Université de Franche-Comté **UFR Sciences et Techniques**

École doctorale SPIM

Contribution à la génération automatique de tests à partir de modèles et de schémas de test comme critères de sélection dynamiques

THÈSE

présentée et soutenue publiquement le 8 décembre 2009

pour l'obtention du grade de

Docteur de l'université de Franche-Comté

(Spécialité Informatique)

par

Régis Tissot

Composition du jury

Président :	Bruno Legeard, Professeur, Université de Franche-Comté, LIFC
Directeurs :	Jacques Julliand, Professeur, Université de Franche-Comté, LIFC Pierre-Alain Masson, Maître de conférences, Université de Franche-Comté, LIFC
Rapporteurs :	Pascale Le Gall, Professeur, École Centrale Paris, MAS Marie-Laure Potet, Professeur, Institut Polytechnique de Grenoble, VERIMAG
Examinateur :	Boutheina Chetali, Security Research Group Manager, GEMALTO

Laboratoire d'Informatique de l'Université de Franche-Comté

Mis en page avec la classe thloria.

Remerciements

En premier lieu, je souhaite adresser mes remerciements à Jacques Julliand et Pierre-Alain Masson, d'abord pour m'avoir offert la possiblité de réaliser cette thèse, puis pour tout le temps qu'ils ont consacré à m'accompagner et à me soutenir durant ces trois années.

Je tiens à remercier chaleureusement mes rapporteurs Pascale Le Gall et Marie-Laure Potet, d'avoir accepté de plonger dans la lecture de ce manuscrit. J'adresse des remerciements tout aussi chaleureux à Boutheina Chetali et Bruno Legeard pour leur participation à mon jury de thèse. Merci de l'intérêt porté à mes travaux.

Je remercie toutes les personnes avec qui j'ai eu plaisir à travailler dans le cadre du projet POSÉ et en particulier : Julien Botella, Eddie Jaffuel et Bruno Legeard de SMARTESTING ; June Andronick, Boutheina Chetali et Georges Debois de GEMALTO ; Lydie du Bousquet, Yves Ledru et Marie-Laure Potet du Laboratoire d'Informatique de Grenoble ; Lionel Van Aertryck de SILICOMP-AQL ; Fabrice Bouquet et Frédéric Dadeau du LIFC.

Un grand merci à Fred pour nos collaborations scientifiques et pédagogiques fructueuses. Et de manière générale, merci à tous les membres du laboratoire, à ceux qui m'ont fait confiance en me confiant des responsabilités d'enseignement (Alain, Jacques, Fabien, Fabrice, Violetta), et à tous les autres, pour avoir rendu toutes ces années passées au LIFC agréables et formatrices.

Un immense merci à tous mes compagnons de "galère", doctorants et ingénieurs, pour tous les bons moments partagés. D'abord, merci à mes collègues du bureau 410, Jérôme, Pierre-Christophe et Sékou pour la bonne ambiance qui a toujours règnée entre nos murs. Et merci à tous ceux avec qui j'ai eu beaucoup de plaisir à partager les pauses cigarettes, les pauses café, la bouffe du RU, les soirées autour de pintes de bière, celles autour de verres de rhum, les parties d'échecs, celles de tarot, les blagues du vendredi, celles du reste de la semaine,... Enfin, merci à vous tous pour toutes ces petites choses qui rendent la vie agréable : Adrien, Alex, Aloïs, Doudou, David, Émilie, JB, John, Kalou, Muriel, Paco, Paul, Philippe, Roméo, Seb, Steph, Vincent (improvisé ambulancier un certain vendredi matin), Yaka... et tous ceux et celles que j'oublie peut-être dans la précipitation. Je vous souhaite à tous beaucoup de réussite pour la suite en espérant que nos routes continuerons de se croiser, au détour d'une cigarette, d'une pinte de bière, d'un verre de rhum, d'une partie d'échecs, d'une partie de tarot,...

Merci à mes parents, sans qui je ne serais certainement jamais arrivé où j'en suis, pour m'avoir toujours encouragé tout au long de mes études et pour tout le reste...

Enfin, je conclurai par quelqu'un, ou plutôt quelqu'une, que je ne remercierai jamais assez de me supporter depuis plus de cinq ans maintenant... Et qui j'espère me supportera de nombreuses années encore... Merci Anaïs.

Table des matières

Première partie

Contexte et problématique

Introduction

Nous présentons, dans cette thèse, une contribution à la conception d'une méthode de génération automatique de tests à partir de modèles (ou MBT — Model Based Testing). Nos travaux s'inscrivent dans la continuité de ceux effectués autour des l'outils BZ-Testing Tools (BZ-TT) et Leirios Tests Generator (LTG) dont l'objectif est la génération de tests fonctionnels à partir de modèles décrits en langage B. Les critères de sélection de tests implémentés ces deux outils reposent sur la couverture structurelle des opérations du modèle du système à valider en prenant en compte les structures de données et de contrôle de celui-ci.

Cette approche ne permet pas de générer de tests à partir de propriétés liées au comportement dynamique du système, par exemple en tenant compte de propriétés basées sur des enchaînements d'opérations. Afin de répondre à cette problèmatique, un certain nombre de travaux proposent des méthodes où l'expertise humaine est exploitée afin de définir des critères de sélection de tests "dynamiques". De tels critères permettent à l'ingénieur validation de définir des stratégies basées sur des propriétés et des aspects du système qu'il souhaite valider. Nos contributions s'inscrivent dans cette voie, tout en visant la complémentarité par rapport à la génération automatique de tests par couverture structurelle du modèle dans un objectif de réutilisation des technologies et ressources déployées à cette fin.

Notre première contribution est la définition d'un langage de formalisation d'objectifs de tests qui permet d'exprimer des ensembles de scénarios de tests inspirés de propriétés à valider sur le système. Ce langage permet de décrire des schémas de tests à partir d'un formalisme, basé sur celui des expressions régulières, qui permet de décrire des ensembles de scénarios principalement par des enchaînements d'appels d'opération et d'états symboliques.

Notre seconde contribution est la définition et l'implémentation de méthodes de génération de tests intégrées aux outil BZ-TT et LTG, afin de prendre en compte ce nouveau type de critères de sélection de tests. Cette méthode permet de réutiliser le modèle conçu pour le test, les technologies d'animation symbolique et de résolution de contraintes de ces outils. Il permet également de conserver les fonctionnalités de concrétisation et d'exécution des tests produits. Dans cette méthode, la seule charge supplémentaire pour l'ingénieur validation est la définition des schémas de test utilisés comme critère de sélection.

Nos dernières contributions, visent à évaluer la complémentarité de notre méthode avec celle de génération automatique de tests par couverture structurelle du modèle. Nous proposons une méthode d'évaluation de la complémentarité entre deux suites de tests. Cette méthode est basée sur le calcul de la couverture d'états et de transitions des suites de tests sur une abstraction du système. Enfin, nous appliquons cette méthode à trois études de cas (deux applications de type carte à puce et un système de gestion de fichiers POSIX), et nous montrons la complémentarité qu'elle apporte.

Chapitre 1

Contexte, Motivations et Contributions

Sommaire

1.1 Contexte

L'activité de validation par le test d'un système peut se décomposer en trois phases : la conception de cas de test, leur exécution et l'évaluation de la couverture obtenue par rapport aux exigences initiales, au modèle ou à l'implémentation.

La réalisation de cas de test est une démarche qui, à partir des spécifications informelles (ou semi-formelles) d'un système ou à partir de l'implémentation, va amener un ingénieur validation à rédiger des cas de tests. Ces cas de tests décrivent comment doit être animé le système et comment celui-ci doit réagir, afin de détecter d'éventuels problèmes de conformité entre le système sous test et sa spécification. Ils doivent couvrir l'ensemble des exigences de validation imposées sur le système, dans le but d'augmenter la confiance dans la conformité du système vis-à-vis de sa spécification.

L'exécution des tests sur le système impose de définir et fréquemment d'outiller une interface avec le système qui permettra d'interagir avec ce dernier. La phase d'exécution des tests doit aussi permettre d'établir les verdicts, c'est-à-dire que les interfaces avec le système doivent mettre à disposition suffisamment de points d'observation pour permettre de déterminer si l'exécution d'un test révèle ou non, une non-conformité.

Enfin, la phase d'analyse de couverture des tests exécutés doit permettre de quantifier la partie du système adressée par les tests. Cette quantification peut être réalisée par rapport à divers référenciels :

- les exigences de validation exprimées par le cahier des charges,
- le modèle du système,
- le système lui-même.

On parlera alors de couverture d'exigences, de couverture de modèle ou de couverture de code.

Bien qu'elles offrent une rentabilité à moyen terme, en réduisant les coûts d'entretien, de dépannage et d'évolution des systèmes, toutes ces activités sont coûteuses et représentent un investissement important lors de n'importe quel développement. Une des solutions utilisées pour optimiser ces activités est la génération de tests à partir de modèles, autrement dit le *"Model Based Testing"* (MBT).

1.2 Le test à partir de modèles

Le test à partir de modèles a pour objectif d'automatiser la validation de systèmes. La figure 1.1 présente une vision de l'activité de validation par le test, proposée par Jan Tretmans (Univ. Nijmegen, NL), suivant trois axes principaux : le niveau de détail de ce que l'on souhaite tester, le niveau d'accessibilité du système et les caractéristiques que l'on souhaite valider.

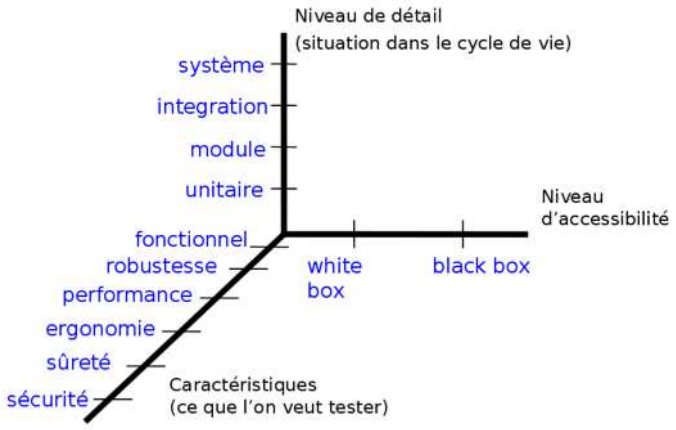


FIG. 1.1 – Vision du test suivant 3 axes

Parmi les niveaux de détail des aspects du système que l'on souhaite valider, on peut distinguer :

- Le test unitaire qui vise à valider indépendamment les élements "unitaires" d'un système, tels que les fonctions ou les méthodes.
- Le test de module qui vise à valider indépendamment chaque module du système.
- Le test d'intégration qui vise à tester le bon comportement de la composition des procédures, modules ou composants d'un système.
- Et finalement, le test du système global (ou test de recette/conformité) qui vise à considérer la validation de l'adéquation du système dans son ensemble vis-à-vis de ses spécifications.

L'accessibilité du système peut être réduite aux concepts de boîte blanche et de boîte noire qui respectivement s'appliquent aux systèmes dont l'état interne peut être observé (boîte blanche) et aux systèmes qui ne peuvent être observés qu'au travers d'une interface réduite (boîte noire). Cette différence prend toute son importance pour les questions d'observation du système sous test et en particulier pour l'établissement des verdicts des tests (résultats des comparaisons entre ce qui est prévu par le modèle d'un système et les comportements réels de ce système).

De même qu'il existe plusieurs niveaux de détail dans les aspects du système que l'on souhaite tester, un grand éventail de caractéristiques différentes peuvent être sujettes à validation. Nous en détaillons quelques-unes :

- Le test fonctionnel vise à valider les comportements nominaux du système en vérifiant que les résultats des appels aux différentes fonctionnalités du système sont conformes à ceux attendus (calculés à partir d'un modèle ou fournis manuellement).
- Le test de robustesse vise à valider le fait que le système réagit correctement à une utilisation non-conforme (par exemple à la saisie de données invalides).
- Le test de performance vise à tester les capacités du système à répondre aux requêtes qui lui sont soumises en un temps donné (par exemple dans le cas d'un grand nombre de requêtes simultanées ou d'une cadence élevée d'arrivée des requêtes).

En plus des différents aspects de l'activité de validation par le test, nous pouvons citer le test de non-régression. Cette approche est commune à la plupart de celles que nous avons présentées. Elle vise à tester l'évolution d'un système en validant le fait que des modifications (évolutions ou corrections) du système n'introduisent pas d'anomalies dans les parties supposées inchangées du système.

Parmi tout cet éventail de facettes de l'activité de validation par le test, nous situons nos travaux dans une approche de validation fonctionnelle d'un système de type boîte noire à partir de son modèle. Par conséquent, dans ce chapitre, nous abordons le thème du test de conformité à partir de modèles. Nous présentons d'abord un aperçu gobal du processus de validation par MBT. Puis, nous nous penchons sur les problématiques de la modélisation et de la génération des tests. Et enfin, nous abordons la concrétisation, l'exécution des tests, l'établissement du verdict et l'analyse de la couverture du système (ou de sa spécification) par les tests.

1.2.1 Démarche MBT

Le principe du Model Based Testing est résumé dans la figure 1.2. Cette figure présente les phases clés du processus de validation d'un système par test à partir de modèles. Les flèches pointillées représentent les étapes manuelles, et celles pleines représentent les étapes automatiques ou automatisables. La partie droite de la figure schématise un processus classique de développement. Le cadre pointillé à gauche représente les moyens complémentaires mis spécifiquement en œuvre dans une approche MBT :

- Un modèle formel du système est rédigé à partir des spécifications.
- Un ou plusieurs critères de sélection de tests sont définis. Ils permettent de définir comment les tests sont dérivés du modèle.
- Des cas de test abstraits sont générés en appliquant les critères de sélection de tests au modèle.
- Une relation entre le modèle et le système doit être définie afin de pouvoir transformer les cas de test abstraits, exprimés au niveau d'abtraction du modèle, en cas de test

concrets, exécutables sur le système sous test (SUT).

Et enfin, la partie en aval du schéma représente le cas général de l'exécution des tests (lors d'une approche de validation MBT). Quelque soit la démarche mise en œuvre pour produire les tests, ceux-ci sont exécutés sur le système sous test et un verdict est calculé en comparant les résultats obtenus à partir de l'implémentation à ceux prévus par le modèle.

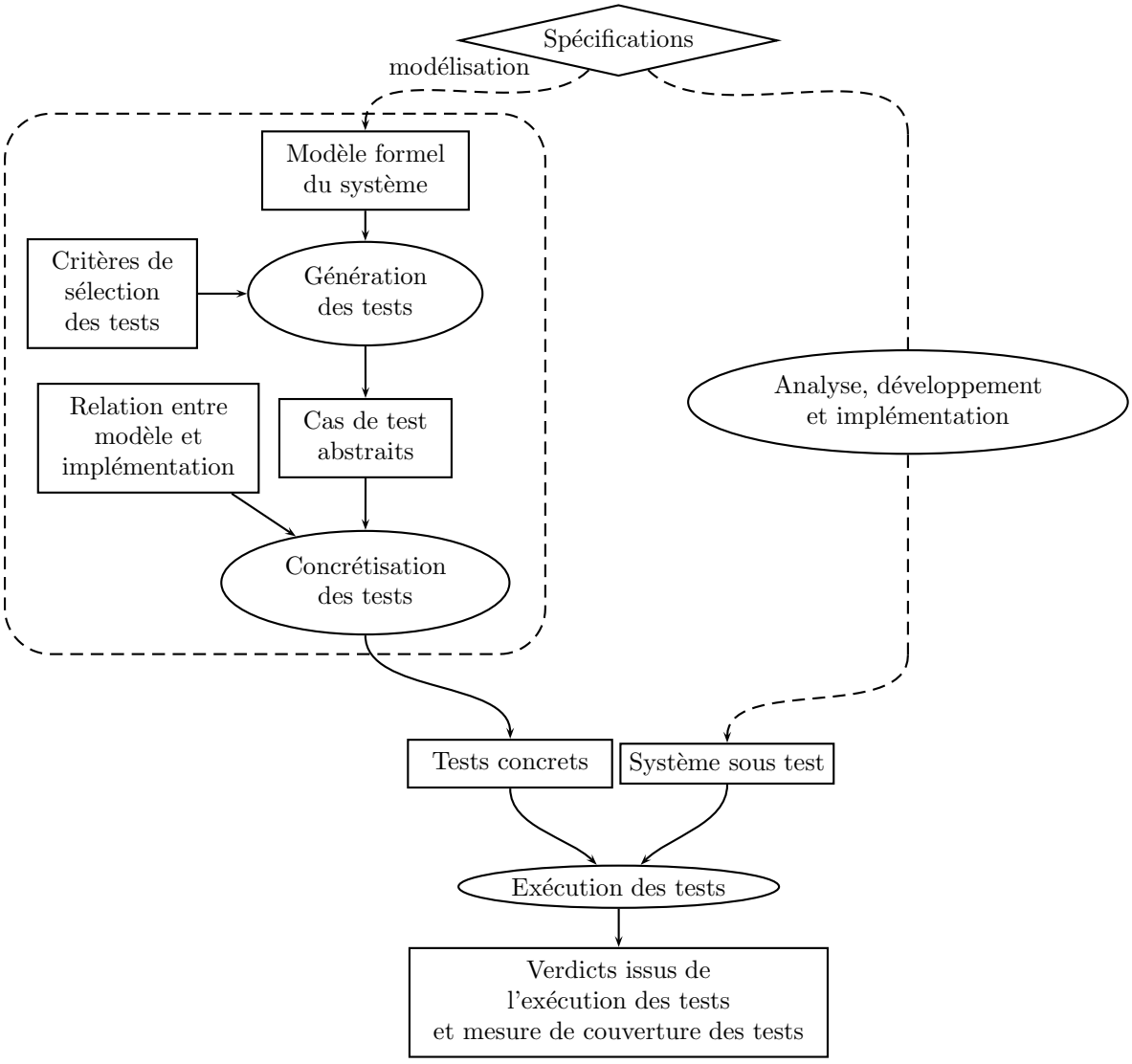


FIG. 1.2 – Processus de Model Based Testing

1.2.2 Modélisation

Dans [PP04], les auteurs débutent leur propos sur l'importance de la modélisation dans l'activité de validation d'un système. Le constat de base est que, trop souvent, les ingénieurs validation utilisent les spécifications uniquement pour s'approprier une connaissance incomplète du système et se basent sur un modèle "mental" de celui-ci pour dériver des cas de test. Cette approche n'est ni structurée, ni reproductible et elle n'offre pas d'argument permettant de justifier son efficacité.

L'utilisation de modèles formels permet de constituer une base solide pour structurer, reproduire (faire évoluer) et argumenter une démarche de validation. Ensuite, se pose la question de savoir comment utiliser les modèles. Les auteurs de [PP04] exposent quatre scénarios distincts :

1. Le modèle tient lieu de spécification, il sert à la fois à générer des tests et à générer le code de l'implémentation.
2. Le modèle est extrait de manière automatique de l'implémentation, ensuite il est ensuite utilisé afin de générer les tests.
3. Le modèle est réalisé manuellement à partir des spécifications et il sert à générer des tests.
4. Deux modèles sont utilisés, l'un pour générer le code de l'implémentation et un second pour générer les tests.

Le premier scénario pose deux problèmes. L'un est dû à l'absence de redondance d'informations entre le comportement attendu d'un système (modèle) et le comportement réel (implémentation). Or cette redondance d'informations est cruciale pour établir un verdict. L'autre est dû aux contraintes portant sur le niveau d'abstraction du modèle qui devra être très proche de l'implémentation pour permettre la génération automatique de code. Le second scénario pose le même problème d'absence de redondance des informations. Ce sont les troisième et quatrième scénarios qui offrent les meilleurs garanties d'effectuer une validation de qualité.

Pour notre part, nous nous intéressons aux troisième et quatrième scénarios exposés qui considèrent un cycle de validation dans lequel un modèle spécifique du système est utilisé afin de générer des tests. L'idée est donc de passer par une étape de formalisation du système à tester à partir de ses spécifications. Cette formalisation va donner une vue abstraite du système autorisant l'utilisation d'outils dédiés à automatiser la production et l'exécution de tests. Certains outils et certaines méthodes dédiés à la validation de systèmes par génération de tests à partir de modèles sont présentés dans les chapitres 2 et 3.

Niveau d'abstraction du modèle

L'abstraction est une notion essentielle pour la modélisation d'un système en vue de sa validation par le test. Dans [UL06], les auteurs mettent l'accent sur l'importance d'une bonne modélisation, car cette phase de travail va influer sur la phase de génération des tests. Quels que soient les critères de sélection de tests appliqués, un modèle très abstrait ne produit qu'un nombre limité de tests ne couvrant que peu d'aspects du système ; alors qu'un modèle dont le niveau d'abstraction est proche de l'implémentation offre la possibilité de générer un grand nombre de tests qui explorent le système dans ses moindres détails. Le choix du niveau d'abstraction est donc motivé d'une part par les objectifs de validation du système et d'autre part, par la nécessité de maîtriser le nombre de tests générés.

Dans [BDLN06], les auteurs pointent du doigt cette difficulté dans le cadre du rapprochement d'un processus de conception assisté par les modèles et d'un processus de validation basé sur le MBT. Dans ce cadre, un modèle d'analyse du système issu des spécifications va servir de point de départ pour le développement et pour la génération de tests. Les auteurs montrent clairement qu'un modèle d'analyse (fait dans une optique de développement) ne convient pas pour produire des tests, certaines informations sont trop détaillées, d'autres pas assez. C'est pourquoi le choix est fait de produire un modèle pour le test à partir de ce modèle d'analyse.

Dans [PP04], les auteurs décrivent deux approches d'abstraction : *l'omission de détails* et l'encapsulation de détail. L'approche *d'omission de détails* consiste à éliminer certaines informations lors de la formalisation. *L'encapsulation de détails* consiste à occulter la complexité d'un système ou d'une sous-partie d'un sytème par des références. Par exemple par l'utilisation d'une référence à un appel de fonction ou par une référence à un ensemble de données.

Pour conclure sur l'abstraction, nous nous devons de dire quelques mots sur le raffinement. On peut considérer le raffinement comme le processus inverse de l'abstraction : il s'agit de partir d'une modélisation très abstraite pour la détailler par étapes successives afin d'arriver à un niveau de modélisation très détaillé, et éventuellement jusqu'à l'implémentation. Cette manière de procéder est appliquée dans la *méthode B* [Abr96]. La méthode B décrit un processus de raffinement de modèles mené conjointement à un processus de vérification (basé sur la preuve), à partir duquel il est possible de produire du code (Ada, C). Un processus de preuve permet de garantir la préservation de propriétés de haut niveau au cours des raffinements successifs.

L'utilisation des modèles

Un modèle de conception (ou d'analyse) est une formalisation des spécifications du système à développer. La rédaction de ce modèle est une phase préalable à la phase de développement proprement dite. Ce modèle peut être destiné à plusieurs utilisations :

- En tant que document de référence, il peut compléter les spécifications pour le suivi de projet.
- Comme document intermédiaire entre les spécifications et le système, il peut être porteur d'informations complémentaires comme des choix de conception.
- S'il est animable, il peut servir de "maquette" que les ingénieurs de validation peuvent utiliser pour comprendre le système en profondeur et en déduire des cas de tests manuellement.
- Certaines propriétés issues des spécifications, une fois formalisées, peuvent être vérifiées sur ce modèle à l'aide de techniques de preuve ou de model-checking. Ceci afin de montrer que les choix faits lors de la modélisation respectent les spécifications.

L'utilisation de modèles pour le test répond à une autre attente. Ici l'utilisation du modèle consiste à extraire des traces qui seront ensuite exécutées sur le système sous test (SUT) afin d'observer ses réponses et de les comparer à celles prévues par le modèle (oracle du test). Ainsi, les questions qui se posent lors de la modélisation pour le test sont différentes de celles qui se posent pour développer un modèle de conception. Par exemple :

- Quelles sont les actions contrôlables, et quelles sont les actions observables ?
- Certains aspects des spécifications sont-ils dépendants des choix d'implémentation ?

Les *actions contrôlables* sont celles qui permettent d'agir sur le système, alors que les *actions observables* sont déclenchées de manière automatique par le système (expiration d'un délai, réponse à une action contrôlable, ...). Si certains aspects des spécifications dépendent de choix d'implémentation, plusieurs réponses se présentent :

- Il est possible de réaliser un modèle qui "colle" au système en concertation avec l'équipe de développement.
- Il est possible de représenter ces choix par un modèle indéterministe.
- Ces choix peuvent être abstraits soit par encapsulation, soit par omission de détails.

La première de ces trois solutions présente l'avantage de rendre la modélisation plus naturelle, mais d'un autre côté, il restreint l'utilisation du modèle à un choix particulier d'implémentation. La seconde solution offre un modèle plus complet du système, par contre les étapes de génération, concrétisation et exécution des tests sont plus complexes. En effet, l'exécution d'une opération d'un cas de test peut influer sur le calcul de l'oracle pour les opérations ultérieures. Enfin, le choix de l'abstraction permet de conserver un modèle général, par contre, des vérifications complémentaires devront être menées si on souhaite valider une partie du système que l'on a abstraite.

La modélisation pour le test est donc délicate, et l'ingénieur de validation doit continuellement faire des compromis entre le fait d'atteindre ses objectifs de validation et la prise en compte des différentes difficultés qui peuvent se présenter comme l'indéterminisme, la gestion du contrôle et de l'observation du système et bien sûr les restrictions technologiques de la génération de tests.

1.2.3 Génération des tests

Constitution d'un cas de test

Un cas de test est une séquence d'actions issue du modèle lors de la génération de test. L'exécution de cette séquence sur le système sous test permet d'obtenir un verdict de conformité entre celui-ci et le modèle. Un cas de test est constitué d'une suite d'appels aux fonctionnalités du système et d'un oracle. La suite d'appels aux fonctionnalités du système permet de faire évoluer ce dernier et l'oracle permet de comparer les réponses du système à celles prévues par le modèle.

Dans le cas d'un système réactif à entrées/sorties, la partie "appel de fonctionnalité" est caractérisée par des actions d'émission (réception pour le système) et la partie oracle est caractérisée par des actions de réception (émission pour le système). Dans le cas d'un système orienté sur les données, la partie "appel de fonctionnalité" est plutôt caractérisée par des appels de fonctions munis de paramètres et la partie oracle plutôt caractérisée par les résultats des appels de fonctions.

L'objectif d'un cas de test est de valider le comportement du système dans une situation particulière dont l'identification dépend du critère de sélection des tests employé (la notion de critère de sélection est présentée plus loin). La figure 1.3 présente la forme générale d'un cas de test.

La suite d'appels d'opérations visant à mettre le système dans un contexte (état) particulier est appelé *"préambule"*. L'appel de l'opération dont on souhaite valider l'exécution est appelé le *"corps de test"*. En plus du préambule et du corps de test, un cas de test peut être constitué d'une partie *"observation"* et d'une partie *"postambule"*. La partie observation[1] est

[1] On peut noter que les opérations présentes dans la partie observation, ne sont pas forcément que des

une suite d'opérations suivant le corps de test dont l'objectif est d'observer l'état du système après le corps de test. La partie "postambule" est une suite d'appels d'opérations à la fin du cas de test dont l'objectif est de positionner le système dans un état particulier qui peut servir de point de départ à l'exécution d'un autre cas de test[2].

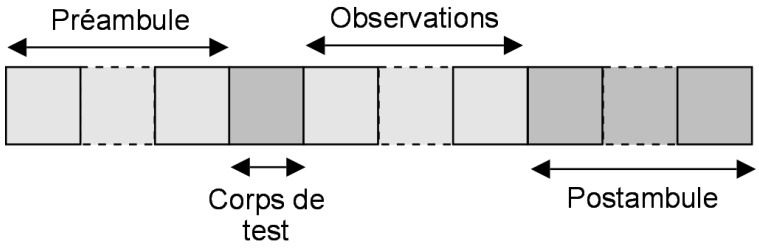


FIG. 1.3 – Forme générale d'un cas de test

Les critères de sélection

La génération des tests à partir du modèle va consister à extraire des séquences d'opérations du modèle en vue de les exécuter sur le système. Afin de sélectionner les séquences d'opérations utilisées pour tester le système, on définit un critère de sélection. Suivant le type de modélisation choisie, divers critères de sélection de tests peuvent être appliqués :

- des critères de sélection basés sur l'expertise humaine (sélection manuelle),
- des critères de *"parcours"* portant sur la couverture d'états, de transitions, de paires de transitions ou de chemins,
- des critères de couverture de comportements (couverture de graphe de flot de contrôle),
- des critères basés sur la couverture de données,
- des critères basés sur des modèles de fautes,
- des critères basés sur des scénarios,
- des critères basés sur le hasard (test aléatoire), potentiellement complétés par un critère de couverture uniforme du modèle.

Nous proposons une présentation de ces critères dans les chapitres 2 et 3.

Dans [Ber03], l'auteur fait d'abord remarquer que l'utilisation de critères de sélection de tests différents peut mener à des résultats très différents en fonction du contexte (type d'application, expertise de l'ingénieur de validation, outils utilisés, ...). Plus loin, l'auteur ajoute que des critères de sélection de tests différents mènent à la détection de différents types "d'erreurs". Par ailleurs, une stratégie visant à utiliser ses ressources humaines et matérielles pour utiliser plusieurs critères de sélection semble être plus efficace qu'une forte concentration de ressources utilisées pour la mise en place d'un seul critère de sélection.

opérations observables. En effet, une action sur le système peut provoquer des réactions qui nous renseignent sur l'état dans lequel il se trouve.

[2]En règle générale, on cherche à ramener le système à son état initial.

1.2.4 Concrétisation, exécution et verdict

Concrétisation

Dans la partie "modélisation", nous avons présenté l'utilisation de l'abstraction qui est utilisée afin de produire un modèle de "plus haut niveau" du système à tester. Le fait de posséder un modèle abstrait du système offre des avantages non négligeables, puisqu'il permet de se concentrer sur les parties du système que l'on souhaite valider. Il permet également de traiter certains problèmes comme l'indéterminisme du système à tester ou la complexité des données, facilitant ainsi la génération de tests. Et si cette étape d'abstraction du système n'est pas simple et requiert une certaine expertise pour être menée, son "miroir" qui est la concrétisation n'est guère plus aisée.
Principalement, la concrétisation des cas de tests va se concentrer sur trois axes :

- La différence de niveau d'abstraction entre le modèle (et par conséquent les tests générés à partir du modèle) et le système à tester doit être comblée. On peut diviser cette problématique en trois cas. En premier lieu, il faut mettre le système dans le même état que celui qui a servi d'état initial du modèle pour générer les tests. Ensuite, il faut assurer la correspondance de type et de valeur entre les constantes du modèle et celle de l'implémentation. Et enfin, il faut concrétiser les fonctionnalités du système qui ont pu être abstraites dans la modélisation (par exemple une fonction de cryptage de données qui aurait été abstraite par une fonction indiquant juste si une donnée est cryptée ou non au niveau du modèle).
- La mise en place d'une solution destinée à rendre possible l'exécution des tests sur le système.
- L'établissement du verdict qui consiste d'une part à observer les réactions du système en réponse aux sollicitations décrites dans le cas de test, et d'autre part à définir une relation qui permet de déterminer si les réponses du système sont conformes à celles prévues par le modèle (oracle du test).

Dans [UL06], trois solutions de concrétisation sont présentées :

- **L'adaptation,** qui consiste à créer un "adaptateur" qui prend en entrée les cas de test abstraits et va assurer la synchronisation entre ce dernier et le système sous test.
- **La transformation,** qui consiste à traduire le cas de test abstrait en un script exécutable sur le système.
- **Une approche mixte,** qui consiste à créer un adapteur un peu plus général qui prend en entrée des cas de tests déjà partiellement traduits.

Il n'y a pas de règle générale pour justifier l'emploi de l'une ou l'autre des méthodes. Le choix est déterminé par un ensemble de paramètres parmi lesquels nous pouvons citer, le système à tester, l'interface avec ce système, la différence de niveau d'abstraction entre le modèle et le système ou encore le fait de savoir si un système proche doit également être testé.

Établissement du verdict

Une fois les tests concrétisés, ceux-ci sont exécutés sur le système afin d'obtenir un verdict pour chacun d'entre eux. Le verdict est calculé par comparaison des observations réalisées sur le système et des oracles fournis par le modèle. Dans les cas simples, cette comparaison est réalisée à l'aide d'une relation de conformité. Cette relation est en général définie par un test d'égalité de valeurs, mais il est tout à fait possible de mettre en place une relation de comparaison plus complexe. Par exemple, si pour des raisons d'abstraction, l'oracle fourni

par le modèle pour une valeur donnée consiste en une fourchette de valeurs, la comparaison peut-être effectuée par des inégalités.

En plus de la problématique du verdict, nous pouvons citer la problématique de l'ordonnancement des tests. En effet, dans certains cas, l'ordre d'exécution des tests peut se révéler important. Parmi ces cas, nous pouvons donner deux exemples :

- Dans le cas où certains tests placent le système dans un état de blocage, ou plus généralement dans un état qui n'est pas réversible, il peut être intéressant d'exécuter ces cas de tests de manière groupée. Car cela permet de minimiser le nombre de réinitialisations (parfois coûteuses) du système qui ne peuvent être réalisées par utilisation d'un postambule.
- Dans le cas où l'on peut déterminer un graphe de dépendance entre les cas de test, il peut être bon d'ordonner les tests de manière à arrêter l'exécution des tests au plus tôt lors de la détection d'une non-conformité pour corriger le système. Puis de relancer uniquement un sous-ensemble de tests une fois le problème résolu.

Analyse des résultats de l'exécution des tests

Suite à l'exécution des tests sur le système, chacun des tests dont le verdict révèle un problème de conformité entre le système et le modèle doit donner lieu à un travail d'analyse qui a pour objectif de déterminer d'où provient ce problème. Quelques pistes sont envisageables :

- *Une erreur de modélisation* peut entraîner l'apparition d'un faux-négatif, c'est alors le modèle qui n'est pas conforme à la spécification. Normalement, ce type de situation devrait être assez rare, car une bonne pratique du test à partir de modèles consiste à vérifier la cohérence interne du modèle entre sa partie descriptive (invariant) et sa partie comportementale (initialisation et opérations). Mais dans les faits, cette pratique coûteuse est fréquemment éludée, et la possibilité d'une erreur de modélisation n'est donc pas à écarter, d'autant plus que bien souvent, il est difficile de vérifier, sur le modèle, l'ensemble des propriétés énoncées dans la spécification. Il faut donc corriger le modèle afin de le rendre conforme aux spécifications.
- *Une erreur d'implémentation* est le cas adressé par la validation par le test. Le système sous test n'est donc pas conforme à ses spécifications et il faut corriger ce problème.
- *Un problème de précision de la spécification* peut entraîner deux interprétations différentes de celle-ci. Il est donc possible d'avoir développé un modèle et une implémentation qui sont chacun conforme aux spécifications, mais qui ne sont pas conformes entre eux vis-à-vis de la relation de conformité utilisée pour établir le verdict. Dans ce cas, les deux solutions envisageables sont la modification de la relation de conformité ou la modification du modèle (resp. de l'implémentation) afin qu'il concorde à l'implémentation (resp. au modèle).

Lorsque les problèmes de conformité ont tous été identifiés et corrigés, il est nécessaire, d'une part de regénérer les tests si le modèle a été modifié et d'exécuter, d'autre part, ce nouveau jeu de tests sur l'implémentation. Ces deux activités visent à valider le fait que toutes les corrections nécessaires ont bien été réalisées, qu'elles sont effectives et que l'on n'a pas introduit de nouvelles erreurs en modifiant le modèle, l'implémentation ou la relation de conformité.

1.2.5 Intérêts et limites de l'approche MBT

Avantages

L'utilisation d'une approche MBT, si elle peut sembler fastidieuse en premier lieu (rédaction des modèles, sélection des tests, concrétisation des tests), a en réalité beaucoup d'avantages. Tout d'abord, elle apporte un gain de temps (et donc d'argent) pour la réalisation et la validation d'un système. Ce gain est en grande partie dû à la rationnalisation de la démarche de validation qui permet son automatisation partielle. A partir d'un modèle du système, cette automatisation permet de générer des séquences de tests complètes qui contiennent à la fois les séquences d'appels au système et les oracles des tests qui servent à établir le verdict. L'utilisation d'un modèle procure d'autres avantages tels qu'un transfert de connaissance facilité et une meilleure adaptabilité, puisqu'il est moins coûteux de modifier un modèle plutôt qu'un listing de cas de tests (potentiellement très nombreux).

L'emploi de critères de sélection de tests permet de justifier le degré de couverture du modèle par les tests qui ont été générés. De plus, leur emploi assure une génération systématique des tests définie par le critère et évite donc certains oublis qui sont le lot de tout processus manuel.

Comme exposé dans [BDLN06] l'utilisation de modèles n'est pas limitée à l'activité de validation, mais peut également être précieuse pour la conception. Ainsi, un même effort de modélisation peut être en partie factorisé pour deux objectifs différents.

Dans une autre optique, un modèle peut également être utilisé par animation manuelle, ou vérification de propriétés dans l'objectif d'augmenter la confiance que l'on peut avoir dans sa conformité avec les spécifications du système. Par conséquent, l'animation augmente le degré de confiance que l'on peut avoir dans les oracles produits lors de la génération des tests.

Enfin, nous concluons cette partie sur le constat que les modèles ne sont pas figés dans le temps contrairement à des suites de tests rédigées manuellement pour un système donné. Ainsi, si le système évolue, le modèle peut évoluer de la même manière pour prendre en compte ces changements. Il en va de même si l'on souhaite réutiliser un modèle pour valider un système proche de celui pour lequel il a été conçu.

Limites

Malgré tous ces avantages, l'approche de validation de systèmes par génération de tests à partir de modèles souffre de quelques limites qui sont essentiellement liées à la complexité des systèmes à valider. Dans le cadre d'applications de taille industrielle, les comportements nominaux des systèmes sont souvent complexes, et cette complexité est généralement décuplée par la gestion de l'environnement des systèmes. Á cause de cette complexité, l'approche MBT souffre de limitations technologiques humaines et méthodologiques.

Les *limites technologiques* sont directement liées à la complexité de systèmes complexes qui peuvent possèder un très grand nombre d'états, éventuellement infini. Même une fois abstrait au cours de la modélisation, de tels systèmes posent des problèmes d'explosion combinatoire. L'explosion du nombre d'états potentiels dans lesquels un système peut se trouver pose des problèmes de temps de calcul et d'espace mémoire importants. Bien que ces problèmes soient pris en compte dans les différentes technologies liées à la génération de tests, ils constituent toujours un obstacle au niveau du passage à l'échelle sur des systèmes de plus en plus complexes.

Les limites humaines résident dans le niveau de compétence des ingénieurs validation. En plus de maîtriser un ou plusieurs langages de modélisation, afin de formaliser un système, ceux-ci doivent gérer bien d'autres aspects. Parmi ces préoccupations, deux sont cruciales. La première découle des limites technologiques, ainsi l'ingénieur validation doit avoir une connaissance assez fine des technologies sous-jacentes utilisées pour la génération des tests, afin de faire des choix de modélisation qui seront en adéquation avec les capacités des outils qui traiteront les modèles et la couverture du système par les tests que l'on souhaite obtenir. La seconde réside dans la connaissance d'autres méthodes et outils orientés vers la vérification. En effet, pour être en mesure d'assurer que le modèle qui a servi à générer des tests est bien conforme aux spécifications du système testé, il est parfois nécessaire de passer par la vérification de propriétés sur le modèle par des techniques de preuve ou de model-checking, par exemple.

La qualité de la traçabilité des tests est une *limite méthodologique* due à l'utilisation d'un critère de sélection uniquement basé sur l'analyse syntaxique du modèle. Les tests ainsi produits ne peuvent pas être liés à des propriétés de haut niveau (par exemple des propriétés temporelles). Ce manque de traçabilité est pénalisant pour l'analyse et la présentation des résultats de l'exécution des tests.

1.3 Problématique et contributions

Notre approche est une extension des travaux déjà réalisés autour de la génération de test utilisant un critère de sélection de test basé sur la couverture structurelle des opérations d'un modèle (c.f. partie 2.2.1) et implémentés dans les outils BZTT et LTG (c.f. partie 2.3.2). Les implémentations actuelles de ces deux outils reposent sur l'utilisation de critères de couverture de comportements (basés sur des critères de couverture du graphe de flot de contrôle du modèle) et sur la couverture des variables et des paramètres aux limites de leurs domaines de définition. L'utilisation de ces critères de couverture trouve ses limites face à des modèles de taille industrielle qui obligent à faire des compromis lors de la modélisation et du paramètrage des critères de couverture du modèle pour éviter des problèmes d'explosion combinatoire du nombre de test à générer. Dans ce cadre, les deux problématiques principales que nous avons adressées sont :

- la mise en place d'une solution de génération de tests à partir de propriétés dynamiques destinées à compléter la génération de tests fonctionnels (tels qu'ils peuvent être produits par LTG) ;
- la maximisation de la réutilisation du matériel mis en place pour le test fonctionnel afin de valoriser l'effort réalisé par l'ingénieur validation.

