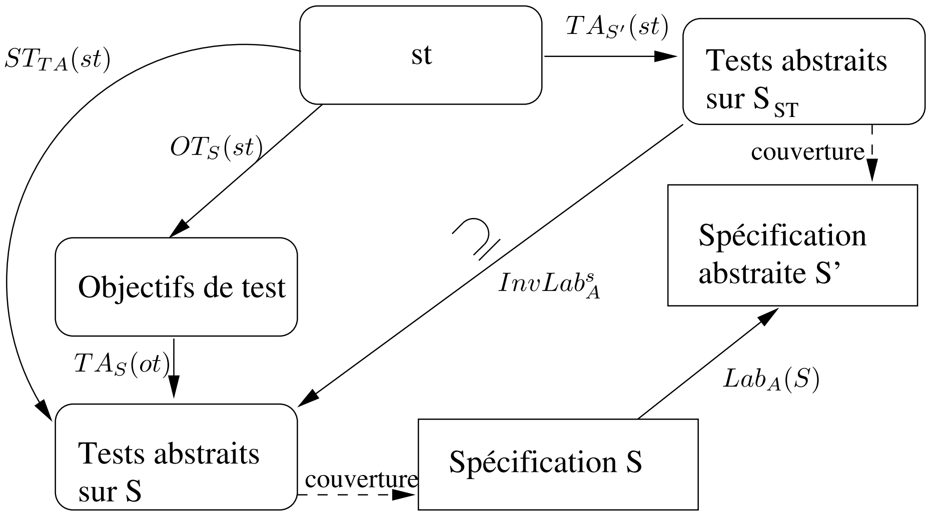


FIG. 9.2 – Abstraction de la spécification S et des tests abstraits sur S

En utilisant la définition de ST_{TA} (définition 21), la partie droite de la formule (2) se réécrit sous la forme suivante : $\exists ot.(\ ta \in TA_S(ot) \wedge F_A(ot) = st)$. Nous pouvons ainsi réexprimer la formule (2) sous la forme suivante :

$$(3)\ ta \in E^s_{ta} \wedge Lab_A(ta) \in TA_{S'}(st) \Rightarrow \exists ot.(\ ta \in TA_S(ot) \wedge F_A(ot) = st)$$

La partie gauche de l'implication permet de déduire, par définition de l'ensemble $TA_{S'}(st)$ (définition 13), les hypothèses suivantes :
- (4) $LLabel(Lab_A(ta)) \in traces'(q^s_{init})$
- (5) $\exists h.\ H_h(LLabel(Lab_A(ta))) = st$

L'hypothèse (4) correspond au fait que $Lab_A(ta)$ est un test abstrait sur S', c'est donc un chemin sur S'. L'hypothèse (5) correspond au fait que si $Lab_A(ta)$ appartient à $TA_{S'}(st)$, alors il existe une fonction h sur les labels de $Lab_A(ta)$ telle que l'application de cette fonction sur ce test abstrait correspond au schéma de test st.

Nous avons également $LLabel(ta) \in traces(q^s_{init})$ puisque ta appartient à l'ensemble E^s_{TA} des tests abstraits sur S. Comme ta a la même longueur que $Lab_A(ta)$, nous pouvons appliquer la fonction H_h à $LLabel(ta)$. Nous pouvons construire un objectif de test ot, tel que $H_h(LLabel(ta)) = ot$. Par construction, on a donc $Realise_{OT}(ot, LLabel(ta))$. Dès lors que $ta \in TA_S(ot)$ (définition 13), il faut que les 3 conditions exprimées dans la section 8.4 soient respectées. Or on sait que $Lab_A(ta)$ respecte ces conditions pour st, puisque $Lab_A(ta) \in TA_{S'}(st)$. Comme la même fonction h associe $Lab_A(ta)$ à st et ta à ot, si le label de la i^{eme} transition de $Lab_A(ta)$ correspond au j^{eme} label de st, il en sera de même pour le label de la i^{eme} transition de ta et le j^{eme} label de ot. Cette correspondance entre les placements des labels de st et de ot dans (respectivement) $Lab_A(ta)$ et ta nous permet de garantir la préservation des conditions 1, 2 et 3 :

– Condition 1. ta est le plus court test abstrait correspondant à ot sur ce chemin de la spécification. Comme $Lab_A(ta)$ comporte des labels plus simples que ta et qu'il respecte cette condition, en ajoutant de l'information aux labels on ne peut pas trouver un test plus court encore.
– Condition 2. L'état final de ta est le premier état stable après reconnaissance de la chaîne de labels correspondant à ot.
– Condition 3. ta ne comporte pas de cycle improductif vis-à-vis de ot.

Nous avons donc $ta \in TA_S(ot)$. Il nous reste à montrer que $F_A(ot) = st$. En reprenant l'hypothèse (5) nous avons $H_h(LLabel(Lab_A(ta))) = st$. Nous allons appliquer les propriétés transitives de la fonction Lab_A pour obtenir une expression de la forme $F_A(H_h(LLabel(ta)) = st$, soit $F_A(ot) = st$.

$$
\begin{aligned}
st \;=\; H_h(LLabel(Lab_A(ta))) \;&=\; H_h(F_A(LLabel(ta))) \\
&=\; H_h(F_A(\alpha_1 \ldots \alpha_k)) \\
&=\; H_h(F_A(\alpha_1) \ldots F_A(\alpha_k)) \\
&=\; H_h(\alpha'_1 \ldots \alpha'_k) \\
&=\; \alpha'_{h1} \ldots \alpha'_{hn} \\
&=\; F_A(\alpha_{h1}) \ldots F_A(\alpha_{hn}) \\
&=\; F_A(\alpha_{h1} \ldots \alpha_{hn}) \\
&=\; F_A(H_h(\alpha_1 \ldots \alpha_k)) \\
&=\; F_A(H_h(LLabel(ta))) \;=\; F_A(ot)
\end{aligned}
$$

Nous obtenons bien $F_A(ot) = st$. C'est à dire que l'objectif de test ot construit à partir de h et de ta est produit par st.

La couverture des schémas de test sur la spécification abstraite S' constitue ainsi un sous ensemble de transitions correspondant à la couverture obtenue par les tests abstraits sur la spécification comportementale S.

9.3 Un exemple d'utilisation

A travers un exemple simple de distributeur de boisson, nous allons illustrer la démarche présentée dans ce chapitre. Dans cet exemple nous reprenons les notions de spécification, test abstrait, objectif de test, schéma de test, fonction d'abstraction et abstraction de spécification à travers une campagne de test.

Nous modélisons un distributeur de boissons avec deux types de boissons au choix, du thé ou du café. Nous offrons aussi la possibilité de sucrer la boisson. Une boisson coûte 20 centimes d'euro, on peut soit mettre deux pièces de 10 centimes, soit une pièce de 20 centimes. Le IOLTS du haut de la figure 9.3 représente la spécification comportementale S de ce système avec des méthodes associées à deux instances, l'instance *commande* qui correspond à l'interface entre l'utilisateur et la machine (panneau de commande et monnayeur), et l'instance *délivre* qui correspond à la partie mécanique du distributeur qui prépare la boisson.

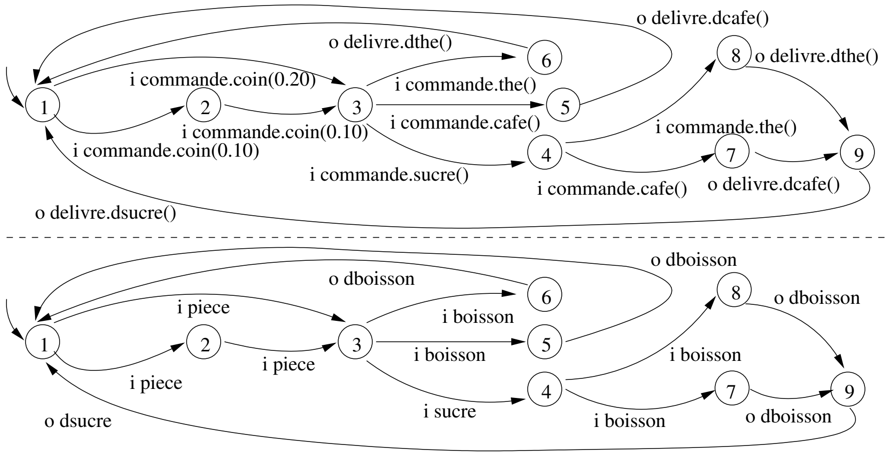


FIG. 9.3 – Spécification comportementale du distributeur de boissons (en haut) et son abstraction (en bas)

Les états ①, ②, ③ et ④ sont les états stables de cette spécification, tout test abstrait s'arrêtera donc dans l'un de ces quatres états.

9.3.1 Une campagne de test

Voici les objectifs de test que nous prenons en compte :

- $ot_1 \hat{=} i\ commande.coin(0.20)$; $o\ delivre.dsucre()$
- $ot_2 \hat{=} i\ commande.coin(0.10)$; $o\ delivre.dsucre()$
- $ot_3 \hat{=} i\ commande.coin(0.20)$; $i\ commande.cafe()$
- $ot_4 \hat{=} i\ commande.coin(0.20)$; $i\ commande.the()$
- $ot_5 \hat{=} i\ commande.coin(0.10)$; $i\ commande.cafe()$
- $ot_6 \hat{=} i\ commande.coin(0.10)$; $i\ commande.the()$

Les objectifs de test ot_1 et ot_2 correspondent au fait qu'on peut obtenir une boisson sucrée quelque soit le mode de paiement utilisé, deux pièces de 0.10 euros ou une pièce de 0.20 euros. Les objectifs de test ot_3 à ot_6 ont pour but de montrer qu'on peut commander du café ou du thé indépendamment du mode de paiement utilisé. Pour décrire les tests abstraits, nous omettons les instances pour avoir une représentation textuelle lisible (les messages de types input correspondent à l'instance *commande*, et les messages de type output correspondent à l'instance *delivre*).

Pour l'objectif de test ot_1, nous obtenons le jeu de tests abstraits TA_1 suivant :
- $1 \xrightarrow{i\ coin(0.20)} 3 \xrightarrow{i\ sucre()} 4 \xrightarrow{i\ cafe()} 7 \xrightarrow{o\ dcafe()} 9 \xrightarrow{o\ dsucre()} 1$
- $1 \xrightarrow{i\ coin(0.20)} 3 \xrightarrow{i\ sucre()} 4 \xrightarrow{i\ the()} 8 \xrightarrow{o\ dthe()} 9 \xrightarrow{o\ dsucre()} 1$

Pour l'objectif de test ot_2, nous obtenons le jeu de tests abstraits TA_2 suivant :
- $1 \xrightarrow{i\ coin(0.10)} 2 \xrightarrow{i\ coin(0.10)} 3 \xrightarrow{i\ sucre()} 4 \xrightarrow{i\ cafe()} 7 \xrightarrow{o\ dcafe()} 9 \xrightarrow{o\ dsucre()} 1$
- $1 \xrightarrow{i\ coin(0.10)} 2 \xrightarrow{i\ coin(0.10)} 3 \xrightarrow{i\ sucre()} 4 \xrightarrow{i\ the()} 8 \xrightarrow{o\ dthe()} 9 \xrightarrow{o\ dsucre()} 1$

L'objectif de test ot_1 a un taux de couverture de 0.54 (7 transitions couvertes pour 13 possibles) avec l'ensemble TA_1. De même l'objectif de test ot_2 a un taux de couverture de 0.62 (8 transitions couvertes sur 13) avec l'ensemble TA_2. En regroupant les couvertures obtenues par TA_1 et TA_2, on obtient une couverture de S de 0.69.

Les objectifs de test ot_3 à ot_6 à eux seuls permettent de couvrir complètement la spécification.

9.3.2 Mise en œuvre de la méthode

Nous créons une fonction F_A associée aux labels de la spécification S. La première abstraction porte sur les paramètres, ainsi les labels $i\ commande.coin(0.10)$ et $i\ commande.coin(0.20)$ deviennent des labels de la forme $i\ commande.coin()$.

La deuxième abstraction faite sur les labels consiste à abstraire les méthodes en créant les groupes :
- *piece* : coin() ;

– *boisson* : cafe() et the() ;
– *sucre* : sucre() ;
– *dboisson* : dcafe() et dthe() ;
– *dsucre* : dsucre().

Nous regroupons des méthodes sous un même groupe parce qu'elles ont un traitement semblable, ceci reste le choix et la responsabilité du testeur. Les méthodes que nous ne souhaitons pas regrouper correspondent à des groupes dont le nom est celui de la méthode. C'est le cas pour les groupes *sucre()* et *dsucre()*.
La dernière abstraction porte sur les instances, ainsi le label $i\ commande.piece$ devient $i\ piece$. Avec ces différentes abstractions, nous obtenons la fonction F_A tel que $S' = Lab_A(S)$. S' est le IOLTS du bas de la figure 9.3.
Nous allons écrire deux schémas de test :

1. $st_1 \hat{=} i\ piece\,;\ o\ dsucre$
2. $st_2 \hat{=} i\ piece\,;\ i\ boisson$

Le schéma de test st_1 signifie qu'après avoir inséré de l'argent, on peut obtenir du sucre et le schéma de test st_2 signifie qu'après avoir inséré de l'argent, on peut commander une boisson.
Nous avons $\{ot_1,\ ot_2\} = OT_S(st_1)$. L'ensemble des tests abstraits sur S' associés à ce schéma offre un taux de couverture de S' de 0.69 (avec 9 transitions couvertes sur 13). $TA_{S'}(st1)$ =

– $1 \xrightarrow{i\ piece} 3 \xrightarrow{i\ sucre} 4 \xrightarrow{i\ boisson} 7 \xrightarrow{o\ dboisson} 9 \xrightarrow{o\ dsucre} 1$
– $1 \xrightarrow{i\ piece} 3 \xrightarrow{i\ sucre} 4 \xrightarrow{i\ boisson} 8 \xrightarrow{o\ dboisson} 9 \xrightarrow{o\ dsucre} 1$
– $1 \xrightarrow{i\ piece} 2 \xrightarrow{i\ piece} 3 \xrightarrow{i\ sucre} 4 \xrightarrow{i\ boisson} 7 \xrightarrow{o\ dboisson} 9 \xrightarrow{o\ dsucre} 1$
– $1 \xrightarrow{i\ piece} 2 \xrightarrow{i\ piece} 3 \xrightarrow{i\ sucre} 4 \xrightarrow{i\ boisson} 8 \xrightarrow{o\ dboisson} 9 \xrightarrow{o\ dsucre} 1$

Ici $InvLab^s_A[TA_{S'}(st_1)] = TA_1 \cup TA_2$ où TA_1 et TA_2 sont les ensembles de tests abstraits respectifs des objectifs de test ot_1 et ot_2. La couverture est équivalente.

Les objectifs de test ot_3, ot_4, ot_5 et ot_6 dérivent du schéma de test st_2. Nous pouvons construire les tests abstraits sur S' qui dérivent de ce schéma de test, ils sont au nombre de huit et couvrent les 13 transitions de S'. Ce schéma de test est suffisant pour obtenir une couverture totale de S'. Nous obtenons cette même couverture totale sur S à partir des tests abstraits produits à partir de st_2.

9.3.3 Exemple d'inclusion des couvertures

Dans l'exemple précédent nous obtenons l'égalité entre les deux types de couverture que nous pouvons calculer à partir des schémas de test. L'égalité entre ces couvertures reste un cas particulier puisque la couverture obtenue sur l'abstraction de la spécification est incluse au sens large dans la couverture obtenue via les objectifs de test.

Une spécification $Spec$: $0 \xrightarrow{a} 1 \xrightarrow{b} 2$.
Un groupe G : $\{a; b\}$
L'abstraction de la spécification $Spec_{Abs}$: $0 \xrightarrow{G} 1 \xrightarrow{G} 2$
Un schéma de test $S1 = G$

FIG. 9.4 – Illustration des analyses différentes faites pour un schéma de test

La figure 9.4 présente un exemple avec une spécification $Spec$ où les états 1 et 2 sont des états stables. Soit le groupe G constitué des appels de méthode a et b, si on définit un schéma de test st tel que $st \hat{=} G$, nous obtenons $TA_{S'}(st) =$

– $0 \xrightarrow{G} 1$

Deux objectifs de test correspondent à ce schéma de test :

– $ot_1 \hat{=} a$
– $ot_2 \hat{=} b$

L'objectif de test ot_1 permet d'obtenir le test abstrait ta_1 :

– $0 \xrightarrow{a} 1$

L'objectif de test ot_2 permet d'obtenir le test abstrait ta_2 :

– $0 \xrightarrow{a} 1 \xrightarrow{b} 2$

Dans ce cas les objectifs de test ot_1 et ot_2 associés à st couvrent plus de transitions que st en utilisant la fonction $TA_{S'}$.

Cet exemple illustre l'inclusion de la couverture obtenue en utilisant l'abstraction de la spécification dans celle obtenue sur la spécification via les objectifs de test. Cela est dû au fait qu'en traitant les schémas de test au niveau d'une abstraction de la spécification, en conservant les conditions exprimées dans la partie 8.4 afin d'obtenir un ensemble fini de tests abstraits, nous perdons de l'information vis-à-vis du traitement fait en utilisant les objectifs de test sur la spécification.

9.4 Conclusion

Au cours de ce chapitre nous avons défini une couverture a priori de la spécification, par les tests abstraits, basée sur les schémas de test et sur l'utilisation d'une abstraction de la spécification. Cependant nous n'avons considéré que le cas où un schéma de test est un chemin (en excluant des constructeurs comme la négation et la co-région). Il reste donc un large sous-ensemble des instructions des schémas de test à considérer dans le cadre de la couverture a priori par les schémas de test.

Dans ce chapitre nous nous sommes intéressés uniquement à l'aspect théorique de cette approche. Le chapitre suivant donne une validation pratique de ce travail en reprenant l'étude de cas d'ascenseurs présentée dans le chapitre 6.

Chapitre 10

Un outil de mesure de couverture

CoPAS est un outil qui permet d'obtenir une mesure de couverture des schémas de tests et des tests abstraits suivant les critères de couverture exposés dans cette troisième partie du mémoire. Cet outil permet aussi de générer les tests abstraits associés à ces schémas de test.

La section 10.1 introduit l'outil CoPAS et présente ses différentes utilisations en termes de couverture de test et de synthèse de test. Après cette présentation nous reprenons l'étude de cas du système d'ascenseurs du chapitre 6 pour illustrer l'utilisation de CoPAS dans deux contextes différents. La première utilisation de CoPAS dans la section 10.2 se fait à partir de schémas de test exprimés dans le chapitre 6 mais dans le cadre que nous avons défini dans les deux chapitres précédents. Au cours de cette section nous évaluons la couverture des schémas de test et nous procédons à la génération des tests abstraits associés en utilisant CoPAS. La deuxième expérimentation effectuée avec CoPAS, et correspondant à la section 10.3, consiste à comparer la couverture des schémas de test avec celle des tests abstraits générés par le couple d'outils TObiAS/TGV dans le chapitre 6. Au terme de ces deux expérimentations nous effectuons dans la section 10.4 un bilan sur les deux approches outillées : CoPAS et TObiAS/TGV.

10.1 CoPAS

CoPAS est l'acronyme de *COverage tool for test Pattern on Abstract Specification*. C'est le deuxième outil qui a été développé au cours de cette thèse. CoPAS implémente le critère de couverture associé aux tests abstraits défini au chapitre 8, ainsi que le critère de couverture associé aux schémas de test défini au chapitre 9.