En réponse à ces deux problématiques, nous présentons quatre contributions :

- la définition d'un langage de description d'objectifs de tests ;
- la conception d'un processus de génération de tests à partir de ces objectifs ;
- la définition de critères de couverture de tests ;
- l'application de nos travaux à différentes études de cas.

En premier lieu, nous avons défini un *langage de description d'objectifs de tests* qui permet de décrire de manière simple des ensembles de scénarios de test au travers d'enchaînements d'appels d'opération et d'états cibles. Les différentes constructions de ce langage, basé sur le langage des expressions régulières, permettent de décrire de manière concise un grand nombre de scénarios destinés à couvrir un objectif de validation. L'utilisation de description d'états

du système au travers de prédicats exprimés sur les variables du modèle permet d'apporter de la précision dans la description des objectifs de tests. Certaines constructions du langage ont pour objectif de "piloter" la génération des tests afin de maîtriser les problèmes d'explosion combinatoire.

Nous avons conçu un *processus de génération de tests* à partir de ces objectifs de tests qui permet la valorisation des efforts déployés pour la génération automatique de tests fonctionnels à partir de modèles. La réutilisation du modèle destiné à la génération de tests fonctionnels, sans modification de ce dernier, permet également la réutilisation des solutions de concrétisation et d'exécution des tests fonctionnels.

Dans l'objectif d'évaluer d'une part les tests produits grâce à l'utilisation de schémas de tests et d'autre part leur complémentarité vis-à-vis des tests fonctionnels, nous proposons une *méthode d'évaluation de la couverture des tests*. Cette méthode permet d'évaluer la couverture des tests en terme d'états, de transitions et de paires de transitions d'une abstraction du modèle du système. L'utilisation d'une abstraction permet de rendre cette évaluation possible sur un modèle complexe constitué d'un grand nombre d'états et de transitions.

Enfin, nous avons expérimenté notre approche dans le cadre de trois *études de cas* différentes afin d'évaluer sa pertinence et sa faisabilité. Ces trois études de cas, qui concernent une application de type carte à puce, un porte-monnaie électronique et un système de gestion de fichiers, nous ont permis de nous confronter à différentes situations de contrôle d'accès, de gestion de cycle de vie ou d'organisation de données (arborescence de fichiers).

1.4 Plan du mémoire

Ce mémoire est organisé en 8 chapitres et 5 parties dont la structure est la suivante :

- La **partie I** présente le contexte et les problématiques qui ont motivé nos travaux. Cette partie est composée de trois chapitres :
  - Le **chapitre 1** présente le contexte de nos travaux, une description de l'activité de validation par Model Based Testing et une présentation des problématiques adressées par nos contributions.
  - Le **chapitre 2** présente un aperçu des techniques de génération tests à partir de modèles et de critères sélection de tests "statiques" basés sur la couverture structurelle du modèle du système à valider.
  - Le **chapitre 3** présente un aperçu des techniques de génération de tests à partir de modèles et de critères de sélection de tests "dynamiques". En plus des informations structurelles contenues dans le modèle, ces critères permettent d'exploiter d'autres informations issues des spécifications et fournies par l'ingénieur validation . Ce chapitre se conclut par un positionnement de nos travaux.
- La **partie II** est consacrée à la présentation de nos contributions. Celles-ci sont présentées dans trois chapitres :
  - Le **chapitre 4** présente le langage de schéma de test, destiné à décrire des d'objectifs de test, que nous proposons comme critère "dynamique" de sélection de tests.
  - Le **chapitre 5** présente les solutions de génération de tests implémentées pour les outils LTG et BZ-TT afin qu'ils puissent prendre en compte un schéma de test comme critère de sélection.
  - Le **chapitre 6** présente, dans un premier temps les solutions de concrétisation et d'exécution des tests. Dans un second temps, nous présentons le problème d'évaluation

des ensembles de tests produits, et notre contribution concernant ce sujet.
- La **partie III** présente, au travers du chapitre 7, les trois expérimentations que nous avons menées dans le but d'évaluer la démarche que nous proposons.
- La **partie IV** termine ce mémoire avec le chapitre 8 qui présente les conlusions de nos travaux et les perspectives envisageables pour les étendre.

Chapitre 2

Model Based Testing

Sommaire

Ce chapitre présente différents travaux concernant la génération de tests à partir de modèles. Ces approches consistent à appliquer un critère de sélection de tests qui permet de déterminer quelles informations présentes dans le modèle doivent être couvertes par les tests générés. Nous présentons d'abord différents paradigmes de modélisation et différents critères statiques de sélection de tests. Nous introduisons les critères dynamiques de sélection qui sont présentés en détail dans le chapitre suivant. Enfin, nous concluons par la présentation d'outils appliquant ces différents critères de sélection aux différents paradigmes de modélisation présentés.

2.1 Les modèles pour le test

Dans cette partie, nous introduisons les trois types de formalismes principaux sur lesquels reposent les modèles présentés dans les travaux que nous présentons. Il s'agit des modèles d'états/transitions, des modèles d'états/transitions symboliques et des modèles comportementaux (à sémantique Pré/Post).

2.1.1 Les modèles d'états/transitions

Les systèmes de transitions étiquetées

Dans cette partie, nous abordons le test de conformité pour les systèmes modélisés à l'aide de systèmes de transitions étiquetées (ou LTS – Labelled Transition Systems). Nous nous intéressons plus particulièrement aux systèmes de transitions étiquetées à entrées et sorties [Tre96] (également nommés IOLTS – Input/Output Labelled Transition System) qui sont largement utilisés dans le cadre du test de conformité. Par rapport aux LTS, les IOLTS sont particulièrement bien adaptés à la modélisation de système réactifs, car ils distinguent les actions contrôlables des actions observables.

Nous rappelons, d'abord, quelques notions et notations sur les systèmes de transitions étiquetées, leur enrichissement par la distinction des entrées/sorties et par la notion de quiescence. Puis, nous présentons la relation de conformité *ioco* utilisée pour établir le verdict de conformité entre le modèle et le système sous test. Ensuite nous définirons la notion de cas de test.

Le formalisme des systèmes de transitions étiquetées (déf. 1) permet de modéliser un système sous la forme d'ensembles d'états (dont un état particulier est l'état initial du système), de nom d'actions (parmi lesquelles nous pouvons distinguer l'action interne τ) et de transitions. La modélisation d'un système et de son environnement peut être décrite par un LTS, mais dans ce cas, il n'y a aucune différenciation entre les émissions et les réceptions. Or, dans le cadre du test, il est important de séparer ce qui peut être contrôlé de ce qui peut être observé. C'est pourquoi l'enrichissement des LTS en IOLTS (déf. 2) par une signature des actions est utile. On peut noter que les ensembles des noms d'actions d'entrée et de sortie sont disjoints.

Définition 1 (Système de transitions étiquetées – ou LTS)**.** Un LTS est défini par un quadruplet $< Q, q_0, A, \rightarrow >$ tel que :
- Q est un ensemble dénombrable et non vide d'états,
- $q_0 \in Q$ est l'état initial du système,
- $A \cup \{\tau\}$ est un ensemble de noms d'actions, l'action spéciale $\tau \notin A$ caractérisant les actions internes (non-observables du système). On note A_τ pour $A \cup \{\tau\}$.
- $\rightarrow \subseteq Q \times A_\tau \times Q$ est la relation de transition. On note $q \xrightarrow{\mu} q'$ pour $(q, \mu, q') \in \rightarrow$.

Définition 2 (Système de transitions étiquetées à entrées et sorties – ou IOLTS)**.** Un IOLTS est un LTS dont l'ensemble d'actions A est scindé en deux ensembles d'actions A_I et A_O afin de différencier les actions de sorties des actions d'entrée. $A = A_I \cup A_O$ et $A_I \cap A_O = \varnothing$ avec A_I l'ensemble des actions d'entrée (réceptions) et A_O l'ensemble des actions de sortie (émissions). Les actions d'entrée sont activables depuis n'importe quel état, le modèle est dit "complet en entrées".

La notion de quiescence (déf. 3) permet d'expliciter les situations normales de blocage du système. Parmi ces situations de blocage, nous pouvons distinguer :
- Les blocages de sortie (ou outputlock) : ces situations de blocage se présentent dans un état à partir duquel aucune action d'émission et aucune action interne ne sont spécifiées.
- Les "deadlock" : ces situations de blocage se présentent dans un état à partir duquel aucune transition n'est activable.
- Les "livelock" : ces situations se présentent lorsque qu'il existe un chemin (non vide) d'actions internes qui part d'un état et aboutit au même état.

Cette quiescence est matérialisée par l'action spéciale δ. Dans la pratique, l'observation de la quiescence à l'exécution d'un cas de test est réalisée par l'expiration d'un "timer".

Définition 3 (Quiescence)**.** Soit un IOLTS $S =< Q, q_0, A, \rightarrow >$, un état $q \in Q$ est dit quiescent (noté $\delta(q)$), si $\forall \mu \in A_O \cup \{\tau\} : q \overset{\mu}{\not\rightarrow}$. On note S^δ l'IOLTS quiescent associé à S. Pour obtenir S^δ, on ajoute l'action de quiescence δ à A et des transitions $q \overset{\delta}{\rightarrow} q$ pour tout les états $q \in Q$ qui sont quiescents.

Afin de confronter le système sous test à son modèle, la relation de conformité formalise l'ensemble des implémentations correctes vis-à-vis du modèle. La relation de conformité *ioco* introduite par Tretmans [Tre96] est basée sur l'inclusion de traces. Ainsi une implémentation I est conforme à sa spécification S (pour la relation de conformité ioco), si pour toute séquence σ de S^δ incluant les blocages, les sorties de I (et donc les blocages) sont inclues dans celles prévues par S^δ.

Un cas de test est un IOLTS dont les états ont été marqués par **Pass**, **Fail** ou **Inconc**. Ce marquage permettra d'établir le verdict :

- *Pass* pour les états appartenant à une séquence d'exécution du système conforme aux spécifications,
- *Fail* pour les états appartenant à une séquence d'exécution du système non conforme aux spécifications,
- *Inconc* pour les états appartenant à une séquence d'exécution du système dont on ne peut pas déterminer la conformité avec les spécifications.

L'exécution d'un cas de test est réalisée par composition parallèle de ce dernier avec le système sous test. Le verdict est donné par le marquage de l'état atteint par une trace complète de la composition parallèle du cas de test et du système sous test. L'exécution des tests peut être effectué *off-line* ou *on-line*. Dans le cas de l'exécution *off-line* des tests, ceux-ci sont d'abord tous générés à partir du modèle puis exécutés sur le système. Par contre, dans le cas du test *on-line*, les cas de tests sont générés parallèlement à leur exécution sur le système, cette approche permet de conditionner la production des tests au déroulement de leur exécution, et par conséquent de résoudre des problèmes d'indéterminisme (du modèle ou du système).

Pour être complète, la génération de cas de test doit consister en l'extraction de toutes les traces du modèle menant à une situation de blocage. En pratique, on se trouve donc rapidement confronté à des situations d'explosion combinatoire dues au nombre d'états, de branchements, de boucles qui peuvent dans le pire des cas aboutir à des traces infinies.

Les machines à états finis – FSM

Les machines à états finis (def. 4) sont très proches des systèmes de transitions étiquetées. Comme le montre la définition 4, les différences résident dans le fait que les ensembles d'états et de transitions sont finis et qu'un ensemble d'états d'acceptation permet de déterminer les exécutions correctes du système.

Définition 4 (Machine à états finis déterministe – FSM)**.** Une machine à états finis et un quintuplet $< Q, q_0, A, \rightarrow, F >$ où :

- Q est un ensemble fini d'états,
- $q_0 \in S$ un état initial,
- A est un ensemble fini d'actions,
- $\rightarrow \subseteq Q \times A \rightarrow Q$ une fonction de transition,

– $F \subseteq Q$ l'ensemble des états finaux.

On peut noter qu'une machine à états finis n'est pas forcément déterministe comme présenté dans la définition 4. La fonction de transition est alors de la forme $\rightarrow: Q \times A \rightarrow \mathcal{P}(Q)$.

L'utilisation de systèmes d'états et de transitions (FSM ou LTS) trouve vite ses limites dans le cadre de la modélisation de systèmes de taille industrielle. Ceci est dû au fait que toutes les données doivent être instanciées ce qui provoque un problème d'explosion combinatoire du nombre d'états.

2.1.2 Les modèles d'états/transitions symboliques

Une des solutions proposée pour répondre au problème d'explosion du nombre d'états est l'utilisation de modèles symboliques. Un modèle symbolique est un LTS muni de variables de données permettant d'abstraire un système.

Systèmes de transitions étiquetées symboliques

Les systèmes de transitions étiquetées symboliques (STES) ou STS (Symbolic Transition System) sont un enrichissement des systèmes de transitions étiquetées par une prise en compte explicite du flot de données. La définition d'un STS (def. 5), issue de [FTW05], montre que l'enrichissement des LTS en STS réside dans l'introduction de :

– variables de données,
– paramètres d'action,
– gardes sur les transitions (exprimées sur les variables de données et les paramètres)
– termes exprimant les modifications des variables de données lors des transitions.

Définition 5 (Système de transitions étiquetées symbolique)**.** Un STS est un t-uple $< Q, q_0, \mathcal{V}, \iota, \mathcal{I}, A, \rightarrow >$ tel que :

– Q est un ensemble d'états,
– $q_0 \in L$ est l'état initial,
– $\mathcal{V}$ est un ensemble de variables de données,
– ι est une initialisation des variables de données,
– $\mathcal{I}$ est un ensemble de paramètres et $\mathcal{I} \cap \mathcal{V} = \varnothing$,
– A est un ensemble de noms d'actions,
– $\rightarrow \in Q \times A \times \mathfrak{F}(\mathcal{V} \cup \mathcal{I}) \times \mathfrak{T}(\mathcal{V} \cup \mathcal{I})^{\mathcal{V}} \times Q$ est une relation de transition. $\mathfrak{F}(\mathcal{V} \cup \mathcal{I})$ est un ensemble de formules logiques du premier ordre sur les variables et les paramètres (garde de l'action). $\mathfrak{T}(\mathcal{V} \cup \mathcal{I})^{\mathcal{V}}$ est un ensemble de termes sur les variables et les paramètres exprimant la modification des variables de données par l'action.

La sémantique d'un STS est un LTS [FTW05, MDJ08]. L'utilisation de systèmes de transitions étiquetées symboliques pour la génération de test nécessite d'utiliser d'autres techniques que le simple parcours de graphe. Ainsi par exemple, l'animation symbolique du modèle peut permettre d'extraire des cas de tests symboliques qui seront valués par résolution de contraintes.

2.1.3 Les modèles Pré/Post

Le langage B

Le langage B [Abr96] est un langage de modélisation basé sur la logique ensembliste. Un système est modélisé sous la forme d'une machine abstraite (ou d'un ensemble de machines abstraites dans le cadre de la composition de machines et du processus de raffinement). Une machine B est constituée de différentes clauses dont les principales sont :

SETS : déclaration d'ensembles constants

CONSTANTS : déclaration de constantes

PROPERTIES : propriétés sur les constantes

VARIABLES : déclaration des variables d'état

INVARIANT : définition des propriétés invariantes des variables d'état du système

ASSERTIONS : définition de propriétés découlant de l'invariant

INITIALISATION : initialisation des variables

OPERATIONS : opérations modélisant la dynamique du système

Les actions du système sont définies par les opérations sous forme de substitutions généralisées. Parmi les opérateurs de substitution, nous trouvons la substitution gardée (SELECT), les substitutions de choix déterministes (IF, CASE), de choix indéterministe borné (CHOICE) ou de choix indéterministe non borné (ANY), les substitutions d'affectation (:=, :∈), la substitution sans effet (skip) et la substitution parallèle (||) qui permet de composer deux substitutions en parallèle. La figure 2.1 présente une machine B modélisant des permutations de valeurs dans un tableau (les explications apparaissent en commentaire dans la figure).

Le langage B a été introduit dans le cadre de la méthode B [Abr96] qui vise au développement d'un logiciel, à partir de ses spécifications, par étapes successives de raffinement et de preuve. Le système est d'abord formalisé à un niveau d'abstraction élevé, puis il est raffiné par étapes successives afin d'arriver au niveau d'abstraction de l'implémentation à partir duquel le code de l'implémentation peut être produit. Au cours de ce processus, la preuve permet :

- de vérifier la cohérence d'une machine abstraite aux travers des obligations de preuves permettant de prouver le respect de l'invariant par l'initialisation des variables du système et sa préservation par les subsitutions effectuées par les opérations,
- de vérifier le raffinement, c'est-à-dire la préservation des propriétés abstraites et la satisfaction des propriétés invariantes introduites au cours du raffinement.

Par la suite de nombreux outils de preuve, model-checking et de génération de tests ont été développés autour du langage B, dont quelques-uns sont présentés dans la section 2.3.2.

JML

Java Modeling Language[LBR99, LC06] est un langage d'annotations destiné aux programmes Java. Il permet de modéliser une spécification à l'intérieur d'un programme sous la forme d'annotations qui peuvent être :

- des invariants de classe,
- des pré-conditions et des post-conditions sur les méthodes,
- des invariants et variants de boucles.

MACHINE
permut

CONSTANTS
/* Déclaration des données : tableau et nombre d'élément maximum */
$TAB, MAXI$

PROPERTIES
/* ''Typage'' de la constant MAXI */
$MAXI \in \mathbb{N} \wedge MAXI > 0$

/*
* ''Typage'' du tableau : représentation sous la forme
* d'une fonction totale
*/
$TAB \in 1..MAXI \rightarrow \mathbb{N}$

VARIABLES
/* Déclaration du tableau résultat */
$newTab$

INVARIANT
/* ''Typage'' du tableau résultat */
$newTab \in 1..MAXI \rightarrow \mathbb{N} \wedge$

/*
* Propriété invariante exprimant le fait que newTab est une permutation
* de TAB par l'existance d'une bijection r qui composée avec TAB (r;TAB)
* formalise cette permutation
*/
$\exists(r).(r \in 1..MAXI \rightarrowtail 1..MAXI \wedge (r;TAB) = newTab)$

INITIALISATION
/* Initialisation du tableau résultat */
$newTab := TAB$

OPERATIONS
/* Opération de permutation de deux valeurs */
$permut(i) \triangleq$
PRE
/* Précondition sur les paramètres de l'opération */
$i \in 1..MAXI - 1$
THEN
/*
* Permutation, des valeurs aux indices i et i+1, réalisée grâce à
* l'opérateur de surcharge ensembliste $\Leftarrow$
*/
$newTab := newTab \Leftarrow \{i \mapsto newTab(i+1), i+1 \mapsto newTab(i)\}$
END
END

FIG. 2.1 – Exemple de machine B : Permutation de valeurs dans un tableau

```
/*@ requires tab != null && tab.length > 1 && i>=0 && i+1<tab.length;
  @ ensures (\forall int j; ((j>=0 && j<i)||(j>i+1  && j<tab.length)) ==> tab[j]==\old(tab[j]))
  @ && tab[i]==\old(tab[i+1])  && tab[i+1]==\old(tab[i]);
  @ assignable tab[i],tab[i+1];
  @*/
void permut(int[] tab, int i){
    int temp0 = tab[i];
    tab[i] = tab[i+1];
    tab[i+1] = temp0;
}
```

FIG. 2.2 – Exemple d'annotation JML : méthode de permutation

JML repose sur une notion de conception par contrats (*design by contract*), dont l'idée principale est que les interactions entre une classe et ses clients sont régies par des contrats exprimés par des propriétés. Ainsi, le "client" garantit le respect de certaines contraintes lors de l'appel d'une méthode d'une classe (respect des pré-conditions de la méthode) ; en retour, la classe garantit certaines propriétés (exprimées dans la post-condition). Les propriétés sont exprimées en logique du premier ordre sur les attributs de la classe et les paramètres des méthodes ou fonctions. Par exemple, la figure 2.2 présente les annotations JML d'une méthode de permutation de deux valeurs dans un tableau d'entiers ($tab[i]$ et $tab[i+1]$) :

- la clause `requires` donne les pré-conditions de la méthode. Elle exprime ici le fait que le tableau ne doit pas être vide et que les indices des deux éléments permutés appartiennent au domaine du tableau.
- la clause `ensures` donne les post-conditions de la méthode. Elle exprime ici le fait que tous les éléments du tableau ont conservé leur valeur sauf les deux élements permutés (dont les valeurs doivent avoir été échangées).
- la clause `assignable` donne les variables dont la valeur peut-être modifiée.

De nombreux outils ont été développés autour de JML, les plus connus étant :

- le compilateur *jmlc* qui permet de compiler les annotations JML dans le bytecode Java permettant ainsi la vérification d'assertions à l'exécution,
- l'outil de test unitaire *JMLUnit* qui combine le compilateur JML et *JUnit* afin d'utiliser les annotations JML pour l'établissement des verdicts,
- l'outil de vérification statique *ESC/Java2*, qui utilise les annotations JML afin de détecter des erreurs,
- le prouveur *Key* basé sur le calcul des séquents qui permet de générer des obligations de preuves à partir des annotations JML et de les prouver de manière automatique ou semi-automatique,

Par ailleurs, nous présentons deux outils de génération de tests à partir de spécifications formalisées en JML : *JML-TT* et Jartege, dans la section 2.3.

AsmL et Spec♯

Les langages AsmL [BGN+03, BGN+04] et *Spec♯* [VCG+08] sont deux langages de modélisation reposant sur les ASM (Abstract State Machine). Les ASM reposent sur la description d'états symboliques et de fonctions de transformation de ces états. Un modèle Spec♯ se présente sous la forme d'annotations dans un programme C♯. Il permet de définir une ASM au travers de la formalisation de pré-conditions et post-conditions. Ces deux langages sont supportés par l'outil de génération de tests *SpecExplorer* (c.f. section 2.3) developpé par Mi-

crosoft.

2.2 Sélection des tests

Nous présentons dans cette section les principaux critères de sélection de tests présents dans la littérature. Nous traitons le point des critères de sélection suivant deux axes, le premier est constitué des critères de sélection statiques, et le second des critères de sélection dynamiques.

2.2.1 Les critères statiques

Critères de couverture du graphe de flot de contrôle

Les critères de couverture des instructions et des *conditions-décisions* permettent de définir des critères de sélection de tests basés sur l'analyse statique des instructions de branchement d'un programme (test boîte blanche) ou d'un modèle (test boîte noire). Les conditions sont des expressions booléennes élémentaires ne contenant aucun connecteur logique ; elles ne peuvent être décomposées. Les décisions sont des expressions booléennes composées contenant une où plusieurs conditions liées par des connecteurs logiques. Les principaux critères de couverture structurelle d'un graphe de contrôle sont :

- *Statement Coverage (SC)* : chaque instruction doit être exécutée au moins une fois ;
- *Decision Coverage (DC)* : chaque décision doit prendre tous les résultats possibles au moins une fois ;
- *Condition Coverage (CC)* : chaque condition doit prendre tous les résultats possibles au moins une fois ;
- *Condition Decision Coverage (C/DC)* : chaque décision et chaque condition doivent prendre tous les résultats possibles au moins une fois ;
- *Modified Condition/Decision Coverage(MC/DC)* : chaque décision prend tous les résultats possibles au moins une fois et l'effet de chaque condition sur sa décision est montré, indépendamment des autres conditions, c'est à dire en fixant la valeur de ces autres conditions dans la décision ;
- *Multiple Condition Coverage (MCC)* : toutes les combinaisons possibles de conditions à l'intérieur des décisions doivent être exécutées au moins une fois.

Il est facile de constater, que parmi les critères *SC, DC, CC C/DC et MCC*, c'est le critère MCC qui offre la meilleure couverture du graphe de flot de contrôle. Mais, le point négatif de ce critère de couverture est qu'il peut amener à une combinatoire très élevée. C'est pourquoi le critère MC/DC a été introduit pour faire face à ce problème. Ces critères de couverture sont présentés, ainsi que leur implémentation dans l'outil BZTT dans [LPU04, BLPT04].

On peut noter que le critère de couverture des décisions (DC) permet d'extraire les comportements d'une opération. Par définition, on considère que les comportements d'une opération sont les différentes exécutions possibles de cette opération en terme d'effets de l'opération (affectation des variables). Ainsi, si l'on souhaite calculer l'ensemble des comportements de l'opération COMMIT_TRANSACTION (validation d'une transaction) présentée dans la figure 2.3, on peut appliquer le critère de couverture DC à son graphe de flot de contrôle (fig. 2.4). On extrait alors trois chemins du graphe de flot de contrôle de cette opération ([1,2,4,6,8], [1,2,5,6,8] et [1,3,6,8]). Chacun de ces chemins correspond à un comportement de l'opération et peut se lire sous la forme d'un prédicat avant/après exprimé sur les variables

du modèle (les noms de variables primés correspondent à l'état des variables après l'opération). Le premier chemin du graphe de flot de contrôle qui est le comportement de succès de l'opération se lit sous la forme du prédicat avant/après suivant :

$$etat\_carte = use \wedge debit \neq 0 \wedge solde' = solde - debit \wedge out = sw\_success \wedge debit' = 0$$

A partir de l'extraction d'un comportement, il est possible de définir des cibles de test qui sont des cas particuliers dérivés de celui-ci. Les cibles de tests peuvent être calculées par application de critères de couverture des conditions qui définissent un comportement ou par un critère de couverture de données sur les variables mises en jeu dans ces conditions. La notion de cible de test est utilisée dans les outils BZ-TT et LTG présentés dans la partie 2.3.2.

```
out <— COMMIT_TRANSACTION =
BEGIN
  IF etat_carte = use THEN
  /* Etat du cycle de vie de la carte cohérent */
    IF debit ≠ 0 THEN
      /* Transaction initiée */
      solde := solde − debit
      || out := sw_Success
    ELSE
      /* Aucune transaction inititiée */
      out := sw_Error_invalid_transaction
    END
  ELSE
    /* Etat de cycle de vie de la carte incohérent */
    out := sw_Error_life_cycle
  END
  /* Réinitialisation de la transaction en cours*/
  || debit := 0
END
```

FIG. 2.3 – Exemple d'opération B

Dans [DF93], les auteurs proposent une méthode de partionnement de l'espace d'états d'entrée des opérations par la transformation de l'expression $OP_{pre} \wedge OP_{post} \wedge Inv$ (avec OP_{pre} la précondition de l'opération, OP_{post} la postcondition de l'opération et Inv l'invariant du système) sous sa forme normale disjonctive (DNF). Puis par la création d'une nouvelle instance de l'opération pour chacune des sous-expressions issue de cette réécriture. Le système est ainsi représenté sous la forme d'une machine à états finis et les tests sont générés par un parcours de cette dernière. Ces travaux sont implémentés dans l'outil ProTest [SLB05].

Critères de couverture sur les graphes d'états/transitions

Afin d'obtenir une couverture exhaustive d'un système représenté sous la forme d'un diagramme d'états/transitions, il faut générer toutes les traces d'exécution possibles menant à des états puits. Or ces traces d'exécutions sont potentiellement infinies, en cas de cycle par exemple. D'autres critères de sélection de cas de tests sont donc nécessaires. Parmi les critères de sélection classiques, on trouve :

- la couverture des états,
- la couverture des transitions,
- la couverture des paires de transitions,
- une couverture aléatoire.

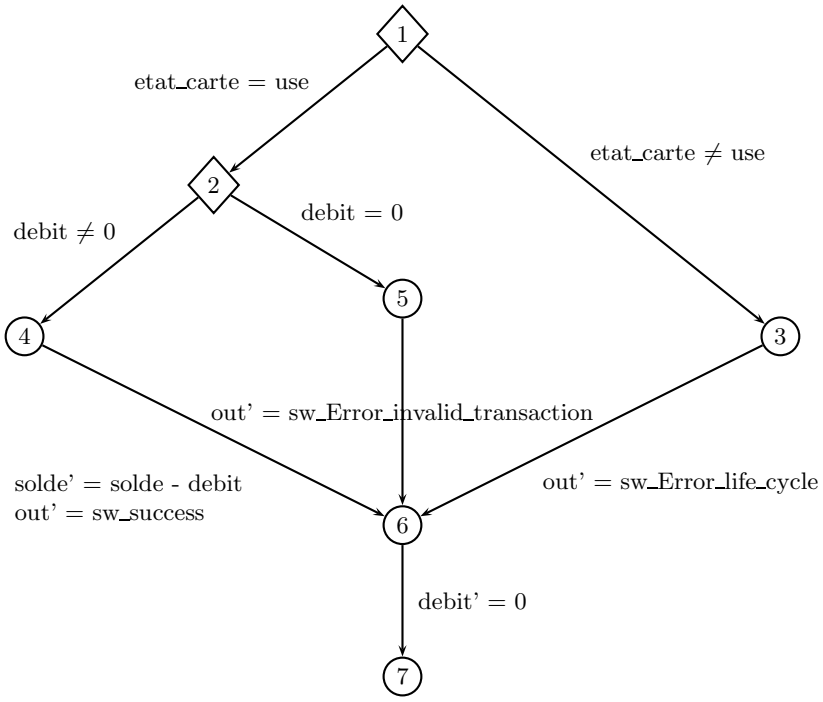


FIG. 2.4 – Exemple de graphe de flot de contrôle

Ces critères de sélection peuvent être mis en œuvre de différentes manières. Par exemple, le critère de couverture de toutes les transitions du système peut donner lieu à la génération d'un ensemble de cas de test dont chacun a pour objectif de couvrir une transition ; mais il peut également donner lieu à la génération d'un cas de test unique visant à mettre en jeu toutes les transitions du système. Ces deux possibilités sont proposées dans l'outil *conformance kit* [MRS+97].

D'autres stratégies peuvent également être appliquées comme la limitation de la longueur (en nombre de transitions) des cas de test produits, ou une limitation du nombre maximal de dépliage des cycles, ou une stratégie de réduction du nombre de transitions sortantes d'un état exploré. Ces stratégies sont implémentées dans l'outil TORX [TB03].

Critères de couverture de données

Dans [LPU02, LPU04], les auteurs proposent des critères de sélection de test basés sur la couverture des données aux limites. Pour chaque cible de tests (définies à partir des comportements du système à tester), les données (paramètres et/ou variables d'état) seront valuées aux bornes de leurs intervalles de définition. Pour déterminer ces valeurs limites, une fonction d'optimisation est appliquée sur les données, et on cherche soit à minimiser, soit à maximiser le résultat suivant la borne que l'on vise. L'ensemble des variables en jeu dans le prédicat définissant le comportement à tester est représenté sous la forme d'un vecteur de variables V_i. On va chercher à résoudre les systèmes de contraintes $minimize(f(V_i), Inv \wedge P)$ et $maximize(f(V_i), Inv \wedge P)$ où Inv est l'invariant du système, P le prédicat décrivant le comportement et f la fonction d'optimisation. La fonction d'optimisation est différente se-

lon le type des variables présentes dans V_i, par exemple pour des variables entières, ce sera la somme de leurs valeurs, pour des ensembles ce sera la somme de leurs cardinalités, et pour des couples ce sera la somme des cardinalités des domaines (ou des co-domaines). On peut noter, que pour un même vecteur de variables, il est possible d'avoir plusieurs valeurs limites minimales ou maximales. Il est donc possible de limiter le nombre de valuations aux limites d'un vecteur de variables ; de même qu'il est également possible d'inclure au vecteur des variables n'entrant pas en jeu dans le prédicat définissant le comportement.

Critères de couverture aléatoire ou probabilistes

Le hasard est également un critère de sélection de test, il est par exemple utilisé dans l'outil Jartege [Ori05]. Le critère aléatoire peut-être utilisé pour définir les séquences d'opérations constituant un cas de test et les valeurs des paramètres des opérations de la séquence. L'utilisation d'un modèle du système à tester dans le cas de la génération aléatoire de cas de test offre deux avantages non négligeables :

- la possibilité de filtrer les cas de test pour ne conserver que les séquence d'opérations dont les préconditions sont respectées,
- la possibilité de calculer un oracle à partir des post-conditions des opérations.

En général, le test aléatoire n'est pas uniquement basé sur un critère de hasard. D'autres critères de sélection de cas de test sont appliqués de manière à obtenir une couverture "intelligente" du système. Par exemple, en partionnant l'espace d'état du système et en assurant une couverture uniforme de ces états par les différents cas de test, on assure la couverture de certains comportements peu probables mais néanmoins intéressants à tester.

2.2.2 Les critères dynamiques

Contrairement aux critères de sélection de tests statiques que nous avons présentés, les critères de sélection de tests que nous qualifions de "dynamiques" ne reposent pas uniquement sur l'analyse syntaxique du modèle du système à tester, mais également sur des informations supplémentaires, fournies par l'utilisateur, exprimant son objectif de validation. Plusieurs points justifient l'utilisation de critères de sélection dynamiques :

- Les critères de sélection dynamiques donnent la possibilité de modéliser des objectifs de tests ciblant des propriétés de haut niveau telles que des propriétés temporelles qui ne sont pas couvertes par l'utilisation de critères de sélection statiques. Par exemple, certains enchaînements d'appels d'opérations.
- L'utilisation des critères de sélection statiques nécessite un compromis constant entre deux problèmes qui sont la couverture du système par les tests et le problème d'explosion combinatoire. Les critères dynamiques n'éliminent pas le problème de l'explosion du nombre de tests, mais ils offrent des outils permettant de gérer cette explosion combinatoire plus finement.
- L'utilisation de critères systématiques basés sur le modèle du système à tester pose également le problème de la traçabilité, c'est à dire la connaissance du lien entre un cas de test généré et une exigence de conformité exprimée sous la forme d'une propriété de haut niveau (par exemple une propriété temporelle).

Nous abordons plus précisément ces points et les différentes solutions déjà proposées dans le chapitre 3.

2.3 Les outils

2.3.1 Modèles états/transitions

L'outil TorX [TB03], implémente quelques critères statiques de sélection de test pour des systèmes modélisés sous la forme d'IOLTS. Ces critères de sélection sont basés sur des heuristiques de parcours de l'IOLTS représentant le système. Les trois heuristiques proposées pour la sélection des tests sont :

- l'heuristique de *"longueur des cas de test"* permet de considérer uniquement les traces de longueur inférieure à une constante donnée,
- l'heuristique de *"longueur des cycles"* permet de ne considérer qu'un nombre donné d'itérations sur les cycles,
- l'heuristique de *"réduction"* permet de ne sélectionner qu'un nombre fini de transitions sortantes d'un état.

Les heuristiques proposées permettent en partie de réduire le problème d'explosion combinatoire, mais celui-ci reste présent à cause des données possèdant un large domaine de définition. Les auteurs de [TB03] proposent donc comme perspective une représentation symbolique des données pour résoudre ce problème.

Dans le domaine de la génération de tests à partir de FSM, nous pouvons également citer *"Conformance Kit"* développé par "Dutch PTT" et son extension *"Phacts"* développée par "Philips Research Laboratories Eindhoven". Cet outil prend en entrée un modèle sous forme d'une EFSM et la transforme en FSM pour générer des tests. Les critères de sélection de tests appliqués permettent de générer :

- soit un unique test parcourant toutes les transitions du modèle,
- soit un ensemble de tests dont chacun active une transition du modèle.

Cet outil, n'étant pas universitaire, il n'a pas fait l'objet de publications, mais il est cité dans un retour d'expérimentation [MRS+97], et dans un article de veille technologique consacré aux outils de génération de tests à partir de modèle d'états/transitions [BFS05].

2.3.2 Modèles pré/post

ProTest [SLB05] est un outil de génération de test basé sur l'outil de model checking *ProB* [LB08]. ProB est un ensemble d'outils dédiés à la validation dans le cadre de la *méthode B*. ProB couvre une grande part du *langage B classique*, il couvre également la syntaxe du *B événementiel* (utilisé pour modéliser des systèmes réactifs). ProB offre des fonctionnalités de vérification de consistance de machines B et de raffinements. Les techniques de model checking employées imposent quelques restrictions. Les domaines des variables doivent être finis et de tailles réduites (ensembles contenant peu d'éléments, intervalles d'entiers réduits) afin de limiter l'explosion combinatoire. Les fonctionnalités offertes par ProB ne sont pas une alternative aux techniques de preuves (implémentées dans *l'Atelier B*[3], *B4Free*[4] ou *Rodin*[5]) dans la mesure où le model checking exhaustif d'une machine n'est envisageable que pour des modèles restreints. Par contre, l'utilisation de ProB peut être intéressante comme une phase préalable à la preuve, car c'est un moyen de détection d'inconsistances simple d'utilisation. Dans ce cadre, l'utilisation de ProB peut éviter de débuter par une phase de preuve

[3] http ://www.atelierb.eu/
[4] http ://www.b4free.com/
[5] http ://www.event-b.org/

potentiellement fastidieuse qui échouera (pour une inconsistance qui aurait pu être découverte par model checking) et qui sera à réitérer une fois l'erreur corrigée. ProTest partionne l'espace d'états à explorer en calculant la DNF (Disjunctive Normal Form) de la précondition pour chaque opération. Puis pour chaque opération de la forme *IF* <*cond*> *THEN* ... *ELSE* ... *END*, le prédicat <*cond*> $\vee \neg$ <*cond*> est ajouté à la précondition (conjonction), puis la précondition résultante et recalculée sous forme DNF. Enfin, pour une opération dont la précondition a été ainsi transformée, une nouvelle instance de l'opération est créée pour chacune des disjonctions (pour chaque instance de l'opération, la précondition est constituée d'une des disjonctions). Une fois le modèle ainsi modifié, ProB est utilisé pour explorer l'espace d'états de la machine afin de créer un FSM. L'objectif est de faire apparaître au moins une instance de chaque opération dans la FSM. Les tests sont générés par un parcours de la FSM ainsi construite.

BZ-TT et sa déclinaison industrielle *LTG* sont deux outils de génération de tests principalement orientés pour utiliser des modèles décrits en langage B. LTG est issu d'un transfert de technologie du Laboratoire d'Informatique de l'Université de Franche-Comté vers la société Leirios (appelée aujourd'hui Smartesting[6]). La génération de tests à l'aide de l'outil BZ-TT [LPU02] repose sur deux critères de sélection qui sont la couverture du graphe de flot de contrôle des opérations du modèle (critères DC,C/DC, MC/DC et MCC) et la couverture des variables d'états et des paramètres aux limites de leurs domaines [LPU04]. D'un point de vue technologique, BZ-TT s'appuie sur un animateur symbolique basé sur un solveur de contraintes. L'utilisateur peut choisir le type de couverture du graphe de flot de contrôle qu'il souhaite appliquer au modèle et également sélectionner les variables et/ou les paramètres dont il veut la couverture des domaines aux limites, afin de limiter l'explosion combinatoire. De nombreux cas d'études ont été traités avec l'un ou l'autre des deux outils, dont un certain nombre d'applications de tailles industrielles notamment dans le domaine des cartes à puces [BLPT04]. Un autre aspect intéressant de ces outils réside dans le calcul de préambule. En effet, lorsque les critères de sélection sont appliqués pour une opération du modèle, on obtient une cible de test. Une cible de test est constituée de deux prédicats décrivant les états du système (au travers des états des variables et des paramètres) avant (contexte) et après (effet) l'exécution de l'opération. Or l'état initial du système n'est donc pas forcément compatible avec le contexte d'une cible de test donnée, il faut donc trouver une séquence d'opérations menant le système dans un état compatible avec le contexte de la cible de test. Certaines stratégies de calcul de préambules ont donc été développées, dont des heuristiques de recherche avant (et arrière) avec heuristique du meilleur d'abord [CLP04].

L'outil *B-CASTING* [VABM97] est une instanciation de l'outil de génération de test CASTING pour le langage B. La sélection des tests est réalisée par analyse statique du modèle. A partir de cette analyse, des spécifications de tests sont générées (sous la forme de prédicats avant/après). La génération des tests est réalisée en deux étapes :

- Un graphe, dont les nœuds symboliques représentent les états du système et les transitions représentent les spécifications de cas de test, est calculé.
- Ensuite, des chemins sont extraits de ce graphe et instanciés par résolution de contraintes afin de produire des séquences de test.