L'implantation de ces critères de couverture permet à CoPAS d'offrir différentes fonctions suivant les entrées qui lui sont fournies. Celles-ci peuvent être des spécifications comportementale, des abstraction de spécification[1], des schémas de test ainsi que des tests abstraits :

- Une abstraction de la spécification et un schéma de test en entrée permettent de calculer la couverture de ce test abstrait sur cette abstraction de la spécification et ainsi d'apprécier la couverture finale des tests abstraits associés.
- Une spécification et un ensemble de tests abstraits en entrée permettent de calculer la couverture des branches de cet ensemble de tests abstraits vis-à-vis de la spécification.
- Une spécification, son abstraction et un ensemble de schéma de test permettent de définir la couverture de cet ensemble de schémas de test sur l'abstraction de la spécification. Mais aussi de calculer l'ensemble exhaustif des tests abstraits associés, ou un ensemble *réduit* de tests abstraits ayant la même couverture.
- Une spécification et son abstraction en entrée avec un ensemble de schémas de test et un ensemble de tests abstraits permet d'effectuer des comparaisons entre les couvertures respectives des schémas de test et des tests abstraits sur l'abstraction de la spécification et sur la spécification.

CoPAS permet de synthétiser un ensemble *réduit* de tests abstraits offrant une couverture de test équivalente que celle qu'offrirait un ensemble de tests abstraits exhaustif pour un schéma de test donné. CoPAS effectue un parcours de la spécification comportementale en profondeur d'abord pour définir la couverture d'un schéma de test. Une fois qu'un chemin satisfaisant est trouvé, CoPAS s'assure qu'au moins une des transitions du chemin n'a jamais été atteinte parmi les chemins satisfaisants déjà définis. Si c'est le cas le chemin est gardé pour produire les tests abstraits correspondants, sinon le chemin est abandonné. Il existe d'autres techniques semblables de réduction de l'ensemble de test nécessaire à la couverture suivant un critère donné [HG04].

[1]actuellement ces deux spécifications sont calculées de manière manuelle. Dans une version future l'abstraction de la spécification sera calculée sur la base des groupes et de la spécification comportementale.

Actuellement CoPAS possède une restriction forte en plus de celles exposées dans le chapitre 9 sur les schémas de test. Cette restriction supplémentaire porte sur la spécification et son abstraction en entrée, les deux spécifications doivent être isomorphes.

10.2 CoPAS : un hybride entre TObiAs et TGV

CoPAS peut se substituer à TObiAs et à TGV pour générer des tests abstraits à partir des schémas de test. Cette section présente de quelle façon cette substitution peut s'effectuer en reprenant des schémas de test de l'étude de cas du système d'ascenseurs du chapitre 6 pour illustrer cette utilisation.

10.2.1 Une substitution à TObiAs et TGV

CoPAS représente une nouvelle voie pour générer des tests abstraits à partir de schémas de test. L'autre voie vue dans cette thèse est l'utilisation du couple d'outils TObiAs et TGV. TObiAs génère des objectifs de test à partir de schémas de test puis TGV génère des tests abstraits à partir de ces objectifs de test et de la spécification comportementale du système observé.

Contrairement au couple TObiAs/TGV, CoPAS ne manipule pas d'objectifs de test à proprement parler. Il utilise le fait d'avoir deux niveaux de description de la spécification et un lien entre ces deux niveaux pour générer, en deux temps, des tests abstraits sur la spécification comportementale à partir des schémas de test. Dans un premier temps CoPAS utilise les schémas de test comme des objectifs de test sur l'abstraction de la spécification pour génèrer des tests abstraits sur cette abstraction. Cette génération respecte les contraintes spécifiées dans le chapitre 8 vis-à-vis de la relation entre les objectifs de test et les tests abstraits. Dans un second temps CoPAS exploite la relation d'abstraction liant la spécification comportementale et son abstraction. Il crée ainsi un test abstrait sur la spécification comportementale pour chaque test abstrait exprimé sur l'abstraction de la spécification.

CoPAS souffre d'une lacune par rapport à TGV en ce qui concerne la prise en compte de la spécification comportementale. L'utilisation de CoPAS nécessite que celui-ci ait une connaissance complète de la spécification pour pouvoir travailler.

Alors que TGV ne travaille que sur l'état courant et une fonction qui en chaque état lui permet de calculer les états suivants. Il en résulte que CoPAS ne peut pas gérer des spécifications de tailles importantes contrairement à TGV qui peut travailler sur des spécifications de taille infinie.

10.2.2 Etude de cas du système d'ascenseurs avec CoPAS

Le tableau 10.1 représente trois des cinq schémas de test définis dans l'étude de cas du système d'ascenseurs. Nous ne considérons pas les schémas de test 3 et 5. Le schéma de test 3 nécessitait des réajustements et le schéma de test 5 comportait une négation qui le rendait incompatible avec les schémas de test acceptés par CoPAS.

1	* !p.ETAT_AS ; * !p.renouveler_affichage
2	u !u.MOUVEMENT_UT^2..3
4	u !u.MOUVEMENT_UT ; u !*.MOUVEMENT_AS ; u !u.MA_POSITION

TAB. 10.1 – Les schémas de test traités avec TObiAs.

Nous avons utilisé CoPAS avec ces trois schémas de test à deux reprises. Une première fois en générant l'ensemble exhaustif des tests abstraits (tableau 10.2), et une deuxième fois en générant l'ensemble réduit des tests abstraits (tableau 10.3).

Les tableaux 10.2 et 10.3 montrent que dans les deux cas (ensemble de tests abstraits exhaustif et ensemble de tests abstraits réduit) le taux de couverture obtenu est le même et les transitions couvertes correspondent exactement aux transitions définies pour les schémas de test correspondants. Dans le cas de l'ensemble de tests abstraits réduit, chaque test abstrait appartenant à cet ensemble permet de couvrir au moins une transition qui n'est pas couverte par le reste de l'ensemble de tests abstraits.

10.2.3 Analyse des résultats

CoPAS peut être utilisé de façon à se passer de TObiAs et de TGV. En effet CoPAS travaille avec les schémas de test sur l'abstraction de la spécification de la même façon que TGV travaille avec les objectifs de test sur la spécification.

		tests abstraits exhaustifs		
	Taux de couverture du schéma de test sur S'	Nombre de tests produits	Taux de couverture des tests sur S	Transitions communes au schéma de test
1	18%	4	18%	100%
2	87%	592	87%	100%
4	82%	1728	82%	100%

TAB. 10.2 – Comparaison des couvertures de test avec CoPAS (1)

		tests abstraits réduits		
	Taux de couverture du schéma de test sur S'	Nombre de tests produits	Taux de couverture des tests sur S	Transitions communes au schéma de test
1	18%	4	18%	100%
2	87%	38	87%	100%
4	82%	22	82%	100%

TAB. 10.3 – Comparaison des couvertures de test avec CoPAS (2)

A cela près que CoPAS explore tous les chemins possibles[2] depuis l'état initial pour trouver des chemins correspondant à un schéma de test et finir dans un état stable. Cela correspond à notre critère pour associer un objectif de test à un test abstrait (définition 12). Nous sommes assurés de la terminaison de ce processus de recherche exhaustif car notre spécification initiale est finie et possède des états stables, son abstraction a donc les mêmes propriétés (section 9.2).

D'après nos critères, dont l'isomorphisme de la spécification et de son abstraction, à chaque chemin de l'abstraction de la spécification couvert par un schéma de test nous pouvons associer un test abstrait ayant un chemin correspondant sur la spécification. Avec CoPAS, comme nous avons la spécification et son abstraction, lors de la recherche de la couverture d'un schéma de test nous pouvons générer un test abstrait par chemin correspondant au schéma de test. Cela permet d'obtenir l'ensemble exhaustif des tests abstraits correspondant à ce

[2]Les chemins correspondent aux trois conditions exprimées dans la partie 8.4.

schéma de test.

CoPAS peut générer un test abstrait pour chaque chemin de la spécification correspondant au schéma de test considéré. L'utilisation de CoPAS avec nos trois schémas de test a généré un ensemble de tests abstraits correspondant à $InvLab^s_A[TA_{S'}()]$ où $TA_{S'}$ est appliqué à chaque schéma de test. La couverture obtenue avec chaque schéma de test et chaque ensemble de tests abstraits associés sont bien les mêmes, le critère de couverture présenté dans le chapitre 9 est bien appliqué.

Nous pouvons toutefois remarquer que le nombre de tests abstraits peut alors devenir très important, voire sans doute trop important, notamment pour le schéma de test 4 qui génère 1728 tests abstraits, si nous voulons exécuter chaque test abstrait produit. En utilisant une méthode de réduction de suite de test, nous pouvons générer un nombre bien plus réduit de tests abstraits. En effet notre critère de couverture nous permet de ne couvrir qu'une fois chaque transition, cela nous permet de réduire le nombre de tests abstraits nécessaire en ne générant pas les tests abstraits redondants du point de vue des transitions couvertes.

Nous avons apporté les modifications à CoPAS pour qu'il puisse générer un ensemble réduit de tests abstraits suivant son algorithme de recherche de chemin et avec comme objectif de couvrir les transitions au moins une fois. Pour tout test abstrait produit, associé à un schéma de test, il existe une transition telle que cette transition n'appartient à aucun autre test abstrait produit associé à ce schéma de test. CoPAS permet ainsi de générer 4 tests abstraits pour le schéma de test 1, 38 pour le schéma de test 2 et seulement 22 pour le schéma de test 4. Chacun de ces ensembles de tests abstraits obtient la même couverture que leur schéma de test correspondant.

CoPAS permet de générer des tests abstraits pour des schémas de test donnés et de valider notre mesure de couverture. Cela sous réserve que la spécification corresponde aux critères que nous avons mis en place pour obtenir cette couverture de test.

10.3 Utilisation de CoPAS avec TObiAs et TGV

Lors de l'étude de cas du système d'ascenseurs du chapitre 6, 608 tests abstraits associés à 5 schémas de test ont été produits. Dans cette deuxième partie de l'étude, nous reprenons les schémas de test 1, 2 et 4. Pour chacun de ces schémas de test nous utilisons CoPAS pour observer les différences qui apparaissent entre la couverture que nous proposons dans le chapitre 9 pour les schémas de test et la couverture obtenue pour les tests abstraits associés obtenus par l'utilisation des outils TObiAs et TGV. Cette étude est motivée par le fait que TObiAs et TGV n'ont pas le comportement décrit dans les chapitres 8 et 9 vis-à-vis des relations existantes entre les différents niveaux d'abstraction et des restrictions prises. Nous pouvons ainsi étudier les différences entre les couvertures obtenues.

10.3.1 Les résultats obtenus

Nous étudions pour chaque schéma de test et leur ensemble de tests abstraits associés obtenu avec les outils TObiAs et TGV leurs différents taux de couverture ainsi que les transitions couvertes (ou qui devraient être couvertes dans le cas des schémas de test). Comme nous avons une abstraction de la spécification isomorphe à la spécification, cette abstraction comporte donc 201 transitions.

Le schéma de test 1 :

Ce schéma de test a une capacité de couverture de l'abstraction de la spécification de 18%, soit 38 transitions. Nous avons produit 128 tests abstraits pour les 128 objectifs de test correspondant à ce schéma. Ces tests abstraits couvrent pour leur part 50 transitions de la spécification, soit 25% de celle-ci. Il y a donc une large différence de couverture entre le schéma de test et les tests abstraits qui lui sont associés. En effet il n'y a que 11 transitions qui soient couvertes à la fois par le schéma de test et les tests abstraits alors que 27 transitions ne le sont que par le schéma de test et 39 seulement par les tests abstraits.

Le schéma de test 2 :

L'outil CoPAS permet de déterminer que ce schéma de test a une capacité de couverture de l'abstraction de la spécification de 87%, soit 176 transitions. Nous avions 252 objectifs de test associés à ce schéma de test, soit 252 tests abstraits correspondant. Ces tests abstraits couvrent 130 transitions de la spécification, soit 64,7% de celle ci. En mettant en correspondance les couvertures des tests abstraits et du schéma de test 2, on voit que toutes les transitions couvertes par les tests abstraits sont aussi couvertes par le schéma de test. 47 transitions sont couvertes uniquement par le schéma de test.

Le schéma de test 4 :

Avec nos critères de couverture, ce schéma de test couvre 165 transitions soit 82% de la spécification abstraite. Les 48 tests abstraits correspondant à ce schéma de test, générés par TGV couvrent 101 transitions, soit 50% de la spécification. Toutes ces transitions sont aussi couvertes par le schéma de test, 64 transitions ne sont couvertes que par le schéma de test.

Le tableau 10.4 résume les différents résultats obtenus avec l'outil de couverture CoPAS pour les taux de couverture des trois schémas de tests et de leurs tests abstraits associés sur la spécification comportementale.

	Taux de couverture du schéma de test	Taux de couverture des tests abstraits	Intersection des transitions couvertes	% couvert seulement par le schéma de test	% couvert seulement par les tests abstraits
1	18%	25%	28.9%	13.4%	19.4%
2	87%	65%	73.9%	23.4%	0%
4	82%	50%	61.2%	31%	0%

TAB. 10.4 – Taux de couverture des schémas de test et des tests abstraits obtenus avec TObiAS et TGV sur la spécification

10.3.2 Analyse des résultats

Les taux de couverture entre tests abstraits et schémas de test divergent nettement. Le critère de couverture mis en place au chapitre 9 ne peut pas s'appliquer à la génération faite par les outils TObiAs et TGV. Cette divergence est le fait de la génération des objectifs de test assurée par TObiAs d'une part, et de l'utilisation faite de TGV d'autre part.

La figure 10.1 correspond aux couvertures obtenues pour le schéma de test 1 et les tests abstraits qui lui sont associés. Cette figure montre deux cas de figure correspondant aux divergences observées. Ces divergences s'expriment soit comme le cas où le schéma de test couvre des transitions qui ne sont pas couvertes par les tests abstraits, soit comme le cas où les tests abstraits couvrent des transitions qui ne le sont pas par le schéma de test.

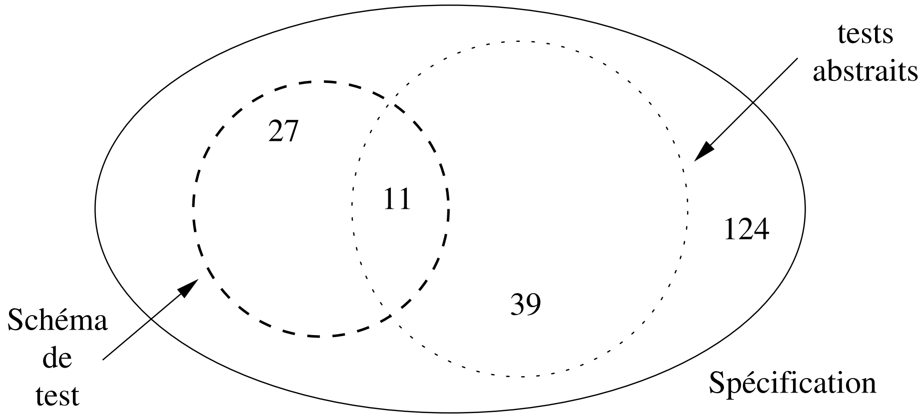


FIG. 10.1 – Nombre de transitions couvertes par le schéma de test 1 et par les tests abstraits qui lui sont associés.

Ces deux cas de figure propres à l'utilisation de TGV et TOBIAS s'expliquent de la façon suivante :

- Le fait que des transitions couvertes par le schéma de test ne le soient pas par les tests abstraits vient du fait que nous n'avons pas la liste exhaustive des tests abstraits associés à ce schéma. Cela est dû au fait que lors de notre utilisation de TGV nous n'obtenons qu'un test abstrait pour un objectif de test. Donc nous ne pouvons assurer un recouvrement total de la couverture du schéma de test par celle des tests abstraits dans ce cas là. Nous retrouvons ce cas de figure pour les schémas de test 1, 2 et 4.
  L'utilisation d'options d'exécution de TGV permet d'obtenir différents tests abstraits pour un même objectif de test, mais ces utilisations nécessitent une manipulation de la part de l'ingénieur de test là où nous cherchons un système automatisé.
- Dans le cas du schéma de test 1, des transitions sont couvertes par les tests abstraits mais pas par le schéma de test. Cette distinction vient d'une restriction que nous apportons à la couverture des schémas de test qui n'apparaît pas dans l'algorithme de déploiement utilisé par TObiAs.

La condition 1 exprimée dans le chapitre 8 apporte comme restriction au niveau de la couverture des schémas de test que le fait qu'un label de l'abstraction de la spécification correspond à un label du schéma de test à identifier, c'est celui ci qui est pris en compte. TObiAs n'effectue pas cette distinction relative à la spécification et effectue un déploiement "aveugle" du schéma de test. Nous illustrons cela avec le schéma de test 1 et le groupe **ETAT_AS**. Dans l'absolu ce groupe se déploie en quatre valeurs : *descendre(a1), descendre(a2), monter(a1), monter(a2).* C'est le déploiement qu'effectue TObiAs. CoPAS interprète le schéma de test en fonction de l'abstraction de la spécification, et non pas en fonction des méthodes associées aux groupes. Dans le cas de notre spécification et de ce schéma de test, depuis l'état initial par quelque chemin que ce soit, la première occurrence de **ETAT_AS** rencontrée correspond, soit à *monter(a1)*, soit à *monter(a2)*. Le taux de couverture calculé par CoPAS pour le schéma de test 1 ne prend en compte que ces deux appels de méthode.

L'exemple de la partie 9.3.3 illustre bien ce cas de figure où l'interprétation des shémas de test au niveau de l'abstraction peut nous faire perdre des informations par rapport à leur interprétation via les objectifs de test tel que cela est fait avec TObiAs et TGV.

10.3.3 Analyse de la démarche outillée TObiAs - TGV - CoPAS

TObiAs permet de structurer une campagne de test et de décrire de nombreux tests avec seulement quelques schémas de test, et cela de manière rapide, puis de générer automatiquement tout un ensemble d'objectifs de tests associés à ces schémas de test. TGV permet d'obtenir rapidement et de manière automatisée des tests abstraits correspondant aux objectifs de test et à la spécification comportementale.

Cette étude avec l'utilisation de CoPAS avait pour but de vérifier en quels points le système de couverture proposé dans le chapitre 9 divergeait de la couverture qu'assure l'association de ces deux outils. Les résultats obtenus pour les schémas de tests 1, 2 et 4 avec notre suite d'outils nous permettent de tirer plusieurs conclusions.

Malgré les avantages du point de vue du gain de temps de conception d'une campagne de test et de l'automatisation de la génération des cas de test que fournissent TObiAs et TGV, notre calcul de couverture a priori sur les schémas de test n'est pas applicable avec ces outils du fait des restrictions importantes que nous imposons. En effet les résultats obtenus avec ces outils ne nous permettent jamais de prédire la couverture des tests abstraits puisque celle-ci peut couvrir aussi bien 25% que 65% des transitions couvertes par le schéma de test associé. Les tests abstraits peuvent même couvrir des éléments de la spécification qui ne sont pas compris dans la couverture du schéma de test correspondant.

Avec l'utilisation de CoPAS, la notion de schéma de test est plus restrictive que dans TObiAs d'un point de vue syntaxique avec moins de constructions autorisées, mais aussi d'un point de vue contextuel. Dans CoPAS les schémas de test sont vus comme des objectifs de test et l'interprétation faite pour calculer la couverture s'effectue au niveau de l'abstraction de la spécification tandis que dans TObiAs toutes les abstractions sont "dépliées" sans tenir compte de la spécification. TObiAs permet ainsi d'associer plus de comportements à un seul schéma de test. TGV est un outil puissant qui offre de nombreuses configurations d'exécution. Dans cette expérimentation nous avons choisi une forme simple et n'avons généré qu'un seul test abstrait par objectif de test. Cette démarche ne peut pas nous assurer un ensemble exhaustif de tests abstraits tel que CoPAS le considère dans ses hypothèses de couverture. Une meilleure utilisation de TGV nous aurait permis de multiplier le nombre de tests abstraits différents associés à un objectif de test avec de multiples exécutions de TGV. Mais cette démarche ne serait plus automatisée comme précédemment.