L'outil de génération de tests *JML-TT* [BDL06, Dad06] a été développé sur la même base technologique que BZ-TT. Cet outil permet ainsi de générer des tests à partir d'un programme java annoté en JML. Les critères de couverture utilisés sont les mêmes que ceux

[6] `http ://smartesting.com`

mis en place dans BZ-TT, c'est-à-dire les critères de couverture du graphe de flot de contrôle des opérations et la valuation des données aux limites. Bien que les annotations JML soient directement présentes dans le code JAVA, cet outil trouve sa place dans un contexte de test boîte noire, puisque la connaissance du code de l'application à tester n'est pas un préalable. En effet, les comportements sont extraits directement des annotations JML et les oracles sont fournis par ces mêmes annotations.

L'outil *Jartege* [Ori05] (Java Random Test Generator), permet de générer aléatoirement des cas de test unitaires à partir de classes Java annotées en JML. Les annotations JML sont utilisées d'une part pour filtrer les cas de tests générés (éliminer les appels de méthodes ou les valeurs de paramètres qui ne respectent pas les pré-conditions), et d'autre part comme oracle de test afin de pouvoir établir les verdicts. En pratique, la génération des tests sera paramétrée par le nombre de cas de test qui doivent être générés et par le nombre d'appels de méthodes pour chaque cas de test. En plus de ces deux paramètres de base, d'autres contraintes peuvent être utilisées pour la génération de tests :

- Des pondérations peuvent être attribuées afin de modifier les probabilités qu'une classe ou qu'une méthode soit appelée dans un cas de test. En particulier, il est possible d'interdire l'appel d'une classe ou d'une méthode en lui associant un poids nul.
- Une fonction de probabilité de création d'objets définit la probabilité qu'un nouvel objet soit créé ou qu'un objet existant soit réutilisé en fonction du nombre d'objets déjà existants.
- Il est possible de définir des domaines réduits pour la valuation des paramètres. Cette possibilité se révèle utile lorsque la probabilité de valuation d'un paramètre de méthode en dehors des pré-conditions est très élevé.
- Enfin, l'outil Jartege propose un mécanisme similaire à celui de l'outil JUnit dans lequel la création de méthodes *setUp* et *tearDown* permet de définir des préambules et des postambules communs à tous les cas de test.

2.4 Résumé

Dans ce chapitre, nous avons présenté un ensemble de formalismes utilisés pour la modélisation de systèmes pour le test, un ensemble de critères de sélection de tests et certains outils de génération de tests associés à ces formalismes et critères de sélection. Les critères de sélection de tests présentés dans ce chapitre ne sont définis que par rapport aux informations structurelles présentes dans le modèle, c'est pourquoi nous les qualifions de "statiques". L'avantage des ces critères de sélection est de pouvoir être appliqués de manière automatique sur un modèle pour générer des cas de tests, limitant de ce fait l'intervention humaine à la rédaction d'un modèle. Leur principal inconvénient est la difficulté de trouver un équilibre entre la couverture du système par les tests et l'explosion combinatoire du nombre de tests. Dans le chapitre suivant, nous présentons un ensemble de travaux qui définissent des critères de sélection de tests qui donnent à l'utilisateur la possibilité d'apporter des informations complémentaires à celles présentes dans un modèle afin de guider de manière plus fine la génération de tests. Ces critères, que nous qualifions de "dynamiques", permettent de définir de manière plus précise la couverture souhaitée des différentes parties d'un système par les tests.

Chapitre 3

La génération de tests à partir d'objectifs de tests

Sommaire

Ce chapitre est centré sur l'utilisation d'objectifs de test, que nous qualifions de critères "dynamiques" de sélection de tests par opposition aux critères "statiques". Dans une première section, nous présentons la problématique liée à l'utilisation d'objectifs de test. La deuxième section présente les différents travaux et outils liés aux critères de sélection dynamiques. Enfin, nous concluons dans la troisième section sur ces travaux.

3.1 Problématique

Comme nous l'avons vu, la génération de tests basée sur l'application d'un critère de sélection à un modèle soulève un certain nombre de problèmes parmi lesquels nous trouvons :

- la maîtrise du nombre de cas de tests,
- la gestion de la traçabilité des tests,
- la définition du périmètre du système à tester.

Ces trois problèmes peuvent être adressés par quelques astuces lors de l'utilisation de critères de sélection statiques. Par exemple, pour la maîtrise du nombre de tests produits, il est possible :

- de choisir un critère de sélection de test "adapté" à l'objectif visé, par exemple en choisissant, parmi les différents critères de couverture des décisions et des conditions, celui qui offre le meilleur rapport entre la couverture du modèle et le nombre de tests générés.
- de choisir une modélisation plus abstraite du système.

Ces deux solutions possèdent chacune un inconvénient. La première offre une solution assez grossière puisqu'il n'est possible de choisir un critère de sélection que parmi l'éventail de ceux proposés. La seconde implique soit de prendre en compte la génération des tests au moment de la modélisation, soit de modifier le modèle ultérieurement. Or, toutes les informations abstraites ou omises dans le modèle auront potentiellement une utilité ultérieure.

Le problème de la traçabilité des tests peut également être adressé par des efforts tels que l'annotation du modèle par des exigences (mécanisme mis en place dans l'outil LTG). Mais cette solution impose un travail supplémentaire non négligeable d'annotation du modèle, et reste partielle puisque, s'il est simple de faire le lien entre une annotation et un comportement d'une opération à observer, il est plus complexe de "tracer" des propriétés de haut niveau (par exemple des propriétés temporelles) par ce biais.

Enfin, la définition du périmètre du système à tester peut être réalisée de manière assez simple par un choix de modélisation qui consiste :

- soit à ne modéliser que ce que l'on souhaite tester,
- soit par la possibilité offerte à l'utilisateur de choisir les transitions, états, opérations,... qui doivent être pris en compte par le critère de sélection choisi.

Ces deux solutions soulèvent également des problèmes d'imprécision de l'approche ou de perte d'informations.

Les travaux autour des critères de sélection dynamiques que nous présentons dans la section suivante visent à offrir des moyens d'adresser de manière plus simple et efficace ces problèmes. Ces travaux visent à offrir la possibilité de maîtriser le nombre de tests produits sans avoir recours à des astuces de modélisation, tout en apportant des améliorations sur la gestion de la traçabilité des tests et la précision de la sélection des tests.

3.2 Travaux existants sur les objectifs de test

3.2.1 Les différentes formes d'objectifs de test

Approches basées sur des modèles de fautes

Dans [AWW08], les auteurs proposent d'utiliser un critère de sélection de tests basé sur un modèle de fautes. Cette approche est mise en œuvre à partir d'une spécification décrite sous la forme d'un IOLTS. L'idée développée est d'introduire une faute dans le modèle original grâce à un opérateur de mutation. Une fois l'erreur introduite, une trace d'exécution menant à la détection de celle-ci est extraite grâce à un model-checker. Cette trace d'exécution est ensuite utilisée comme objectif de test par l'outil TGV [JJ05].

Approches basées sur la description d'objectifs de test

Dans [JJ05] les auteurs proposent une solution pour faire face au problème de la sélection des tests dans le cadre d'un système modélisé sous forme d'IOLTS. En effet, comme nous l'avons mentionné plus haut, pour être exhaustive, la génération de cas de test devrait extraire l'ensemble de toutes les exécutions possibles du système menant à un état puits (état de livelock ou de deadlock), ce qui dans le cas général n'est pas possible. Les auteurs proposent donc une méthode (implémentée dans l'outil TGV) qui permet de formaliser un objectif de test sous la forme d'un IOLTS. L'objectif de test décrit la sous-partie du système à tester sous la forme d'un IOLTS doté du même ensemble d'actions que le modèle du système à tester et de deux ensembles d'état pièges *Accept* et *Refuse*. Les états de refus (*Refuse*) servent à

élaguer au plus tôt les traces d'exécution que l'on ne souhaite pas voir apparaître dans les cas de test générés. Les états d'acceptation (*Accept*) servent de critère d'arrêt pour la génération de tests. Un nouvel IOLTS est produit par le produit synchronisé de l'objectif de test ainsi formalisé et du modèle du système à tester. C'est à partir de ce nouveau modèle que sont générés les tests.

Dans [dVT01], les auteurs proposent une notion de complétude et de correction d'une suite de tests produite par application d'objectifs à des modèles décrits sous la forme d'IOLTS. On peut noter une différence de terminologie, car les auteurs préférent la dénomination d'*objectifs d'observation* dans le sens où l'on doit :

- décrire ce que l'on souhaite observer (tester) dans le système,
- définir un lien clair entre l'objectif d'observation et les tests qui doivent le satisfaire,
- savoir comment interpréter le résultat de l'exécution des tests par rapport à l'objectif d'observation initial.

Dans [GGRT06, Tou06], les auteurs proposent une méthode de génération d'objectifs de test pour la génération de tests à partir d'IOSTS. L'idée développée repose sur l'animation symbolique de l'IOSTS décrivant le système afin d'extraire l'ensemble des chemins d'exécution symbolique utilisés ensuite comme objectifs de test. Deux critères de sélection des chemins d'exécution symbolique sont proposés. Le premier est basé sur l'extraction de tous les chemins d'exécution symbolique de taille n donnée par l'utilisateur. Le second critère nommé "critère de k-inclusion" repose sur la détection d'inclusion d'états permettant de déterminer le plus long chemin d'exécution symbolique sans redondance entre les états parcourus (de longueur p_{max}). La longueur $k \times p_{max}$ est utilisée pour choisir la longueur maximale des chemins d'exécution symbolique que l'on souhaite extraire afin de les utiliser comme objectif de test. On obtient alors un ensemble de chemins représentant toutes les combinaisons de k comportements basiques. Dans [FGG07], une approche similaire est appliquée au test de composants. L'exécution symbolique du système global composé de plusieurs IOSTS est utilisée pour dériver un objectif de test destiné à la validation d'un de ces composants.

Dans [BGN+04, VCG+08], les auteurs proposent une méthode de sélection de cas de test assez proche des objectifs de test. Dans cette démarche, le modèle de départ est une ASM formalisée en langage AsmL ou Spec♯, mais la génération des tests est réalisée à partir d'une machine à états finis. La transformation de l'ASM en FSM est guidée par l'utilisateur qui définit un certains nombre de contraintes d'abstraction et de restrictions qui vont guider cette transformation. Ainsi, il peut définir :

- des contraintes qui vont limiter les différentes valuations possibles des paramètres des opérations,
- des filtres qui ont comme effet d'éliminer certaines transitions du système,
- des filtres qui permettent d'éliminer certains états,
- certains critères permettent de limiter la longueur maximale des chemins ou d'arrêter la construction d'un chemin lorsqu'un certain état cible est atteint,
- de regrouper certains états du modèle sous certaines conditions (respect du déterminisme, par exemple).

Approche par scénarios

Dans [BMM05, BBM02], les auteurs décrivent une méthodologie de dérivation de cas de test grâce à des scénarios dérivés de diagrammes UML. Les principaux diagrammes utilisés sont les diagrammes de cas d'utilisation et les diagrammes de séquence. La démarche consiste

à construire une vue globale du système grâce aux différents diagrammes qui le décrivent. De cette vue globale est extrait un ensemble d'arbres ; chaque arbre posséde comme racine les acteurs du système, les branches étant les différentes actions possibles pour les acteurs (raffinées par niveaux successifs). Les différents scénarios sont extraits par un parcours en largeur des différents arbres obtenus. Deux critères de sélection complémentaires permettent de contrôler le nombre de tests produits :

- La sélection de la partie du système à tester permet d'élaguer certaines branches de comportement que l'on ne souhaite pas tester (par exemple, les comportements qui ne sont pas encore implémentés dans le système réel).
- Une pondération des différents nœuds de l'arbre permet de donner une priorité sur certaines classes de comportements ou sur certains comportements et ainsi de contrôler le nombre de tests produits via deux critères de sélection qui sont soit le nombre de tests produits, soit le pourcentage de comportements couverts.

Cette approche est implémentée dans la suite d'outils de génération de tests Cow_suite.

Approche combinatoire

Dans [LdBMP04], les auteurs proposent une méthode de génération de tests basée sur le dépliage combinatoire de schémas de test. Cette méthode est implémentée dans l'outil *TOBIAS*. Le point de départ de cette approche repose sur le fait que souvent les différents tests d'une même suite de tests partagent les mêmes enchaînements d'appels d'opérations à la différence près de la valuation des paramètres. La méthode proposée offre la possibilité de décrire des patterns de tests sous la forme d'expressions régulières sur les appels d'opérations et de contraintes sur leurs paramètres. Les différents patterns produits peuvent être également combinés entre eux afin de réaliser d'autres patterns. L'outil TOBIAS assure le dépliage des expressions régulières sur les appels d'opérations et les différentes valuations des paramètres dans les domaines définis par le schéma. Cette approche permet de générer un grand nombre de cas de tests différents à moindre coût et très rapidement. Du fait de l'absence de modèle pour la génération des tests, un nombre potentiellement grand de tests générés ne sont potentiellement pas exécutables. Cela est dû au fait que certains enchaînements d'opérations décrits dans le schéma ne sont pas forcément conformes avec la spécification et que les valuations des paramètres ne respectent pas forcément les pré-conditions des opérations. Cependant, cette méthode est très pratique lorsque l'on dispose de spécifications exécutables. Par exemple dans le cadre d'un programme Java annoté en JML, il est très facile d'éliminer les tests qui ne sont pas exécutables et d'obtenir un verdict en vérifiant les propriétés décrites dans les annotations JML après chaque exécution d'opération.

Un travail intéressant, décrit dans [MLdB03], a consisté en l'étude de trois modes de connexion possibles entre l'outil TOBIAS et l'outil UCASTING avec pour objectif le filtrage des tests non exécutables. UCASTING permet de générer des tests à partir de spécifications UML, il permet d'animer des spécifications et de valuer des séquences d'opérations (non instanciées ou partiellement instanciées) grâce à un solveur de contraintes. Les trois modes de connexion étudiés sont :

- UCASTING est utilisé pour valider les séquences de tests, complètement instanciées, produites par TOBIAS.
- UCASTING est utilisé pour instancier les valeurs des paramètres à partir de séquences d'opérations, dont les valeurs des paramètres ne sont pas instanciées, produites par TOBIAS.

– UCASTING est utilisé pour compléter l'instanciation des valeurs des paramètres à partir de séquences d'opérations, dont les valeurs des paramètres sont partiellement instanciées, produites par TOBIAS.

Les trois méthodes permettent d'éliminer les séquences de tests non exécutables sur l'implémentation. La méthode la plus prometteuse est la troisième, car elle est plus rapide que la première, et permet de piloter la génération de tests plus finement que la seconde.

3.2.2 Les outils

Les outils TGV [JJ05] et TorX [dVT01, TB03] implémentent tous deux une solution pour la sélection des tests basée sur des objectifs de tests. Dans les deux cas, le système et l'objectif de test sont modélisés par des IOLTS et les tests sont produits à partir du modèle obtenu par la synchronisation du modèle avec l'objectif de test.

Des solutions de génération de tests à partir de systèmes formalisés sous la forme d'IOSTS ont été proposées. Ces solutions permettent de réduire les problèmes d'explosion combinatoire dus à la complexité des données. Dans [JJRZ05], les auteurs présentent une solution (implémentée dans l'outil STG) dans laquelle un objectif de test est manuellement défini sous la forme d'un IOSTS. Le produit synchronisé de cet objectif de test avec le modèle du système (également un IOSTS) permet de réduire le modèle du système aux parties que l'on souhaite tester. Dans [GGRT06, FGG07], les auteurs présentent une solution (implémentée en utilisant l'outil AGATHA) basée sur l'extraction automatique d'objectif de test par exécution symbolique du modèle (IOSTS) du système à tester. Les critères de sélection des chemins d'exécution symbolique sont basés sur la longueur de ces chemins et la non redondance (basée sur un critère d'inclusion d'états) des états parcourus dans un même chemin.

Dans [AWW08], les auteurs proposent une méthode de sélection de tests à partir d'une spécification formalisée par un IOLTS et par un opérateur de mutation. L'idée générale est de sélectionner les cas de tests qui sont les traces menant à la détection des fautes introduites par les mutations de la spécification. Un model-checker est utilisé pour générer une trace menant à la détection d'une faute. Ensuite, cette trace est utilisée comme un objectif de test pour générer un graphe de test grâce à l'outil TGV.

Nous classons l'outil *Spec Explorer* [VCG+08], développé par *Microsoft Research*, parmi les outils de génération de tests à partir de modèles à états/transitions. Car bien que cet outil utilise des modèles de type pré/post (AmsL [BGN+03, BGN+04] ou Spec♯ [VCG+08]), ces modèles sont d'abord transformés en FSM. Spec Explorer, utilise un mécanisme de *"scenarios"* comme critère de sélection. Ces scénarios permettent :

– de limiter la sélection de valeurs pour les paramètres des méthodes,
– de supprimer des transitions du modèle,
– de filtrer les états du modèle pour ne considérer que ceux statisfaisant un prédicat donné,
– de guider la recherche de chemins d'exécution en bornant les profondeurs de recherche ou en définissant des états à atteindre,
– d'abstraire certains états en les regroupant, sous condition que le modèle reste déterministe.

La suite d'outils *Cow_test* [BMM05, BBM02] offre une solution de dérivation de cas de tests à partir d'un système modélisé sous la forme de diagrammes UML. Comme nous l'avons expliqué dans la section 3.2.1, la méthode proposée repose sur l'utilisation des diagrammes de cas et de séquences afin de dériver des suites de tests en utilisant des critères de sélection

de test basés sur la sélection de la partie du système à tester et sur une pondération des comportements et des classes de comportements à tester (permettant de définir une relation de priorité sur les aspects du système à tester). Cette approche est essentiellement intéressante d'un point de vue méthodologique car elle offre des solutions :

- pour faciliter la modélisation, car cette méthode repose sur une modélisation classique en UML, sans apporter d'informations complémentaires dans les différents diagrammes,
- pour la dérivation de cas de test dès le début de la modélisation,
- pour l'organisation des comportements à tester de manière hiérarchique et par ordre d'importance (sous forme d'arbre),
- pour la traçabilité des tests qui découle de l'organisation des comportements à tester.

L'outil *Telling TestStories* [FBCO+09] propose une approche méthodologique assez proche de celle proposée dans *Cow_test*. Cette méthode est basée sur la définition d'un modèle de tests construit à partir de briques élémentaires (TestSequenceElement) constituées :

- d'un état initial,
- d'une *"test story"*,
- de données de tests.

L'état initial donne l'état du système avant l'exécution de la *"test story"*. La *"test story"* décrit un scénario d'appels de services qui peuvent être conditionnés ou parallélisés et permet de définir des assertions qui permettront de définir des verdicts. Enfin, les données de tests définissent les valeurs des paramètres d'appels de services et les valeurs des résultats. Cette approche met en jeu quatre élements principaux :

- une implémentation,
- un modèle de l'implémentation,
- une implémentation des tests,
- un modèle de test.

Le modèle de test, constitué de *"test stories"* qui décrivent des objectifs de test. L'implémentation des tests définit le lien entre les tests décrits dans le modèle de test et l'implémentation. Le modèle du système est utilisé d'une part pour vérifier la consistance des cas de test et d'autre part pour évaluer la couverture des tests produits grâce à divers critères de couverture. Comme l'approche proposée par *cow_test*, l'outil *telling test stories* offre des aspects méthodologiques très organisés, orientés vers la construction de tests manuels et la traçabilité des exigences.

De manière annexe, nous présentons un travail portant sur les tests unitaires paramétrés [TS05]. Bien que reposant sur un concept de test boîte blanche, ce travail est assez proche de la notion de description d'objectif de test. Ce travail est implémenté dans l'outil *Unit Meister* pour la production de tests pour des applications développées avec le framework *.NET*. Le principe des tests unitaires paramétrés repose sur la rédaction de tests unitaires où les données de test ne sont pas complétement instanciées, mais sont passées en paramètres de la méthode de test. Une méthode de test est composée :

- d'une précondition exprimée sur les paramètres de la méthode et les variables du système,
- d'appels aux fonctionnalités du système,
- d'assertions utilisées pour établir les verdicts.

Un mécanisme d'animation symbolique est utilisé pour déterminer les exécutions du système satisfaisant les préconditions de la méthode de test et les appels aux fonctionnalités du système. A partir de la résolution du système de contraintes issu de l'animation symbolique du programme, il est possible d'obtenir des valuations pour les paramètres des méthodes de test

afin de couvrir les différentes branches des graphes de flot de contrôle observées lors de l'animation et ainsi produire des tests valués. Cette approche permet donc de générer des tests unitaires à partir de spécifications de cas de test symboliques (paramétrés) dont les oracles sont donnés et de l'implémentation du système.

3.3 Bilan

3.3.1 Conclusion sur l'existant

Le tableau 3.1 présente un résumé des différents critères de sélection de test que nous avons présentés. Les critères de sélection statiques sont présentés dans la partie haute du tableau, et les critères dynamiques dans sa partie basse. Les colonnes présentent les différents critères de sélection de test que nous avons explorés, les paradigmes de modélisation auxquels ils sont appliqués et les outils qui les implémentent. Nous avons déjà parlé des inconvénients des critères de sélection statiques dans l'introduction de ce chapitre, ce sont principalement la maîtrise du nombre de cas de test, la gestion de la traçabilité des tests et la finesse de paramétrisation de la génération des tests.

	Critères de sélection	Paradigme de modélisation	outils
Critères statiques	couverture du graphe de flot de contrôle	Pré/post	LTG/BZTT
			JMLTT
			B-Casting
	couverture d'états et de transitions	États/Transitions	TorX
			Conformance Kit
		Pré/Post	ProB
	couverture de données aux limites	Pré/post	LTG/BZTT
	critère de couverture aléatoire	Pré/Post	Jartege
Critères dynamiques	modèles de fautes	Etats/transitions	[AWW08]
	objectifs de tests	Etats/transitions symboliques	STG, AGATHA
		Etats/transitions	TGV
		Pré/Post	Spec Explorer
	approche par scénarios	Pré/Post	Cow Test
			Telling TestStories
	approche combinatoire	Pré/post	TOBIAS

Tab. 3.1 – Récapitulatif des critères de sélection

Tous les critères de sélection dynamiques de test présentés dans le tableau 3.1 comblent en partie ces lacunes. Ainsi, l'utilisation de modèles de fautes permet de cibler précisément une vulnérabilité potentielle du système et par conséquent de réduire le nombre de cas de test produits et d'augmenter la traçabilité puisque les tests sont directement liés à une vulnérabilité potentielle. Cette approche nécessite beaucoup d'expertise pour la définition des opérateurs de mutation.

L'utilisation d'objectifs de test comme elle est utilisée dans l'outil TGV permet de réduire le périmètre d'application d'un critère de sélection de test statique sur le système. Cette approche offre la possibilité d'une part de maîtriser l'explosion combinatoire et d'autre part d'exprimer plus finement ce que l'on souhaite tester. Cette approche est toutefois limitée dans son application aux IOLTS puisque toutes les données doivent être instanciées, de ce fait la taille des modèles est vite rédhibitoire dans le cas d'applications de taille industrielle.

La description d'objectifs de test telle qu'elle est implémentée dans l'outil Spec Explorer repose essentiellement sur des moyens de filtrage des tests au travers de fonctions de transformation du modèle de départ et de contraintes de parcours du modèle pour la génération de test. Cette approche ne propose pas de moyens clairement définis pour formaliser des propriétés ou exigences sur le système qui guideraient la génération de tests.

L'extension des travaux sur la modélisation d'objectif de tests pour des systèmes d'états et de transitions étiquetées symboliques implémentée dans les outils STG et AGATHA offre une solution pour appréhender des systèmes de taille plus conséquente en réduisant la complexité due aux données.

L'approche par scénario, implémentée dans les outils Cow_test ou Telling TestStories s'appuie sur une grande quantité de documents de travail afin d'extraire directement des séquences de test. Cette approche est certainement celle qui donne le plus de latitude pour la finesse de sélection des tests et la plus grande efficacité pour la traçabilité des tests. Par contre, peu de choses sont automatiques, et c'est le point faible de cette approche.

Enfin, l'approche combinatoire est un intermédiaire entre l'approche par objectifs de test et celle par scénarios. Elle permet de dériver des objectifs de test de manière combinatoire à partir de scénarios partiels.

Chacune de ces approches apporte des réponses méthodologiques et techniques aux trois problèmes que nous avons soulevés concernant les critères de sélection statiques de tests. Toutes sont efficaces sur la maîtrise du nombre de cas de test produits. Par contre, concernant la traçabilité, les approches basées sur l'utilisation de scénarios (éventuellement partiels) sont plus pertinentes, car chaque scénario peut être associé à un objectif d'observation particulier. Concernant les approches par objectifs de test, celles que nous avons abordées reposent sur une transformation du modèle original du système en un modèle plus restreint ou abstrait puis par application d'un critère de couverture statique sur ce nouveau modèle.

3.3.2 L'approche proposée et son originalité

Après cet aperçu des différents travaux autour de la génération de tests à partir de modèles, nous présentons ici, l'approche de génération de tests que nous proposons. Nous abordons le positionnement de notre approche par rapport au contexte dans lequel nous avons développé cette approche et plus généralement par rapport aux travaux existants dans le domaine.

Contexte

Nos travaux sont initialement prévus pour s'appliquer à la génération de tests pour des systèmes modélisés à l'aide de machines B. Ils s'inscrivent dans la continuité des travaux déjà réalisés autour des outils BZTT et LTG. Les travaux réalisés autour de BZTT et LTG ont été présentés dans le chapitre 2, il s'agit de génération de tests *fonctionnels* pour des modèles décrits en langage B. Les tests fonctionnels sont générés en utilisant des critères de couverture structurelle du modèle du système. Parmi ces critères, les outils BZTT et LTG exploitent essentiellement les critères de couvertures des conditions et décisions et de valuation des données (variables d'état et paramètres des opérations) aux limites de leurs domaines.

Comme nous l'avons souligné précédemment, ces critères de sélection de tests statiques ne permettent pas la sélection de tests à partir de propriétés de haut niveau ou d'objectifs de tests reposant sur d'autres critères que la couverture structurelle des opérations du modèle. Notre objectif est par conséquent de proposer une approche permettant de modèliser des objectifs

de tests à partir d'enchaînements d'opérations et d'états afin de les utiliser comme critères de sélection et ainsi dépasser les limitations d'expressivité des critères de sélection structurels.

Toutefois, les critères de sélection reposant sur une couverture structurelle du modèle offrent un moyen simple et totalement automatisable de production de tests. Notre but, n'est pas d'écarter cette approche de génération de tests, mais de proposer une approche complémentaire. Dans cette optique, nous considérons la génération de tests fonctionnels comme une étape préalable à la génération de tests à partir d'objectifs de tests. Et par conséquent, nous nous plaçons dans une optique de réutilisation des ressources mises en œuvre pour la génération de tests fonctionnels, et nous déployons une solution d'évaluation de la complémentarité des tests qui sont produits par chacune des approches.

Positionnement

Notre démarche vise principalement à adresser le problème de l'expression d'objectifs de test de manière précise et à donner les moyens de gérer le problème d'explosion combinatoire du nombre de tests. L'idée principale est de conserver la phase de génération automatique de tests à partir d'un critère de sélection basé sur la couverture structurelle du modèle, tout en proposant une approche complémentaire destiner à augmenter la couverture du modèle par des tests issus de propriétés ou d'exigences définies par l'utilisateur. L'objectif est de limiter cette partie de la génération de test, par couverture structurelle du modèle, à ce qui peut être réalisé de manière automatique sans chercher à résoudre les problèmes de couverture ou d'explosion combinatoire par des ajustements au niveau du modèle ou des paramètres de génération de tests (critères de couverture, profondeur de recherche, timeout, ...). Après cette première phase de génération de test, notre proposition vise à décrire des objectifs de validation complémentaires et à produire des tests complémentaires à l'aide d'une méthode outillée à cet effet. La description des objectifs de validation par des schémas de test peut alors donner lieu à une nouvelle phase de génération de tests dont l'objectif est de compléter la première par rapport à des exigences de validation et des propriétés de haut niveau.

Les différentes approches que nous avons présentées dans ce chapitre ne pose pas la possibilité d'exploiter le modèle du système pour générer des tests grâce à l'utilisation d'un critère de sélection statique par un processus totalement automatisé. Pour notre part, nous pensons qu'une telle approche permet dans un premier temps de générer des tests qui permettront de couvrir une grande part des cas d'utilisation les plus simples du système à moindre coût, puis d'envisager ensuite la production de tests complémentaires.

Dans cette optique de réutilisation et valorisation des efforts mis en œuvre pour la génération de tests par couverture structurelle du modèle, les éléments qui sont réutilisés sont :

- le modèle du système à tester,
- la technologie de génération de test (animateur et solveur de contraintes).
- la couche de concrétisation,

Le modèle du système qui a servi à la génération des tests fonctionnels peut être réutilisé tel quel, puisque les informations qui servent à guider la génération des tests complémentaires sont contenues dans les schémas de test. La technologie sur laquelle repose la génération des tests fonctionnels est également réutilisée. La génération de tests à partir de schémas exploite les outils d'animation et de résolution de contraintes implémentés dans LTG et BZTT. Et enfin, du fait de la réutilisation du modèle du système, la couche d'adaptation développée pour exécuter les tests fonctionnels peut également être réutilisée pour l'exécution des tests qui sont générés à partir des schémas.

Le langage de modélisation des schémas de test que nous proposons repose sur la notion d'objectif de test telle que nous avons pu la développer dans ce chapitre. Ce langage est un intermédiaire entre les intentions de validation (exigences et propriétés de haut niveau) et les outils de génération de test. Afin d'avoir un langage générique permettant d'atteindre cet objectif, nous proposons un langage basé sur les expressions régulières (simples d'utilisation et faciles à traiter) qui permet de manière simple et concise d'exprimer des enchainements d'opérations ou d'états et la manière dont on souhaite les couvrir. On peut noter que la description d'états, dans les séquences d'opérations, offre un formalisme permettant d'exprimer certaines contraintes de manière plus simple et plus générique que l'expression de contraintes sur les paramètres des opérations. La construction du langage que nous proposons le rend générique par la séparation des différents niveaux de construction d'un schéma :

- description des éléments du modèle (appel d'opération, description d'états et de comportements),
- dynamique des enchaînements d'états et d'opérations (décrite à l'aide des opérateurs des expressions régulières : concaténation, choix et répétition),
- choix de couverture des exécutions du système décrites par les schémas (couverture des choix et des comportements des opérations).

Cette dernière possibilité d'agir finement sur la couverture des schémas de test par le biais des directives de couverture sur les choix et les comportements des opérations permet d'influer sur la sélection des cas de test de manière à mettre l'accent sur les aspects du système dont la validation est prioritaire.

Les différentes approches basées sur la définition d'objectifs de test que nous avons présentées dans ce chapitre peuvent se diviser en deux catégories :

- Celles basées sur l'utilisation de propriétés ou d'exigences de validation comme objectif de test (par exemple sous la forme d'IOLTS dans le cas de l'outil TGV ou d'IOSTS pour STG).
- Celles, comme Spec Explorer par exemple, qui reposent sur la définition de contraintes qui vont permettre de limiter l'exploration du système en réduisant les domaines des données, en supprimant certaines transitions ou certains états, en définissant une longueur maximale pour les cas de test, ...

Le langage que nous proposons permet de combiner ces deux approches, en permettant à la fois de modéliser la dynamique associée à une propriété ou une exigence de validation sous la forme d'enchainements d'états symboliques et d'appels d'opération et des indications concernant la manière de couvrir ces enchainements. Cette approche permet d'une part de définir précisément la sous-partie du système que l'on souhaite valider et d'autre part de donner des directives de pilotage afin d'affiner la couverture que l'on souhaite obtenir.

L'approche que nous proposons ne résoud pas le problème de l'explosion combinatoire du nombre de tests puisque nous nous basons sur les même technologies et outils que ceux utilisés pour la génération des tests fonctionnels. Par contre, elle offre la possibilité de guider la génération de tests pour les parties du système qui doivent être la cible d'efforts de validation importants. Le langage de description des schémas de test permet d'orienter la génération du système selon deux axes :

- la réduction des exécutions du système à celles décrites par les enchaînements d'états et d'opérations,
- les directives de couverture des choix et des comportements des opérations qui permettent de donner une description encore plus fine des tests qui doivent être générés.

En dernier lieu, au sens où un schéma de test prend en compte une propriété de haut

niveau ou une exigence de validation et décrit la manière dont celle-ci doit être couverte, notre approche adresse le problème de la traçabilité. En effet, il est simple de maintenir le lien entre un schéma de test et les tests produits grâce à lui. De ce fait, il est possible, pour un ensemble de tests, de savoir quelle exigence de validation est visée et comment celle-ci a été couverte. De plus, un schéma de test peut être composé de plusieurs autres schémas de test, ce qui donne la possibilité de partitionner un ensemble de tests issus d'un objectif en plusieurs sous-ensembles répondant chacun à un aspect différent de l'objectif initial.

Deuxième partie

Contributions

Chapitre 4

Le langage de description d'objectifs de test

Sommaire

Dans ce chapitre, nous proposons un langage de description d'objectifs de test destinés à être utilisés comme critère de sélection dynamique. Comme nous avons pu le constater dans le chapitre précédent, les critères de sélection "dynamiques" peuvent prendre de multiple formes. Dans l'approche que nous proposons, nous considérons qu'un objectif de test et issu d'un ou plusieurs *besoins de test* et une ou plusieurs propriétés. Nous considérons qu'un besoin de test est l'énoncé informel d'intentions de validation. Ainsi, si on considère la spécification du porte-monnaie électronique *Demoney* présentée dans la section suivante, un besoin de test pourrait être : *"observer les différents scénarios possibles d'une transaction de crédit de la carte, incluant les scénarios de succès et ceux d'échec"*. Les propriétés associées étant par exemple : *"l'absence d'authentification de l'utilisateur avant l'initialisation d'un crédit fait échouer celle-ci"* ou *"toute transaction initialisée à l'aide de la commande INITIALIZE_TRANSACTION doit être immédiatement suivi d'une validation réalisée grâce à la commande COMMIT_TRANSACTION, sans quoi cette transaction est annulée"*.

Le langage que nous proposons permet de formaliser de tels objectifs de test sous la forme de schémas de test à l'aide d'un langage d'expression d'enchaînements d'appels d'opérations et d'états symboliques définis au moyen d'expressions régulières. Á ce langage s'ajoute des

directives de pilotage destinées à guider la production des tests en filtrant les comportements des opérations et en définissant le caractère optionnel de certains chemins.

Nous débutons ce chapitre par la présentation d'une partie du cas d'étude "Demoney" sur lequel reposent les exemples que nous introduisons. Puis, nous présentons le langage de formalisation d'objectifs de tests et sa sémantique sous forme d'automates.

4.1 Exemple fil rouge

Afin d'illustrer notre démarche, nous introduisons ici un exemple de système à tester. Nous avons choisi d'utiliser un exemple jouet inspiré de la spécification du porte-monnaie électronique nommé "Demoney". Cette spécification a été proposée par la compagnie *Trusted Labs*[7] dans un but expérimental. La spécification sur laquelle nous nous basons et le modèle complet du système sont présentés en annexe 1. Nous nous contentons ici de présenter les commandes concernant la réalisation d'une transaction. Tous les exemples que nous présentons supposent implicitement que la phase de personnalisation de la carte est déjà réalisée. Nous avons par ailleurs proposé ce cas d'étude comme projet semestriel pour des étudiants de première année de master. Cette expérience pédagogique a été présentée lors de deux workshops, l'un sur l'enseignement des méthodes formelles [DT08], et l'autre sur la méthode B et le transfert de connaissance de la recherche vers l'enseignement [DJT08].

4.1.1 Spécification

Demoney est un porte-monnaie électronique permettant de gérer des transactions de crédit (rechargement de la carte) et de débit (utilisation de la carte pour réaliser des achats). La carte possède un cycle de vie décomposé en quatre phases :

- La phase de *personalisation* de la carte permet de fixer les codes PIN de l'utilisateur et de la banque, et de fixer le solde et le débit maximal de la carte. Une fois la personnalisation effectuée et validée, la carte ne peut plus revenir en phase de personnalisation.
- La phase *d'utilisation normale* de la carte, permet à l'utilisateur d'utiliser sa carte afin de faire des achats. Pour la recharger, celui-ci doit s'authentifier sur un terminal bancaire.
- Le *blocage* de la carte intervient lorsque tout les essais d'authentification autorisés de l'utilisateur ont échoué. La carte est alors désactivée temporairement jusqu'à ce que la banque la débloque (la carte repasse alors en phase d'utilisation).
- La *désactivation* de la carte, intervient lorsque celle-ci est bloquée et qu'un nombre trop élevé d'essais d'authentification infructeux a été réalisé sur le code PIN de la banque. La carte est alors irrémédiablement désactivée.

La carte possède deux codes PIN, l'un permet à l'utilisateur de s'authentifier afin de recharger sa carte. L'autre permet à la banque de s'authentifier lorsque la carte est bloquée, afin de la débloquer, réinitialiser le compteur d'essais associé au PIN de l'utilisateur et modifier la valeur du code PIN. Chaque authentification acquise n'est valable que pour une utilisation correcte (rechargement de la carte ou déblocage de celle-ci) et un échec d'authentification provoque la perte des authentifications en cours.

Les trois opérations auxquelles nous nous intéressons sont : *INITIALIZE_TRANSACTION*, *COMMIT_TRANSACTION* et *VERIFY_PIN*. La commande VERIFY_PIN permet à l'utili-

[7] `http ://www.trusted-labs.com/`

sateur ou à la banque de s'authentifier sur la carte. Cette opération possède deux paramètres, le premier pour indiquer quel PIN est utilisé et le second pour donner la valeur du code PIN. La commande INITIALIZE_TRANSACTION est utilisée pour débuter une transaction ; elle possède deux paramètres, le premier pour indiquer si la transaction est un crédit ou un débit, le second pour indiquer le montant de la transaction. Enfin, la commande COMMIT_TRANSACTION permet de valider une transaction initiée en vérifiant que le contexte de la transaction est correct (cycle de vie, authentification et montant de transaction corrects). La commande INITIALIZE_TRANSACTION doit être directement suivie d'un appel à la commande COMMIT_TRANSACTION. Dans le cas contraire, la transaction en cours est annulée.

4.1.2 Modèle

Dans cette partie, nous décrivons en partie la machine B, rédigée à partir des spécifications du porte-monnaie électronique Demoney. Nous présentons d'abord le modèle de données de cette machine B, puis certaines opérations nécessaires à l'exécution d'une transaction.

Modèle de données

La première partie du modèle de données (fig.4.1) définit deux ensembles énumérés. L'ensemble *ETAT_CARTE* contient les différents états du cycle de vie par lesquels la carte peut passer. L'ensemble *sw* définit les messages d'erreurs ou de succès qui peuvent être émis par les différentes commandes du système.

```
SETS
  /*Etats de cycle de vie de la carte*/
  ETAT_CARTE = {perso, use, invalid, dead};

  /* STATUS_WORD */
  sw ={sw_Success, /*codé 9000 au niveau de l'implémentation*/
  sw_Error_life_cycle, /*9101*/
  sw_Error_parameter,/*9100*/
  sw_Error_perso_not_correct,/*9402*/
  sw_Error_perso_not_completed, /*9401*/
  sw_Error_pin_value, /*9501*/
  sw_Error_pin_value_no_more_try_user, /*9502*/
  sw_Error_pin_value_no_more_try_bank, /*9503*/
  sw_Error_invalid_credit,/*9601*/
  sw_Error_invalid_debit,/*9602*/
  sw_Error_invalid_transaction, /*9701*/
  sw_Error_invalid_pin_value /*9801*/
  }
```

FIG. 4.1 – Demoney : ensembles énumérés

Dans les clauses *CONSTANTS*, *PROPERTIES* et *DEFINITIONS* (fig. 4.2), nous avons défini des constantes de différents types : entiers, ensembles d'entiers et fonctions. Parmi ces définitions, on trouve principalement la déclaration de contantes identifiant par exemple les différents codes PIN présents sur la carte (PIN_BANK et PIN_USER). On trouve également, une fonction constante (MAX_RETRY) qui associe chacun des codes PIN présents avec le nombre d'essais maximum pour s'authentifier.