10.4 Conclusion

CoPAS est la mise en œuvre des concepts présentés dans les chapitres 8 et 9. A partir d'un schéma de test et d'une abstraction de la spécification considérée, CoPAS peut calculer une couverture à priori des tests abstraits associés à ce schéma de test. CoPAS offre aussi la possibilité de calculer les tests abstraits correspondants et de les générer, cela fait que CoPAS peut se substituer au couple d'outils TObiAs/TGV lorsque la spécification considérée est finie.

Notre approche comporte néanmoins des restrictions d'ordre syntaxique sur les schémas de test et au niveau de leur interprétation faite sur l'abstraction de la spécification. Ces restrictions font que notre notion de couverture a priori des tests abstraits à partir des schémas de test n'est pas utilisable avec des outils tels que TObiAs et TGV de façon automatique. D'un côté TObiAs peut générer des objectifs de test invalides parce qu'il ne prend pas en comtpe la spécification comportementale et d'autre part TGV ne génère pas automatiquement des tests abstraits couvrant toutes les transitions que le schéma de test peut couvrir.

Dans le cas où l'ingénieur de test possède une spécification finie et qu'il définit des groupes de méthodes permettant d'avoir une abstraction de la spécification tels que cette spécification et son abstraction soient isomorphes, on peut dire que CoPAS est plus performant que TObiAs et TGV. CoPAS permet pour un ensemble de schémas de test d'obtenir des sorties que ne proposent pas TObiAs et TGV :

- une prédiction de la couverture de la suite de test ;
- la production d'une suite de test réduite assurant une couverture équivalente aux prédictions.

Une suite de test restreinte peut être nécessaire pour certaines applications où l'exécution d'un test peut coûter cher. Nous pouvons prendre l'exemple de tests de cartes à puce qui amènent à la destruction de celle-ci, réduire le nombre de tests semblables est alors intéressant.

CoPAS reste tout de même limité du fait que les schémas de test sont plus contraints que dans TObiAs et que contrairement à TGV, il ne peut pas travailler sur des spécifications comportementales infinies ou dans le cas où l'isomorphisme n'est pas respecté entre la spécification et son abstraction. Cette limite sur la taille des spécifications, fait de CoPAS un outil plus adapté au test de parties bien déterminées d'un logiciel que de l'ensemble d'un système au comportement complexe.

Chapitre 11

Conclusion et perspectives

Conclusion

Dans cette thèse nous avons traité de la synthèse des tests dans le cadre du test de conformité, ainsi que de la mise en place d'une mesure de couverture a priori. Ce travail a été motivé par des besoins industriels tout en constituant un vrai travail de recherche académique dans le cadre du projet RNTL COTE. Le besoin exprimé se rapporte principalement à un gain de temps dans la phase d'élaboration des tests afin de réduire le temps et le coût de la phase de test. Pour optimiser cette phase de développement nous avons travaillé sur deux axes : la création d'un niveau d'abstraction plus élevé et la mise en place d'outils de synthèse et de couverture de test à ce niveau d'abstraction.

Il existe différents niveaux d'abstraction reconnus dans le domaine du test de logiciels : test exécutable, test abstrait et objectif de test. La similarité de nombreux objectifs de test pour une application donnée nous a laissé voir qu'une nouvelle forme d'abstraction des tests pouvait être bénéfique à l'ingénieur de test. Nous avons ainsi défini les schémas de test en effectuant des abstractions sur les éléments constitutifs des tests permettant ainsi de factoriser leurs expressions. Les abstractions portent sur les paramètres des méthodes, les méthodes elles-mêmes ainsi que sur les instances des méthodes et la répétition des appels de méthode.

La définition des schémas de test nous a amené à les outiller pour permettre l'utilisation de cette notion dans la création d'une campagne de test. Nous avons donc créé l'outil TObiAs pour exprimer des schémas de test et générer de manière automatique les objectifs de test correspondants. Deux études menées dans le chapitre 6 nous ont permis d'apprécier l'apport des schémas de test ainsi que de

TObiAs dans la définition d'une campagne de test.

La définition de la notion de schéma de test nous a amené à porter notre réflexion sur un autre domaine du test de logiciel : la couverture de test. Notre but est d'offrir un ensemble complet d'outils à l'ingénieur de test afin qu'il n'ait à travailler qu'à un seul niveau d'abstraction, celui des schémas de test.

Notre travail, pour définir une notion de couverture au niveau des schémas de test [BP03, BP04] a consisté à définir formellement une notion de couverture de test basée sur la couverture des transitions pour les tests abstraits. Nous avons ensuite défini une notion de couverture identique pour les objectifs de test en s'appuyant sur celle des tests abstraits. Nous avons fait de même pour les schémas de test afin de produire une notion de couverture de test d'un haut niveau d'abstraction cohérente avec les notions de couverture habituellement utilisées. Dans un second temps nous avons redéfini une notion de couverture des schémas de test non plus basée sur les tests abstraits mais sur une abstraction de la spécification afin d'obtenir une valeur de couverture directement calculable sans avoir à créer les tests abstraits associés.

Tout comme pour la création des schémas de test, nous avons cherché à outiller cette notion de couverture des schémas de test. Nous avons créé l'outil CoPAS à ces fins. L'outil CoPAS permet d'une part de définir une couverture de la spécification par les schémas de test, mais il permet aussi par le procédé d'abstractions fait sur la spécification de produire directement les tests abstraits associés à un schéma de test sans passer par la génération d'objectifs de test.

Dans le chapitre 10 nous avons pu observer l'efficacité de CoPAS, mais nous avons aussi observé ses limitations face aux outils TObiAS et TGV sur deux points : le pouvoir d'expressivité des schémas de test pris en compte et la limitation à des spécifications comportementales finies et petites.

Perspectives

Le travail effectué offre des possibilités d'évolution sur différents axes. Une évolution outillée peut s'effectuer avec l'intégration de CoPAS dans TObiAs. Une autre évolution peut porter sur l'extension de l'expressivité des schémas de test pris en compte pour la notion de couverture de test avec CoPAS.

Intégration de CoPAS dans TObiAs

TObiAs offre une bonne interface utilisateur pour créer des campagnes de test et exprimer des schémas de test, CoPAS ne s'exécute qu'en mode texte et n'offre aucune possibilité d'édition. Actuellement CoPAS ne considère pas les abstractions telles que les groupes de la façon dont ils ont été défini dans le chapitre 4 et fonctionne par analogie entre la spécification comportementale et son abstraction qui lui sont donnés en entrée.

Une intégration de CoPAS dans TObiAs permettrait à CoPAS de bénéficier de l'interface de TObiAs pour définir et éditer les schémas de test ainsi que les abstractions portant sur les méthodes et leurs paramètres. Cette prise en compte des abstractions permettrait de calculer l'abstraction de la spécification automatiquement à partir de la spécification comportementale ou directement à partir des diagrammes d'états-transitions de chaque instance d'une campagne de test en calculant les fonctions d'abstractions.

Cette intégration de CoPAS dans TObiAs signifirait que dans le cadre de spécifications finies, on utiliserait le moteur de CoPAS à la place de celui de TObiAs pour générer des cas de test. D'autres évolutions propres à TObiAs peuvent aussi être mises en place pour améliorer son fonctionnement actuel comme nous avons pu le voir dans le chapitre 5 avec notamment une meilleur prise en compte de la spécification exprimée en UML en entrée, ou en perfectionnant la génération de tests abstraits.

Extension de l'expressivité des schémas de test pour la couverture de test

Dans les travaux que nous avons menés sur la couverture des schémas de test, ceux-ci (ainsi que les objectifs de test et tests abstraits) sont réduits à de simples séquences d'appels, ceci limite largement le pouvoir d'expressivité des schémas de test puisqu'à la base ceux-ci permettent d'exprimer les notions d'alternative, de séquence stricte, de co-région, de boucle ou encore de négation.

Intuitivement toutes ces notions semblent compatibles avec la notion de couverture mise en place. Pour procéder à cette mise en place nous pouvons procéder de deux façons : soit en découpant les schémas de test, soit en redéfinissant les notions de couverture du chapitres 8 et en s'assurant que la démonstration du chapitre 9 reste valide dans ce nouveau cadre.

En “découpant” un schéma de test utilisant l’alternative, la co-région et/ou la boucle nous pouvons obtenir plusieurs schémas de test ne comportant que des séquences qui peuvent ainsi nous permettre d’effectuer notre couverture. La figure 11.1 montre le découpage d’un schéma de test en quatre schémas de test exprimés uniquement à partir de séquences.

$$\boxed{(a|b); c^{\wedge 1..2} \rightarrow a; c \wedge b; c \wedge a; c; c \wedge b; c; c}$$

FIG. 11.1 – Exemple de “découpage” d’un schéma de test

Les séquences strictes et la négation sont assez semblables puisque la séquence stricte implique des négations sur tous les appels non définis dans le schéma de test au niveau de la séquence stricte. Ces deux notions qui peuvent donc être vues comme une seule, nécessitent de reprendre la formalisation des tests abstraits, objectifs de test et schémas de test pour y être introduites. Ensuite il faut s’assurer que la démonstration du chapitre 9 reste valable ou qu’elle peut être adaptée.

Les travaux présentés dans cette thèse définissent de nouvelles méthodes pour exprimer des tests et pour exprimer une couverture des tests a priori. Des outils reprenant ces méthodes ont été développés. Cet ensemble de méthodes et d’outils devrait permettre de réduire le temps et les efforts des ingénieurs de test lors de la création d’une campagne de test.

Bibliographie

[ABM97a] L. Van Aertryck, M. Benveniste, and D. Le Metayer. CASTING : une méthode formelle de génération de cas de tests. In *AFADL : Approches formelles dans l'assistance au développement de logiciel*, pages 99–112, Toulouse, France, May 1997.

[ABM97b] L. Van Aertryck, M. Benveniste, and D. Le Métayer. Casting : A formally based software test generation method. In *Proceedings of the 1st International Conference on Formal Engineering Methods : ICFEM'97*, page 101, Hiroshima, Japan, November 1997. IEEE Computer Society.

[Abr77] J.-R. Abrial. Manuel du langage z (z/13). Technical report, Electricité de France, 1977.

[Abr87] S. Abramsky. Observational equivalence as a testing equivalence. *Theoretical Computer Science*, 53(3), 1987.

[Abr96] J.-R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, August, 1996.

[ACV93] J. Alilovic-Curgus and S. T. Vuong. A metric based theory of test selection and coverage. In *Proceedings of the IFIP TC6/WG6.1 Thirteenth International Symposium on Protocol Specification, Testing and Verification XIII*, pages 289–304, Liege, Belgium, 1993. North-Holland.

[ADKM02] A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov. Evaluating the small scope hypothesis, September 2002.

[Aer98] L. Van Aertryck. *Une méthode et un outil pour l'aide à la génération de jeux de tests de logiciels*. Thèse de doctorat, Université de Rennes 1, 1998.

[AJ03] L. Van Aertryck and T. Jensen. UML-CASTING : Test synthesis from UML models using constraint resolution. In *AFADL : Ap-*

proches formelles dans l'assistance au développement de logiciel, Rennes, France, January 2003.

[BBF+01] A. Belinfante, E. Brinksma, J. Feenstra, J. Tretmans, and R. G. de Vries. Côte de Resyste – automatic model-based testing of communication protocols. In *Mobile Communications in Perspective – 7th Annual CTIT Workshop*, pages 49–51, University of Twente, The Netherlands, Februray 2001.

[BDM97] P. Behm, P. Desforges, and F. Mejia. Application de la méthode B dans l'industrie ferroviaire. In OFTA, editor, *Application des techniques formelles au logiciel*, pages 59–88. Observatoire Français des Techniques Avancéees & Lavoisier TEC & DOC, 1997.

[Beg02] S. Beghdadi. Exploitation d'objectifs de test par ajout d'un langage de contrainte. Dea d'informatique : systèmes et communications, Université Joseph Fourier, June 2002.

[Bei90] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 2nd edition edition, 1990.

[BGM91] G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications : a theory and a tool. *Softw. Eng. J.*, 6(6) :387–405, November 1991.

[BGMR03] J. Bézivin, S. Gérard, P.-A. Muller, and L. Rioux. MDA components : Challenges and opportunities. In *Metamodelling for MDA, York*, York, England, November 2003.

[BH89] F. Belina and D. Hogrefe. The CCITT-specification and description language SDL. *Comput. Netw. ISDN Syst.*, 16(4) :311–341, 1989.

[BKM02] C. Boyapati, S. Khurshid, and D. Marinov. Korat : Automated testing based on java predicates. In Phyllis G. Frankl, editor, *Proceedings of the ACM SIGSOFT 2002 International Symposium on Software Testing and Analysis (ISSTA-02)*, volume 27, 4 of *SOFTWARE ENGINEERING NOTES*, pages 123–133, Roma, Italy, july 2002. ACM Press.

[BL03] F. Bouquet and B. Legeard. Reification of executable test scripts in formal specification-based test generation : The java card transaction mechanism case study. In Araki K., Gnesi S., and Mandrioli D., editors, *Formal Methods Europe, FME 2003*, volume

2805 of *Lecture Notes in Computer Science*, pages 778–795. Springer, 2003.

[BLP00] F. Bouquet, B. Legeard, and F. Peureux. Constraint logic programming with sets for animation of b formal specifications. In *(C)LPSE : (Constraint) Logic Programming and Software Engineering*, London, Great Britaine, July 2000.

[BLP02] F. Bouquet, B. Legeard, and F. Peureux. Clps-b : a solver for b. In *TACAS : Tool and Algoritjms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 188–204, Grenoble, France, April 2002. Springer.

[BLPP00] F. Bouquet, B. Legeard, F. Peureux, and L. Py. Un système de résolution de contraintes ensemblistes pour l'évaluation de spécifications b. In *JFPLC : Journées Francophones de Programmation en Logique et Contraintes*, Marseille, France, June 2000. Hermès.

[BMdB+01a] P. Bontron, O. Maury, L. du Bousquet, Y. Ledru, C. Oriat, and M.-L. Potet. Tobias : un environnement pour la création d'objectifs de tests à partir de schémas de tests. In J.C. Rault, editor, *In International Conference on Software and Systems Engineering and their Applications (ICSSEA)*, Paris, France, December 2001. CNAM Paris.

[BMdB+01b] P. Bontron, O. Maury, L. du Bousquet, Y. Ledru, C. Oriat, and M.-L. Potet. Cote document o-tela. Rapport sous-projet 3 - activité 3.1, Projet COTE, Juin 2001.

[BMdB+02] P. Bontron, O. Maury, L. du Bousquet, Y. Ledru, C. Oriat, M.-L. Potet, and J. Vassy. Outils de synthèse des données. Rapport sous-projet 4 - activité 4.5, Projet COTE, Novembre 2002.

[Bou02] H. Boudjedri. Exploitation d'objectifs de test avec prise en compte des diagrammes d'objets. Dea communication et coopération dans les systèmes à agents, Université Joseph Fourier, June 2002.

[BP03] P. Bontron and M.-L. Potet. Stratégie de couverture de test à haut niveau d'abstraction. In *AFADL : Approches formelles dans l'assistance au développement de logiciel*, Rennes, France, January 2003.

[BP04] P. Bontron and M.-L. Potet. Stratégie de couverture de test à haut niveau d'abstraction. *Technique et Science Informatiques, RSTI*, 23(7/2004) :905–928, (à paraître) 2004.

[BR02] L. Burdy and A. Requet. Jack : Java applet correctness kit. In *4th Gemplus Developer Conference*, Singapore, November 2002.

[BY98] K. Burr and W. Young. Combinatorial test techniques : Table-based automation, test generation and code coverage. In *Proc. of the Intl. Conf. on Software Testing Analysis & Review*, San Diego, 1998.

[CDFP97] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The aetg system : An approach to testing based on combinatorial design. *IEEE Trans. Softw. Eng.*, 23(7) :437–444, July 1997.

[Cha97] O. Charles. *Application des hypothèses de test à une définition de la couverture*. Thèse de doctorat, Université Henri Poincaré - Nancy 1, October 1997.

[Cho78] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3) :178–187, 1978.

[CL02] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing : The jml and junit way. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 231–255. Springer-Verlag, June 2002.

[Com02] F. Combret. Dossier de spécification d'un service bancaire. Rapport sous-projet 7 - activité 7.1, Projet COTE, March 2002.

[COT] Project COTE. Projet COTE : http ://www.irisa.fr/cote/.

[dBLM+04] L. du Bousquet, Y. Ledru, O. Maury, C. Oriat, and J.-L. Lanet. A case study in jml-based software validation. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, Linz, Austria, september 2004.

[dBMJ01] L. du Bousquet, H. Martin, and J.-M. Jézéquel. Conformance testing from UML specification, experience report. In Andy Evans, Robert France, Ana Moreira, and Bernhard Rumpe, editors, *Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists. Workshop of the pUML-Group held together with the UML 2001 October 1st, 2001 in Toronto, Canada*, volume P-7 of *LNI*, pages 43–56, Toronto, Canada, 2001. German Informatics Society.

[DF93] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes In Computer Science*, pages 268–284. Springer, 1993.

[DN81] J. W. Duran and S. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering, Vol. SE-10, No.*, pages 438–444, July 1981.

[Fer89] J.C. Fernandez. Aldebaran : A tool for verification of communicating processes. Technical report spectre c14, LGJ-IMAG Grenoble, 1989.

[FJJ+96] J.-C. Fernandez, C. Jard, T. Jéron, L. Nedelka, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 348–359, New Brunswick, NJ, USA, 1996. LNCS Springer Verlag.

[FL98] J. Fitzgerald and P. G. Larsen. *Modelling systems : practical tools and techniques in software development*. Cambridge University Press, 1998.

[FPdV01] J. Feenstra, L. Heerink (Philips), and R. G. de Vries. Specification based formal testing : The easylink case study. In F. Karelse, editor, *2nd PROGRESS Workshop on Embedded Systems*, pages 73–82, Veldhoven, The Netherlands, Oct 2001. STW Technology Foundation, Utrecht, The Netherlands.

[Gau93] M.C. Gaudel. Test selection based on adt specifications. In Gregor von Bochmann, Rachida Dssouli, and Anindya Das, editors, *Protocol Test Systems, V, Proceedings of the IFIP TC6/WG6.1 Fifth International Workshop on Protocol Test Systems*, volume C-11 of *IFIP Transactions*, pages 31–40, Montreal, Quebec, Canada, 28-30 September 1993. North-Holland.

[GCR96] R. Groz, O. Charles, and J. Renévot. Relating conformance test coverage to formal specifications. In *IFIP TC6/ 6.1 international conference on formal description techniques IX/protocol specification, testing and verification XVI on Formal description techniques*

IX : theory, application and tools, pages 195–210, Kaiserslautern, Germany, 1996. Chapman & Hall, Ltd.

[Gro00] The VDM Tool Group. *VDM-SL Toolbox User Manual*. IFAD, October 2000. ftp ://ftp.ifad.dk/pub/vdmtools/doc/userman_letter.pdf.

[Gue01] A. Le Guennec. *Génie Logiciel et Méthodes Formelles avec UML Spécification, Validation et Génération de tests*. Thèse de doctorat, Université de Rennes 1, 29 juin 2001.

[Ham94] R. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of software Engineering*, pages 970–978, Wiley, 1994.