La figure 4.3, présente la déclaration des variables (dans la clause *VARIABLES*) et leur "typage" (dans la clause "INVARIANT"). Les variables *max_solde* et *max_debit* (uniquement

```
CONSTANTS
   /* PIN */
   PIN_BANK,
   PIN_USER,
   PIN_NONE,

   /*Type de code pin (erroné, banque ou utilisateur)*/
   TYPE_PIN_WITH_NONE,
   /*Type de code pin (banque ou utilisateur)*/
   TYPE_PIN,

   /*Compteur d'essais d'autentification des PIN*/
   MAX_RETRY,

   /* Identifiants de données */
   SET_MAX_BALANCE,
   SET_MAX_DEBIT,
   SET_HOLDER_PIN,
   SET_BANK_PIN,

   /*Type de transaction (débit ou crédit)*/
   TYPE_TRANSACTION,
   TRANSACTION_CREDIT,
   TRANSACTION_DEBIT


PROPERTIES
   /* PIN */
   PIN_BANK = 0
∧ PIN_USER = 1
∧ PIN_NONE = 13
∧ TYPE_PIN = {PIN_BANK, PIN_USER}
∧ TYPE_PIN_WITH_NONE = TYPE_PIN ∪ {PIN_NONE}

∧ MAX_RETRY ∈ TYPE_PIN → NAT1
∧ MAX_RETRY = {PIN_USER ↦ 3, PIN_BANK ↦ 4}

   /* PUT_DATA */
∧ SET_MAX_BALANCE    = 0
∧ SET_MAX_DEBIT      = 1
∧ SET_HOLDER_PIN     = 2
∧ SET_BANK_PIN       = 3

   /*INITIALIZE_TRANSACTION*/
∧ TRANSACTION_CREDIT = 0
∧ TRANSACTION_DEBIT = 1
∧ TYPE_TRANSACTION = {TRANSACTION_CREDIT, TRANSACTION_DEBIT}


DEFINITIONS
   SHORT == −32768..32767;
   BYTE == −128..127;

   PIN_SET == 0..9999
```

FIG. 4.2 – Demoney : constantes et définitions

modifiables en phase de personnalisation de la carte) définissent respectivement le montant maximum d'argent présent sur le porte-monnaie électronique et le montant maximum d'un retrait. La variable *etat_carte* permet de connaître dans quel état du cycle de vie la carte se trouve. La variable *solde* contient le solde courant. La variable *debit* contient le montant d'une transaction initiée mais non finalisée ; si le montant est positif, il s'agit d'un débit et s'il est

négatif, il s'agit d'un crédit. La fonction *pin_2_value* fait le lien entre chaque PIN de la carte et le code secret qui lui est associé. La variable *authenticated_pin* contient l'identifiant du code pin authentifié sur la carte (PIN_BANK, PIN_USER ou PIN_NONE). Enfin, la fonction *retry_counter* associe chaque code PIN avec le nombre d'essais restants pour son authentification. Le nombre de tentatives d'authentification de l'utilisateur (resp. de la banque) est compris entre 0 et MAX_RETRY(PIN_USER) (resp. MAX_RETRY(PIN_BANK)).

VARIABLES

```
  /* perso */
  max_solde ,
  max_debit ,

  /* use */
  etat_carte ,
  solde ,
  debit ,
  pin_2_value ,
  authenticated_pin ,
  retry_counter
```

INVARIANT

```
/* ******************************** */
/* Invariant de typage des variables */
/* ******************************** */

/* Etat du cycle de vie de la carte (perso, use, invalid ou dead)*/
  etat_carte ∈ ETAT_CARTE

/* Solde maximum de la carte --- Fixé à la personnalisation de la carte */
∧ max_solde ∈ SHORT

/* Montant maximum d'un débit --- Fixé à la personnalisation de la carte */
∧ max_debit ∈ SHORT

/* Solde courant de la carte */
∧ solde ∈ SHORT

/* Montant du débit (debit > 0) ou crédit (debit < 0) en cours */
∧ debit ∈ SHORT

/* Association entre un code PIN et sa valeur */
∧ pin_2_value ∈ TYPE_PIN → SHORT

/* Code PIN authentifié utilisateur (PIN_USER), banque (PIN_BANK) ou aucun (PIN_NONE) */
∧ authenticated_pin ∈ TYPE_PIN_WITH_NONE

/* Association entre un code PIN et le nombre d'essais d'authentification restant */
∧ retry_counter ∈ TYPE_PIN → SHORT
∧ retry_counter(PIN_BANK) ≤ MAX_RETRY(PIN_BANK)
∧ retry_counter(PIN_USER) ≤ MAX_RETRY(PIN_USER)
∧ retry_counter(PIN_BANK) ≥ 0
∧ retry_counter(PIN_USER) ≥ 0
```

FIG. 4.3 – Demoney : variables et invariant

4.2 Syntaxe concrète du langage de schémas de test

Nous pouvons décomposer le langage de description des schémas de test suivant trois couches différentes : *modèle*, *séquence* et *directive*. Cette séparation a pour but d'assurer l'indépendance de ce langage vis-à-vis du langage de modélisation utilisé pour représenter le système et des outils utilisés pour générer les tests.

La couche modèle (fig. 4.4) permet de faire le lien entre le modèle comportemental du système à tester et l'objectif de test au travers de prédicats et de noms d'opérations. Les prédicats sont exprimés sur les variables d'état du modèle (terminal prédicat d'état). L'expression d'un état sous la forme de prédicat permet de définir un sous-ensemble d'états du système dont on souhaite atteindre l'un des éléments par une suite d'opérations.

Les noms d'opération du modèle (terminal nom d'opération) permettent de définir un appel à une commande du système.

Les descriptions de comportements (terminal comportement) permettent de définir quel(s) comportement(s) d'une opération sélectionner pour la génération de tests. Les comportements sont décrits par des identificateurs faisant référence à des comportements extraits automatiquement à l'aide d'un critère de couverture structurelle appliqué sur les opérations du modèle ; ils peuvent également être définis manuellement par l'utilisateur. Dans ces deux cas, les identificateurs de comportements font référence à des prédicats avant/après exprimés sur les variables du modèle, sur les paramètres ou les valeurs de retour de l'opération. Par exemple, $x = 1 \wedge x' = 2$ signifie que l'on vise le comportement de l'opération dans lequel la variable x vaut 1 avant l'exécution de l'opération, et 2 après.

SP	::=	prédicat d'état
OP_NAME	::=	nom d'opération
CPT	::=	comportement

FIG. 4.4 – Syntaxe concrète : couche modèle

La couche séquence (fig. 4.5) dont le non-terminal SEQ est la racine de la grammaire, est bâtie sur le langage des expressions régulières. Elle permet de décrire un objectif de test comme un ensemble d'exécutions (séquences de couples (état,action)) du système. Les états peuvent être contraints par des prédicats sur les variables d'états du modèle. Ces contraintes sont introduites par le non-terminal *LEADSTO*. Parmi les constructions héritées du langage des expressions régulières, nous trouvons la concaténation (non-terminal CONCAT), le choix (non-terminal CHOICE), les répétitions (non-terminal REPEAT). Les transitions sont introduites par les appels d'opérations (non-terminal SIMPLE_OP et PILOT_OP). Le terminal *$OP* représente l'ensemble de toutes les opérations du modèle (éventuellement privé de certaines opérations grâce à la construction de soustraction notée $OP\OP_LIST).

La couche directive (fig. 4.6) permet de piloter la génération de tests à partir du schéma par des contraintes sur les chemins à parcourir[8]. D'une part avec la possibilité d'utiliser soit un choix exclusif, soit un choix inclusif entre plusieurs chemins. Et d'autre part avec la possibilité de sélection des comportements à explorer dans une opération (un, tous ou une

[8]La notation $^{0/1}$ exprime le caractère optionnel de certaines sous-expressions

```
SEQ          ::= CONCAT eof

CONCAT       ::= CHOICE (”.”CONCAT)^0/1
CHOICE       ::= LEADSTO (CHOICE_OP CHOICE)^0/1
LEADSTO      ::= REPEAT (⇝ ”(” SP ”)”)^0/1
REPEAT       ::= SIMPLE_EXP
                 (”?” | ”*” | ”+” | (”{” n (”,” m^0/1)^0/1 ”}”) CHOICE_PARAM)^0/1

SIMPLE_EXP   ::= ”(” CONCAT ”)”
               | SIMPLE_OP
               | PILOT_OP

SIMPLE_OP    ::= OP_NAME
               | ”$OP” CHOICE_PARAM (”\” OP_LIST)^0/1

OP_LIST      ::= ”{” operation name (”,” operation name)* ”}”
```

FIG. 4.5 – Syntaxe concrète : couche séquence

sélection de comportements). La différence entre les opérateurs de choix inclusif "|" et exclusif $\oplus$ est introduite par le non-terminal CHOICE_OP. Un choix inclusif signifie que chacune des branches doit faire l'objet de la génération de tests, alors qu'un choix inclusif indique que seule une des branches donnera lieu à la génération de tests pour le chemin courant. Le pilotage sur le choix des comportements des opérations est introduit par le non-terminal PILOT_OP. Les crochets autour d'un nom d'opération indiquent que tous les comportements applicables de l'opération doivent être explorés (par défaut, seul le premier comportement qui convient à la valuation de la séquence est appliqué). Le suffixe "/e{liste de comportements}" permet d'exclure certains comportements des comportements à observer. Le suffixe "/w{liste de comportements}" permet de limiter les comportements à explorer à ceux présentés dans la liste. Le non-terminal CHOICE_PARAM introduit un moyen de qualifier les choix implicites dans l'utilisation des boucles et du terminal $OP (dans le cas par défaut, on considère qu'un choix est inclusif). Pour les boucles, il permet de préciser si toutes les branches générées par la boucle doivent être explorées (_all), ou si une seule suffit (_one). Appliqué au terminal $OP, il permet de préciser si chaque opération qui peut être appelée doit donner lieu à une séquence ou si l'une de ces opérations suffit.

```
CHOICE_OP    ::= ”|” | ”⊕”

PILOT_OP     ::= ”[” OP_NAME ((”/w” | ”/e”) CPT_LIST)^0/1 ”]”
CPT_LIST     ::= ”{” CPT (”,” CPT)* ”}”

CHOICE_PARAM ::= ”_one” | ”_all”
```

FIG. 4.6 – Syntaxe concrète : couche directive

4.3 Exemple d'application du langage

Nous proposons ici quelques exemples de description d'objectifs de test décrits à l'aide du langage que nous proposons. L'objectif est de présenter les différentes constructions du

langage en les appliquant à l'exemple du porte-monnaie électronique Demoney.

4.3.1 Premier exemple

Nous allons tout d'abord nous intéresser à un objectif de test assez simple dont le but est d'observer une opération de crédit sur la carte. Nous prenons comme hypothèse que la carte est déjà personnalisée. Pour réaliser cet objectif de test, nous allons devoir utiliser un enchaînement de trois opérations qui sont :

1. VERIFY_PIN afin que l'utilisateur s'authentifie pour ensuite pouvoir recharger sa carte ;
2. INITIALIZE_TRANSACTION pour définir les paramètres de la transaction ;
3. COMMIT_TRANSACTION pour valider la transaction.

Cet enchaînement d'opérations seul ne garantit pas l'observation d'un crédit couronné de succès. Il nous faut d'abord garantir le fait que l'opération COMMIT_TRANSACTION se déroule avec succès. Cette garantie peut être aquise en ciblant les comportements de succès de cette opération dans le schéma. Ce comportement est identifié par l'étiquette CPT_COMMIT_SUCCESS, il correspond au cas où la valeur de la variable de retour de l'opération (*out*) vaut *sw_Success*. De plus, si nous souhaitons observer un crédit et non un débit, il faudra préciser le type de transaction initiée. Pour cela, nous pouvons préciser l'état dans lequel le sytème doit être après l'initialisation de la transaction. Plus particulièrement, nous pouvons dire que la variable *debit* devra avoir une valeur strictement négative[9]. L'objectif de test produit est présenté dans la figure 4.7.

VERIFY_PIN.INITIALIZE_TRANSACTION$\rightsquigarrow (debit < 0)$
.[COMMIT_TRANSACTION/$_w${CPT_COMMIT_SUCCESS}]

FIG. 4.7 – Demoney : Objectif de test pour le crédit

Supposons que l'on veuille, en plus de ce premier objectif, observer ce qui se passe si l'utilisateur doit faire plusieurs essais avant de réussir à s'authentifier. Il nous suffit d'ajouter un nombre d'essais potentiellement nul menant à un échec d'authentification au début de l'objectif de test (fig. 4.8). Si on ne souhaite observer qu'un essai infructeux d'authentification, il suffit de remplacer le symbole de répétition "*" par le symbole " ?" (répétition bornée entre 0 et 1).

(VERIFY_PIN$\rightsquigarrow (authenticated_pin = PIN_NONE)$)*
.VERIFY_PIN.INITIALIZE_TRANSACTION$\rightsquigarrow (debit < 0)$
.[COMMIT_TRANSACTION/$_w${CPT_COMMIT_SUCCESS}]

FIG. 4.8 – Demoney : Objectif de test pour le crédit (seconde version)

4.3.2 Second exemple

Au travers de ce second exemple, nous souhaitons présenter la rédaction d'un schéma de test plus complet. Cet exemple est toujours basé sur le rechargement de la carte, mais en plus

[9] Ce qui correspond à un crédit de la carte en cours.

d'observer des situations de succès. Nous souhaitons également observer différentes situations d'échecs de crédit de la carte.

Parmi les situations d'échec du crédit, nous souhaitons observer les causes suivantes :

- L'absence d'authentification (éventuellement après des essais d'authentification infructueux).
- Le choix d'un montant d'argent à transférer trop important.
- La rupture de l'enchaînement entre les opérations d'initialisation d'une transaction et sa validation.
- La carte n'est pas dans le bon état du cycle de vie.
- Aucune transaction n'a été initiée.

La figure 4.9 présente le schéma de test que nous proposons afin de guider l'observation des points cités précédemment. La ligne 1 permet de produire plusieurs essais d'authentification infructueux successifs (dans le cas où il y a trois essais infructueux, la carte passe en état de cycle de vie "bloqué"). La ligne 2 introduit un succès d'authentification. Les lignes 3 , 4 et 5 permettent d'explorer les différents cas d'utilisation occasionnés par le montant choisi pour le rechargement de la carte (montant correct, nul ou trop important). La ligne 6 permet d'introduire une rupture, dans l'enchaînement des commandes de paramètrage et de validation d'une transaction, par l'introduction d'une opération quelconque du modèle[10]. Et enfin, la ligne 7 introduit la validation de la transaction.

```
1 | (VERIFY_PIN⇝ (authenticated_pin = PIN_NONE)){0,3}
2 | .(VERIFY_PIN⇝ (authenticated_pin = PIN_USER)) ?
3 | .(INITIALIZE_TRANSACTION⇝ (debit < 0 ∧ solde − debit ≤ max_solde)
4 |   | [INITIALIZE_TRANSACTION /w{CPT_INIT_TRANS_CREDIT_ERROR_MAX}]
5 |   | [INITIALIZE_TRANSACTION /w{CPT_INIT_TRANS_CREDIT_NULL}]) ?
6 | .$OP_all ?
7 | .COMMIT_TRANSACTION
```

Fig. 4.9 – Demoney : Objectif de test pour le crédit (troisième version)

La combinatoire induite dans ce dernier schéma de test (fig. 4.9) mène à la production de 512 séquences d'appels d'opérations et d'états. On peut noter que parmi ces 512 séquences, toutes ne sont pas animables sur le modèle B. Par exemple, parmi toutes les séquences où l'utilisateur n'est pas authentifié, on ne pourra pas animer celles qui initialisent un crédit avec succès. Cela est dû au fait que certaines de ces séquences proposent des enchaînements d'appels d'opérations et d'états qui ne sont pas acceptés par le modèle. Par exemple, si trois échecs d'authentification de l'utilisateur surviennent (ligne 1), alors il sera impossible d'atteindre un état où l'utilisateur sera authentifié uniquement par appel à la commande VERIFY_PIN (ligne 2). Par conséquent, certaines des séquences proposées par cet objectif de test ne pourront pas donner lieu à la génération de tests.

4.4 Sémantique du langage

Dans cette partie, nous présentons la syntaxe abstraite associée au langage de description d'objectifs de test décrit dans la section 4.2. Puis nous définissons les automates associés à

[10] Le suffixe _all précise qu'une séquence doit être produite pour chacune des opérations disponibles.

la syntaxe abstraite et les règles d'association. Finalement, nous définissons la sémantique du langage comme un sous-ensemble des chemins de l'automate associé à un schéma de test.

Exemple

Nous introduisons ici l'exemple qui sera détaillé au cours de cette section. Il s'agit du schéma de test suivant, rédigé dans le cadre de la validation de l'application Demoney :

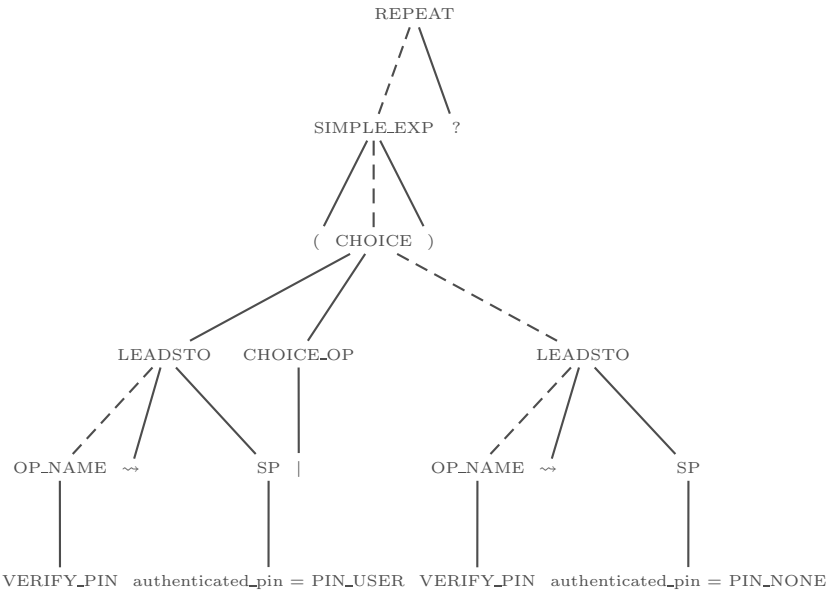
```
(1) | (VERIFY_PIN⇝(authenticated_pin = PIN_USER) | VERIFY_PIN⇝(authenticated_pin = PIN_NONE))?
(2) | .[INITIALIZE_TRANSACTION].COMMIT_TRANSACTION
```

Ce schéma volontairement simple décrit un ensemble de scénarios exerçant les différents comportements possibles de la réalisation d'une transaction (enchaînement des deux opérations INITIALIZE_TRANSACTION et COMMIT_TRANSACTION) en considérant trois cas différents d'authentification :

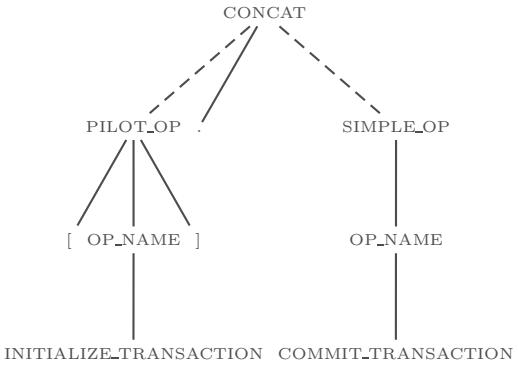
- soit l'authentification de l'utilisateur est réalisée avec succès ;
- soit l'essai d'authentification a échoué ;
- soit aucune tentative d'authentification n'a été réalisée.

On peut noter que ce schéma aurait pu être plus complexe en prenant en compte des scéarios complémentaires, par exemple en faisant une différence entre tentative d'authentification de la banque et de l'utilisateur, ou en envisageant un gain suivi d'une perte d'authentification. Mais étant dans le cas d'un exemple illustratif, nous proposons un schéma simple. Ci-dessous est présenté l'arbre d'analyse syntaxique du schéma à partir de la grammaire présentée dans la section 4.2. Les branches de dérivation représentées en pointillés sont les branches où certains nœuds intermédiaires n'ayant qu'un seul fils ont été omis lors de la dérivation.

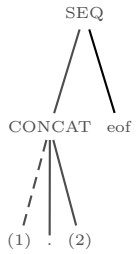
Reconnaissance de *(1)* :



Reconnaissance de *(2)* :



Reconnaissance de *(1).(2)* :



4.4.1 Syntaxe abstraite

Dans cette partie, nous présentons les nœuds de l'arbre de syntaxe abstraite associé à un schéma de test. La définition de la syntaxe abstraite minimise le nombre de constructions afin de minimiser le nombre de règles de transformation en automate. La figure 4.10 présente la correspondance entre la syntaxe concrète du langage (c.f. 4.2) et sa syntaxe abstraite. $i designe l'arbre de syntaxe abstraite associé au i^{e}non-terminal de la règle où $i apparaît. *Concat*, *ChoixI*, *ChoixE*, *Leadsto* et *Op* sont les différents types de nœuds de la syntaxe abstraite.

SEQ	::=	CONCAT eof	$1
CONCAT	::=	CHOICE	$1
	|	CHOICE "." CONCAT	Concat($1,$3)
CHOICE	::=	LEADSTO	$1
	|	LEADSTO "|" CHOICE	ChoixI($1,$3)
	|	LEADSTO "$\oplus$" CHOICE	ChoixE($1,$3)
LEADSTO	::=	REPEAT	$1
	|	REPEAT "$\rightsquigarrow$(" SP ")"	Leadsto($1,$3)
REPEAT	::=	SIMPLE_EXP	$1
SIMPLE_EXP	::=	"(" CONCAT ")"	$2
	|	SIMPLE_OP	$1
	|	PILOT_OP	$1
SIMPLE_OP	::=	OP_NAME	Op($1,$\perp$)
PILOT_OP	::=	"[" OP_NAME "/$_w$ {" CPT "}]"	Op($2,$4)
SP	::=	prédicat d'état	$1
OP_NAME	::=	nom d'opération	$1
CPT	::=	comportement	$1

FIG. 4.10 – Lien entre la syntaxe concrète et la syntaxe abstraite

Certaines constructions de la syntaxe concrète (fig. 4.6) n'apparaissent pas dans la figure 4.10. Ces constructions sont réécrites à partir d'autres constructions primitives. Il s'agit des constructions suivantes :

– les répétitions sont toutes réécrites sous la forme de séquences ayant un nombre constant d'éléments répétés. La constante fixant le nombre de répétitions est fournie par l'utili-

sateur (voir fig. 4.11 pour les règles de réecriture) ;
- le terminal $OP (voir fig. 4.12 pour les règles de réecriture) ;
- le filtrage des comportements d'une opération (voir fig. 4.13 pour les règles de réécriture).

$a*_{all}$	$\Longrightarrow$	$a\{0, max\}_{all}$ avec $max > 1$ une constante
$a*_{one}$	$\Longrightarrow$	$a\{0, max\}_{one}$ avec $max > 1$ une constante
$a+_{all}$	$\Longrightarrow$	$a\{1, max\}_{all}$ avec $max > 1$ une constante
$a+_{one}$	$\Longrightarrow$	$a\{1, max\}_{one}$ avec $max > 1$ une constante
$a?_{all}$	$\Longrightarrow$	$a\{0, 1\}_{all}$
$a?_{one}$	$\Longrightarrow$	$a\{0, 1\}_{one}$
$a\{n\},\ n > 0$	$\Longrightarrow$	$\underbrace{a.a.\cdots.a}_{n\ fois}$
$a\{0\}$	$\Longrightarrow$	ε
$a\{n, m\}_{all},\ n \geq 0\ et\ n < m$	$\Longrightarrow$	$a\{n\}.\underbrace{(a.(a.\cdots)\|\varepsilon)\|\varepsilon}_{(m-n)\ fois}$
$a\{n, m\}_{one},\ n \geq 0\ et\ n < m$	$\Longrightarrow$	$a\{n\}.\underbrace{(a.(a.\cdots)\oplus\varepsilon)\oplus\varepsilon}_{(m-n) fois}$
$a\{n,\}_{all},\ n \geq 0$	$\Longrightarrow$	$a\{n, max\}_{all}$ avec $max > 1$ une constante
$a\{n,\}_{one},\ n \geq 0$	$\Longrightarrow$	$a\{n, max\}_{one}$ avec $max > 1$ une constante

FIG. 4.11 – Règles de réécriture des répétitions

Soit $O = \{o_1, \ldots, o_n\}$ l'ensemble des opérations du modèle et $O' = O - \{op_1, \ldots, op_m\} = \{o'_1, \ldots, o'_k\}$

$\$OP_{all}$	$\Longrightarrow$	$(o_1\|\ldots\|o_n)$
$\$OP_{one}$	$\Longrightarrow$	$(o_1 \oplus \cdots \oplus o_n)$
$\$OP_{all}\backslash\{op_1, \ldots, op_m\}$	$\Longrightarrow$	$(o'_1\|\ldots\|o'_k)$
$\$OP_{one}\backslash\{op_1, \ldots, op_m\}$	$\Longrightarrow$	$(o'_1 \oplus \cdots \oplus o'_k)$

FIG. 4.12 – Règles de réécriture de $OP

Soit $C = \{c_1, \ldots, c_n\}$ l'ensemble des comportements de l'operation "*op*".

[op]	$\Longrightarrow$	$([\text{op}/_w\{c_1\}]\|\ldots\|[\text{op}/_w\{c_n\}])$
$[\text{op}/_w\{cpt_1, \ldots, cpt_m\}]$	$\Longrightarrow$	$([\text{op}/_w\{cpt_1\}]\|\ldots\|[\text{op}/_w\{cpt_m\}])$
$[\text{op}/_e\{cpt_1, \ldots, cpt_m\}]$	$\Longrightarrow$	$[\text{op}/_w(C - \{cpt_1, \ldots, cpt_m\})]$

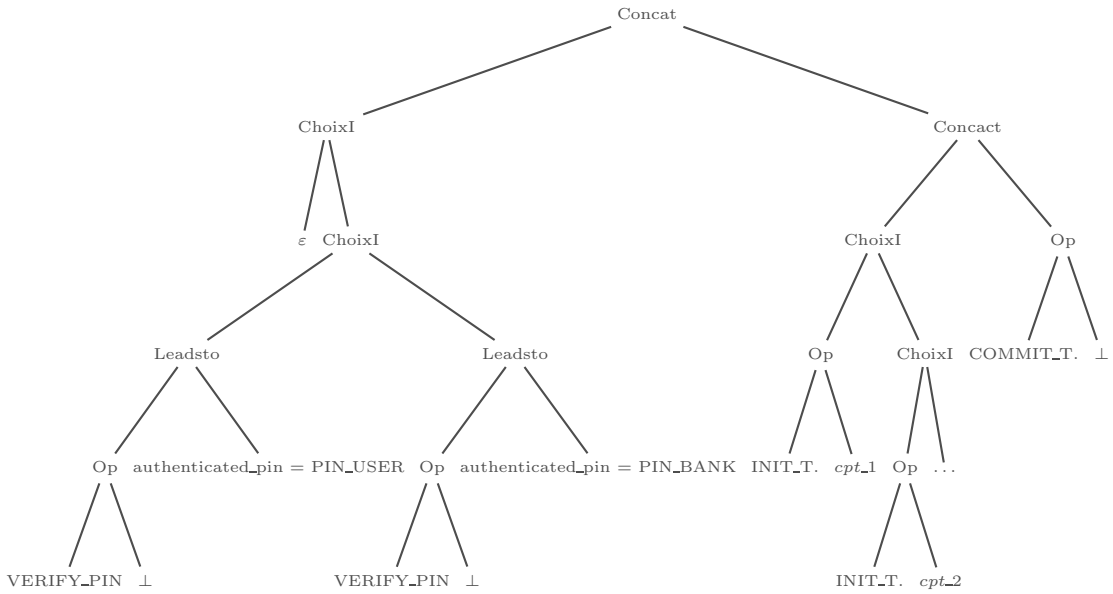
FIG. 4.13 – Règles de réécriture du filtrage de comportements

Exemple

Nous présentons ici l'arbre de syntaxe abstraite issu de l'exemple de schéma de test qui illustre ce chapitre :

```
(VERIFY_PIN⇝(authenticated_pin = PIN_USER) | VERIFY_PIN⇝(authenticated_pin = PIN_NONE))?
.[INITIALIZE_TRANSACTION].COMMIT_TRANSACTION
```

On peut remarquer la disparition du nœud de répétition borné entre 0 et 1 qui a été réécrit à l'aide de choix et de concaténations par application des règles de transformation présentées dans la figure 4.11.



4.4.2 Transformation en automate

Automate

Dans la définition 6 nous définissons un automate associé à un schéma de test sans directives de pilotage portant sur l'utilisation du choix exclusif. Il s'agit d'un automate dont les transitions sont étiquetées par les noms d'opérations du modèle et éventuellement par un identifiant de comportement de cette opération. Les états par des prédicats exprimés sur les variables d'état du modèle. La définition 7 présente l'automate associé à un schéma utilisant des directives de pilotage.

Définition 6 (Automate associé à un schéma sans directive de pilotage)**.** Un automate sans directive de pilotage est un 5-uplet : $\langle Q, q_0, T, \lambda, q_f \rangle$, avec

- Q un ensemble d'états ;
- q_0 l'état initial ($q_0 \in Q$) ;
- T un ensemble de transitions étiquetées par un couple nom d'opération, nom de comportement ($T \subseteq Q \times OP \times CPT \times Q$ où OP est l'ensemble des noms des opérations du modèle et CPT l'ensemble des noms de comportements) ;
- λ une fonction d'étiquetage des états ($\lambda : Q \rightarrow SP$ avec SP l'ensemble des prédicats exprimables sur les variables d'état du modèle) ;

– q_f l'état terminal ($q_f \in Q$).

L'exécution d'un automate $A = \langle Q, q_0, T, \lambda, q_f \rangle$ est une séquence de transitions $\sigma = t_0\, t_1\, \ldots\, t_n$ telle que :

– t_0 part de l'état initial ($t_0 = q_0 \xrightarrow{op,cpt} q_1$),

– pour tout $i \geq 0$, t_i est une transition de T ($q_i \xrightarrow{op_i,cpt_i} q_{i+1} \in T$) et l'état cible de t_i est égal à l'état source de t_{i+1},

– t_n atteint l'état final ($q_{n+1} = q_f$).

L'ensemble des exécutions de l'automate est noté Σ.

Définition 7 (Automate associé à un schéma avec directives de pilotage)**.** Un automate avec directives de pilotage A' est un 6-uplet $A' = \langle A, Q_e \rangle$ où Q_e est l'ensemble des états qui font l'objet d'un choix exclusif.

L'automate avec directive de pilotage (déf. 7) correspond au 5-uplet qui définit l'automate associé à un schéma (déf. 6) auquel nous ajoutons un ensemble d'état Q_e où Q_e est l'ensemble des états dont les transitions sortantes sont choisies de manière exclusive lors du parcours de l'automate pour l'animation du modèle.

Opérations sur les automates

Nous donnons, ici, trois définitions d'opérations sur les automates tels que décrits dans la définition 7 qui nous sont utiles pour définir les règles de transformation d'un schéma en automate. Parmi ces définitions, nous trouvons les opérations : de concaténation de deux automates (Déf. 8), l'union de deux automates correspondant soit à un choix inclusif (déf. 9), soit à un choix exclusif(déf. 10).

Définition 8 (Concaténation d'automates)**.** Soient deux automates $A_1 = \langle Q^1, q_0^1, T^1, \lambda^1, q_f^1, Q_e^1 \rangle$ et $A_2 = \langle Q^2, q_0^2, T^2, \lambda^2, q_f^2, Q_e^2 \rangle$ tels que $Q_1 \cap Q_2 = \varnothing$. La concaténation $A' = A_1.A_2$ est définie par : $A' = \langle Q^1 \cup Q^2, q_0^1, T^1 \cup T^2 \cup \{q_f^1 \xrightarrow{\varepsilon} q_0^2\}, \lambda^1 \cup \lambda^2, q_f^2, Q_e^1 \cup Q_e^2 \rangle$.

Définition 9 (Union de deux automates – pour un choix inclusif)**.** Soit $A_1 = \langle Q^1, q_0^1, T^1, \lambda^1, q_f^1, Q_e^1 \rangle$ et $A_2 = \langle Q^2, q_0^2, T^2, \lambda^2, q_f^2, Q_e^2 \rangle$ deux automates tels que $Q_1 \cap Q_2 = \varnothing$. L'union $A' = A_1 \cup A_2$ est définie par $A' = \langle Q^1 \cup Q^2 \cup \{q_\alpha, q_\beta\}, q^\alpha, T^1 \cup T^2 \cup \{q_\alpha \xrightarrow{\varepsilon} q_0^1, q_\alpha \xrightarrow{\varepsilon} q_0^2, q_f^1 \xrightarrow{\varepsilon} q_\beta, q_f^2 \xrightarrow{\varepsilon} q_\beta\}, \lambda^1 \cup \lambda^2 \cup \{q_\alpha \mapsto true, q_\beta \mapsto true\}, q_\beta, Q_e^1 \cup Q_e^2 \rangle$, où q_α et q_β sont deux nouveaux états n'appartenant ni à Q^1, ni à Q^2.

Définition 10 (Union de deux automates – pour un choix exclusif)**.** Soit $A_1 = \langle Q^1, q_0^1, T^1, \lambda^1, q_f^1, Q_e^1 \rangle$ et $A_2 = \langle Q^2, q_0^2, T^2, \lambda^2, q_f^2, Q_e^2 \rangle$ deux automates tels que $Q_1 \cap Q_2 = \varnothing$. L'union $A' = A_1 \oplus A_2$ est défine par $A' = \langle Q^1 \cup Q^2 \cup \{q_\alpha, q_\beta\}, q^\alpha, T^1 \cup T^2 \cup \{q_\alpha \xrightarrow{\varepsilon} q_0^1, q_\alpha \xrightarrow{\varepsilon} q_0^2, q_f^1 \xrightarrow{\varepsilon} q_\beta, q_f^2 \xrightarrow{\varepsilon} q_\beta\}, \lambda^1 \cup \lambda^2 \cup \{q_\alpha \mapsto true, q_\beta \mapsto true\}, q_\beta, Q_e^1 \cup Q_e^2 \cup \{q_\alpha\} \rangle$, où q_α et q_β sont deux nouveaux états n'appartenant ni à Q^1, ni à Q^2.

La figure 4.14 illustre la concaténation de deux schémas formalisée par la définition 8. La figure 4.15 présente le choix entre deux schémas. La différence entre les définitions 9 et 10, qui se traduit par l'appartenance ou non de l'état q_α à l'ensemble Q_e, n'est pas illustrée dans cette figure. La figure 4.16 illustre l'automate produit pour la construction *Leadsto*.

Règles de transformation

La définition 11 présente la fonction de transformation qui permet de traduire un schéma en automate, en fonction du type de noeud de l'arbre de syntaxe abstraite passé en paramètre.

L'automate produit par application des règles de transformation (déf. 11) contient un certain nombre d'ε-transitions qui sont en partie supprimées par un algorithme de suppression des productions vides.

Définition 11 (Transformation d'un schéma en automate)**.** La fonction f :Nœud$\rightarrow$ *Automate* transforme un schéma représenté sous forme d'arbre de syntaxe abstraite en automate tels qu'ils sont définis dans la définition 7. Soient s_1 et s_2 deux nœuds de syntaxe abstraite, p un prédicat exprimé sur les variables d'état du modèle, *op* un nom d'opération et *cpt* un nom de comportement (y compris $\perp$), la définition de la fonction f est la suivante :

Concatenation :

$f(Concat(s_1, s_2) = f(s_1).f(s_2)$

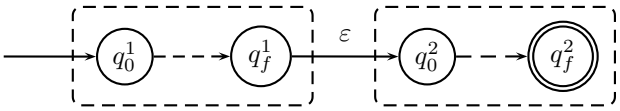


FIG. 4.14 – Concatenation de 2 schémas

Choix inclusif :

$f(ChoixI(s_1, s_2)) = f(s_1) \cup f(s_2)$

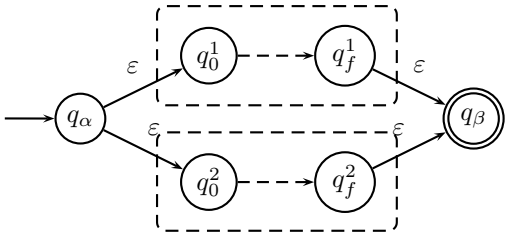


FIG. 4.15 – Choix entre 2 schémas

Choix exclusif :

$f(ChoixE(s_1, s_2)) = f(s_1) \oplus f(s_2)$

LeadsTo :

Soit $\langle Q, q_0, T, \lambda, q_f, Q_e\rangle = f(s_1)$ et p un prédicat, alors $f(Leadsto(s_1, p)) = \langle Q \cup \{q_\beta\}, q_0, T \cup \{q_f \xrightarrow{\varepsilon} q_\beta\}, \{q_\beta \mapsto p\}, q_\beta, Q_e\rangle$

Avec q_β un nouvel état($q_\beta \notin Q$).

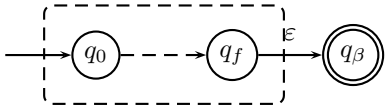


FIG. 4.16 – Etat à atteindre par un schéma (Leadsto)

Opération :
Soit *cpt* un nom de comportement ou $\perp$.
$f(Op(op, cpt)) =$
$\langle \{q_\alpha, q_\beta\}, q_\alpha, \{q_\alpha \xrightarrow{op,cpt} q_\beta\}, \{q_\alpha \mapsto true, q_\beta \mapsto true\}, q_\beta, \varnothing \rangle$

Exemple

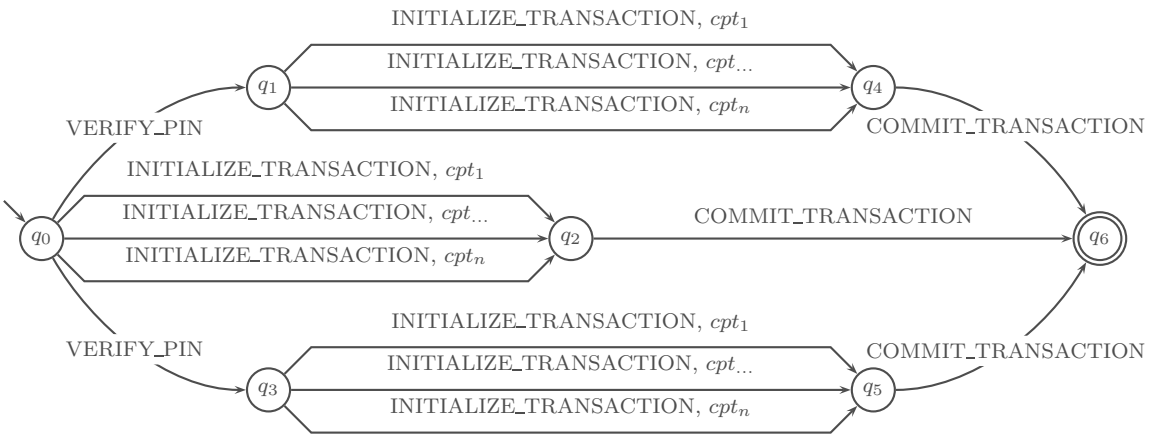
Nous donnons ici l'automate issu de l'exemple de schéma de test qui illustre ce chapitre après suppression des ε-transitions :

```
(VERIFY_PIN⇝(authenticated_pin = PIN_USER)
   | VERIFY_PIN⇝(authenticated_pin = PIN_NONE))?
.[INITIALIZE_TRANSACTION].COMMIT_TRANSACTION
```

La notation $cpt_{...}$ signifie qu'il y a autant de transitions que de comportements de l'opération concernée, $INITIALIZE\_T, cpt_{...}$ sur l'exemple.