[Har87] D. Harel. Statecharts : A visual formalism for complex systems. *Science of Computer Programming*, 8(3) :231–274, June 1987.

[HG04] Ma. P.E. Heimdahl and D. George. Test-suite reduction for model based tests : Effect on test quality and implications for testing. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, Linz, Austria, september 2004.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[HT90] R. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering, Vol. 16, No. 12*, pages 1402–1411, December 1990.

[IEE90] IEEE. IEEE Standard Glossary of Software Engineering Terminology. Technical Report IEEE Std 610.12-1990, Institute of Electrical and Electronic Engineers, New York, December 1990.

[Inc92] RTCA Inc. *Software Considerations in Airborne Systems and equipment certification*. DO-178B/ED-12B, december 1992.

[ISO89] ISO. Open systems interconnection, lotos - a formal description technique based on the temporal ordering of observational behaviour. Technical Report 8807, International Organization for Standardization, Geneva, Switzerland, 1989.

[ISO91] ISO. *Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework*. International Standard IS-9646, 1991.

[IT94] ITU-T. Message sequence chart (msc). Recommendation z.100, International Telecommunication Union, General Secretariat, Geneva, Switzerland, 1994.

[JHGP99] J.-M. Jézéquel, W.-M. Ho, A. Le Guennec, and F. Pennaneac'h. UMLAUT : an extendible UML transformation framework. In Robert J. Hall and Ernst Tyugu, editors, *Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99*, Florida, October 1999. IEEE.

[JJ02] C. Jard and T. Jéron. Tgv : theory, principles and algorithms. In C. V. Ramamoorthy, B. Krämer, and J.P. Tsai, editors, *Proceedings of the 2002 IDPT Conference - The Sixth World Conference on Integrated Design & Process Technology*, Pasadena, California, USA, June 2002.

[Jon90] C. B. Jones. *Systematic Software DEvelopment Using VDM (Second Edition).* Prentice-Hall, London, 1990.

[Kes03] M. Kessis. Test de conformité des programmes java. Dea en systèmes d'informations, Université Joseph Fourier, June 2003.

[Lan02a] J.-L. Lanet. Etude de cas bancaire : Jeux de test abstraits de l'aplication. Rapport sous-projet 7 - activité 7.3, Projet COTE, July 2002.

[Lan02b] J.-L. Lanet. Etude de cas bancaire : Modèle UML testable de l'application gemplus. Rapport sous-projet 7 - activité 7.2, Projet COTE, july 2002.

[LdBMB04] Y. Ledru, L. du Bousquet, O. Maury, and P. Bontron. Filtering tobias combinatorial test suites. In Michel Wermelinger and Tiziana Margaria, editors, *Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004*, volume 2984 of *Lecture Notes in Computer Science*, Barcelona, Spain, 2004. Springer.

[LP01] B. Legeard and F. Peureux. Génération de séquences de tests à partir d'une spécification b en plc ensembliste. In *AFADL : Approches formelles dans l'assistance au développement de logiciel*, pages 113–130, Nancy, France, June 2001.

[LPU02] B. Legeard, F. Peureux, and M. Utting. Automated Boundary Testing from Z and B. In L.-H. Eriksson and P. Lindsay, editors, *Formal Methods Europe, FME 2002*, volume 2391 of *Lecture Notes in Computer Science*, pages 21–40. Springer, July 2002.

[Lyn88] N. A. Lynch. I/o automata : A model for discrete event systems. In *22nd Conf. on Information Sciences and Systems*, pages 29–38, Princeton, NJ, USA, March 1988.

[Mar01] H. Martin. *Une méthodologie de génération automatique de suites de tests pour applets Java-Card*. Thèse de doctorat, Université de Lille 1, March 2001.

[ML02] O. Maury and Y. Ledru. Using tobias for the automatic generation of VDM test cases. In Andrei Voronkov, editor, *In Third VDM workshop (in conjunction with FME2002)*, Copenhagen, Danemark, 2002. LNCS 2392, Springer-Verlag.

[Mor00] P. Morel. *Une algorithmique efficace pour la génération automatique de tests de conformité*. Thèse de doctorat, Université de Rennes 1, February 2000.

[MV95] M. Mori and S. T. Vuong. On finite covering of infinite spaces for protocol test selection. In *Proceedings of the fourteenth of a series of annual meetings on Protocol specification, testing and verification XIV*, pages 237–251, Vancouver, Canada, 1995. Chapman & Hall, Ltd.

[Mye79] G. J. Myers. *The Art of Software Testing*. New York, Chichester (etc.), J. Wiley, 1979.

[NH84] R. De Nicolas and M. Henessy. Testing equivalences for processes. *Theoretical Computer Science*, 34 :83–133, 1984.

[OMG] OMG. MDA : http ://www.omg.org/mda/.

[OMG02a] OMG. UML 2.0 testing profile specification. Technical report, Object Management Group, 2002.

[OMG02b] OMG. Xml metadata interchange (xmi) specification. Technical report, Object Management Group, January 2002.

[OMG03] OMG. Omg unified modeling language specification. version 1.5, Object Management Group, March 2003.

[Ori02] C. Oriat. Jartege : a tool for random generation of unit tests for java classes. Rapport de recherche rr1069 lsr 19, LSR-IMAG, 2002.

[Pac03] C. Pachon. Une approche pour la génération automatique de tests de robustesse. In *Actes (CD-ROM) de la conférence MAJECSTIC'03 (MAnifestation des JEunes Chercheurs dans le domaine STIC)*, Marseille, October 2003.

[Pha94] M. Phalippou. *Relations d'implantations et Hypothèses de test sur les automates à entrées et sorties*. Thèse de doctorat, Université de Bordeaux, 1994.

[PJH+01] S. Pickin, C. Jard, T. Heuillard, J.-M. Jézéquel, and P. Desfray. A UML-integrated test description language for component testing. In Andy Evans, Robert France, Ana Moreira, and Bernhard Rumpe, editors, *Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists. Workshop of the pUML-Group held together with the UML 2001*, volume P-7 of *LNI*, pages 208–223, Toronto, Canada, October 2001. German Informatics Society.

[PLT00] L. Py, B. Legeard, and B. Tatibouët. Evaluation de spécifications formelles en programmation logique avec contraintes ensemblistes - application à l'animation de spécifications formelles. In *AFADL : Approches formelles dans l'assistance au développement de logiciel*, pages 21–35, Grenoble, France, January 2000.

[SLB05] M. Satpathy, M. Leuschel, and M. Butler. Protest : an automatic test environment for b specifications. In *Electronic Notes in Theoretical Computer Science*, volume 111, pages 113–136, 2005.

[TFWY95] P. Thévenod-Fosse, H. Waeselynck, and Y.CROUZET. Software statistical testing. In B. Randell, J.-C. Laprie, H. Kopetz, and B.Littlewood, editors, *Predictably dependable computing systems*, pages 253–272. Springer, 1995.

[Tre91] J. Tretmans. An overview of osi conformance testing. In Samson, editor, *Conformance Testen, in Handboek Telematica,*, volume II, pages 4400 1–19, 1991.

[Tre95] J. Tretmans. Testing labelled transition systems with inputs and outputs. In *8th International Workshop on Protocols Test Systems*, Evry, France, September 1995.

[VAC92] S. T. Vuong and J. Alilovic-Curgus. On test coverage metrics for communication protocols. In *Proceedings of the IFIP TC6/WG6.1 Fourth International Workshop on Protocol Test Systems IV*, pages 31–45, Leinschendam, the Netherlands, 1992. North-Holland.

[WK98] J. Warmer and A. Kleppe. *The Object Constraint Language*. Addison-Wesley, 1998.

Annexe A

Les schémas de test pour le service bancaire

Cet annexe présente les exigences, schémas de test et groupes se rapportant à l'étude de cas exposée dans le chapitre 6. La section A.1 liste les 40 exigences définies dans le dossier de spécification [Com02] qui correspond au cahier des charges détaillé. La section A.2 présente les groupes et schémas de test résultant des premiers efforts d'abstraction. Enfin la section A.3 présente les groupes et schémas de test correspondant à la deuxième passe pour réduire leur nombre.

La nomenclature utilisée pour dénommer les groupes et les schémas de test de la section A.2 sert à associer ces différents éléments aux exigences définies dans le cahier des charges et à un type d'exécution pour cette exigence. Ces exécutions sont celles définies dans le document [Lan02a].

A.1 Les exigences du cahier des charges

Les exigences :

1. L'application doit être testable.
2. L'utilisateur doit être authentifié par code secret.
3. L'authentification doit s'opérer avant la connexion au serveur.
4. Le mot de passe n'est pas modifiable par l'utilisateur.
5. Le nombre de saisies du mot de passe est illimité.

6. Toutes les transactions terminal/serveur s'opèrent par « secure messaging ».
7. L'application permet de consulter le solde des comptes de l'utilisateur.
8. Les comptes peuvent être domiciliés dans plusieurs banques.
9. Le solde peut être affiché en Dollar US, Franc français, Euro.
10. Le taux de conversion est fixe durant toute la durée d'une connexion.
11. Le solde est arrondi à deux chiffres après la virgule.
12. Les calculs sont faits, en interne, avec une précision de 12 chiffres après la virgule.
13. Le solde d'un compte n'est pas plafonné.
14. Le solde d'un compte ne peut être négatif.
15. L'utilisateur peut faire des transferts ponctuels entre ses comptes.
16. Un transfert s'opère toujours entre deux comptes distincts.
17. Le montant d'un transfert ponctuel doit être supérieur au solde du compte débité.
18. Le montant d'un transfert peut être exprimé soit en Dollar US, soit en Franc français, soit en Euro.
19. Des règles d'épargne périodique peuvent être définies.
20. Une règle d'épargne périodique doit spécifier un compte à débiter.
21. Une règle d'épargne périodique doit spécifier un compte à créditer.
22. Une règle d'épargne périodique doit spécifier le solde maximum du compte à débiter.
23. Si le solde du compte à débiter est supérieur au solde maximum spécifié alors l'application crédite le solde à créditer du montant de (solde courant) - (solde maximum).
24. Si le solde du compte à débiter est inférieur au solde maximum spécifié alors l'application ne fait rien.
25. Une règle d'épargne doit être vérifiée périodiquement.
26. Une période de vérification doit être spécifiée pour une règle d'épargne.
27. L'unité de la période de vérification est la seconde pour une règle d'épargne.
28. Des règles de dépenses périodiques peuvent être définies.

29. Une règle de dépense périodique doit spécifier un compte à débiter.
30. Une règle de dépense périodique doit spécifier un compte à créditer.
31. Une règle de dépense périodique doit spécifier le solde minimum du compte à débiter.
32. Si le solde du compte à créditer est inférieur au solde minimum spécifié alors l'application débite le solde à débiter du montant de (solde minimum) - (solde courant) (du compte à créditer).
33. Si le solde du compte à créditer est supérieur au solde minimum spécifié alors l'application ne fait rien.
34. Une règle de dépense doit être vérifiée périodiquement.
35. Une période de vérification doit être spécifiée pour une règle de dépense.
36. L'unité de la période de vérification est la seconde pour une règle de dépense.
37. On peut conserver les règles d'épargne ou de dépense entre deux connexions.
38. La périodicité des transferts est exprimée en secondes.
39. Pendant une session la somme des soldes des comptes doit être constante.
40. Il est possible d'annuler les règles au cours d'une connexion.

A.2 La première passe

Le tableau TAB. A.1 présente les 14 groupes définis dans un premier temps. Ces groupes sont associés aux 14 schémas de test correspondant aux différentes exécutions des exigences du cahier des charges. Ces 14 schémas de test génèrent 1900 cas de test différents.

1. Ex7EN01_Ex7ER01 :
   MTC !balances.AccountsMethods
   "Cas de test 1" : getAccountsVector(0)
   "Cas de test 2" : getAccountsVector(1)
   "Cas de test 3" : getAccountsVector(2)
   "Cas de test 4" : getAccountsVector(3)
   "Cas de test 5" : getBalances(0)
   "Cas de test 6" : getBalances(1)
   "Cas de test 7" : getBalances(2)

AccountsMethods	getAccountsVector({0,1,2,3}) getBalances({0,1,2,3})
Transfer	transfer(1 ;2 ;100)
BocreateEN	Bocreate (1 ;"BNP" ;1000.0)
GetBalancesVector	getAccountsVector({1,2}) getBalances({1,2})
getBalancesEN8	BOcreate(1 ;"BNP" ;1000.0) getBalances(1)
BOcreateEN_2	BOcreate(1,"CE",2000.0)
BOdeleteEN803	BOdelete(4)
BOcreateEx8ER02	BOcreate(1 ;"BNP" ;1000.0)
BOcreateEx8ER03	BOcreate(1 ;"LSR" ;1000.0)
setcurrencyEx9	setCurrency({"USD","EUR"})
amountToDisplayEx	amountToDisplay(1.0)
setcurrencyEx9ER	setCurrency("CHF")
transferEx14	transfer({1,2} ;{1,2,100} ;{100.0,1000,2000})
transferEx15_39	transfer({1,2,3} ;{1,2,3} ;{5,10,15,20,25,30,35})

TAB. A.1 – Les groupes créés pour la première passe

"Cas de test 8" : getBalances(3)

2. Ex7EN02 :
   MTC !transfers.transfer ;MTC !balances.getBalancesVector
   "Cas de test 1" : transfer(1,2,100),getAccountsVector(1)
   "Cas de test 2" : transfer(1,2,100),getAccountsVector(2)
   "Cas de test 3" : transfer(1,2,100),getBalances(1)
   "Cas de test 4" : transfer(1,2,100),getBalances(2)

3. Ex8EN01 :
   MTC !accman.BOcreateEN ;MTC !balances.getBalancesEN8
   "Cas de test 1" : BOcreate(1,"BNP" : 1000.0),getBalances(1)

4. EX8EN02 :
   MTC !accman.BOcreateEN ;MTC !accman.BOcreateEN_2 ;
   MTC !balances.getBalancesEN8

"Cas de test 1" : BOcreate(1,"BNP" : 1000.0),BOcreate(1,"CE" : 2000.0),getBalances(1)

5. Ex8EN03 :
   MTC !accman.BOcreateEN ;MTC !accman.BOdeleteEN803 ;
   MTC !balances.getBalancesEN8
   "Cas de test 1" : BOcreate(1,"BNP" : 1000.0),BOdelete(4),getBalances(1)

6. Ex8ER01 :
   MTC !accman.BOcreateEx8ER01
   "Cas de test 1" : BOcreate(1,"BNP" : 1000.0)

7. Ex8ER02 :
   MTC !accman.BOcreateEx8ER02
   "Cas de test 1" : BOcreate(1,"BNP" : 1000.0)
8. Ex8ER03 :
   MTC !accman.BOcreateEx8ER03
   "Cas de test 1" : BOcreate(1,"LSR" : 1000.0)

9. Ex9EN01 :
   MTC !currency.setcurrencyEx9 ; MTC !currency.amountToDisplayEx
   "Cas de test 1" : setCurrency("USD"),amountToDisplay(1.0)
   "Cas de test 2" : setCurrency("EUR"),amountToDisplay(1.0)

10. Ex9ER01 :
    MTC !currency.setcurrencyEx9ER
    "Cas de test 1" : setCurrency("CHF")

11. Ex14ENER :
    MTC !transfers.transferEx14 (18 cas de test sont générés)
    "Cas de test 1" : transfer(1,1,100.0)
    "Cas de test 2" : transfer(2,1,100.0)
    "Cas de test 3" : transfer(1,2,100.0)
    "Cas de test 4" : transfer(2,2,100.0)
    "Cas de test 5" : transfer(1,100,100.0)
    ...

"Cas de test 16" : transfer(2,2,2000.0)
"Cas de test 17" : transfer(1,100,2000.0)
"Cas de test 18" : transfer(2,100,2000.0)

12. Ex15_Ex39 :
MTC !transfers.transferEx15_39 (63 cas de test sont générés)
"Cas de test 1" : transfer(1,1,5)
"Cas de test 2" : transfer(2,1,5)
"Cas de test 3" : transfer(3,1,5)
"Cas de test 4" : transfer(1,2,5)
"Cas de test 5" : transfer(2,2,5)
...
"Cas de test 61" : transfer(1,3,35)
"Cas de test 62" : transfer(2,3,35)
"Cas de test 63" : transfer(3,3,35)

13. Ex19EN01_02_06_Ex01_02_03 :
MTC !transfers.RegisterSavingRule (750 cas de test sont générés)
"Cas de test 1" : registerSavingRule("12082002" : 1,100.0,1,0)
"Cas de test 2" : registerSavingRule("12082002" : 2,100.0,1,0)
"Cas de test 3" : registerSavingRule("12082002" : 3,100.0,1,0)
"Cas de test 4" : registerSavingRule("12082002" : 100,100.0,1,0)
"Cas de test 5" : registerSavingRule("12082002" : 200,100.0,1,0)
...
"Cas de test 749" : registerSavingRule("12082002" : 100,100.0,200,1)
"Cas de test 750" : registerSavingRule("12082002" : 200,100.0,200,1)

14. Ex28all :
MTC !transfers.RegisterSpendingRule (1100 cas de test sont générés)
"Cas de test 1" : registerSpendingRule("12082002" : 1,100.0,1,0)
"Cas de test 2" : registerSpendingRule("12082002" : 0,100.0,1,0)
"Cas de test 3" : registerSpendingRule("12082002" : 1,100.0,1,0)
"Cas de test 4" : registerSpendingRule("12082002" : 2,100.0,1,0)
"Cas de test 5" : registerSpendingRule("12082002" : 3,100.0,1,0)
...
"Cas de test 1099" : registerSpendingRule("12082002" : 2,900.0,3,3)
"Cas de test 1100" : registerSpendingRule("12082002" : 3,900.0,3,3)

A.3 La deuxième passe

Les groupes et schémas de test de la première passe ont été repris. Le nombre de groupes a été réduit à 6 (TAB. A.2). Cette réduction du nombre de groupes (par agrégation de groupes) permet de redéfinir un ensemble plus petit de 5 schémas de test couvrant l'ensemble des exigences. Pour tous ces nouveaux schémas de test nous précisons à quels schémas de test de la première passe ils se rapportent.

G1	getAccountsVector({0,1,2,3}) getBalances({0,1,2,3})
G2	transfer({1,2,3,100} ;{1,2,3,100} ;{0,100.0,1000})
G3	Bocreate(1 ;{"CE","BNP","LSR"} ;1000.0) getBalances(1)
G4	BOdelete(3)
G5	setCurrency({"USD","EUR","CHF"})
G6	amountToDisplay(1.0)

TAB. A.2 – Les groupes créés pour la deuxième passe

1. S1 (Ex7EN02, Ex7EN01_Ex7ER01, Ex14ENER, EX15_Ex39) :
   MTC !transfers.G2 ; MTC !balances. G1
   Ce schéma de test permet de tester les opérations de transfert.
2. S2 (Ex8EN03,Ex8EN1, Ex8EN2, Ex8ER01, Ex8ER02, Ex8ER03) :
   MTC !accman.G3 ; MTC !accman.G4 ; MTC !balances.G1
   Avec ce schéma on peut s'assurer que la création et la destruction des comptes s'effectuent de manière conforme.
3. S3 (Ex9EN01, Ex9ER01) :
   MTC !currency.G5 ; MTC !currency.G6
   Ce schéma s'assure que l'affichage des soldes dans toutes les devises est correct
4. S4 (Ex19EN01_02_06_Ex01_02_03) :
   MTC !transfers.RegisterSavingRule
   Avec ce schéma de test on définit des règles d'épargne périodique.

5. S5 (Ex28all) :
   MTC !transfers.RegisterSpendingRule
   Avec ce schéma de test on définit des règles de dépense périodique.