Définition de l'automate :
$\mathcal{A} = \langle Q, q_0, T, \lambda, q_f, Q_e \rangle$
avec
$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$
$q_0 = q_0$
$T = \{(q_0, VERIFY\_PIN, \perp, q_1),$
$(q_0, INITIALIZE\_TRANSACTION, cpt_1, q_2),$
$(q_0, INITIALIZE\_TRANSACTION, cpt_{...}, q_2),$
$(q_0, INITIALIZE\_TRANSACTION, cpt_n, q_2),$
$(q_0, VERYFY\_PIN, \perp, q_3),$
$(q_1, INITIALIZE\_TRANSACTION, cpt_1, q_4),$
$(q_1, INITIALIZE\_TRANSACTION, cpt_{...}, q_4),$
$(q_1, INITIALIZE\_TRANSACTION, cpt_n, q_4),$
$(q_2, COMMIT\_TRANSACTION, \perp, q_6),$
$(q_3, INITIALIZE\_TRANSACTION, cpt_1, q_5),$
$(q_3, INITIALIZE\_TRANSACTION, cpt_{...}, q_5),$
$(q_3, INITIALIZE\_TRANSACTION, cpt_n, q_5),$
$(q_4, COMMIT\_TRANSACTION, \perp, q_6),$
$(q_5, COMMIT\_TRANSACTION, \perp, q_6)\}$
$\lambda = \{q_0 \mapsto true, q_1 \mapsto (authenticated\_pin = PIN\_USER), q_2 \mapsto true,$
$q_3 \mapsto (authenticated\_pin = PIN\_NONE), q_4 \mapsto true, q_5 \mapsto true, q_6 \mapsto true\}$
$q_f = q_6$
$Q_e = \varnothing$

L'illustration suivante représente l'automate du schéma que nous avons pris pour exemple :



4.4.3 Sémantique d'un schéma de test

L'automate produit à partir d'un schéma de test représente un ensemble d'exécutions constituées de séquences de transitions. Les tests produits à partir d'un schéma sont un sous-ensemble des exécutions, décrits par l'automate associé.

La restriction qui implique que les tests produits, à partir du parcours de l'automate associé à un schéma de test, ne correspondent qu'à un sous-ensemble des exécutions de cet automate provient de la sémantique de l'opérateur de choix exclusif. Dans l'automate produit, lorsqu'un état q est étiqueté comme correspondant à un choix exclusif, cela signifie que pour chaque séquence de transitions menant de l'état initial q_0 jusqu'à un état q, une seule transition sortant de l'état q mènera à la production d'une ou plusieurs séquences de test.

Nous définissons l'ensemble des tests correspondant à un automate $A' = \langle A, Q_e \rangle$ où $A = \langle Q, q_0, T, \lambda, q_f \rangle$ comme un sous-ensemble Σ' des exécutions Σ de A. Soit Σ l'ensemble des exécutions de A, nous définissons $\Sigma' \subseteq \Sigma$ comme un sous-ensemble des exécutions de Σ respectant cette condition :

$\Sigma' \subseteq \Sigma \wedge$

$\forall \sigma \in \Sigma', ((\sigma = \sigma_1.q \overset{op,cpt}{\longrightarrow} q'.\sigma_2 \wedge q \in Q_e) \Rightarrow (\nexists \sigma' = \sigma_1.q \overset{op',cpt'}{\longrightarrow} q''.\sigma_3 \wedge q' \neq q''))$
$\wedge (\nexists \Sigma'' \subseteq \Sigma \wedge \Sigma' \subset \Sigma'' \wedge$

$\forall \sigma \in \Sigma'', ((\sigma = \sigma_1.q \overset{op,cpt}{\longrightarrow} q'.\sigma_2 \wedge q \in Q_e) \Rightarrow (\nexists \sigma' = \sigma_1.q \overset{op',cpt'}{\longrightarrow} q''.\sigma_3 \wedge q' \neq q''))).$

Cette condition est exprimée en deux temps. D'abord, elle définit qu'il n'existe pas deux exécutions appartenant à Σ' qui partagent la même séquence de transitions entre l'état initial q_0 et un état de choix exclusif q', et qui activent des transitions sortantes de q' différentes. Ensuite, elle définit qu'il n'existe pas d'ensemble d'exécutions Σ'', respectant les mêmes conditions que Σ', qui contiendrait plus d'exécution que Σ' ($\Sigma' \subset \Sigma''$).

Exemple

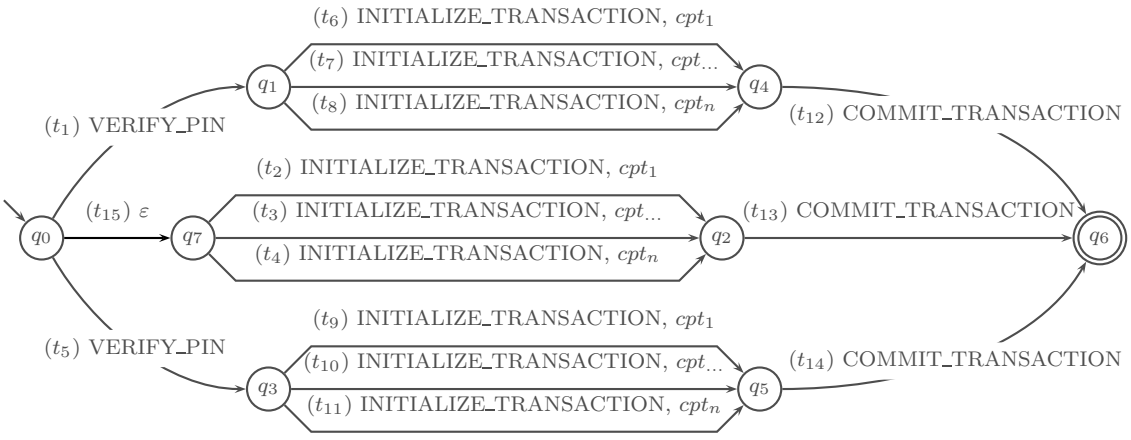
Nous reprenons l'exemple qui illustre ce chapitre en le modifiant de manière à faire correspondre le premier état de l'automate à un choix exclusif :

```
(VERIFY_PIN⇝(authenticated_pin = PIN_USER)
   ⊕ VERIFY_PIN⇝(authenticated_pin = PIN_NONE))?one
```

`.[INITIALIZE_TRANSACTION].COMMIT_TRANSACTION`

L'automate devient :
$\mathcal{A} = \langle Q, q_0, T, \lambda, q_f, Q_e \rangle$
avec
$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}$
$q_0 = q_0$
$T = \{(q_0, VERIFY\_PIN, \bot, q_1),$
$(q_0, \varepsilon, \bot, q_7),$
$(q_0, VERYFY\_PIN, \bot, q_3),$
$(q_1, INITIALIZE\_TRANSACTION, cpt_1, q_4),$
$(q_1, INITIALIZE\_TRANSACTION, cpt_{...}, q_4),$
$(q_1, INITIALIZE\_TRANSACTION, cpt_n, q_4),$
$(q_7, INITIALIZE\_TRANSACTION, cpt_1, q_2),$
$(q_7, INITIALIZE\_TRANSACTION, cpt_{...}, q_2),$
$(q_7, INITIALIZE\_TRANSACTION, cpt_n, q_2),$
$(q_2, COMMIT\_TRANSACTION, \bot, q_6),$
$(q_3, INITIALIZE\_TRANSACTION, cpt_1, q_5),$
$(q_3, INITIALIZE\_TRANSACTION, cpt_{...}, q_5),$
$(q_3, INITIALIZE\_TRANSACTION, cpt_n, q_5),$
$(q_4, COMMIT\_TRANSACTION, \bot, q_6),$
$(q_5, COMMIT\_TRANSACTION, \bot, q_6)\}$
$\lambda = \{q_0 \mapsto true, q_1 \mapsto (authenticated\_pin = PIN\_USER), q_2 \mapsto true,$
$q_3 \mapsto (authenticated\_pin = PIN\_NONE), q_4 \mapsto true, q_5 \mapsto true, q_6 \mapsto true, q_7 \mapsto true\}$
$q_f = q_6$
$Q_e = \{q_0\}$

L'illustration suivante représente l'automate décrit ci-dessus où l'état q_0 est un état correspondant à un choix exclusif :



La sémantique de cet automate est définie par l'un des trois ensembles d'exécution suivants :
$\{t_1 t_6 t_{12},\ t_1 t_7 t_{12},\ t_1 t_8 t_{12}\}$

ou $\{t_{15}t_2t_{13},\ t_{15}t_3t_{13},\ t_{15}t_4t_{13}\}$
ou $\{t_5t_9t_{14},\ t_5t_{10}t_{14},\ t_5t_{11}t_{14}\}$
La sémantique du même automate, où l'état q_0 ne correspondrait pas à un choix exclusif dans le schéma de test, serait constituée de l'ensemble des exécutions suivantes :
$\{t_1t_6t_{12},\ t_1t_7t_{12},\ t_1t_8t_{12},\ t_{15}t_2t_{13},\ t_{15}t_3t_{13},\ t_{15}t_4t_{13},\ t_5t_9t_{14},\ t_5t_{10}t_{14},\ t_5t_{11}t_{14}\}$

4.5 Résumé

Dans ce chapitre, nous avons présenté le langage de description d'objectifs de test que nous proposons. Ce langage permet d'exprimer à l'aide de schémas de tests des ensembles de scénarios de tests décrits à l'aide d'enchaînements d'appels d'opérations et d'états du système à atteindre (décrits symboliquement par des prédicats). A cette grammaire de base s'ajoute des directives de pilotage destinées à guider plus finement la génération des tests au travers de la sélections des chemins et du filtrage des comportements des opérations.

Nous avons également présenté la sémantique de ce langage à l'aide d'automates définis sous la forme de systèmes de transitions étiquetés symboliques. La génération de tests à partir d'un automate issu d'un schéma de test et du modèle du système à tester est présentée de manière détaillée dans le chapitre suivant.

Chapitre 5

Génération des tests

Sommaire

Nous présentons ici la démarche de génération de tests à partir d'objectifs de test décrivant des critères de sélection dynamiques, appliquée à des modèles formalisés en langage B. Notre objectif est de produire des tests complémentaires aux tests fonctionnels générés automatiquement à partir de critères de couverture structurelle appliqués sur les opérations du modèle. Nous détaillons, dans ce chapitre, la génération de tests à partir d'objectifs de test. Nous présentons successivement :

- La démarche générale de génération, afin de placer les divers éléments dans leur contexte.
- Une méthode de conception et de formalisation d'objectifs de test, qui constituent la principale intervention humaine (avec la modélisation du système) et qui par conséquent est une étape importante du processus de validation.
- Une solution de génération de tests basée sur la synchronisation de l'automate représentant l'objectif de test avec le modèle du système qui permet de construire un modèle dont l'ensemble des exécutions possibles est restreint par l'objectif de test.
- Une solution de génération de tests basée sur l'animation du modèle B du système guidée par l'automate associé au schéma de test.

5.1 Démarche de génération de tests

Nous présentons ici la démarche de génération de tests illustrée par la figure 5.1. La mise en œuvre de cette démarche débute avec l'identification de propriétés définies sur le système et des besoins de test qui leur sont associés. Ces informations sont ensuite formalisées sous forme d'objectifs de tests. La formalisation d'un objectif de test, qui est la description d'un

ensemble de scénarios visant à couvrir des propriétés et des besoins de test est réalisée par la conception d'un schéma de test.

Cet objectif de test est ensuite traduit en un automate conforme à la définition 7 du chapitre 4. Cet automate (constitué de noms d'opérations, de contraintes sur les états et les comportements d'opérations, et de directives de pilotage) représente les chemins d'exécutions symboliques pour lesquels on souhaite générer des tests. Les tests abstraits sont générés à partir du modèle comportemental du système et de l'automate issu du schéma de test qui est utilisé comme critère de sélection.

Les tests abstraits, ainsi générés, sont constitués d'une séquence d'appels d'opérations dont les paramètres sont valués et les valeurs de retour sont connues. Ces tests ont le même niveau d'abstraction que celui du modèle comportemental et ne sont pas exécutables en l'état. Ces tests doivent donc être concrétisés, c'est-à-dire, qu'ils doivent être amenés au même niveau d'abstraction que celui de l'implémentation pour les rendre exécutables (voir le chapitre 6 pour plus de détails sur la concrétisation et l'exécution des tests).

Une fois les tests concrets obtenus, soit ceux-ci sont directement exécutables sur le système (par exemple des tests concrétisés sous la forme de scripts), soit ils sont exécutés à l'aide d'un banc de tests qui sert d'interface entre les tests et le système. Lors de l'exécution des tests, les verdicts sont établis par comparaison des réponses du système sous test et de celles prévues par le modèle. Étant dans une situation de test boîte noire, les verdicts sont obtenus en comparant les valeurs de retour des opérations.

Une fois que les tests ont été exécutés, l'ingénieur validation dispose pour chaque test d'un verdict qui indique si le test a été exécuté avec succès ou non. Les verdicts permettent de savoir si un test a révélé une non-conformité (cas d'un test en échec) entre le système sous test et le modèle de ce dernier. Si une non-conformité a été détectée, il faut alors déterminer d'où provient cette dernière, la corriger puis relancer à nouveau l'exécution des tests afin de valider la correction apportée. Dans le cadre d'un processus de validation correctement mené, le modèle à partir duquel sont générés les tests (et par conséquent calculés les oracles) doit avoir été l'objet de vérification de conformité vis-à-vis des spécifications du système. Dans ce cas, le risque d'erreurs de modélisation est fortement réduit et par conséquent les causes du problème de conformité relevé par un cas de test en échec peuvent-être recherchées en priorité du côté de l'implémentation du système. Souvent, le modèle n'étant pas l'objet d'une vérification exhaustive de sa conformité avec les spécifications, il convient tout de même de vérifier si l'échec de l'exécution d'un test n'est pas due à un problème de modélisation.

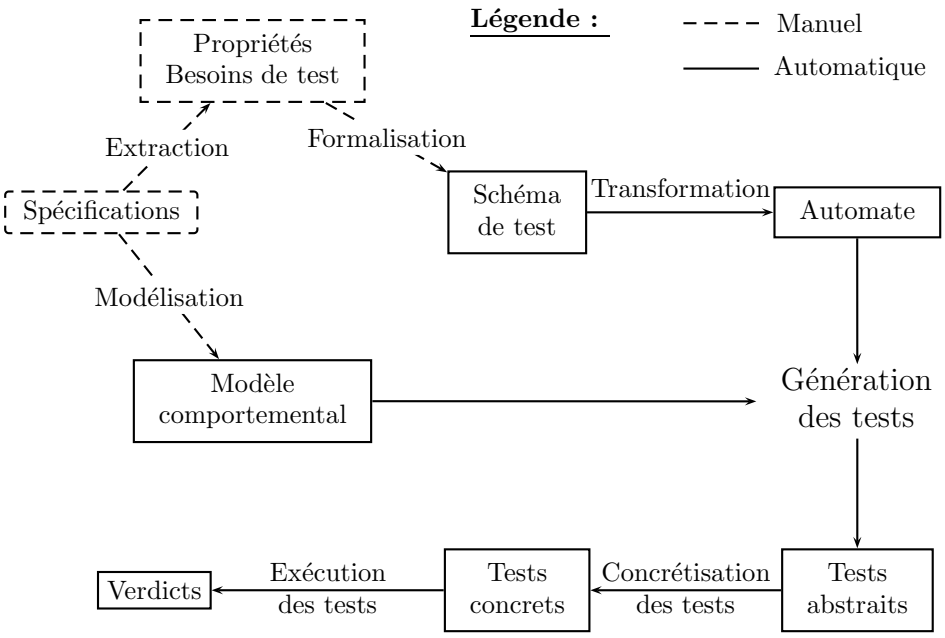


FIG. 5.1 – Processus de génération de tests

5.2 Conception et formalisation de l'objectif de test

5.2.1 Des propriétés aux objectifs de test

La première étape qui s'impose dans la démarche de génération de tests ayant pour but de valider la sécurité (et plus généralement des propriétés de haut niveau) d'un système réside dans le fait d'identifier l'ensemble des propriétés que le système doit satisfaire. Deux options se dégagent : soit ces propriétés sont clairement définies dans les spécifications du système, le cahier des charges ou tout autre document de travail (par exemple, une analyse de vulnérabilité) ; soit elles doivent être déduites des documents de spécification par l'ingénieur de validation. Dans ce dernier cas, l'ingénieur de validation doit posséder une expertise suffisante et une connaissance appronfondie des spécifications pour déduire, des éléments dont il a connaissance, une série de propriétés.

Exemple

Nous nous plaçons dans le cadre de la validation du porte-monaie électronique Demoney (c.f. 4.1 et Annexe 1) et plus particulièrement dans le cadre de la validation de l'opération de crédit. La spécification précise que le rechargement de la carte nécessite l'authentification de l'utilisateur (à l'aide de la commande VERIFY_PIN), l'initialisation d'une transaction (par un appel à INITIALIZE_TRANSACTION) et une validation de la transaction (par la commande COMMIT_TRANSACTION). Une lecture plus précise de la spécification nous permet de dégager un certain nombre de propriétés qui devront faire l'objet de validation sur le système :

- l'utilisateur doit être authentifié pour initier une transaction ;
- trois échecs successifs d'authenfication de l'utilisateur bloquent la carte (celle-ci peut éventuellement être débloquée par la banque) ;
- la carte doit être dans l'état du cycle de vie "USE" pour pouvoir initier une transaction ;
- le solde de la carte doit être toujours inférieur à une valeur maximum qui a été définie à la personnalisation de la carte ;
- pour pouvoir être validée, une transaction doit avoir été initiée ;
- tout appel à une autre commande que COMMIT_TRANSACTION suivant l'initialisation d'une transaction a pour effet d'annuler la transaction en cours.

Les propriétés étant identifiées, il reste à déterminer les besoins de tests complémentaires aux tests fonctionnels qui ont pu être générés à partir de critères de sélection statiques. Il s'agit donc, dans un premier temps, de constater comment les propriétés identifiées ont été couvertes par la (ou les) campagne(s) de tests fonctionnels déjà réalisée(s). Ce travail pourrait être réalisé par une revue des tests déjà générés, mais le niveau de connaissance de l'ingénieur validation, concernant la mise en œuvre des campagnes de test fonctionnel, pourra lui suffire pour inférer sur la manière dont chaque propriété aura été couverte par ces tests.

Exemple

Dans le cadre de la validation d'une transaction de crédit d'un porte-monaie électronique de type Demoney, nous considérons les critères de couverture appliqués au modèle afin de générer des tests fonctionnels. Ces critères de sélection sont basés sur la couverture des comportements des opérations VERIFY_PIN, INITIALIZE_TRANSACTION et COMMIT_TRANSACTION. Nous pouvons donc supposer que chaque comportement de succès ou d'échec de ces opérations aura été couvert. Mais les tests générés en utilisant ce critère de sélection sont les séquences d'opération les plus courtes permettant d'activer le comporte-

ment visé. Par conséquent, un grand nombre de scénarios restent non adressés. Le tableau ci-dessous reprend les différentes propriétés que nous avons énoncées sur les transactions de crédit et pointe différents scénarios qui ne sont pas couverts par les tests fonctionnels :

Propriétés	Scénarios non adressés pour un crédit
L'utilisateur doit être authentifié pour initier une transaction.	– L'authentification a été acquise puis perdue. – Il y a eu un ou plusieurs échecs d'authenfication (éventuellement avant un succès).
Trois échecs successifs d'authenfication de l'utilisateur bloquent la carte (celle-ci peut éventuellement être débloquée par la banque).	Scénarios débutant par trois échecs successifs d'authentification de l'utilisateur puis : – divers essais de crédits (avec différentes erreurs possibles), – différents essais d'authentifications (banque, utilisateur) avant des essais de crédit, – différents essais de déblocage de la carte (succès, échecs) avant des essais de crédit.
La carte doit être dans l'état de cycle de vie "USE" pour pouvoir initier une transaction.	– La carte est dans l'état "bloqué" – La carte est dans l'état "définitivement désactivée"
Pour pouvoir être validée, une transaction doit avoir été initiée.	– Une tentative d'initialisation a eu lieu, mais a été un échec pour une raison ou une autre. – Une transaction a bien été initialisée, mais elle a été annulée par l'exécution d'une autre commande.
Tout appel à une autre commande que COMMIT_TRANSACTION suivant l'initialisation d'une transaction a pour effet d'annuler la transaction en cours.	Scénarios de rupture de séquence entre INITIALIZE_TRANSACTION et COMMIT_TRANSACTION avec toutes les opérations possibles (y compris) des appels à INITIALIZE_TRANSACTION et COMMIT_TRANSACTION.

En plus des différents scénarios que nous avons envisagés dans le tableau, les tests fonctionnels ne prennent pas en compte les différentes combinaisons possibles de ces scénarios au sein d'un même test. Or, il peut être intéressant d'observer par exemple l'enchaînement de plusieurs comportements d'erreur. De la même manière, il peut être intéressant d'observer la succession de plusieurs transactions de crédit.

L'énoncé des besoins de test est par la suite guidé par la question de savoir comment doit être complétée la couverture d'une propriété par rapport aux tests fonctionnels précédement obtenus. Une propriété peut donner lieu à plusieurs besoins de test, chacun devra couvrir une partie bien définie de la propriété initiale. De la bonne définition de la portée du besoin de test dépendra plus tard la facilité de l'analyse des verdicts des tests exécutés, puisqu'il sera alors simple de rattacher un test à la propriété et au besoin de test dont il dépend.

L'objectif de test naît de l'application d'un besoin de test à une propriété. Il s'agit d'une instanciation d'un (ou de plusieurs) besoin(s) de test exprimée sur les éléments du modèle et éventuellement complétée par un ensemble de directives de pilotage qui permettent de définir de manière précise la portée de l'objectif de test. Dans le cas où l'objectif de test couvre plusieurs besoins de test, il est important que ceux-ci adressent des problématiques

de validation suffisamment proches ; d'une part pour faciliter la formalisation de cet objectif sous forme de schémas et d'autre part pour ne pas nuire à la traçabilité des tests produits par cet objectif qui devront être facilement rattachés à des propriétés.

5.2.2 Formalisation de l'objectif de test

Le fait qu'un objectif de test adresse un certain nombre de scénarios suffisamment proches rend sa formalisation aisée sous forme de schéma. Le langage de description des objectifs de test (c.f. chapitre 4) étant basé sur les expressions régulières, celui-ci facilite la formalisation rapide et intuitive d'un objectif de test. Ainsi, la première partie du schéma que l'on peut qualifier de préambule[11] est souvent commune à l'ensemble des scénarios décrits par l'objectif de test. Les constructions du langage de formalisation telles que le choix permettent aisément de factoriser cette partie de l'objectif de test.

Les scénarios qui nécessitent la répétition d'une opération, ou d'une séquence d'opérations tireront profit des opérateurs de répétition bornée ou non. Dans ce cas, les directives de couverture de chemins permettront de distinguer deux cas différents :

- soit l'on souhaite obtenir des tests mettant en jeu tous les chemins produits par le dépliage d'une boucle, ce qui est le cas "par défaut" de l'utilisation de l'opérateur de répétition.
- soit la répétition d'une opération ou d'un ensemble d'opérations n'a pour unique objectif que d'atteindre un état spécifique du système, et l'on peut modifier la sémantique de l'opérateur de répétition en suffixant l'opérateur de boucle par "*_one*". Cette modification aura pour effet de limiter le dépliage de la boucle en un unique chemin permettant d'atteindre l'état visé.

La construction la plus importante pour formaliser un objectif de test est sans doute le *leadsto* : cet opérateur permet de spécifier un état à atteindre par une sous-séquence du schéma. Le *leadsto* va permettre de contraindre les comportements des opérations de la sous-séquence de manière plus simple que de définir pour chaque opération le comportement que l'on souhaite activer. Le *leadsto* va également permettre de définir de manière simple un point de passage pour un scénario sous la forme d'un prédicat exprimé sur les variables d'état du système.

Exemple

Nous avons vu que pour effectuer un rechargement d'un porte-monaie électronique Demoney, l'utilisateur doit être authentifié. Exprimer le passage d'un état où l'utilisateur n'est pas authentifié à un état où il a acquis l'authentification grâce à la commande VERIFY_PIN est aisé. Il suffit d'exprimer un appel à la commande VERIFY_PIN qui doit mener dans un état où la variable *authenticated_pin* a la valeur *PIN_USER* :

VERIFY_PIN$\rightsquigarrow$(authenticated_pin = PIN_USER)

L'opération générique notée "*$OP*" permet de définir un appel à n'importe quelle opération du modèle. Cette construction, conjuguée avec l'utilisation d'un *leadsto* et éventuellement d'un opérateur de répétition, permet de spécifier facilement une séquence d'opérations pourvue d'un objectif exprimé sous forme d'état sans se soucier du nom et de l'ordre des opérations qui seront utilisées. D'autres utilisations peuvent également être intéressantes, par exemple, si plusieurs

[11] Nous entendons par préambule, une séquence ou un ensemble de séquences qui visent à positionner le système dans un (ou plusieurs) état(s) nécessaire(s) pour atteindre le but visé par l'objectif de test.

opérations peuvent avoir le même effet sur l'état du système, plutôt que de les énumérer, il est plus simple d'utiliser l'opération générique en décrivant l'effet attendu sur le système sous la forme d'un prédicat et de laisser le solveur de contraintes instancier correctement le nom de l'opération et ses paramètres. De plus, dans un cas comme celui-ci, les directives de couverture de chemins nous permettent de définir si l'on souhaite générer un test par opération possible, ou uniquement un test avec la première opération possible. Il est bon de noter que l'utilisation d'un opérateur de répétition sur le terminal $OP est source d'explosion combinatoire, par exemple si on imagine un schéma de la forme $\$OP\{7\} \rightsquigarrow (etat\_carte = dead)$ alors que le modèle possède 7 opérations, la combinatoire induit un nombre de chemins à explorer égal à $7^7 = 823543$.

Exemple

La spécification Demoney précise que l'enchaînement des opérations d'initialisation et de validation de la transaction doit être atomique. C'est à dire que les deux opérations doivent se suivre sans occurence intermédiaire d'une autre commande. Si nous souhaitons exprimer le besoin de valider cet aspect de la spécification, l'instruction "$OP" nous permet de modéliser l'occurence de n'importe quelle autre opération lors de la séquence :

INITIALIZE_TRANSACTION$\rightsquigarrow (debit < 0 \wedge solde - debit \leq max_solde)$
.$OP_all
.COMMIT_TRANSACTION

L'utilisation du suffixe "_all" est une directive de pilotage qui permet d'exprimer le fait que nous souhaitons que toutes les instanciations de "$OP" par une opération du modèle donnent lieu à un test. Si pour des raisons de maîtrise du nombre de tests, nous ne souhaitons obtenir qu'une seule instanciation de cette instruction, il faut utiliser le suffixe "_one".

5.2.3 Utilisation des directives de pilotage

L'utilisation des directives de pilotage va permettre de préciser la couverture de l'objectif par l'introduction de l'opérateur de choix exclusif et par la quantification des comportements que l'on souhaite couvrir pour une opération donnée.

Directives reposant sur l'opérateur de choix exclusif

L'introduction de l'opérateur de choix exclusif va permettre de définir des alternatives optionnelles à un chemin du schéma.

La première utilisation du choix exclusif s'exprime avec l'utilisation des opérateurs de répétition. Dans ce cas, le choix exclusif est introduit par le suffixe *_one* après l'opérateur de répétition. Sa sémantique exprime le fait que seul le plus petit nombre d'itérations nécessaires pour construire un cas de test sera considéré.

Exemple

Considérons le schéma suivant où nous appliquons entre 1 et 5 fois l'opération VERIFY_PIN pour atteindre un état où la carte est bloquée :

$$VERIFY\_PIN\{1,5\}_{all} \rightsquigarrow (etat\_carte = invalid)$$

Le nombre de répétition compris entre 1 et 5 peut dénoter le fait que l'on ne sache pas combien d'erreurs d'authentification mènent à un état de blocage de la carte, ou que cette valeur est susceptible de changer. Si le nombre d'échecs d'authentification qui mènent à un

état de blocage est égal à 3, ce schéma produit 3 séquences de tests constituées de 3, 4 et 5 appels à l'opération VERIFY_PIN.

Considérons ce même schéma dans lequel on remplace le suffixe *_all* par *_one* :

$$VERIFY\_PIN\{1,5\}_{one} \rightsquigarrow (etat\_carte = invalid)$$

Si, le nombre d'échec d'authentification menant à un état de blocage de la carte est de 3, on obtient alors une unique séquence de test composée de trois appels successifs à l'opération VERIFY_PIN. Ceci nous permet de réduire le nombre de tests générés en ne conservant que ceux qui nous semblent pertinents.

La seconde utilisation du choix exclusif a déjà été présentée, c'est également une utilisation implicite du choix exclusif. Il s'agit du suffixe *_one* appliqué au terminal $OP (utilisé pour exprimer un appel à n'importe quelle opération du modèle). Le suffixe *_one* signifie dans ce cas que le le terminal $\$OP_{one}$ est remplacé par un choix exclusif entre toutes les opérations du modèle.

La dernière utilisation du choix exclusif repose sur l'utilisation explicite de l'opérateur de choix exclusif. Il peut être utile pour limiter la combinatoire induite par la succession de plusieurs choix qui mènent rapidement à l'explosion du nombre de chemins à explorer.

Exemple

Considérons le schéma suivant où plusieurs séquences d'actions (les pointillés) mène à l'initialisation d'une transaction de crédit ou de débit. Ici, nous avons modélisé le choix entre un crédit et un débit car pour certaines séquences d'actions seule l'une ou l'autre de ces transactions est possible (solde de la carte nul ou solde maximal atteint) alors que l'on souhaite uniquement initier une transaction valide (montant différent de 0).

$$\begin{array}{l}\ldots \\ .(INITIALIZE\_TRANSACTION \rightsquigarrow (debit > 0) \\ |(VERIFY\_PIN.INITIALIZE\_TRANSACTION) \rightsquigarrow (debit < 0))\end{array}$$

Ainsi rédigé, ce schéma implique que lorsqu'une séquence d'actions menant au choix permet d'initialiser à la fois un crédit et un débit, alors les deux solutions sont explorées. Si l'on ne souhaite voir apparaître qu'une seule transaction pour chaque séquence d'actions menant au choix, il nous faut utiliser un choix exclusif :

$$\begin{array}{l}\ldots \\ .(INITIALIZE\_TRANSACTION \rightsquigarrow (debit > 0) \\ \oplus(VERIFY\_PIN.INITIALIZE\_TRANSACTION) \rightsquigarrow (debit < 0))\end{array}$$

Directives reposant sur la couverture de comportements

Nous avons introduit dans le langage la possibilité de demander à ce que tous les comportements d'une opération soient explorés. Contrairement à l'utilisation du choix exclusif qui limite le nombre de chemin de l'automate à explorer, la couverture de tous les comportements d'une opération va les multiplier.

Exemple

On souhaite valider le comportement de l'opération de validation d'une transaction (COMMIT_TRANSACTION) lorsqu'aucune transaction n'est initialisée. On sait que les tests fonctionnels couvrent le cas où aucun appel à l'opération INITIALIZE_TRANSACTION n'a été réalisé, et par conséquent, on souhaite produire des tests où un appel à la commande INITIALIZE_TRANSACTION a été réalisé, mais a été un échec. Un premier schéma pourrait être :

$$INITIALIZE\_TRANSACTION \rightsquigarrow (debit = 0).COMMIT\_TRANSACTION$$

Cette solution, n'est pas la plus exhaustive, car seul un comportement d'échec de la commande INITIALIZE_TRANSACTION sera exploré. Aussi, nous pouvons modifier notre schéma de la manière suivante :

$$[INITIALIZE\_TRANSACTION] \rightsquigarrow (debit = 0).COMMIT\_TRANSACTION$$

Cette modification induit que tous les comportements de l'opération INITIALIZE_TRANSACTION qui laisseront inchangé le montant du débit ou crédit en cours seront explorés.

5.3 Production de tests par synchronisation du modèle et du schéma

Nous décrivons dans cette section la première solution de génération de tests à partir de schémas de tests que nous avons expérimentée. Cette solution de génération de tests était destinée à être utilisée avec l'outil LTG. N'ayant pas la possibilité d'ajouter des contraintes sur l'état des variables du système et sur les paramètres des opérations pour guider l'animation lors de la génération de tests, nous avons dû mettre en place une solution basée essentiellement sur l'instrumentation du modèle du système à valider.

La solution que nous proposons est basée sur la synchronisation du modèle B avec le schéma de test dans le but de restreindre les exécutions du modèle à celles décrites dans le schéma de test. Ensuite, les chemins extraits de l'automate (sous la forme de séquences de transition) sont utilisés pour guider l'animation du modèle afin de produire les tests. La figure 5.2 présente l'instanciation de notre démarche avec cette solution de génération de tests.

La figure 5.3 présente la synchronisation de la machine B, modélisant le système, avec l'objectif de test. La partie gauche de la figure présente la machine B modélisant le système, celle-ci est constituée de différentes clauses :

- CONSTANTS qui contient les déclarations des noms de constantes du modèle ;
- PROPERTIES qui contient les propriétés exprimées sur les constantes ;
- SETS qui contient les ensembles constants énumérés ou non ;
- VARIABLES qui contient les déclarations des noms de variables ;
- INVARIANT qui contient les propriétés invariantes exprimées sur les variables ;
- INITIALISATION qui contient l'initialisation des variables du modèle ;
- OPERATIONS qui contient les opérations modélisant la dynamique du système.

La partie droite de la figure 5.3 présente la machine synchronisée avec l'objectif de test. Afin de synchroniser le modèle initial et l'objectif de test, une nouvelle machine B est créée, elle inclut la machine B initiale. En plus de l'inclusion de la machine B initiale, elle comporte un ensemble $Q = \{q_0, \ldots, q_n\}$ qui représente les états de l'automate et une variable $currentState \in Q$ qui représente l'état courant lors du parcours de l'automate.

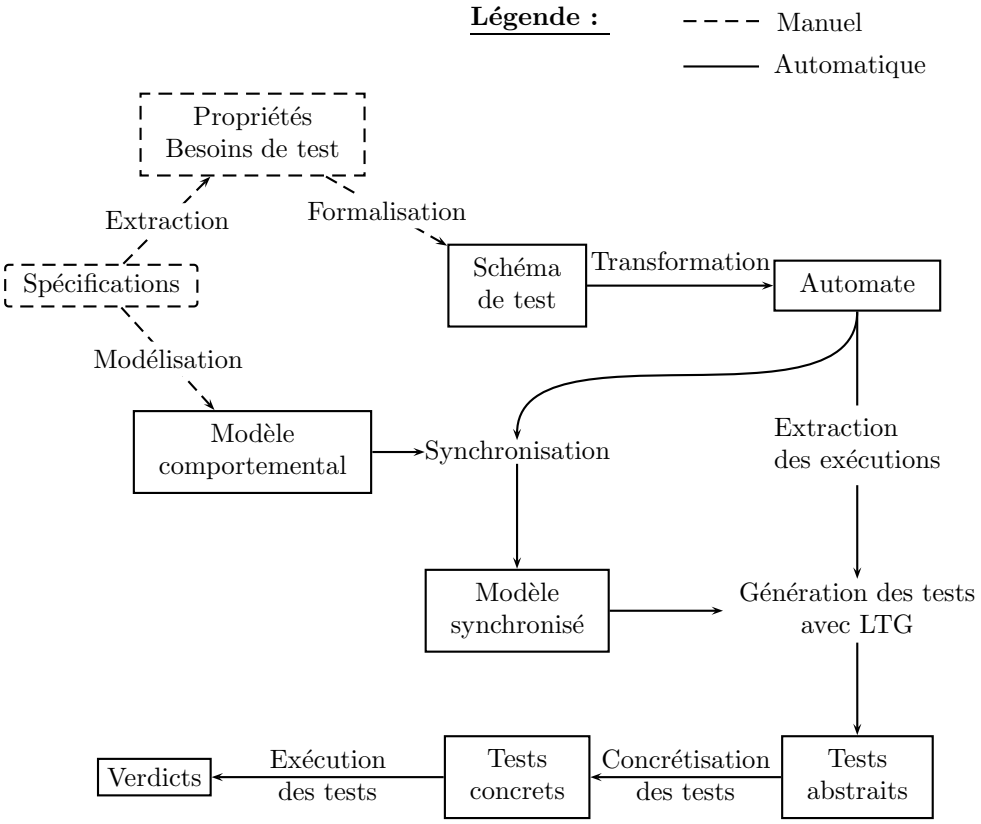


FIG. 5.2 – Processus de génération de tests

Les opérations de la machine synchronisée représentent les transitions de l'automate. Chacune de ces opérations correspond à une transition t_i issue de l'automate de la forme $q \overset{op_i,cpt}{\longrightarrow} q'$. Ces nouvelles opérations possèdent les mêmes paramètres, les mêmes valeurs de retour et les mêmes préconditions que l'opération décorant la transition qu'elle représente. Les préconditions de l'opération sont renforcées afin de modéliser les contraintes sur les états d'entrée et de sortie de la transition et les contraintes sur le comportement de l'opération. Ainsi, en plus des préconditions de l'opération décorant la transition, les préconditions de l'opération modélisant la transition contiennent :

- une contrainte sur l'état courant du parcours de l'automate afin de déterminer si la transition est activable : $currentState = q$;
- une contrainte portant sur l'état d'arrivée de la transition ($\lambda(q')$) et sur le comportement de l'opération (cpt) : $\exists(\overline{x'}, \overline{sw_i'}) \cdot (prd_{\overline{x}}(S_i) \wedge [\overline{x} := \overline{x'}]\lambda(q') \wedge cpt)$ où $\overline{x}$ dénote le n-uplet des variables du modèle. Cette contrainte est exprimée à l'aide du prédicat avant/après noté $prd_x(S)$ qui étant donné une substitution S définit la relation entre les valeurs d'une (ou plusieurs) variable(s) x avant et après la subtitution S (x' dénotant l'état de la variable x après la substitution S).

L'ajout de ces éléments à la précondition de l'opération permet d'assurer que cette opération n'est déclenchable que si les contraintes exprimées sur la transition qu'elle représente sont respectées.

Le corps de l'opération t_i représentant la transition est quant à lui constitué d'un appel à

l'opération op_i décorant la transition avec les paramètres p_i. Ces paramètres respectent à la fois les préconditions de op_i et les contraintes complémentaires exprimées dans la précondition de l'opération t_i. En plus de l'appel à op_i, le corps de l'opération t_i contient une affectation qui permet de modifier l'état courant du parcours de l'automate et ainsi de marquer le fait que la transition a été franchie.

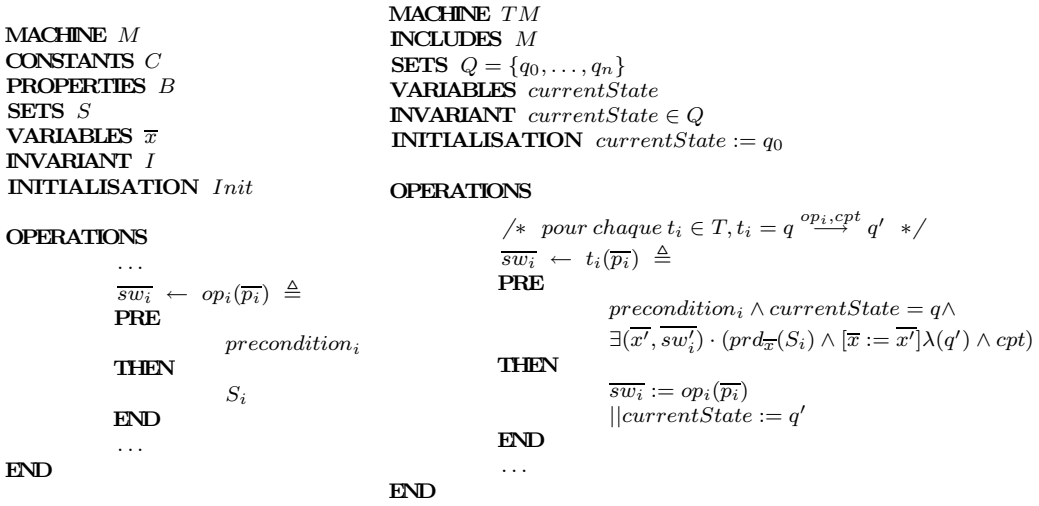


FIG. 5.3 – Synchronisation d'une machine B avec un objectif de test

La machine B synchronisée obtenue modélise donc les exécutions du système décrites par le schéma de test. Afin de produire des séquences de tests valuées (dont les paramètres ont été valués), il reste à guider le processus d'animation du modèle pour produire des séquences correspondant aux chemins décrits par le schéma de test. Les séquences de transitions de la forme $(t_0, t_n, \ldots, t_m)$ représentant les différents chemins de l'automate sont donc extraites et utilisées afin de forcer l'outil LTG à produire une séquence de test pour chacune d'elles à partir du modèle synchronisé.

On peut noter que toutes les séquences de transitions extraites de l'automate ne donnent pas nécessairement lieu à la génération de cas de tests. En effet, il se peut que certains scénarios décrits par le schéma de test ne soient pas animables sur le modèle du système, il se peut également que l'outil LTG ne dispose pas d'assez de temps de calcul ou de mémoire pour calculer une séquence de tests. Nous devons noter que l'implémentation de cette solution de génération de tests ne supporte pas les directives de pilotage du langage reposant sur l'utilisation du choix exclusif. Il serait néanmoins envisageable de compléter cette implémentation de manière à supporter cet aspect du langage. Une solution possible serait d'extraire les chemins de l'automate au fur et à mesure de la génération des tests.

5.4 Production des tests par animation du modèle

Nous présentons dans cette section une seconde implémentation de la méthode de génération de tests que nous proposons, celle-ci repose sur l'animation du modèle initial du système à tester. Cette animation est guidée par l'automate représentant le schéma de test. Cette solution de génération de tests a fait l'objet d'une implémentation dans l'outil *jSynoPsys* [DT09] qui exploite les fonctionnalités d'animation symbolique et de résolution de contraintes de

l'outil BZ-TT. La figure 5.4 représente une instanciation de notre démarche à cet outil.

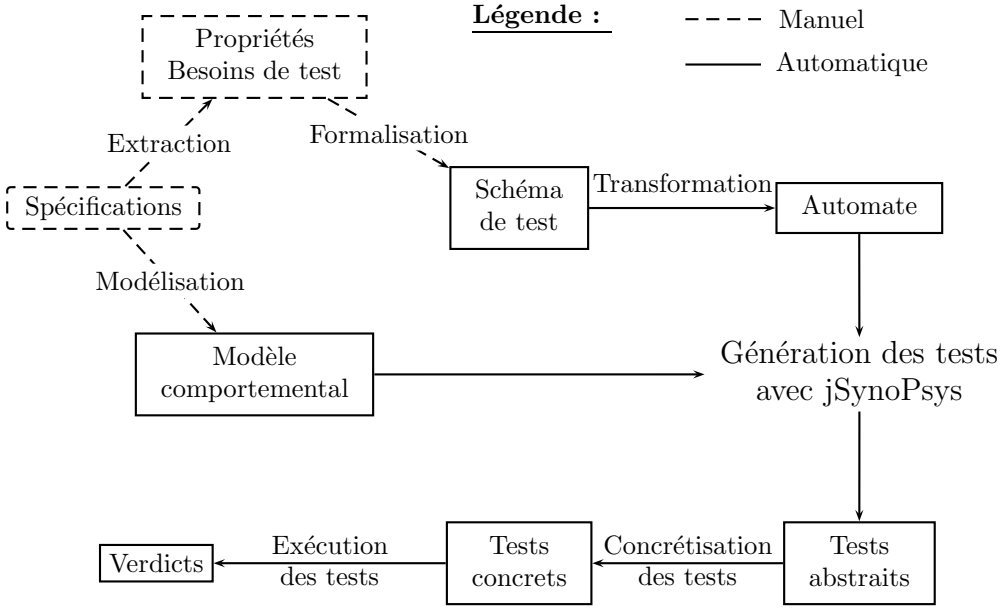


FIG. 5.4 – Processus de génération de tests avec jSynoPsys

Nous présentons l'algorithme de génération de tests par animation du modèle dans la figure 5.5. Cet algorithme est constitué de la fonction d'exploration *explorer* qui prend en paramètres l'état courant de l'automate et une exécution symbolique et retourne un ensemble de tests abstraits. Une exécution symbolique est une liste d'objets de type *OpSymb* constitués du nom de l'opération exécutée ainsi que des contraintes s'appliquant sur ses paramètres, ses valeurs de retour et sur les variables du système après exécution de l'opération. Un test abstrait est une séquence d'opérations issue de la valuation d'une séquence d'exécution symbolique où les paramètres des opérations et leurs valeurs de retour ont été valués.

L'algorithme que nous présentons met en jeu d'autres fonctions :

- La fonction *valuer* qui prend en paramètre une séquence d'exécution symbolique permet de fixer les paramètres des opérations de cette séquence et produit un cas de test abstrait.
- Les fonctions *sauvegarder* et *restaurer* permettent de sauvegarder l'ensemble des contraintes s'exprimant sur les variables du modèle à un instant donné de l'animation dans une pile, et de les restaurer par la suite. Cet ensemble de contraintes sur les variables du modèle est modifié par l'animation d'une opération réalisée grâce à la fonction *executerOp*.
- La fonction *executerOp* prend trois paramètres, le nom de l'opération que l'on souhaite exécuter (op), des contraintes sur le comportement de l'opération (cpt) et des contraintes sur l'état du système après exécution de l'opération (λ(q')). Cette fonction anime l'opération souhaitée (depuis l'environnement courant) de manière symbolique en tenant compte des contraintes données, et renvoie un objet de type OpSymb ou la valeur *null* si l'opération n'a pas pu être animée. L'objet de type OpSymb stocke les contraintes sur les paramètres et les valeurs de retour de l'opération, et sur les variables du modèle après animation de l'opération.

Le principe de l'algorithme présenté (fig. 5.5) repose sur un parcours en profondeur de l'automate depuis l'état initial. Les tests sont construits de manière symbolique au cours de

```
1  tests ← explorer(Etat q,ListOfOpSymb symbTest){
2      SetOfAbsTest testsRes;
3      OpSymb opSymb;
4      testsRes := ∅;

6      si (q ∈ Q_f) alors
7          /* si q est un état final, on value la séquence de test symbolique */
8          testsRes := {valuer(symbTest)} ;
9      sinon
10         pour chaque transition q -op,cpt-> q' ∈ T faire
11             si q ∉ Q_e ∨ testsRes = ∅ alors
12                 sauvegarder(environnement);
13                 opSymb := executerOp(op,cpt,λ(q'))
14                 si opSymb ≠ null alors
15                     /* la transition est activable, on poursuit l'exploration */
16                     testsRes := testsRes ∪ explorer(q',concat(symbTest, opSymb));
17                 sinon
18                     /* la transition n'a pas pu être activée */
19                     afficher("la transition q -op,cpt-> q' n'a pu être activée.")
20                 fsi

22                 restaurer(environnement);
23             fsi
24         fpour
25     fsi

27     return testsRes
28 }
```

FIG. 5.5 – Algorithme de génération de tests

l'exploration, puis valués lorsque l'état final est atteint. On peut noter deux particularités dans ce parcours en profondeur, l'une est l'élagage de branches de l'automate (condition ligne 11) lorsque l'état en cours correspond à un choix exclusif et qu'une de ses transitions sortantes a déjà donné lieu à la génération d'au moins un test. L'autre correspond au cas où une transition décrite dans l'automate n'a pu être activée (condition ligne 14), cette situation peut survenir pour deux raisons : soit cette transition ne correspond pas à l'exécution autorisée d'une opération du modèle, soit le temps de calcul ou l'espace mémoire disponible pour résoudres les contraintes définissant une exécution symbolique et ainsi valuer les paramètres des opérations.

Afin de lancer la procédure de génération de tests, il faut :

- Fournir la machine B qui modélise le système ;
- Initialiser le système au travers de son *environnement* qui correspond à la clause d'initialisation des variables de la machine B ;
- Donner l'automate $A = \langle Q, q_0, T, \lambda, q_f, Q_e \rangle$ issu du schéma de tests à partir duquel on souhaite générer les tests ;
- Faire un appel à la fonction *explorer* avec comme premier paramètre q_0 qui est l'état initial de l'automate et une liste vide pour la séquence de test symbolique.

L'algorithme, que nous présentons ici, est une version simplifiée de celui implémenté dans l'outil jSynoPsis. La principale simplification repose sur l'utilisation de la fonction d'animation d'une opération. Dans la version implémentée, la fonction *executerOp* ne permet d'animer qu'un seul des comportements[12] de l'opération fournie en paramètre, qui doit être précisé. Il

[12]Les comportements des opérations sont extraits à l'aide d'un critère de couverture structurel appliqué sur les opérations du modèle.

en découle que :

- Pour chaque transition, une boucle d'itération supplémentaire est nécessaire pour explorer tous les comportements (de l'opération décorant la transition) tant qu'il reste des chemins passant par cette transition qui n'ont pu être animés complètement.
- En plus de l'ensemble de tests valués, la fonction *explorer* doit retourner une information qui précise si l'ensemble des chemins situés après la transition en cours ont pu être explorés. Ceci afin de déterminer s'il est nécessaire d'itérer sur les comportements de l'opération de la transition en cours de traitement.
- Il faut marquer les transitions déjà complètement explorées.

La figure 5.6 présente les quatres vues de l'outil jSynoPsys. De haut en bas et de gauche à droite :

- la vue principale où apparaissent les différents schémas du projet en cours ;
- l'éditeur de schéma de test ;
- la vue du contenu des opérations du modèle B du projet courant ;
- la vue de présentation des cas de test générés à partir d'un schéma.

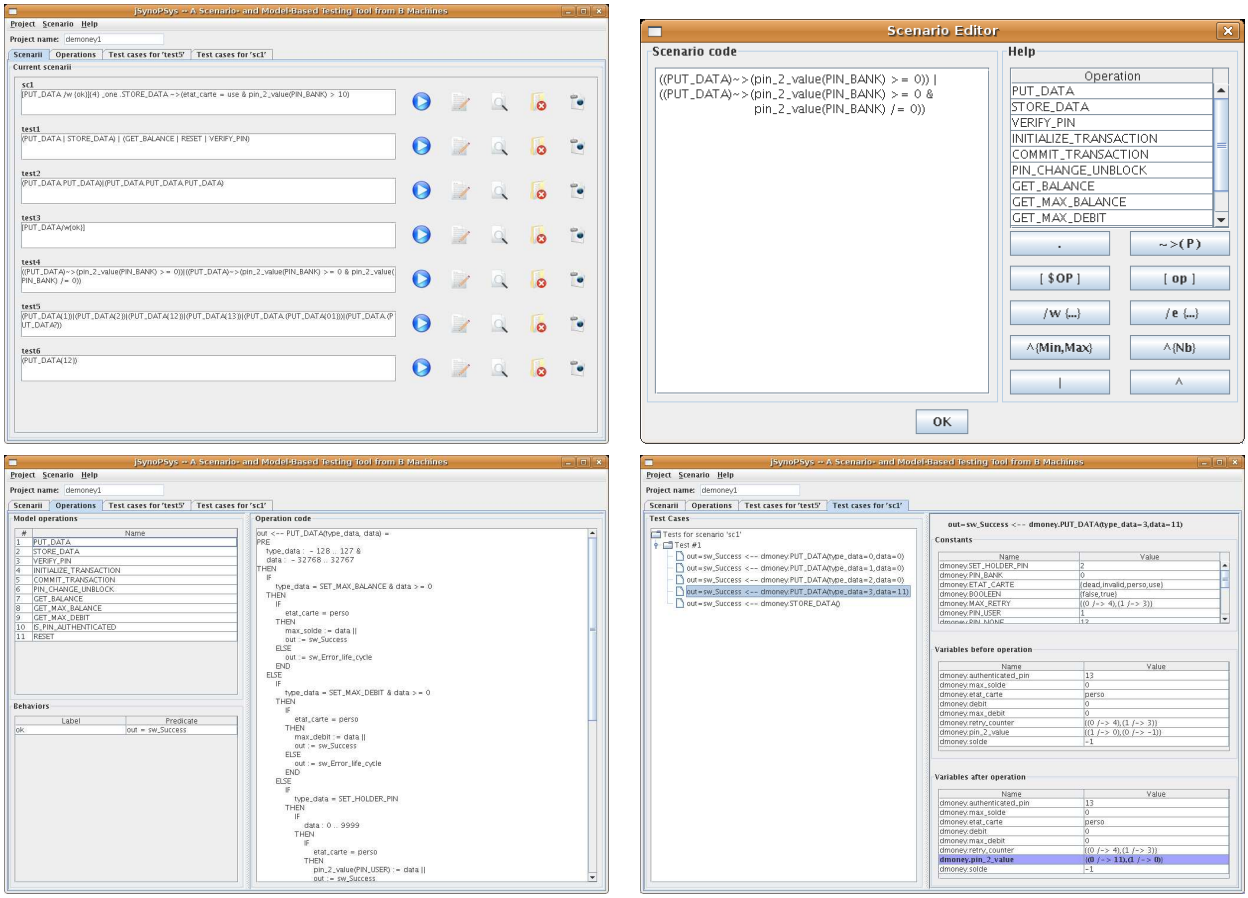


FIG. 5.6 – Interface de l'outil jSynoPsys

5.5 Résumé

Dans ce chapitre, nous avons présenté deux implémentations réalisées dans le but de générer des tests à partir d'objectifs de test formalisés à l'aide du langage de schémas de test présenté dans le chapitre 4.

La première de ces implémentations, réalisée pour exploiter les fonctionnalitées de l'outil LTG, repose sur la synchronisation d'un schéma de test et d'un modèle comportemental. La machine B obtenue par cette synchronisation modélise les exécutions du système restreintes à celles décrites dans le schéma de test. Les tests sont produits par animation du modèle synchronisé à partir des séquences de transitions extraites de l'automate associé au schéma de test.

Dans la seconde implémentation, réalisée pour exploiter les fonctionnalités de l'outil BZ-TT, les contraintes exprimées dans un schéma de test ne sont pas ajoutées au modèle, elles sont ajoutées au cours de l'animation. Cette solution de génération de tests présentée ici offre une amélioration de celle basée sur la synchronisation d'un modèle avec un schéma de test. Cette amélioration repose sur l'algorithme de parcours de l'automate pour l'animation du modèle. En effet, l'algorithme de parcours en profondeur de l'automate, présenté dans la figure 5.5, permet de ne parcourir qu'une seule fois (quand cela est possible) une séquence de transitions débutant à l'état initial de l'automate et commune à plusieurs chemins de l'automate. Alors que dans la solution basée sur le produit synchronisé d'un modèle et d'un schéma, chacun des chemins est animé indépendamment. Bien qu'aucune mesure comparative n'ait été effectuée, on peut supposer que cette différence d'implémentation induise une différence de performance entre les deux solutions à l'avantage de la seconde.

Nous soulignons que les expérimentations présentées dans le chapitre 7 ont toutes été réalisée avec l'implémentation de notre démarche qui repose sur l'utilisation de l'outil LTG et du produit synchronisé entre un schéma de test et un modèle.

En conclusion, les deux implémentations présentées, aux travers de leurs particularités offrent chacune certains avantages. L'intérêt du produit synchronisé entre un schéma de test et un modèle réside dans la production d'une machine B pouvant être "directement" utilisée par un outil, alors que la solution basée sur l'animation du modèle à l'aide du schéma de test requiert une implémentation spécifique pour l'outil utilisé. L'avantage de la seconde méthode est d'offrir la possibilité, au travers des fonctionnalités de l'outil BZ-TT, de définir de manière précise l'algorithme de production de tests.

Chapitre 6

Concrétisation, exécution et couverture des tests

Sommaire

Dans ce chapitre, nous présentons les moyens mis en œuvre visant à concrétiser les tests abstraits fonctionnels et produits à partir de schémas de tests. La concrétisation des tests abstraits, qui sont constitués de séquences d'opérations du modèle du système dont les paramètres sont valués et les valeurs de retour connues, consiste d'une part à faire le lien entre les niveaux d'abstraction du modèle et du système réel, et d'autre part à mettre en place une solution d'exécution des tests. Dans un second temps, nous proposons une solution d'évaluation de la couverture des tests produits par rapport au modèle du système, et une méthode destinée à évaluer de manière quantitative la complémentarité entre deux ensembles de tests en terme de couverture. L'évaluation de la complémentarité entre deux ensembles de tests a pour objectif de mesurer, pour un système donné, la complémentarité entre les tests fonctionnels générés par l'outil LTG et ceux produits à partir de schémas.

6.1 Concrétisation et exécution des tests

Le problème de la concrétisation des tests recoupe trois problématiques qui sont :
- combler la différence de niveau d'abstraction entre le modèle à partir duquel les tests ont été générés et le système sur lequel ils doivent être exécutés ;
- rendre possible l'exécution des tests sur le système sous test, soit en instrumentant le système à l'aide d'un banc de test, soit en rendant les tests exécutables sur le système (sous forme de script par exemple) ;
- définir l'établissement du verdict qui permet de déterminer si un test est un succès ou un échec en fonction des réponses produites par le système et de celles prévues par le modèle.

Notre contribution par rapport à cette problématique se borne au fait que notre approche de génération de tests permet d'utiliser les mêmes moyens de concrétisation et d'exécution pour les tests fonctionnels (générés à l'aide d'un critère de couverture structurelle des opérations du modèle) et ceux produits à partir de schémas.
Cette partie permet principalement :

- d'illustrer une phase importante de la démarche de validation par génération de tests à partir de modèle,
- et de présenter l'intégration de cette phase dans notre démarche (illustrée par les différentes expérimentations que nous présentons dans le chapitre suivant).

Les tests produits par animation du modèle du système à valider sont appelés tests abstraits. Leur niveau d'abstraction correspond au niveau d'abstraction du modèle du système. Or, afin de pouvoir exécuter ces tests, il est nécessaire de les transformer en tests concrets dont le niveau d'abstraction est compatible avec celui du système sous test. Dans les expérimentations où nous avons dû prendre en compte ce problème, la différence de niveau d'abstraction se situait principalement au niveau des types de données. Par exemple, dans le cadre de l'application Demoney les messages d'erreurs sont gérés sous forme de chaînes de caractères dans le modèle pour des raisons de lisibilité, alors qu'ils correspondent à des valeurs entières sur le système réel. Dans ce type de cas, la fonction de transformation des tests abstraits en tests concrets repose sur une fonction de mapping qui fait le lien entre les valeurs constantes du modèle et celle de l'implémentation.

Nous avons mentionné plus haut deux solutions possibles pour exécuter les tests sur le système. La première impose de créer un banc de tests constitué d'une interface de communication avec le système qui permet de réaliser les séquences d'appels de commandes décrites par les tests et d'observer les réponses du système. La seconde solution consiste à rendre les tests exécutables, cette solution quand elle peut être employée et certainement la plus simple car elle évite le développement d'un banc de test dédié au système. Dans le cas des expérimentations où nous avons dû gérer l'exécution des tests sur le système, celui-ci reposait soit sur des programmes en langage Java (Demoney et Posix), soit sur des programmes en langage C (Posix). Pour ces deux langages, il est assez aisé de rendre les tests exécutables. Dans le cas d'un système développé en langage Java, l'utilisation de la librairie Junit permet l'exécution de séquences de tests décrites sous la forme de fonctions contenues dans des classes Java. Dans le cas de Posix dont une des implémentations est réalisée sous la forme de librairies C, nous avons transformé nos tests en programmes C utilisant les fonctionnalités des librairies implémentant la norme Posix.

Une fois les tests rendus exécutables, le problème de l'établissement d'un verdict se pose. Le verdict sert à informer l'ingénieur de validation du succès ou de l'échec de l'exécution d'un test. Le succès ou l'échec de l'exécution d'un test peuvent être définis de plusieurs manières différentes. Par exemple, dans le cadre du test de robustesse d'un système, le verdict est uniquement basé sur les capacités du système à répondre aux sollicitations décrites par la séquence de test (par exemple, toujours fournir une réponse à une requête dans un temps imparti). Notre approche de validation se situe dans le domaine du test de conformité qui consiste à valider le fait que le système se comporte conformément à ses spécifications. Les spécifications sont formalisées au travers du modèle du système qui sert donc de référence pour la validation du système. Les tests produits à partir de ce modèle sont constitués d'une part de la séquence d'appels d'opérations destinée à solliciter le système et d'autre part de l'oracle qui représente les réponses à ces sollicitations telles qu'elles sont prévues par le modèle. C'est la comparaison entre l'oracle fourni par le modèle et les réponses produites par le système sous

test qui permet d'établir le verdict. Le contexte des diverses expérimentations que nous avons menées entre dans le cadre du test boîte noire où le système n'est observable qu'au travers des valeurs de retour des opérations qu'il exécute. Par conséquent l'établissement du verdict est effectué par comparaison des valeurs de retour des opérations appelées dans les séquences de tests et de celles prévues par le modèle. Dans notre cas, l'implémentation établit le verdict par évaluation d'assertions exprimant l'égalité des valeurs de retour des opérations observées sur le système sous tests et de celles prévues par le modèle. Ces assertions sont vérifiées à chaque exécution d'une opération.

Exemple

A titre d'illustration, nous donnons ci-dessous une séquence de test abstrait générée à partir du modèle de l'application Demoney et sa transformation en une fonction Java utilisée pour son exécution. La séquence de test abtrait est un document XML où les tests sont représentés sous la forme de suites d'opérations définies par leurs noms, les noms et les valeurs de leurs paramètres d'entrée et les noms et les valeurs de leurs sorties.
La séquence de test abstraite au format XML est la suivante :

```
<Project name="dmoney">
    <Campaign name="utilisation">
        <Test name="dmoney_00">
            <Operation name="VERIFY_PIN">
                <Inputs>
                    <Input name="pin">
                        <Atom value="1"/>
                    </Input>
                    <Input name="value">
                        <Atom value="1234"/>
                    </Input>
                </Inputs>
                <Outputs>
                    <Output name="out">
                        <Atom value="sw_Success"/>
                    </Output>
                </Outputs>
            </Operation>
            <Operation name="INITIALIZE_TRANSACTION">
                <Inputs>
                    <Input name="transaction">
                        <Atom value="0"/>
                    </Input>
                    <Input name="data">
                        <Atom value="5"/>
                    </Input>
                </Inputs>
                <Outputs>
                    <Output name="out">
                        <Atom value="sw_Success"/>
                    </Output>
                </Outputs>
            </Operation>
            <Operation name="COMMIT_TRANSACTION">
                <Inputs>
                </Inputs>
                <Outputs>
                    <Output name="out">
                        <Atom value="sw_Success"/>
                    </Output>
                </Outputs>
            </Operation>
            <Operation name="GET_BALANCE">
                <Inputs>
                </Inputs>
```

```
                <Outputs>
                    <Output name="out">
                        <Atom value="5"/>
                    </Output>
                </Outputs>
            </Operation>
        </Test>
    </Campaign>
</Project>
```

Le cas de test abstrait décrit dans le fichier XML est transformé, dans une classe de test Java, sous la forme d'une fonction. Cette fonction décrit la création d'une instance de l'application demoney suivie d'une séquence d'appels aux fonctions mises à disposition par cette instance (correspondant à la séquence d'appel d'opérations décrite par le test). Cette transformation permet de passer d'un cas de test abstrait qui n'est pas exécutable à une classe de test Java dont chaque fonction est un cas de test exécutable sur le système.
La séquence de test concrète en Java associée à la séquence abstraite précédente et la suivante :

```
public void dmoney_00() {
    /* création de l'instance de Demoney */
    Demoney b = new Demoney();

    /* Initialisation du système */
    b.PUT_DATA((byte)0,(short)199);
    b.PUT_DATA((byte)1,(short)100);
    b.PUT_DATA((byte)2,(short)1234);
    b.PUT_DATA((byte)3,(short)4321);
    b.STORE_DATA();

    /* Séquence de test */
    Assert.assertTrue(b.VERIFY_PIN((byte)1,(short)1234) == 9000);
    Assert.assertTrue(b.INITIALIZE_TRANSACTION((byte)1,(short)5) == 9000);
    Assert.assertTrue(b.COMMIT_TRANSACTION() == 9000);
    Assert.assertTrue(b.GET_BALANCE() == 5);
}
```

Cet exemple permet d'observer les différents aspects de la concrétisation que nous avons mentionnés plus haut. Ici, la différence d'abstraction entre le modèle et l'implémentation est comblée par les deux transformations suivantes :

- Les valeurs de retour des opérations sont traduites (la valeur de retour *sw_success* du modèle qui est une constante de type chaîne de caractères devient une valeur entière valant 9000 pour correspondre à l'implémentation).
- L'ajout d'une séquence d'initialisation permet de mettre le système réel dans un état où la phase de personnalisation de la carte est déjà effectuée. Cette séquence d'initialisation n'apparaît pas dans le cas de test abstrait car l'état initial du modèle a été modifié afin de correspondre à l'état voulu sans qu'il soit besoin d'utiliser les commandes de personnalisation, ce qui n'est évidemment pas possible sur le système réel.

Le verdict quant à lui est réalisé par l'utilisation de la fonction *assertTrue* mise à disposition par la librairie Junit. Cette fonction permet de définir les comparaisons entre les valeurs de retour des opérations prévues par le modèle et celles fournies par l'application lors de l'exécution des tests. A l'exécution, une différence entre ces deux valeurs est traduite par la production d'un rapport d'exécution des tests où le test en question est mentionné comme étant en échec. De plus le numéro de ligne correspondant à l'assertion violée est également mentionné afin de faciliter la recherche de l'erreur qui est la cause de la non-conformité.

6.2 Évaluation de la couverture des tests

6.2.1 Méthode d'évaluation de la couverture d'une suite de tests

Nous présentons ici, les moyens que nous avons mis en œuvre afin d'évaluer la qualité des suites de tests produites avec notre approche, mais surtout leur complémentarité vis-à-vis de suites de tests fonctionnels produites grâce à l'outil LTG. LTG met à disposition de l'utilisateur un indicateur de couverture du modèle par les tests générés. Celui-ci est basé sur le même critère de couverture que celui utilisé pour la sélection des tests, c'est à dire par un critère de couverture structurel des conditions et des décisions. En fait, cet indicateur fournit le pourcentage et la liste des cibles de tests qui n'ont pu donner lieu à la génération de tests, soit parce qu'elles ne sont pas atteignables, soit parce que les séquences de tests menant à ces cibles n'ont pu être trouvées dans le temps imparti ou pour la profondeur de recherche maximum fixée (exprimée en nombre d'appels d'opérations). Ces informations ont une utilité car elles permettent par exemple de détecter des comportements inatteignables dans le modèle ou de souligner la nécessité de modifier les paramètres des stratégies de recherche des séquences de tests (heuristique, profondeur de recherche, temps de génération, ...). Mais, cette évaluation reposant sur le même critère de couverture du modèle que celui utilisé pour la sélection des tests fonctionnels produits par LTG, elle n'est pas adaptée aux objectifs que nous visons qui sont de comparer la couverture de deux ensembles de tests et d'évaluer leur complémentarité le plus objectivement possible.

Parmi les critères permettant d'évaluer la qualité de suites de tests, nous pouvons citer la capacité des suites de tests à détecter des erreurs. Une manière simple de mettre en place une telle évaluation peut consister à confronter les suites de tests produites à des implémentations contenant des fautes. Nous avons mis en place une méthode pour l'expérimentation que nous avons menée sur la validation d'implémentations de la norme POSIX (c.f. 7.2). Cette méthode est très empirique et les résultats dépendent beaucoup des erreurs introduites. Dans notre cas, nous avons introduit manuellement les erreurs dans le système à tester ; on peut considérer que cette méthode induit un biais puisque le choix des erreurs introduites n'est pas forcément objectif. De plus, cette méthode permet dans une certaine mesure de montrer la complémentarité entre deux suites de tests (en considérant les erreurs detectées uniquement par l'une ou l'autre des deux suites de tests), mais elle ne permet pas d'avoir une évaluation fine de cette complémentarité.

Nos expérimentations s'effectuant d'une manière générale dans un contexte de test boîte noire, nous avons éliminé la possibilité de procéder à une analyse de couverture de code. Finalement, nous avons choisi de procéder à l'évaluation de la couverture de nos suites de tests en appliquant un critère de couverture des états, des transitions (et des paires de transitions dans le cas de l'expérimentation Demoney) sur le modèle du système à valider. Les modèles des systèmes que nous avons étudiés au cours de nos expérimentations étant trop complexes et étant constitués de nombres trop important d'états et de transitions, nous avons décidé d'utiliser des abstractions de ces modèles pour évaluer la couverture de nos suites de tests.

Pour calculer une abstraction de notre modèle, nous avons utilisé l'outil GénéSyst [Sto07]. GénéSyst permet, à partir d'un modèle B et du partionnement des domaines des variables sur lesquels on veut bâtir une abstraction, de calculer cette abstraction sous la forme d'un système de transitions étiqueté symbolique (STES). L'animation des séquences de tests sur l'abstraction obtenue permet de déterminer les taux de couverture d'états et de transitions des suites des tests. Les avantages de cette méthode d'évaluation sont :

– d'offrir un critère d'évaluation de la couverture des tests sur le modèle différent du critère de sélection de test ;
– de rendre possible une évaluation plus fine de la complémentarité entre deux suites de tests en prenant en compte les états ou les transitions qui ne sont couvertes que par l'une ou l'autre des suites de tests.

Bien qu'offrant un critère d'évaluation plus fin que la simple évaluation basée sur la couverture structurelle des opérations du modèle, l'évaluation de la couverture des tests en terme d'états et de transitions possède néanmoins un défaut. Celui-ci repose sur l'utilisation d'une abstraction, car le choix des variables et de la partition de leurs domaines de définition peut introduire un biais. En effet, ces choix, pour la construction de l'abstraction, vont avoir un impact sur la forme générale de cette dernière en détaillant plus certains aspects du système et en en masquant certains autres. Afin de rendre plus objectifs les résultats d'évaluation de la couverture d'une suite de tests sur une abstraction d'un modèle, il faut essayer dans la mesure du possible :

– de baser l'abstraction sur des variables dont l'état est "important" pour le système ;
– d'équilibrer le partionnement des domaines de ces variables de manière à ce qu'il soit représentatif d'une utilisation "normale" du système ;
– de fournir le choix des variables et des partionnements de leurs domaines en plus des résultats d'évaluation de couverture d'un ensemble de tests sur cette abstraction.

Au cours des expérimentations que nous avons menées, au travers des cas d'étude IAS (c.f. 7.1) et Demoney (c.f. 7.3), nous avons essayé de respecter ces contraintes. D'ailleurs, les comptes rendus de ces deux expérimentations contiennent une description des paramètres choisis afin de construire les abstractions, des deux modèles de ces deux systèmes, sur lesquels ont été effectuées les évaluations de couverture des tests produits.

Exemple

A titre d'exemple, nous pouvons présenter les variables et les partitionnements de leurs domaines choisis pour effectuer l'évaluation de la couverture des tests produits pour l'application Demoney. Les variables choisies pour bâtir l'abstraction sont :

– *etat_carte* qui indique l'état courant du cycle de vie de la carte ;
– *authenticated_pin* qui indique l'état d'authentification des différents objets de type PIN sur la carte ;
– *solde* qui indique le solde courant de la carte ;
– *debit* qui indique le montant de la transaction courante ;

Le partionnement du domaine de la variable *etat_carte* comporte les trois valeurs possibles de cette variable (*use*, *invalid* et *dead*) car chacune d'elle est représentative d'un état de la carte que l'on souhaite observer.

Par contre, le domaine de définition de la variable *authenticated_pin* est partionné en deux ($authenticated\_pin = PIN\_NONE \vee authenticated\_pin \neq PIN\_NONE$) alors que cette variable peut prendre trois valeurs. Ces trois valeurs sont : PIN_USER, PIN_BANK et PIN_NONE qui indiquent respectivement si l'utilisateur ou la banque sont authentifiés ou si personne ne l'est. Finalement, nous ne conservons que l'information indiquant si quelqu'un est authentifié ou non, car le fait de savoir qui est authentifié et redondant avec l'information sur l'état courant du cycle de vie de la carte (l'utilisateur ne pouvant s'authentifier qu'en état *use* et la banque en état *invalid*).

Le domaine de la variable *solde*, est partionné en deux ensembles de valeurs ($solde = 0 \vee solde > 0$) permettant de savoir si la carte est vide ou non. Il aurait été possible d'envisager une troisième valeur correspondant au cas où la carte a atteint son solde maximal.

Enfin, la variable *debit* a été partionnée en trois ensembles de valeurs ($debit = 0 \vee debit > 0 \vee debit < 0$) identifiant les situations d'absence de transaction en cours, de transaction de débit ou de crédit. Le choix de partitionnement du domaine de cette variable est une bonne illustration du biais que l'on peut introduire au travers du choix d'abstraction utilisée pour effectuer les mesures de couverture. En effet, pour cette variable, nous aurions pu choisir de partitionner le domaine en deux ($debit = 0 \vee debit \neq 0$) de manière à ne conserver que l'information indiquant si une transaction est en cours ou non. Mais ce choix aurait biaisé les résultats d'évaluation de couverture des tests produits lors de l'expérimentation. Car lors de cette expérimentation, les tests fonctionnels produits par LTG couvrent à la fois l'initialisation de transactions de crédit et de débit, alors que les tests générés à partir des schémas de tests ne couvrent que des situations de crédit. Un tel choix de partitionnement de domaine pour cette variable aurait gommé une différence importante entre deux ensembles de tests lors de l'évaluation de leur couverture sur l'abstraction.

6.2.2 Analyse de la complémentarité entre deux suites de tests

Un de nos objectifs étant de proposer une méthode de génération de tests complémentaire à la génération automatique de tests par couverture structurelle du modèle, nous proposons une solution destinée à évaluer la complémentarité entre deux suites de tests. Le critère de mesure que nous proposons est basé sur l'évaluation de la couverture (exprimée en nombre d'états, de transitions, de paires de transitions, ...) des suites de tests dont on souhaite évaluer la complémentarité.

Nous avons appliqué ce critère d'évaluation de complémentarité d'une suite de tests par rapport aux couvertures de transitions (expérimentation IAS et Demoney) et aux couvertures de paires de transitions (expérimentation Demoney). La définition 12 introduit la fonction *tc* qui, étant donnés deux ensembles de tests et une fonction d'évaluation de couverture, donne le nombre d'éléments couverts à la fois par l'une et l'autre des suites de tests. Étant données deux suites de tests, la fonction *comp*, présentée dans la définition 13, permet de calculer la proportion d'éléments couverts par l'une des suites de tests qui ne sont pas couverts par l'autre.

Définition 12 (Éléments couverts par deux suites de tests)**.** Soit $cov(e)$ le nombre d'éléments (états, transitions, paires de transitions, ...) couverts par l'ensemble de tests e. $tc(e_1, e_2)$ est la fonction qui définit le nombre d'éléments identiques couverts par les deux ensembles de tests e_1 et e_2 :

$$tc(e_1, e_2) = cov(e_1) + cov(e_2) - cov(e_1 \cup e_2)$$

Définition 13 (Complémentarité entre deux suites de tests)**.** Soient deux ensembles de tests e_1 et e_2, nous définissons la complémentarité de e_1 par rapport à e_2 à l'aide de la fonction $comp(e_1, e_2)$ qui définit la proportion d'éléments (états, transitions,...) couverts par l'ensemble de tests e_1 qui ne le sont pas par l'ensemble de tests e_2. La fonction *comp* est définie par :

$$comp(e_1, e_2) = \frac{cov(e_1) - tc(e_1, e_2)}{cov(e_1)} = \frac{cov(e_1 \cup e_2) - cov(e_2)}{cov(e_1)}$$

6.3 Résumé

Dans ce chapitre, nous avons présenté les solutions mises en œuvre pour la concrétisation, l'exécution et l'évaluation de la couverture des tests abstraits produits à partir d'un modèle B. Ces tests abstraits englobent à la fois des tests fonctionnels générés par l'outil LTG à l'aide d'un critère de couverture structurelle des opérations du modèle et des tests produits à partir du modèle et de schémas de test. Ces solutions sont celles utilisées dans les différents cas d'étude présentés dans le chapitre 7.

Les moyens de concrétisation des tests abstraits et d'exécution des tests concrets décrits dans ce chapitre reposent sur la transformation des tests abtraits en classes de tests Java ou en programmes C. Dans le cas des expérimentations concernées (Demoney et Posix), cette transformation permet d'obtenir des tests directement exécutables sur le système sous test. L'établissement du verdict d'exécution de ces tests est assuré par l'évaluation, lors de l'exécution des tests, d'assertions testant l'égalité des valeurs de retour des opérations exécutées sur le système avec celles prévues par le modèle (oracle du test).

Nous avons présenté deux solutions d'évaluation de la couverture des tests obtenus. L'une est basée sur la détection d'erreurs a été expérimentée au cours du cas d'étude concernant la validation d'un système de fichiers POSIX (c.f. 7.2). L'autre est basée sur le calcul de la couverture des tests sur une abstraction du modèle du système (produite grâce à l'outil GénéSyst) en terme d'états et de transitions. Cette solution peut induire un biais dans l'évaluation de la couverture des tests et nous donnons quelques indications qui permettent de réduire ce défaut.

Enfin, à partir de l'évaluation de la couverture des tests sur une abstraction du modèle du système, nous définissons une solution d'évaluation de la complémentarité entre deux ensembles de tests. Cette solution est définie par une fonction qui, étant donné deux ensembles de tests, permet de déterminer la proportion d'éléments (états, transitions, paires de transitions, ...) couverts par l'un des ensembles de tests exclusivement (éléments qui ne sont pas également couverts par l'autre ensemble de test).

Troisième partie

Applications

Chapitre 7

Etudes de cas

Sommaire

Nous présentons dans ce chapitre différentes expérimentations au cours desquelles nous avons appliqué notre démarche de génération de tests à partir d'objectifs de tests. Les différents cas d'études que nous avons étudiés sont deux applications de type carte à puce (IAS et Demoney) et une application de gestion de système de fichiers (POSIX). Le but de ces différentes expérimentations était de montrer la faisabilité de notre approche et sa complémentarité avec une approche de génération de tests à partir de modèle, reposant en l'occurence sur la sélection de tests par couverture structurelle du modèle du système à valider.

7.1 IAS

7.1.1 Présentation générale

La spécification IAS [GIX04] (*Identification, Authentication, and electronic Signature*) est une spécification technique de services pour les cartes à puces intégrant la Plateforme commune pour l'e-Administration. Elle intégre les fonctions requises afin de réaliser les opérations d'authentification, d'identification et de signature électronique. Ce cas d'étude a été abordé dans le cadre du projet POSE[13] dont l'objectif visait la mise en place d'outils, de techniques et de méthodes pour la validation de conformité d'un système vis-à-vis des politiques de sécurité qui lui sont assignées. Les travaux réalisés sur ce cas d'étude ont fait l'objet de plusieurs publications [JMT08a, JMT08b, DPT08].

Notre travail, dans ce projet, s'est centré autour de la validation du contrôle d'accès de la plateforme IAS, et plus particulièrement autour de l'authentification de l'utilisateur par code PIN. Notre objectif principal visait à mettre au point une "méthodologie" permettant de générer des tests complémentaires aux tests fonctionnels générés à partir de critères statiques de couverture. Dans la section 7.1.2, nous détaillons certaines notations propres au cas d'étude IAS et utilisées dans le reste de cette partie. Dans la section 7.1.3, nous présentons le fonctionnement du contrôle d'accès de la plateforme IAS, sa formalisation par une machine B est détaillée dans la section 7.1.4. Dans la section 7.1.5, nous donnons quelques éléments concernant les différents aspects de la génération des tests à partir du modèle B de la plateforme IAS et des différentes techniques de génération de tests que nous avons employées (LTG et utilisation de schémas de test). Dans la section 7.1.6, nous présentons la méthode de mesure de couverture des différentes suites de tests que nous avons utilisé et les résultats obtenus par cette méthode. La section 7.1.7 présente une interprétation des différents résultats obtenus au cours de cette expérimentation, d'une part pour expliquer ceux-ci et d'autre part pour apporter quelques réponses quant à la complémentarité des tests générés à l'aide des schémas par rapport aux tests fonctionnels générés avec LTG.

7.1.2 Notations

Nous nous proposons de présenter ici quelques abréviations que nous utilisons dans la suite de cette partie, consacrée au cas d'étude IAS. Ces abréviations sont celles utilisées dans la spécication IAS [GIX04].

DF — Dedicated File : Les DF sont les objets de type "répertoire" de la carte, ils peuvent contenir d'autres DF, des EF ou des SDO.

EF — Elementary File : Les EF sont les fichiers "normaux", ils permettent de stocker des données. Dans notre étude, ce type d'objets n'a pas été pris en compte, leur contrôle d'accès étant similaire à celui des DF.

ADF — Application Dedicated File : Les ADF sont des DF particuliers, ils permettent de cloisonner les applications entres elles (en permettant de définir des politiques de contrôle d'accès locales). Comme les EF, ce type particulier d'objets n'a pas été modélisé.

SDO — Security Data Object : Les SDO sont des fichiers de données particuliers, puisqu'ils contiennent des données relatives à l'authentification. Parmi les objets de type

[13] `http ://rntl-pose.info`

SDO, nous trouvons : les objets de type PIN (utilisés pour l'authentification de l'utilisateur par code secret) ; les clés de cryptage et les données biométriques (utilisées pour l'authentification de l'utilisateur). Notre étude concernant exclusivement le contrôle d'accès basé sur l'authentification de l'utilisateur par code PIN, nous n'avons modélisé que ce dernier type de SDO.

AMB — Access Mode Byte : L'AMB permet, à l'intérieur d'une règle d'accès, de spécifier pour quel mode d'accès particulier celle-ci devra s'appliquer. Par exemple, la lecture de données (resp. l'écriture) ou l'activation d'un DF (resp. sa désactivation, terminaison ou suppression).

SCB — Security Condition Byte : Dans une règle d'accès, l'octet SCB contient deux informations. La première indique quelle(s) vérification(s) doi(ven)t être réalisée(s) pour satisfaire la règle d'accès. Par exemple la règle est toujours satisfaite (resp. jamais satisfaite) ou l'utilisateur doit s'authentifier sur un code PIN. La deuxième information est une référence sur un SE.

SE — Security Environment : Un SE contient des références sur des SDO, son utilité est donc de préciser, dans une règle d'accès, quels sont les objets de sécurité qui doivent être utilisés pour valider cette règle.

7.1.3 Spécification du contrôle d'accès pour IAS

Dans cette section, nous présentons les aspects d'IAS qui ont été formalisés dans le modèle et pour lesquels des suites de tests ont été générées. Les trois éléments principaux que nous présentons sont :

- la structure des différents fichiers et répertoires que nous avons considérés dans notre représentation,
- la gestion du contrôle d'accès, aux fichiers et répertoires, basée sur l'authentification de l'utilisateur par code PIN,
- la gestion du cycle de vie des fichiers et répertoires.

Nous considérons deux types d'objets différents. Le premier type d'objets est constitué des objets de type DF qui sont les repertoires présents sur la carte, ils constituent une arborescence. Deux DF possèdent un statut particulier, il s'agit :

- de la racine de l'arborescence, aussi nommée MF — Master File,
- du DF courant (le répertoire sélectionné à un instant donné) dont l'état influe sur l'exécution des opérations et sur l'établissement des conditions d'accès.

Le second type d'objets manipulé est constitué des objets de sécurité (SDO) de type PIN. Les SDO de type PIN sont utilisés par le mécanisme d'authentification de l'utilisateur par code PIN. Les objets de type PIN sont réferencés dans le système par un mécanisme de double référence :

- les références longues qui identifient de manière unique chaque objet PIN,
- les références courtes qui identifient également les objets PIN, mais dont l'unicité est seulement garantie à l'intérieur d'un même DF.

La figure 7.1 présente un exemple d'arborescence possible. Sur la partie gauche, on voit apparaître :

- un DF root (c'est à dire le MF),
- deux fichiers de type DF (file_01 et file_02),
- un SDO de type PIN (pin_02).

La partie extérieure aux pointillés laisse apparaître un type de DF particulier : les ADF (Application Dedicated File) qui sont utilisés pour isoler les applications du reste de l'arborescence. Nous ne traitons pas ce cas particulier de DF.

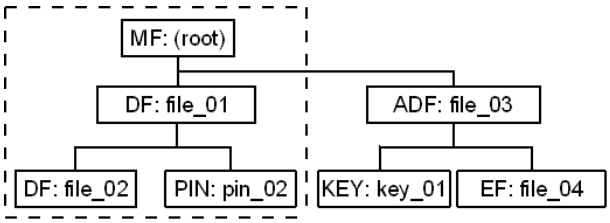


FIG. 7.1 – IAS : exemple de structure de données arborescente

La gestion du contrôle d'accès est gérée par un ensemble de règles qui permet de définir la protection des fichiers (DF ou SDO de type PIN dans notre cas). Chaque fichier est associé à un ensemble de règles qui contiennent :

- un octet AMB qui détermine pour quel mode d'accès s'appliquera cette règle. Par exemple, pour une règle s'appliquant à un DF, si l'octet AMB vaut 0100 0000, alors la règle sera vérifiée avant toute tentative de suppression du DF.
- un octet SCB composé de 4 bits définissant les vérifications à réaliser (les trois règles que nous considérons dans notre expérimentation sont : authentification utilisateur, aucune vérification et opération interdite) et de 4 bits faisant référence à un SE qui définit comment doivent être réalisées les vérifications.

Enfin, à chaque fichier est associé un état de cycle de vie qui determine quelles opérations peuvent être réalisées sur ce dernier. La figure 7.2 illustre comment peut évoluer le cycle de vie d'un fichier au cours du temps. Les quatres états de cycle de vie possibles pour un fichier sont :

Activated : le fichier est dans un *état opérationnel activé*, toutes les opérations peuvent être activées.

Deactivated : le fichier est dans un *état opérationnel désactivé*, seules quelques opérations comme la suppression, la terminaison, l'activation, la sélection, ... peuvent être activées.

Terminated : le fichier est dans un *état terminé (fin de vie)*, aucune opération ne peut être activée, sauf la suppression et la sélection.

Deleted : le fichier a été physiquement supprimé, plus aucune opération ne peut être activée.

7.1.4 Modélisation du contrôle d'accès pour IAS

Afin de modéliser le contrôle d'accès de la plateforme IAS [PTJ07], nous avons utilisé l'outil MECA [HM07, Had07]. Cet outil permet de générer un "noyau de sécurité" à partir d'un *modèle de règles* et d'un *modèle dynamique* décrits en langage B. Dans le cas d'IAS, nous nous sommes basés sur un format de contrôle d'accès des utilisateurs en fonction de conditions portant sur des attributs de sécurité (UAC —User Access Control [DHM07]). Le *modèle de règles* formalise, de manière déclarative, la politique de contrôle d'accès du système. Ce modèle décrit les paramètres de contrôle d'accès (sujets, objets, opérations et attributs de sécurité) et les règles de contrôle d'accès. Le modèle *dynamique* décrit les modifications possibles des attributs de sécurité et l'état initial.

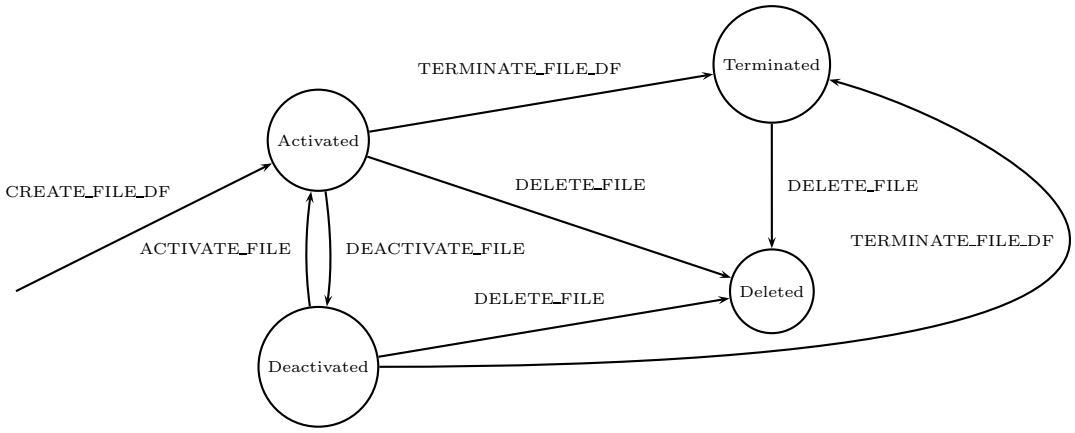


FIG. 7.2 – IAS : États et transitions du cycle de vie des fichiers

L'outil MECA assure le "tissage" de ces deux modèles afin de produire le noyau de sécurité. Le noyau de sécurité est un modèle B où sont présents à la fois :

- la dynamique de modification des attributs de sécurité, au travers des opérations,
- la politique de contrôle d'accès, dont chaque règle est exprimée sous forme d'une substitution gardée conditionnant la modification des attributs de sécurité à l'intérieur des opérations.

La figure 7.3 présente un extrait du modèle de règles concernant l'opération de sélection de répertoire : *SELECT_FILE_DF_CHILD*. Cette opération n'est pas protégée par une règle d'accès basée sur une authentification de l'utilisateur, par contre son exécution n'est possible que si le statut de cycle de vie du DF courant est "activé". La figure 7.4 est un extrait du modèle dynamique (correspondant à l'opération SELECT_FILE_DF_CHILD). On voit que cette opération ne prend pas en compte la condition d'accès, mais se limite à modéliser les modifications des attributs de sécurité suite à un comportement "licite" (où les conditions d'accès sont vérifiées) de l'opération. Enfin, la figure 7.5 présente un extrait du noyau de sécurité tissé par l'outil MECA. Dans cette figure, apparait l'opération SELECT_FILE_DF_CHILD où sont présentes les informations contenues dans le modèle de règles (conditions de la première substitution gardée de l'opération où l'on voit apparaître la condition d'accès) et celles contenues dans le modèle dynamique (la modification des attributs de sécurité, figurant dans le corps de la substitution gardée par la condition d'accès). On voit également, que l'opération renvoie deux valeurs différentes : *"rf"* qui correspond au résultat fonctionnel de l'opération dans le cas où celle-ci a été appelée avec succès vis-à-vis de ses conditons d'accès et *"rs"* qui correspond au résultat de la vérification de la condition d'accès.

Modèle de données

Nous présentons ici les différentes variables du modèle de la plateforme IAS que nous avons utilisées afin de générer les différentes suites de tests produites au cours de notre ex-

MACHINE ias_rule

SETS
DF_ID ;
LIFE_CYCLE_STATE = {activated, deactivated, terminated} ;
OPERATION = {SELECT_FILE_DF_CHILD, CREATE_FILE_DF, ...}

CONSTANTS
/* security attributes */
DF_list, /* the list of the directory files of the application */
DF_2_DFParent, /* the hierarchy of directory files */
current_DF, /* the directory file where we are positioned */
DF_2_life_cycle_state, /* the life cycle of files */

/* Permission relationship */
permission_df

PROPERTIES
DF_list $\subseteq$ DF_ID $\wedge$
DF_2_DFParent $\in$ DF_list $\rightarrow$ DF_list $\wedge$
current_DF $\in$ DF_list $\wedge$
DF_2_life_cycle_state $\in$ DF_list $\rightarrow$ LIFE_CYCLE_STATE $\wedge$
permission $\in$ OPERATION $\leftrightarrow$ DF_list $\wedge$

/* access rule relative to the command SELECT_FILE_DF_CHILD */
$\forall$(df_id).(
/* df_id denotes a child of the current directory file */
df_id $\in$ DF_2_DFParent^{-1}[{current_DF}]
/* the current directory file is activated. */
$\wedge$ ran({current_DF}$\lhd$DF_2_life_cycle_state) = {activated}
$\Longrightarrow$ (SELECT_FILE_DF_CHILD $\mapsto$ df_id) $\in$ permission)

/* access rule relative to the command CREATE_FILE_DF */
$\wedge$ ...

END

FIG. 7.3 – IAS : extrait du modèle de règles

```
rf ← SELECT_FILE_DF_CHILD(df_id) =
PRE
   df_id ∈ DF_ID ∧
   df_id ∈ DF_2_DFParent⁻¹[{current_DF}]
THEN
   rf := ok
   || current_DF := df_id /* update of the current df*/
   || LET pin_loosing_auth BE
        pin_loosing_auth = dom(pin_authenticated_2_df▷{current_DF}) ∩ ran(dom(PIN_2_dfParent▷{df_id})◁PIN_2_short_ref)
      IN
        pin_authenticated_2_df := pin_authenticated_2_df ∪
          (dom(pin_authenticated_2_df▷{current_DF}) - pin_loosing_auth)×{df_id}
      END
END ;
```

FIG. 7.4 – IAS : extrait du modèle dynamique

```
rf, rs ← SELECT_FILE_DF_CHILD(df_id) =
PRE
      df_id ∈ DF_ID
THEN
   IF
      df_id ∈ DF_2_DFParent⁻¹[{current_DF}] ∧
      ran({current_DF}◁DF_2_life_cycle_state) = {activated}
   THEN
      rs := ok
      || rf := ok
      || current_DF := df_id /* update of the current df*/
      || LET pin_loosing_auth BE
           pin_loosing_auth = dom(pin_authenticated_2_df▷{current_DF}) ∩ran(dom(PIN_2_dfParent▷{df_id})◁PIN_2_short_ref)
         IN
           pin_authenticated_2_df := pin_authenticated_2_df ∪
             (dom(pin_authenticated_2_df▷{current_DF}) - pin_loosing_auth)×{df_id}
         END
   ELSE
      rs := ko
   END
END ;
```

FIG. 7.5 – IAS : extrait du noyau de sécurité

périmentation. Ce modèle est le noyau de sécurité produit par MECA à partir du modèle de règles et du modèle dynamique. Soient les ensembles suivants :

PIN_list : un ensemble d'identifiants (uniques) d'objets de type PIN,

DF_list : un ensemble d'identifiants d'objets de type DF (répertoires),

PIN_short_ref_list : un ensemble d'identifiants courts (non uniques) d'objets de type PIN,

rules_list : un ensemble d'identifiants de règles de contrôle d'accès,

ACCESS_MODE_BYTE : un ensemble de modes d'accès,

SE_list : un ensemble d'environnements de sécurité,

SECURITY_CONDITION_BYTE : un ensemble de conditions de sécurité,

LIFE_CYCLE_STATE : un ensemble d'états de cycle de vie.

Nous définissons les variables suivantes :

current_DF ($\in$ DF_list) est la variable utilisée pour déterminer le répertoire courant.

pin_2_dfParent ($\in$ PIN_list $\rightarrow$ DF_list) est la fonction qui associe un objet PIN au DF sous lequel il est enregistré.

pin_2_short_ref ($\in$ PIN_list $\rightarrow$ PIN_short_ref_list) est la fonction qui associe l'identifiant unique d'un objet PIN avec sa référence courte (identifiant non-unique).

pin_authenticated_2_df ($\in$ PIN_short_ref_list $\leftrightarrow$ DF_list) est une relation mettant en correspondance les objets de type PIN avec les DF pour lesquels ils peuvent être considérés comme authentifiés (car leur état d'authentification ne peut être utilisé pour établir la vérification d'une condition d'accès qu'à l'intérieur d'un DF pour lequel ils sont authentifiés).

rule_2_obj ($\in$ rules_list $\rightarrow$ (PIN_list $\cup$ DF_list)) est la fonction qui associe une règle de contrôle d'accès à l'objet qu'elle protège.

rule_2_AMB ($\in$ rules_list $\rightarrow$ ACCESS_MODE_BYTE]) est la fonction qui associe une règle de contrôle d'accès au mode d'accès qu'elle protège (lecture, écriture, ...).

rule_2_SE ($\in$ rules_list $\rightarrow$ SE_list) est une fonction qui associe une règle de contrôle d'accès à un environnement de sécurité. Nous considérons que l'environnement de sécurité est une référence vers l'objet de type PIN utilisé pour établir les règles basées sur l'authentification de l'utilisateur via un code PIN.

rule_2_SCB ($\in$ rules_list $\rightarrow$ SECURITY_CONDITION_BYTE) est la fonction qui associe une règle de contrôle d'accès à une description de la vérification qui doit être effectuée pour vérifier cette règle (authentification sur PIN, accès toujours autorisé ou accès toujours interdit).

ID_2_life_cycle_state ($\in$ (PIN_list $\cup$ DF_list) $\rightarrow$ LIFE_CYCLE_STATE) est la fonction qui associe les objets de type DF et PIN à l'état du cycle de vie dans lequel ils sont.

Les opérations

Dans cette partie, nous présentons succintement les différentes opérations formalisées dans notre modèle. Chacune de ces opérations correspond à une APDU présente sur la plateforme IAS.

RESET permet de réinitialiser la carte en phase d'utilisation ; toutes les authentifications sont perdues et le DF courant est positionné à la racine de la carte.

VERIFY permet soit à l'utilisateur de s'authentifier via un objet de type PIN, soit de se renseigner sur l'état d'authentification d'un objet de type PIN.

CHANGE_REFERENCE_DATA permet de modifier le code secret associé à un objet de type PIN (et d'autres information telles que la taille du code PIN, mais ces autres paramètres sont abstraits dans le modèle que nous présentons). L'application de cette commande, si elle est effectuée avec succès, annule l'authentification sur l'objet visé.

RESET_RETRY_COUNTER permet de réinitialiser le compteur d'essais d'un objet PIN. Comme la commande CHANGE_REFERENCE_DATA, cette commande, si elle est effectuée avec succès, annule l'authentification sur l'objet PIN visé.

SELECT_FILE_DF_PARENT permet de sélectionner le DF père du DF courant dans l'arborescence des DF, cette commande a pour effet de faire perdre l'authentification sur tous les objets PIN authentifiés dans le DF courant.

SELECT_FILE_DF_CHILD permet de selectionner l'un des DF fils du DF courant ; tous les objets PIN authentifiés dans le DF courant le seront dans le DF sélectionné, à moins qu'une instance locale (objet PIN possèdant la même référence courte) y soit présente.

CREATE_FILE_DF permet de créer un nouveau DF dans le DF courant.

PUT_DATA_OBJ_PIN_CREATE permet de créer un nouvel objet de type PIN.

DELETE_FILE permet de supprimer un fichier ou un dossier.

ACTIVATE_FILE permet positionner un DF dans l'état de cycle de vie "actif".

DEACTIVATE_FILE permet de positionner un DF dans l'état de cycle de vie "désactivé".

TERMINATE_FILE_DF permet de positionner un DF dans l'état de cycle de vie "terminé".

7.1.5 Génération de tests

Dans cette section, nous présentons la phase de génération des tests au cours de laquelle nous avons utilisé le modèle B de la plateforme IAS (présenté dans la section 7.1.4) pour générer des tests. Les critères de sélection utilisés sont :

- des schémas de test utilisés comme critères dynamiques de sélection,
- la couverture des comportements des opérations utilisant le critère de couverture des conditions et décisions (C/DC) et la couverture des paramètres aux limites utilisés par LTG.

Nous rappellons que les schémas de test nous permettent de définir des critères dynamiques de sélection pour la génération des tests. Ici, notre objectif était de couvrir (pas forcément de manière exhaustive) différents aspects du contrôle d'accès basé sur l'authentification de l'utilisateur grâce à des codes PIN. Nous nous sommes limités à la rédaction de trois schémas de test différents, que nous nommerons *TP1*, *TP2* et *TP3*.

Chacun de ces trois schémas de test est né du besoin de tester un aspect particulier du contrôle d'accès. Le premier schéma de test (*TP1*) a pour but de tester les différentes opérations d'accès à une ressource lorsque les conditions d'accès à cette ressource ne sont pas satisfaites. Ayant constaté que la génération de tests fonctionnels avec LTG privilègie les chemins les plus courts pour couvrir ces situations ; nous avons décidé de contraindre la couverture de ces comportements par le cas où les conditions d'accès sont préalablement vérifiées, mais ne le sont plus au moment de l'accès à la ressource à la suite d'une perte

d'authentification. Afin d'être le plus exhaustif possible, nous avons décidé de prendre en compte (dans le schéma de test) le fait que nous voulions explorer les différents moyens de perdre l'authentification (requise pour vérifier les conditions d'accès). La figure 7.6 présente la formalisation du schéma de test TP1 dans le langage présenté au chapître 4. Pour des raisons de lisibilité, les états e_n sont décrits de manière informelle dans le tableau situé en bas de la figure et leur expression formelle est donnée à titre indicatif. La figure 7.7 donne une représentation de ce même schéma de test sous la forme d'un automate.

VERIFY.CREATE_FILE_DF$\rightsquigarrow(e_1)$
.PUT_DATA_OBJ_PIN_CREATE.VERIFY$\rightsquigarrow(e_2)$
.CREATE_FILE_DF$\rightsquigarrow(e_3)$
.SELECT_FILE_DF_PARENT$\rightsquigarrow(e_4)$
.(RESET_RETRY_COUNTER | (RESET.SELECT_FILE_DF_CHILD)
| VERIFY | CHANGE_REFERENCE_DATA
| (SELECT_FILE_DF_PARENT.SELECT_FILE_DF_CHILD)$\rightsquigarrow(e_5)$
.SELECT_FILE_DF_CHILD$\rightsquigarrow(e_6)$
.(CREATE_FILE_DF | DELETE_FILE | ACTIVATE_FILE
| DEACTIVATE_FILE | TERMINATE_FILE_DF | PUT_DATA_OBJ_PIN_CREATE);

e_1	le PIN qui protège l'écriture dans le DF racine a été authentifié, un DF a été crée à la racine, et le DF nouvellement crée est devenu le DF courant
	rule_2_SE[rule_2_obj^{-1}[{file_fid_01}]] = {ref_SDO_00001b} $\wedge$ file_fid_01 $\in$ pin_authenticated_2_df[SE_2_SDO[{ref_SDO_00001b}]] $\wedge$ current_DF = file_fid_01
e_2	un nouveau PIN a été crée dans le DF courant, et il a été authentifié
	file_fid_01 $\in$ PIN_2_dfParent[PIN_2_short_ref^{-1}[SE_2_SDO[{ref_SDO_00010b}]]] $\wedge$ file_fid_01 $\in$ pin_authenticated_2_df[SE_2_SDO[{ref_SDO_0001Ob}]] $\wedge$ rule_2_SCB[PIN_2_short_ref^{-1}[SE_2_SDO[{ref_SDO_00010b}]]] $\cap$ PIN_2_dfParent^{-1}[{file_fid_01}] = {SCB_cond_0000b}
e_3	un nouveau DF a été crée dans le répertoire courant et tous les modes d'accès à ce nouveau DF sont protégés par le PIN crée précédemment
	rule_2_SE[rule_2_obj^{-1}[{file_fid_02}]] = {ref_SDO_00010b} $\wedge$ current_DF = file_fid_02
e_4	le DF courant est remonté d'un niveau dans l'arborescence
	current_df = file_fid_01
e_5	l'authentification sur le PIN qui avait été crée a été perdue par l'application de l'une des opérations ou des suites d'opérations définies
	current_df = file_fid_01 $\wedge$ file_fid_01 $\notin$ pin_authenticated_2_df[SE_2_SDO[{ref_SDO_00010b}]]
e_6	le DF courant redescend d'un niveau dans l'arborence
	current_DF = file_fid_02

FIG. 7.6 – IAS : schéma de test TP1

Le besoin de test à l'origine du deuxième schéma de test est quant à lui issu d'un livrable [VA06] du projet POSE (L5.2 "Cible de sécurité pour la plateforme IAS"), et plus précisément de la partie intitulée "analyse de vulnérabilité". Ce besoin de test résulte du fait que les objets de sécurité (dans notre cas particulier les objets de type PIN) sont à la fois identifiés par des identifiants uniques et par des références courtes (c.f. 7.1.3). Sans entrer plus loin dans les détails, cette situation d'homonymie pourrait potentiellement entraîner des non conformités dans l'interprétation des règles de contrôle d'accès associées à une ressource. Le schéma de test TP2 a donc pour objectif de mener à la génération de tests concernant l'activation d'une opération d'accès, lorsque la règle d'accès protégeant celle-ci n'est pas vérifiée, mais dont l'interprétation pourrait être ambigüe et mener à une non-conformité du comportement du système vis à vis de sa spécification. La figure 7.8 présente la configuration de la carte mise en jeu par TP2. On voit que deux SDO de type PIN possédant la même réference courte situés à deux niveaux d'arborescence différents sont créés. A partir de cette configuration, le schéma définit que l'on souhaite tester les différents modes d'accès (protégés

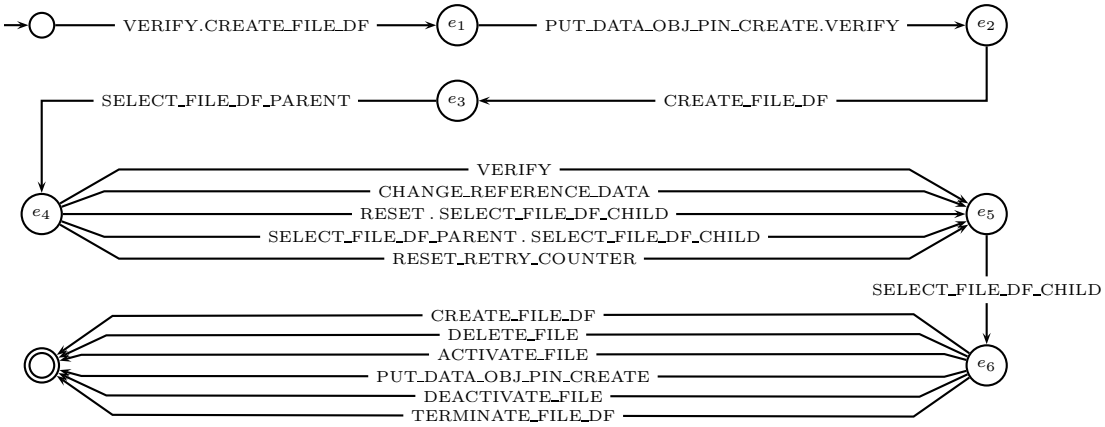


FIG. 7.7 – IAS : schéma de test TP1 (automate)

par authentification sur *pin1* et *pin2*) dans les DF *file_fid_1* et *file_fid_2* en explorant toutes les combinaisons possibles d'état d'authentification des deux objets PIN.

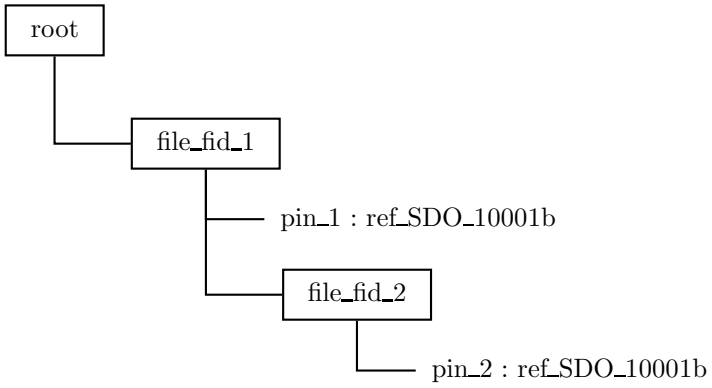


FIG. 7.8 – IAS : structure arborescente utilisée par TP2

Le troisième schéma (TP3) est pour sa part construit autour de l'idée de combiner les notions d'authentification et d'état du cycle de vie de la ressource dont on souhaite tester l'accès. Ainsi, ce schéma guide la sélection de tests en mettant en jeu la combinaison authentification/cycle de vie dans les différents cas possibles. Afin d'augmenter le nombre de tests et de prendre en compte la dimension temporelle, le schéma exprime le fait que le changement d'état du cycle de vie de la ressource peut s'exprimer à deux moments différents. Cette prise en compte induit que certains tests mettent en jeu deux changements successifs de l'état du cycle de vie de la ressource, et que d'autres mettent en jeu ce changement à un moment ou à un autre.

Le tableau 7.1 présente quelques métriques sur les trois schémas de test. Les colonnes présentent consécutivement le nombre d'opérations mises en jeu par le schéma, puis les nombres d'états et de transitions de l'automate associés au schéma. Le tableau 7.2 donne quelques

métriques concernant les tests produits. La première ligne concerne la suite de tests générée par LTG, les autres concernent les suites de tests générées à l'aide des schémas de test. Les colonnes présentent respectivement : le nombre de tests, leurs longueurs moyennes, mininales et maximales en nombre d'appels d'opérations pour chaque suite de tests.

Tests purposes	♯ Operations	♯ Transitions	♯ États
TP1	12	20	12
TP2	10	63	18
TP3	9	30	12

TAB. 7.1 – IAS : Taille des schémas de test

Tests	♯ Tests	Longueur moyenne	Longueur min	Longueur max
LTG	65	2,5	1	5
TP1	35	9,4	9	10
TP2	66	9,5	8	11
TP3	88	6,9	5	8

TAB. 7.2 – IAS : Taille des tests générés

7.1.6 Couverture des tests

Afin d'évaluer les tests produits par les différentes solutions de génération de tests, nous avons évalué leur couverture en terme d'états et de transitions sur une abstraction du modèle de l'application IAS (c.f. 6.2 pour une description de la méthode d'évaluation de la couverture basée sur une abstraction). Dans notre cas, l'abstraction produite est composée de 18 états et 497 transitions.

Le choix des variables et de la décomposition de leurs domaines ont été orientés de manière à privilégier l'observation de la dynamique du système correspondant au contrôle d'accès basé sur l'authentification de l'utilisateur sur code PIN. Les variables choisies sont :

current_df qui permet de connaître la position du DF (répertoire) courant dans l'arborescence des DF. Son domaine a été partionné en trois : le DF racine, les fils du DF racine et enfin les autres DF.

df_2_dfParent qui donne l'organisation de l'arborescence des DF. Son domaine a été divisé en deux : soit le système contient la racine et éventuellement des fils, soit le système contient plus d'un niveau de DF imbriqués.

pin_authenticated_2_df qui décrit l'état d'authentification des différents objets de type PIN dans les différents DF présents sur le système. Le domaine de cette variable a été partionné en trois : aucun PIN n'est authentifié, des PIN sont authentifiés et ils le sont uniquement dans le DF racine et enfin des PIN sont authentifiés et ils doivent l'être dans la descendance du DF racine.

La figure 7.3 présente la décomposition symbolique des domaines des ces variables. Du fait des différents états symboliques de cette abstraction, celle-ci nous offre un réferentiel bien adapté à l'observation du système pour les tests générés à partir des schémas de test TP1

Variables	Décomposition du domaine
current_DF	$current\_DF = ROOT\_ID \vee$ $current\_DF \in DF\_2\_DFParent^{-1}[\{ROOT\_ID\}] \vee$ $current\_DF \notin (\{ROOT\_ID\} \cup DF\_2\_DFParent^{-1}[\{ROOT\_ID\}])$
DF_2_DFParent	$ran(DF\_2\_DFParent) = ROOT\_ID \vee$ $ran(DF\_2\_DFParent) \neq ROOT\_ID$
pin_authenticated_2_df	$pin\_authenticated\_2\_df = \varnothing \vee$ $pin\_authenticated\_2\_df \neq \varnothing \wedge pin\_authenticated\_2\_df \rhd \{ROOT\_ID\} = \varnothing \vee$ $pin\_authenticated\_2\_df \neq \varnothing \wedge pin\_authenticated\_2\_df \rhd \{ROOT\_ID\} \neq \varnothing$

Tab. 7.3 – IAS : Décomposition des domaines des variables de l'abstraction

et TP2. Par contre, le schéma de test TP3 est basé sur l'idée de mettre en jeu différentes situations d'authentification combinées avec différents états de cycle de vie des objets auxquels on souhaite accéder (via l'authentification). Or, le cycle de vie des objets n'a pas été pris en compte dans cette abstraction, celle-ci n'est donc pas bien adaptée pour l'observation du système pour les tests générés avec le schéma de test TP3. Il aurait été intéressant, soit de considérer le cycle de vie dans l'abstraction, mais le nombre d'états symboliques résultant aurait été beaucoup plus élevé, rendant la génération de l'abstraction, la mesure de la couverture des tests et l'interprétation des résultat plus difficiles ; soit de considérer une abstraction différente pour évaluer la couverture des tests générés à partir du schéma de test TP3. Mais l'ajout d'un nouveau référentiel aurait rendu plus délicat l'analyse des résultats de couverture des tests.

Les tableaux 7.4 et 7.5 présentent les résultats du calcul de la couverture des différentes suites de tests sur l'abstraction du modèle choisie comme référentiel. Le tableau 7.4 donne le nombre de tests, la couverture des états ($\frac{\sharp\, états\ atteints}{\sharp\, états\ du\ référentiel}$) et la couverture des transitions ($\frac{\sharp\, transitions\ activées}{\sharp\, transitions\ du\ référentiel}$) par les suites de tests :

- générées par LTG avec un critère de couverture des conditions et décisions (C/DC) et une couverture des paramètres des opérations aux limites,
- générées à partir des trois schémas de tests TP1, TP2 et TP3,
- résultant de l'union des trois suites de tests générées à partir des trois schémas de tests TP1, TP2 et TP3.

En vue d'étudier la complémentarité des suites de tests issues des schémas de test avec la suite de test générée par LTG, nous avons réalisé des mesures de couverture supplémentaires. Celles-ci sont présentées dans le tableau 7.5. Il s'agit des résultats de mesure de couverture de l'union de la suite de tests générée par LTG avec, respectivement, les suites de tests TP1, TP2, TP3 et TP123 (union des ensembles de tests TP1, TP2 et TP3). Des valeurs de couverture de transitions des tableaux 7.4 et 7.5, nous avons déduit des résultats de complémentarité (c.f. 6.2).

7.1.7 Analyse des résultats et bilan

Le tableau 7.2 montre que les tests issus des différents schémas sont en moyenne entre 2,5 et 4 fois plus long en terme de nombre d'appels d'opération que les tests générés par LTG. Ce résultat confirme que la génération de tests telle qu'elle est réalisée par LTG, consiste à trouver

Tests	♯ tests	Couverture d'états	Couverture de transitions
LTG	65	5/18 = 27,78 %	33/497 = 6,64 %
TP1	35	9/18 = 50,00 %	35/497 = 7,04 %
TP2	66	12/18 = 66,67 %	52/497 = 10,46 %
TP3	88	5/18 = 27,78 %	23/497 = 4,63 %
TP123	189	13/18 = 72,22 %	87/497 = 17,51 %

TAB. 7.4 – IAS : Couverture d'états et de transitions de l'abstraction

			% complémentarité	
Tests	♯ tests	Couverture de transitions	comp(LTG,TP_i)	comp(TP_i,LTG)
TP1 ∪ LTG	100	63/497 = 12,68 %	28/33 = 84,8%	30/35 = 85,7%
TP2 ∪ LTG	131	83/497 = 16,70 %	31/33 = 93,9%	50/52 = 96,2%
TP3 ∪ LTG	153	51/497 = 10,26 %	28/33 = 84,8%	18/23 = 78,3%
TP123 ∪ LTG	254	109/497 = 21,93 %	22/33 = 66,7%	76/87 = 87,4%

TAB. 7.5 – IAS : Complémentarité de la couverture des tests

le chemin le plus court permettant d'atteindre une cible de test (un comportement particulier d'une opération). Par contre les schémas de test nous offrent la possibilité de contraindre la génération de tests à "emprunter" des chemins plus longs pour tester des enchaînements de comportements, mis en évidence par l'expertise d'un ingénieur de validation et potentiellement révélateurs de non-conformités.

Le tableau 7.4 laisse apparaître que les résultats de couverture des tests sur l'abstraction sont en moyenne meilleurs. Par contre, ces résultats ne sont meilleurs qu'au regard du référentiel choisi. Ici, l'abstraction a été calculée pour observer les mécanismes de contrôle d'accès qui sont justement les mécanismes mis en jeu par les schémas de test. Les tests générés par LTG ne sont pas particulièrement orientés vers cet aspect du système. D'ailleurs, il est aisé de se convaincre de l'importance du référentiel dans ces résultats au vu des scores de couverture obtenus par la suite de tests issue du schéma TP3, pour laquelle nous avons mentionné que le référentiel n'était pas adapté.

La dernière ligne du tableau 7.4, qui présente les résultats obtenus concernant la couverture de l'union des suites de tests issues des schémas TP1, TP2 et TP3, montre que les scores obtenus sont loin de correspondre à la somme des résultats obtenus pour chacune des suites de tests. Ce constat peut laisser penser que les tests issus des différents schémas de test ne sont pas très complémentaires entre eux. Cela s'explique essentiellement par le fait que les transitions mises en jeu par les schémas TP1 et TP2 sont essentiellement les mêmes ; la complémentarité de ces deux suites de tests aurait nécessité l'utilisation d'un critère de mesure de couverture de paires de transitions pour être mise en valeur.

Finalement, les résultats qui nous semblent les plus intéressants sont ceux présentés dans le tableau 7.5, car ils montrent de manière claire la complémentarité des tests générés à partir de schémas de tests par rapport à ceux générés par LTG en terme de couverture de transitions. Ces résultats nous montrent qu'individuellement chaque schéma permet de générer des ensembles de tests fortement complémentaires aux tests fonctionnels (en terme de couverture de transition) :

- On constate qu'entre 78% et 96% des transitions couvertes par les ensembles de tests générés par chacun des schémas ne le sont pas par les tests générés par LTG.
- De même, pour chacune des comparaisons, on constate qu'entre 84% et 94% des transitions couvertes par LTG ne sont pas redondantes avec celles couvertes par chacun des ensembles de tests générés à partir des schémas.
- Enfin, globalement 66,7% des transitions couvertes par les tests produits avec LTG ne sont pas couvertes par l'ensemble des tests produits à partir des schémas. Et réciproquement, 87,4% des transitions couvertes par l'ensemble des tests générés à partir des schémas sont complémentaires avec celles couvertes par l'ensemble de tests générés grâce à LTG.

Cette expérimentation effectuée sur une application de taille significative nous a permis d'apporter quelques arguments en faveur de la complémentarité de l'utilisation des schémas de test par rapport à l'utilisation d'une démarche de génération de tests basée sur la couverture de comportements, de décisions/conditions et de valuation des paramètres aux limites. De plus, cette expérimentation nous a permis d'utiliser quelques outils (MECA et GeneSyst) développés autour de la méthode B et du langage B en général.

7.2 POSIX

Dans le cadre du mini-challenge POSIX [JH07] dont l'objectif est la conception et la vérification d'un système destiné à des supports de type flash (clé-usb,...) respectant les normes POSIX, nous avons expérimenté la génération de tests à l'aide de schémas de tests. Ce travail a fait l'objet d'une publication [DdKT08].

7.2.1 Présentation générale

La norme POSIX définit un ensemble de standards pour les API des logiciels fonctionnant sur des systèmes d'exploitation UNIX. Dans le cadre du mini-challenge, nous nous sommes intéressés à la gestion de fichiers. L'organisation d'un système de fichiers POSIX repose sur les notions de nœuds et de descripteurs de fichiers. Les nœuds servent à identifier les entités existantes dans le système de fichiers, qui sont soit des fichiers, soit des répertoires. Les descripteurs quant à eux servent à identifier les fichiers ouverts (en écriture, lecture ou lecture/écriture) et indiquent la position (offset) courante du curseur de lecture ou d'écriture. Les commandes liées à la gestion de fichiers auxquelles nous nous sommes intéressées sont :

- mkdir/rmdir : Création/suppression de répertoires.
- chdir : Déplacement dans l'arborescence des repertoires.
- open/close : Ouverture/fermeture de fichier.
- rename : renommage/déplacement d'un fichier ou d'un repertoire
- unlink : suppression d'un fichier
- read/write : Lecture/écriture dans un fichier.
- (f)truncate[14] : Modification de la taille d'un fichier.
- (f)stat[14] : Affichage des informations concernant un fichier (taille, date, droits,...).
- opendir/closedir : Ouverture/fermeture d'un répertoire (uniquement en lecture).

7.2.2 Modélisation

A partir des spécifications, nous avons fait un modèle B ($\approx$ 1400 lignes) représentant un tel système de fichier. Le modèle de données comprend :

- Des ensembles énumérés :
  - types de nœud (fichiers, répertoires),
  - modes d'ouverture (lecture, écriture, lecture/écriture),
  - codes d'erreurs
- Des constantes entières :
  - taille maximum d'un fichier,
  - nombre maximum de fichiers ouverts simultanément.
- Des variables :
  - un ensemble de répertoires,
  - un ensemble de fichiers,
  - un ensemble de nœuds,
  - une fonction pour associer les nœuds à leur nom,
  - une fonction pour associer les nœuds à leurs parents dans l'arborescence dans le système de fichiers,

[14] le modificateur f est employé pour désigner un fichier ouvert par son decripteur au lieu de le désigner de manière absolue par son nœud

- un relation définissant la fermeture transitive de l'arborescence du système de fichiers,
- un répertoire courant,
- une fonction pour associer un fichier à sa taille,
- une fonction pour associer un fichier à son contenu,
- un ensemble de descripteurs de fichiers,
- un fonction pour associer un descripteur de fichier à un nœud,
- un fonction pour associer un descripteur de fichier au mode d'ouverture sous lequel il est ouvert.

La gestion du contenu des fichiers étant complexe et notre objectif étant de nous concentrer sur le contrôle d'accès, nous avons décidé d'abstraire cette information. L'abstraction choisie pour le contenu des fichiers consiste à ramener le contenu du fichier à sa taille et à un numéro de version qui permet de connaître l'historique des changements de taille du fichier.

La partie dynamique du modèle B spécifie l'ensemble des commandes de gestion du système de fichiers (c.f.7.2.1) que nous avons présentées sous la forme d'opérations B.

7.2.3 Génération des tests

La génération de tests fonctionnels sur le modèle à l'aide de l'outil LTG a permis de produire 78 tests. Nous avons rédigé quatre schémas de tests dans le but de compléter cette campagne de tests fonctionnels. Ces quatres schémas, sans couvrir l'intégralité des spécifications adressent chacun un point critique de ces dernières :

- Schéma 1 : ensemble de scénarios visant à exercer le système sur les questions de nombre maximum de fichiers ouverts (4 tests).
- Schéma 2 : ensemble de scénarios destinés à valider la gestion de la taille maximale des fichiers (24 tests).
- Schéma 3 : ensemble de scénarios jouant sur l'utilisation (lecture, écriture, ...) de fichiers supprimés/déplacés ou de descripteurs de fichiers fermés (147 tests).
- Schéma 4 : ensemble de scénarios mettant en jeu l'ouverture multiple d'un fichier et son édition (lecture/changement de taille, ...) au travers des différents descripteurs de fichiers pointant sur celui-ci (64 tests).

Les schémas 1 et 2 avaient pour objectif de faciliter la génération de tests pour des cibles de tests que LTG n'avait pu atteindre étant donné les limitations de profondeur de (nombre d'appels d'opérations trop élevé pour mettre le système dans le contexte permettant de tester le comportement visé) et de temps de recherche. Les schémas 3 et 4 avaient pour objectif d'étendre la campagne de tests fonctionnels afin d'augmenter la couverture du système par les tests.

7.2.4 Résultats

Contrairement aux deux autres expérimentations IAS (c.f. 7.1) et Demoney (c.f. 7.3), nous n'avons pas réalisé de mesures de couverture sur les tests générés. Par contre, nous avons confronté ces tests à deux implémentations différentes d'un système de gestion de fichiers "respectant" la norme POSIX.

Le premier système sur lequel les tests ont été joués est une distribtion Linux (Ubuntu). Afin de jouer les tests sur ce système, nous avons développé une couche d'adaptation basée sur la génération de C à partir des tests. Au cours de cette expérimentation, 7 tests fonctionnels ont révélé des non-conformités entre le modèle et l'implémentation. L'analyse des

non-conformités entre les tests et l'implémentation a permis de déterminer que les causes de ces erreurs étaient dues à des choix d'implémentation laissés libres dans les spécifications et qui ont donné lieu à deux interprétations différentes entre le modèle et les spécifications. Par contre l'exécution des tests issus des schémas n'a pas révélé d'erreurs.

Le second système confronté aux tests produits est une implémentation Java d'un système de gestion de fichiers basé sur la norme POSIX que nous avons réalisée pour l'occasion. Nous avons réalisé une couche d'adaptation reposant sur un traducteur produisant un fichier de tests JUnit à partir des tests. L'exécution des tests sur cette implémentation a permis de trouver quelques bugs parmi lesquels des problèmes de pointeurs nuls ou des accès en dehors des bornes de tableaux. Dans un second temps, nous avons produit quelques versions de l'implémentation contenant des erreurs pour déterminer si celles-ci étaient détectables par les suites de tests. Parmi ces erreurs, l'une interdisait l'ouverture de plus d'un descripteur par fichier. Cette erreur a été capturée par les tests issus du schémas 4, mais pas par les tests fonctionnels. Une autre erreur introduite entrainait un mauvais déroulement de la fermeture d'un fichier et par conséquent sa possible utilisation ultérieure. Cette erreur, qui n'a pas été mise en évidence par l'exécution des tests fonctionnels, l'a été par l'exécution des tests issus du schéma 3.

7.3 Demoney

Dans le cadre des expérimentations que nous avons faites, nous nous sommes intéressé au cas d'étude Demoney. Cette application de gestion de porte-monnaie électronique possède quelques similarités avec l'application IAS, bien que son système de contrôle d'accès et de gestion du cycle de vie soit plus simple.

7.3.1 Présentation générale

Une présentation de l'application Demoney, de ses spécifications et de sa modélisation est réalisée dans la partie 4.1. Par ailleurs, les spécifications complètes et le modèle sont disponibles dans l'annexe 1. Nous nous limitons donc ici à un rappel sur le fonctionnement de l'application Demoney et à une présentation des commandes PIN_CHANGE_UNBLOCK et RESET qui n'ont pas encore été introduites.

Cycle de vie

La figure 7.9 montre les différentes évolutions possibles des états du cycle de vie du porte-monnaie électronique. Le premier état correspond à l'étape de *personnalisation* durant laquelle sont fixés les différents paramètres de la carte comme les valeurs des codes PIN de l'utilisateur et de la banque, ou les valeurs maximales du solde de la carte et du montant des débits. Nous ne nous intéresserons pas à cette phase du cycle de vie de la carte lors de nos expérimentations.

L'état d'*utilisation* de la carte correspond à la période durant laquelle celle-ci peut-être utilisée normalement pour réaliser des crédits ou des débits.

L'état de *blocage* de la carte intervient lorsque plusieurs échecs successifs d'authentification de l'utilisateur ont été commis (dans notre cas : trois échecs). Cet état est réversible sous réserve que la banque s'authentifie sur la carte pour la débloquer et changer la valeur du code PIN de l'utilisateur grâce à la commande PIN_CHANGE_UNBLOCK.

Lorsque la carte est en état de blocage et que quatre échecs successifs d'authentification de la banque interviennent, la carte est détruite. La *destruction* de la carte rend l'exécution de toute commande impossible, celle-ci n'est donc plus utilisable.

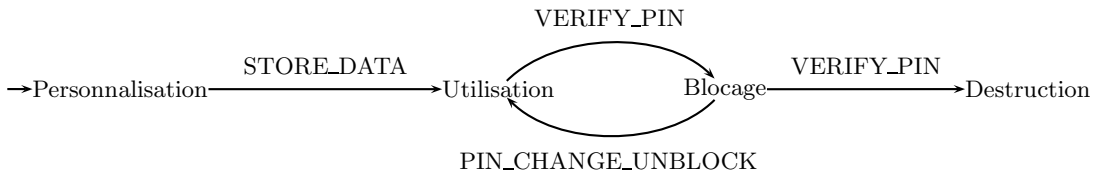


Fig. 7.9 – Demoney : Evolution du cycle de vie de la carte

Authentification

Le système d'authentification a deux objectifs, il permet d'une part à l'utilisateur de créditer de l'argent sur sa carte, et d'autre part, à la banque de débloquer une carte. La commande d'authentification est *VERIFY_PIN*, elle possède deux paramètres qui sont l'identifiant du PIN (soit celui de l'utilisateur, soit celui de la banque) et la valeur du code PIN. En cas d'échec de la commande, le compteur d'essais associé à l'identifiant de PIN passé en paramètre est

décrémenté et toute autre authentification réalisée avec succès auparavant est annulée. En cas de succès, l'authentification est aquise pour une unique utilisation correcte[15] et le compteur d'essai associé au PIN est réinitialisé à son maximum.

On peut noter que l'utilisateur ne peut s'authentifier que pendant la phase d'*utilisation* de la carte et que la banque ne peut s'authentifier que lorsque la carte est dans l'état de *blocage*. Les commandes nécessitant une authentification pour se dérouler avec succès sont :

- INITIALIZE_TRANSACTION pour initialiser une transaction de crédit,
- PIN_CHANGE_UNBLOCK pour débloquer une carte et changer la valeur du code PIN utilisateur.

Transaction de crédit

La transaction de crédit de la carte se déroule en deux étapes : d'abord la transaction doit être initialisée (grâce à la commande *INITIALIZE_TRANSACTION*) puis elle doit être exécutée (grâce à la commande COMMIT_TRANSACTION). Pour que ces commandes s'exécutent avec succès, certaines contraintes doivent être observées. Pour initialiser une transaction de crédit :

- La carte doit être en phase d'*utilisation*.
- L'utilisateur doit être authentifié.
- Le montant de la transaction de crédit ajouté au solde de la carte ne doit pas dépasser le solde maximal de la carte.

Pour exécuter une transaction de crédit, seules deux conditions sont nécessaires, la carte doit être en phase d'utilisation et une transaction de crédit doit avoir été initialisée. Il faut tout de même noter que l'exécution d'une transaction doit survenir directement après son initialisation. L'exécution de toute autre commande que COMMIT_TRANSACTION après une initialisation correcte a pour effet d'annuler la transaction en cours.

7.3.2 Génération de tests

Le modèle de l'application Demoney (présenté en annexe 1) a été utilisé pour générer des tests à l'aide des deux méthodes dont nous souhaitons évaluer la complémentarité. En premier lieu, nous avons généré des tests fonctionnels par couverture des comportements des opérations du modèle aux limites avec l'outil LTG. Puis, nous avons généré des tests à l'aide de schémas de tests répondant à des exigences de validation sur les transactions de crédit. Dans les deux cas, nous avons considéré un état initial du système où la personnalisation était déjà réalisée (cette phase du cycle de vie ne faisant pas partie du périmètre de validation que nous nous étions fixé).

Génération des tests fonctionnels

Les tests fonctionnels ont été générés en considérant la couverture de comportements pour les opérations :

- VERIFY_PIN
- INITIALIZE_TRANSACTION
- COMMIT_TRANSACTION

[15] Par exemple une authentification de l'utilisateur n'est valable que pour un crédit réussi, si ce dernier souhaite faire un autre crédit par la suite, il doit à nouveau s'authentifier.

– PIN_CHANGE_UNBLOCK
– RESET

Ces opérations ont été choisies car elles sont toutes applicables en phase d'utilisation et elles peuvent toutes avoir une influence dans une séquence d'opérations menant à une tentative de crédit de la carte. Le critère de sélection des comportements utilisé est le critère "*modified condition/decision coverage*" (c.f. 2.2.1), il s'agit d'un des critères qui offre une des meilleures couvertures des conditions et décisions tout en offrant une combinatoire raisonnable. En plus de ce critère de couverture des conditions et décisions, nous avons appliqué un critère de couverture aux limites des domaines de définition des paramètres des opérations pour chaque comportement.

La couverture des comportements a permis d'extraire 34 cibles de tests différentes. L'application du critère de couverture des paramètres aux limites sur ces cibles de test a donné lieu à la production de 47 tests d'une longueur moyenne de 2,4 appels d'opérations.

Exigences de validation liées au crédit

Nous avons extrait, des spécifications de l'application Demoney, un certain nombre d'exigences de validation liées aux transactions de crédit. Ces exigences concernent des aspects d'authentification, de cycle de vie, de contraintes sur les montants d'argent disponibles sur la carte qui entrent tous en jeu pour les transactions de crédit et pour la succession des commandes utilisées pour réaliser un crédit.

Pour initialiser une transaction de crédit, la commande INITIALIZE_TRANSACTION doit être exécutée et les conditions suivantes doivent être satisfaites :

– La carte doit être en phase d'utilisation.
– L'utilisateur doit-être authentifié.
– Le montant du crédit ne doit pas être nul.
– Le montant du crédit additionné au solde de la carte ne doit pas excéder le solde maximal.

L'authentification de l'utilisateur est réalisée à l'aide de la commande VERIFY_PIN. Le succès de l'exécution de cette commande dépend de la satisfaction des conditions suivantes :

– La carte doit être en phase d'utilisation.
– Le code PIN passé en paramètre doit être celui de l'utilisateur.

Concernant le cycle de vie, on notera :

– Trois échecs d'authentification successifs de l'utilisateur font passer l'application de phase d'*utilisation* en phase de *blocage*.
– Quatres échecs d'authentification successifs de la banque font passer l'application de phase de *blocage* en phase de *destruction* de la carte.

A partir de ces informations, nous avons déduit certaines situations qui nous semblaient intéressantes à tester et certains "patterns" communs à ces situations. L'utilistation de ces patterns nous permet de factoriser certaines parties communes à plusieurs schémas. Cette pratique permet de faciliter d'une part le travail de conception des schémas, et d'autre part les modifications qui pourraient être apportées aux schémas par la suite. Les situations que nous avons choisies sont :

– différents scénarios de succès et d'échecs d'authentification en phase d'utilisation avant une tentative de crédit ;
– des ruptures de séquences entre l'initialisation et la validation d'un crédit ;
– diverses tentatives de crédit lorsque la carte est en phase de blocage ;

- diverses tentatives de crédit lorsque la carte est en phase de blocage après des essais infructeux de déblocage ;
- diverses tentatives de crédit après un déblocage de la carte ;
- diverses tentatives de crédit lorsque la carte est en phase de destruction.

Les "patterns" communs que nous avons identifiés sont :

1. Trois comportements possibles pour l'initialisation d'un crédit dépendant du montant d'argent à créditer : un montant nul, un montant trop élevé ou un montant correct.
2. Trois comportements d'échec d'authentification de l'utilisateur en phase d'utilisation de la carte : l'utilisation d'une valeur de code PIN erronée (soit la valeur du code PIN de la banque, soit une valeur erronée quelconque) ou un essai d'authentification de l'utilisateur sur le code PIN de la banque.
3. Trois comportements de perte de l'authentification de l'utilisateur : une erreur d'authentification du code PIN utilisateur due à une valeur de code PIN erronée (soit la valeur du code PIN de la banque, soit une valeur érronée quelconque) ou l'initialisation d'une transaction de crédit.
4. Quatres comportements de tentatives réussies ou infructueuses de déblocage dues aux différentes valeurs possibles de code PIN passées en paramètres de l'opération de déblocage de la carte : PIN_CHANGE_UNBLOCK. Les valeurs utilisées sont celles des codes PIN de la banque et de l'utilisateur, une valeur de code PIN quelconque, ou valeur ne correspondant pas à un code PIN (c.a.d. inférieure à 1 ou supérieure à 9999).

Schémas de tests

Schema *init_credit* : Ce schéma représente le pattern identifié pour couvrir les différents comportements de l'initialisation de transaction de crédit. Les labels IT_CREDIT_ERROR_0, IT_CREDIT_ERROR_MAX et IT_CREDIT_PARAM_OK correspondent aux comportements de l'opération d'initialisation en fonction du montant à créditer (c.a.d. un montant nul, un montant trop élevé ou un montant correct).

$$\begin{aligned}&\text{INITIALIZE\_TRANSACTION}/_w\{\text{IT\_CREDIT\_ERROR\_0}\}\\ &| \ \text{INITIALIZE\_TRANSACTION}/_w\{\text{IT\_CREDIT\_ERROR\_MAX}\}\\ &| \ \text{INITIALIZE\_TRANSACTION}/_w\{\text{IT\_CREDIT\_PARAM\_OK}\}\end{aligned}$$

Schéma *fail_auth_user* : Ce schéma décrit les différentes erreurs d'authentification de l'utilisateur dues à de mauvais choix de paramètres pour l'opération VERIFY_PIN. Les labels VERIFY_ERROR_USERBANK et VERIFY_ERROR_USEROTHER correspondent aux comportements d'erreur de l'opération VERIFY_PIN lorsque l'utilisateur essaie de s'authentifier avec la valeur du code PIN de la banque ou une autre valeur de code PIN différente de celle de l'utilisateur. Le label VERIFY_ERROR_BANK_USER correspond au comportement où l'utilisateur essaie de s'authentifier sur le code PIN de la banque avec son propre code secret.

$$\begin{aligned}&\text{VERIFY\_PIN}/_w\{\text{VERIFY\_ERROR\_USERBANK}\}\\ &| \ \text{VERIFY\_PIN}/_w\{\text{VERIFY\_ERROR\_USEROTHER}\}\\ &| \ \text{VERIFY\_PIN}/_w\{\text{VERIFY\_ERROR\_BANKUSER}\}\end{aligned}$$

Schéma *perte_auth_user* : Ce schéma représente le pattern identifié pour couvrir les différents comportements identifiés pour la perte d'authentification de l'utilisateur. Les labels de comportements sont les mêmes que ceux utilisés dans le schéma *fail_auth_user*.

(VERIFY_PIN/$_w${VERIFY_ERROR_USERBANK}
| VERIFY_PIN/$_w${VERIFY_ERROR_USEROTHER}
| INITIALIZE_TRANSACTION)$\rightsquigarrow$(authenticated_pin = PIN_NONE)

Schéma *déblocage* : Ce schéma décrit différents comportements activables pour l'opération PIN_CHANGE_UNBLOCK en fonction de la valeur du code PIN passée en paramètre et destinée à remplacer celle du code PIN de l'utilisateur. Les labels CHANGE_UNBLOCK_BANK, CHANGE_UNBLOCK_USER, CHANGE_UNBLOCK_OTHER et CHANGE_UNBLOCK_KO correspondent aux valeurs de code pin passées en paramètre : valeur du code PIN de la banque, valeur actuelle de code PIN de l'utilisateur, valeur quelconque de code PIN et une valeur qui n'est pas une valeur de code PIN.

PIN_CHANGE_UNBLOCK/$_w${CHANGE_UNBLOCK_BANK}
| PIN_CHANGE_UNBLOCK/$_w${CHANGE_UNBLOCK_USER}
| PIN_CHANGE_UNBLOCK/$_w${CHANGE_UNBLOCK_OTHER}
| PIN_CHANGE_UNBLOCK/$_w${CHANGE_UNBLOCK_KO}

schéma 1 : Ce schéma résulte de la combinaison de différentes situations d'authentification et des trois possibilités d'initialisation du crédit. Parmi les situations d'authentification, nous considérons les suivantes :

- Aucune authentification n'est réalisée ;
- L'authentification de l'utilisateur est un échec ;
- L'authentification de l'utilisateur est un succès ;
- L'authentification de l'utilisateur est un succès, mais elle est perdue ensuite ;
- L'authentification de l'utilisateur est un succès, elle est perdue ensuite, mais elle est réussie à nouveau.

Chacune de ces situations d'authentification est suivie d'une initialisation de transaction de crédit puis d'un essai de validation de la transaction. Pour chacune des sous-situations correspondant à un des patterns que nous avons cités (échec d'authentification, perte d'authentification et initialisation de transaction), toutes les possibilités sont explorées.

```
(
   (
      VERIFY_PIN⇝(authenticated_pin = PIN_USER)
      .(
         (perte_auth_user)
         .VERIFY_PIN⇝(authenticated_pin = PIN_USER)?
      )?
   )
   |(fail_auth_user)
)?
.(init_credit)
.COMMIT_TRANSACTION
```

schéma 2 : Ce schéma explore des scénarios qui mettent en jeu la succession des commandes d'initialisation de crédit et de validation du crédit :

- Aucune initialisation avant la validation de la transaction.
- Échec d'initialisation (montant de la transaction trop important ou nul) avant la validation de la transaction.
- Appel d'une commande entre une initialisation de crédit correcte et sa validation.

$$
\begin{array}{l}
\text{VERIFY\_PIN}{\rightsquigarrow}(\text{authenticated\_pin} = \text{PIN\_USER}) \\
.(\\
\quad (\\
\quad\quad \text{INITIALIZE\_TRANSACTION}{\rightsquigarrow}(\text{debit} < 0) \\
\quad\quad .\$\text{OP\_all}{\rightsquigarrow}(\text{debit} = 0) \\
\quad) \\
\quad |(\boldsymbol{init\_credit}){\rightsquigarrow}(\text{debit} = 0) \\
)? \\
.\text{COMMIT\_TRANSACTION}
\end{array}
$$

schéma 3 : Ce schéma vise à explorer diverses situations où des essais de transactions de crédit sont réalisées lorsque la carte est bloquée. Le label VERIFY_UNKNOW_USER_USER correspond au comportement de l'opération VERIFY_PIN lorsque l'utilisateur tente de s'authentifier sur le PIN utilisateur avec le bon code secret (le résultat de l'opération dépend alors de l'état de cycle de vie de la carte). Ces essais ont lieu lorsque :

- La carte est bloquée.
- La carte est bloquée et l'utilisateur a tenté de s'authentifier.
- La carte est bloquée et la banque s'est authentifiée.

$$
\begin{array}{l}
(\text{VERIFY\_PIN}\{3\}){\rightsquigarrow}(\text{etat\_carte} = \text{invalid}) \\
.(\\
\quad \text{VERIFY\_PIN}/_w\{\text{VERIFY\_UNKNOW\_USER\_USER}\} \\
\quad |\text{VERIFY\_PIN}{\rightsquigarrow}(\text{authenticated\_pin} = \text{PIN\_BANK}) \\
)? \\
.(\boldsymbol{init\_credit}) \\
.\text{COMMIT\_TRANSACTION}
\end{array}
$$

schéma 4 : Ce schéma décrit l'essai de transactions de crédit lorsque la carte est bloquée et après des tentatives de déblocage infructueuses dues à un mauvais état d'authentification. Les tentatives de déblocage ont lieu soit sans aucune tentative d'authentification préalable, soit avec une tentative d'authentification de l'utilisateur préalable. Les tentatives de déblocage sont réalisées avec la commande PIN_CHANGE_UNBLOCK qui prend en paramètre une valeur de code PIN qui servira à écraser l'ancienne valeur du code PIN utilisateur en cas de succès. Dans les deux cas, quatres valeurs sont utilisées pour le paramètre de cette commande : la valeur du code PIN de la banque, celle de l'utilisateur, une valeur de code PIN quelconque et une valeur qui ne correspond pas à un code PIN (n'appartenant pas à l'intervalle [0..9999]).

$$
\begin{array}{l}
(\text{VERIFY\_PIN}\{3\}){\rightsquigarrow}(\text{etat\_carte} = \text{invalid}) \\
.\text{VERIFY\_PIN}/_w\{\text{VERIFY\_UNKNOW\_USER\_USER}\}? \\
.(\boldsymbol{déblocage}) \\
.(\boldsymbol{init\_credit}) \\
.\text{COMMIT\_TRANSACTION}
\end{array}
$$

schéma 5 : Le point de départ de ce schéma est une situation dans laquelle la carte est bloquée. Ensuite la banque s'authentifie avec succès, et la commande PIN_CHANGE_UNBLOCK est appliquée avec ses différents comportements possibles (décrits pour le schéma 4). Enfin, des tentatives de crédit sont effectuées avec authentification préalable ou non de l'utilisateur.

(VERIFY_PIN{3})⇝(etat_carte = invalid)
.VERIFY_PIN⇝(authenticated_pin = PIN_BANK)
.(***déblocage***)
.VERIFY_PIN/$_w${VERIFY_UNKNOW_USER_USER}? .(***init_credit***)
.COMMIT_TRANSACTION

schéma 6 : Ce dernier schéma vise à exercer des tentatives de crédit lorsque la carte est "détruite". Ces tentatives de crédit sont réalisées : sans tentative d'authentification préalable, après une tentative d'authentification de l'utilisateur et après une tentative d'authentification de la banque.

(VERIFY_PIN{3})⇝(etat_carte = invalid)
.VERIFY_PIN{4}⇝(etat_carte = dead)
.(
  VERIFY_PIN/w{VERIFY_UNKNOW_BANKBANK}
  | VERIFY_PIN/w{VERIFY_UNKNOW_USERUSER}
)?
.(***init_credit***)
.COMMIT_TRANSACTION

Tests générés à partir des schémas

La génération des tests à partir des schémas a été réalisée à l'aide de l'outil LTG et du modèle B de l'application Demoney. Chaque schéma de test a été synchronisé avec le modèle Demoney et a donné lieu à la génération d'une campagne de tests. Les résultats de ces générations sont détaillés dans le tableau 7.6 qui présente pour chaque schéma, le nombre de séquences de tests produites et les longueurs moyennes de ces séquences exprimées en nombre d'appels d'opérations.

Nom du schéma	Nombre de tests	Longueur moyenne
Schéma 1	33	3.7
Schéma 2	11	3.6
Schéma 3	9	5.7
Schéma 4	24	6.5
Schéma 5	24	7.5
Schéma 6	9	9.7
Total	110	5.8

TAB. 7.6 – Demoney : Tests générés à partir des schémas

7.3.3 Couverture des tests

Afin d'analyser les résultats obtenus, en l'occurence les tests générés, nous avons procédé par évaluation de la couverture des tests sur une abstraction du modèle Demoney en terme de couverture d'états, de transitions et de paires de transitions (c.f. 6.2). Nous avons donc choisi un ensemble de variables issues du modèle B de l'application Demoney dont nous avons partionné les domaines afin de produire une abstraction de ce modèle grâce à l'outil GénéSyst.

Nous avons choisi deux types de variables : d'une part des variables concernant le cycle de vie de la carte (*etat_carte*) et les différents états possibles d'authentification (*authenticated_pin*) ; d'autre part des variables permettant d'observer une transaction en cours (*debit*) et l'effet d'une transaction (*solde*). Le tableau 7.7 présente la décomposition choisie pour chaque variable.

Variables	Décomposition du domaine
etat_carte	$etat\_carte = use \vee$ $etat\_carte = invalid \vee$ $etat\_carte = dead$
authenticated_pin	$authenticated\_pin = PIN\_NONE \vee$ $authenticated\_pin \neq PIN\_NONE$
solde	$solde = 0 \vee$ $solde > 0$
debit	$debit = 0 \vee$ $debit < 0 \vee$ $debit > 0$

TAB. 7.7 – Demoney : Décomposition des domaines des variables de l'abstraction

Le domaine de la variable *authenticated_pin* n'est partionné qu'en deux ensembles permettant uniquement de savoir si un PIN est authentifié ou non, sans précision sur l'identité du PIN authentifié. Ceci vient du fait que cette information serait redondante avec celle concernant l'état de la carte (partionnement du domaine de la variable *etat_carte*) ; en effet l'utilisateur ne peut s'authentifier qu'en phase d'utilisation de la carte (etat_carte = use) et la banque uniquement en phase de blocage (etat_carte = invalid).

A partir de cette décomposition, l'outil GénéSyst a produit une abstraction du modèle Demoney sous la forme un système de transitions étiquetées symbolique constitué de 14 états et 135 transitions. Le produit cartésien des ensembles de partitions des variables que nous avons choisi devrait mener à la production d'un graphe de 36 états, mais 22 sont en fait inatteignables.

Nous avons utilisé l'abstraction calculée par GénéSyst pour évaluer la couverture des différentes suites de tests que nous avons produites en terme de couverture d'états, de transitions et de paires de transitions. Ces résultats sont présentés dans le tableau 7.8, les lignes F, S et F $\cup$ S correspondent respectivement aux résultats de couverture pour les tests fonctionnels produits avec LTG, pour les tests générés à partir des schémas et pour l'union des suites de tests fonctionnelles et générées à partir des schémas.

Nous avons calculé la complémentarité des deux ensembles de tests obtenus à l'aide des schémas et de l'outil LTG en terme de couverture de transitions et de paires de transitions en appliquant la formule (Déf. 13) que nous avons définie pour l'expérimentation IAS. Ces

		Couverture en terme de :		
Tests	♯ tests	états (/14)	transitions (/135)	paires de transitions (/1270)
F	47	8	19	15
S	110	7	31	51
F ∪ S	157	8	35	54

TAB. 7.8 – Demoney : Évaluation de la couverture des suites de tests

résultats sont présentés dans le tableau 7.9, sous la forme de pourcentage de complémentarité de couverture (de transitions et de paires de transitions) de l'ensemble des tests fonctionnels (F) par rapport aux tests générés à partir des schémas (S) et réciproquement.

Critère de comparaison	% complémentarité de	
	F par rapport à S	S par rapport à F
Transitions	4/19 = 21,1%	16/31 = 51,6%
Paires de Transitions	3/15 = 20%	39/51 = 76,5%

TAB. 7.9 – Demoney : Complémentarité de la couverture des tests

7.3.4 Analyse des résultats et bilan

La lecture du tableau 7.8 souligne en premier lieu qu'un des états qui n'est pas couvert par les tests issus des schémas l'est par les tests fonctionnels. Il s'agit d'un état qui n'est accessible que par une transition représentant une opération de débit, or les schémas que nous avons présentés ne mettent en jeu que des transactions de crédit.

Concernant les six états qui ne sont couverts ni par les tests fonctionnels, ni par les tests issus des schémas, il s'agit d'états accessibles uniquement après un premier crédit sur la carte. Ils n'entrent pas dans le champ que nous nous étions fixé, il est donc normal qu'ils ne soient pas couverts par les tests issus des schémas. Par contre, le fait qu'ils ne soient pas couverts par les tests fonctionnels indique qu'il serait certainement intéressant de déployer des schémas de tests visant à les couvrir.

En terme de couverture de transitions et de paires de transitions, on observe que les tests générés à partir des schémas offrent une meilleure couverture que les tests générés avec LTG. Si l'on observe les résultats de complémentarité de couvertures entre les ensembles de tests (tab. 7.9), on constate que les tests générés à partir des schémas sont bien complémentaires, puisque plus de 50% des transitions et 75% des paires de transitions couvertes par ceux-ci sont complémentaires à celles couvertes par les tests fonctionnels.

On peut toutefois noter que, pour un rapport assez proche entre les nombres de tests fonctionnels et générés à partir des schémas, ces chiffres sont en retrait par rapport à ceux obtenus lors de l'expérimentation IAS (c.f. 7.1.7). Les chiffres qui mettent le plus en valeur ce retrait sont ceux qui montrent que seuls environ 20% des transitions et paires de transitions couvertes par les tests fonctionnels ne le sont pas par les tests générés à partir des schémas. On peut supposer que cette différence a deux causes principales :

– L'abstraction utilisée pour mesurer la couverture est relativement proche du modèle

dans le sens où les transitions de l'abstraction recouvrent en grande partie les comportements des opérations du modèle.
- Les schémas de tests rédigés sont essentiellement axés sur des enchaînements de comportements plus que sur des états par lesquels le système doit transiter.

Cette expérimentation, en plus de montrer que les tests générés à partir de schémas de tests ont un intérêt en terme de complémentarité avec les tests fonctionnels, nous a permis :
- d'expérimenter la modularité dans la rédaction des schémas au travers de l'identification de patterns et de leur factorisation,
- d'utiliser la mesure de couverture pour identifier des aspects du système qui ne sont pas ou peu couverts par les tests déjà produits (dans le cas de l'application Demoney, l'évolution du système après un premier crédit).

7.4 Résumé

Au cours de ce chapitre, nous avons présenté l'application de notre démarche de validation par génération de tests à partir de modèles. Les trois cas d'études sur lesquels notre démarche a été appliquée sont deux systèmes de type "carte à puce" (IAS et Demoney) et un système de gestion de fichiers (POSIX).

Ces expérimentations ont d'abord permis de montrer que le langage de schémas de test que nous proposons permet de décrire des ensembles de scénarios de validation. D'un point de vue méthodologique, ces expérimentations illustrent comment un schéma de test permet d'exprimer des scénarios qui sont basés à la fois sur :
- des propriétés issues des spécifications,
- des choix de validation qui restreignent les exécutions du système que l'on souhaite observer.

Le cas d'étude Demoney portant sur la spécification d'un porte-monnaie électronique a permis de soulever un point méthodologique intéressant. Ce point concerne la modularité des schémas de test. Effectivement, durant cette expérimentation, les schémas ont été en partie réalisés de manière modulaire par combinaison de plusieurs schémas différents. Cette manière de procéder permet de factoriser l'information à l'intérieur de "sous-schémas" et facilite par la suite la rédaction et la lecture des schémas de tests.

Enfin, les résultats expérimentaux que nous avons présentés soulignent la complémentarité des tests générés à partir des schémas avec ceux générés avec un critère de couverture structurelle des opérations du modèle. Cette complémentarité est notamment mise en valeur par les évaluations de couverture, des différents ensembles de tests, que nous avons réalisées au cours des expérimentations IAS et Demoney.

Quatrième partie

Epilogue

Chapitre 8

Conclusions et perspectives

Sommaire

8.1 Conclusions

Nos travaux s'inscrivent dans un contexte où les solutions de génération de tests à partir de modèles utilisant des critères de couverture structurelle du modèle comme critères de sélection de tests trouvent leurs limites face à l'augmentation constante de la complexité des systèmes à valider. Dans ce contexte, nous avons proposé une démarche de génération de tests qui étend les travaux menés autour de la génération automatique de tests fonctionnels obtenus par couverture structurelle de modèles décrits en langage B.

Les travaux que nous avons présentés reposent sur la valorisation de l'expérience humaine au travers de la définition d'objectifs de test destinés à offrir un critère de sélection de tests complémentaire au critère de couverture structurelle du modèle. L'utilisation d'objectifs de tests permet de considérer des ensembles de scénarios complexes d'exécution du système (liés à des propriétés de haut niveau) sans tomber dans l'exploration "exhaustive" d'un modèle et par conséquent se trouver face à des problèmes d'explosion combinatoire.

8.1.1 Langage de description de schémas de test

Le langage de description de schémas de tests que nous avons présenté permet d'exprimer des ensembles de scénarios d'exécution du système, partiellement instanciés, grâce à un langage basé sur les expressions régulières. Le lien entre un schéma de test et le modèle du système à valider s'exprime par des noms d'opération qui représentent les appels aux fonctionnalités du système, et des descriptions symboliques d'états qui permettent de contraindre les exécutions du système décrites par des enchaînements d'opérations. Le langage que nous

proposons permet également d'exprimer certaines directives de pilotage par des contraintes d'utilisation de certains des comportements d'opérations et par l'introduction d'un opérateur de choix exclusif. L'ajout de contraintes sur les comportements d'une opération à l'intérieur d'un schéma de test permet d'exprimer simplement le fait que l'on souhaite explorer un ensemble de comportements (un, plusieurs ou tous) de cette opération. Quant à l'opérateur de choix exclusif, il permet d'élaguer des chemins d'exécution décrit dans le schéma de test.

8.1.2 Génération de tests à partir de schémas de test

Nous avons proposé deux implémentations de génération de tests à partir de modèles et de schémas de tests qui exploitent les fonctionnalités de deux outils dédiés à la génération de tests à partir de modèles décrits en langage B. La première de ces implémentations repose sur la synchronisation d'un schéma de tests et d'une machine B qui produit une machine B formalisant l'ensemble des exécutions du système réduites aux scénarios décrits dans le schéma de test. Cette approche permet d'exploiter l'outil LTG afin de produire des séquences de tests à partir de la machine B synchronisée et des séquences d'étiquettes de transitions représentant les exécutions de l'automate.

La seconde implémentation présentée permet de générer des tests directement par animation de la machine B décrivant le système et d'un schéma de test. La machine B est animée de manière symbolique en suivant les enchaînements d'opérations décrits dans un schéma de test et en respectant les contraintes sur les états et les comportements des opérations. Une séquence d'exécution symbolique d'opérations, issue de l'animation, devient un test abstrait lorsque le dernier état du chemin qu'elle représente est atteint et que le système de contraintes qu'elle constitue est résolu pour fixer les valeurs des paramètres des opérations et les oracles (valeurs de retours des opérations).

8.1.3 Évaluation de l'approche

Nous avons également proposé une méthode d'évaluation de la couverture d'un ensemble de tests sur un modèle. Cette méthode a pour objectif d'évaluer la couverture des tests générés à partir d'un modèle et de schémas de test, et plus particulièrement la complémentarité entre les approches de génération automatique de tests à partir de schémas de tests et par couverture structurelle des opérations d'un modèle. La méthode que nous proposons repose sur l'évaluation de la couverture d'états, de transitions et de paires de transitions couverts par un ensemble de tests sur une structure de transitions étiquetées symbolique qui est une abstraction du modèle. Nous avons relevé le fait que cette méthode peut induire un biais dans le processus d'évaluation de la couverture des tests, car les critères qui définissent le calcul de l'abstraction sont founis par l'utilisateur. Mais nous avons proposé quelques règles de bonne pratique visant à diminuer ce problème et ainsi permettre une évaluation objective. Á partir de cette méthode d'évaluation, nous avons défini une fonction qui permet de calculer la complémentarité entre deux ensembles de tests en terme de couverture. Étant donnés deux ensembles de tests, cette fonction permet de calculer, pour chacun d'eux la proportion d'éléments (états, transitions, ...) de l'abstraction qui ne sont couverts que par cet ensemble de tests.

8.1.4 Expérimentations

Nous avons présenté l'application de nos travaux au travers de trois études de cas destinées à évaluer la faisabilité de notre approche et sa complémentarité avec une approche de génération automatique de tests utilisant la couverture structurelle des opérations d'un modèle comme critère de sélection de tests. Les cas d'études traités sont deux applications de type carte à puce : IAS et Demoney ; et un système de gestion de fichiers : Posix.

L'expérimentation réalisée sur la validation de systèmes de fichiers répondant à la norme Posix a été moins détaillée que les deux autres cas d'études, notamment concernant les mesures de couverture des tests engendrés. Néanmoins, ce cas d'étude a permis l'application de notre démarche à un système réel en générant des tests complémentaires aux tests fonctionnels générés par l'outil LTG. Pour ce cas d'étude, la comparaison entre les tests fonctionnels générés par LTG et ceux générés à partir des schémas de test est essentiellement empirique. On constate tout de même que l'utilisation de schémas de tests a permis de générer des tests qui n'avaient pu être atteints par LTG, en raison du nombre d'appels d'opérations nécessaires pour activer certains comportements d'opérations du modèle. Finalement, une évaluation de couverture en terme de détection de fautes a montré que certaines erreurs introduites manuellement dans une implémentation de Posix sont capturées par les tests générés à partir des schémas de tests, mais pas par les tests fonctionnels.

Les cas d'étude basés sur les spécifications IAS et Demoney offrent des résultats expérimentaux plus conséquents. En effet, pour ces deux expérimentations, nous avons évalué la couverture des ensembles de tests produit sur des abstractions des modèles et nous avons calculé les pourcentages de complémentarité en terme de couverture entre les tests fonctionnels et ceux générés en utilisant des schémas de test. Dans les deux cas, ces résultats montrent que l'utilisation de schémas de tests permet de renforcer la couverture du modèle par les tests générés sans pour autant remplacer les tests fonctionnels générés automatiquement par couverture structurelle des opérations du modèle. De plus, le cas d'étude Demoney a permis de soulever un point méthodologique intéressant puisque c'est à l'occasion de cette expérimentation que nous avons ressenti le besoin d'utiliser des "macros" permettant de factoriser certaines parties de schémas de tests afin de faciliter leur rédaction.

8.2 Perspectives

Production semi-automatique de schémas de tests

Notre démarche repose sur la formalisation d'objectifs de tests découlant de propriétés exprimées sur les systèmes à tester. A l'heure actuelle, la formalisation des objectifs de tests sous la forme de schémas de tests est entièrement manuelle. Une piste envisageable serait la production automatique de "patrons" de schémas de tests à partir de propriétés formalisées dans une logique adéquate. Les patrons de schémas de tests pourraient être des schémas fortement combinatoires définissant de larges ensembles de scénarios, destinés à valider une propriété, que l'utilisateur pourrait raffiner afin de cibler ceux qui lui semblent pertinents.

Évolution du langage de schémas de test

Concernant le langage de schémas de test, une première amélioration possible serait d'offrir la possibilité d'introduire des variables rigides à l'intérieur des schémas. Cette extension du

langage permettrait de définir des contraintes entre différents états et appels d'opérations. Il serait alors possible de créer simplement des scénarios visant à exercer des propriétés portant sur les données. Par exemple, il serait assez simple de décrire un scénario destiné à exercer la propriété : *"si le solde d'une carte de paiement vaut x, après le crédit d'un montant y sur la carte et un débit d'une somme y, le solde de la carte doit être égal à x"*.

Une seconde amélioration du langage de schémas de tests qui nous semble intéressante serait l'ajout de directives de couverture de données qui porteraient sur les paramètres des opérations, les variables du modèle ou les variables introduites dans les schémas. Ainsi, il serait possible de générer plusieurs cas de tests pour un scénario issu d'un schéma en contraignant certaines données à prendre différentes valeurs définies par l'utilisateur ou par un critère de couverture de données (par exemple la valuation de variables aux limites de leurs domaines). L'implémentation de ces améliorations du langage de schémas de test seraient plus simple à réaliser dans l'implémentation de notre démarche pour l'outil BZ-TT.

Extensions méthodologiques

D'un point de vue méthodologique, nous pensons qu'il pourrait être intéressant d'explorer les possibilités d'améliorer la traçabilité entre les tests produits et les schémas de test. Une possibilité pourrait être un mécanisme d'annotations qui permettraient d'étiqueter les différentes branches d'un choix à l'intérieur d'un schéma de manière à identifier les différents scénarios issus d'un même schéma. Un tel mécanisme pourrait être profitable d'une part pour faciliter l'identification des causes d'un test en échec lors de l'exécution de celui-ci, mais il pourrait également être utilisé à des fins de documentation pour fournir une synthèse sur les différents aspects du système ayant fait l'objet de validation.

Nous proposons une solution d'évaluation de la couverture des tests produits sur une abstraction du modèle qui a servi à les générer. Cette méthode d'évaluation a principalement été utilisée dans le but de mesurer la complémentarité entre les ensembles de tests générés à partir de schémas et ceux générés à partir d'un critère de couverture structurelle des opérations du modèle. Au cours de l'expérimentation portant sur Demoney, une autre utilisation de cette méthode d'évaluation est apparue. Nous avons vu que les résultats d'évaluation de la couverture des ensembles de tests pouvaient également être utilisés pour identifier des aspects du système n'étant pas ou peu couverts par les tests. Par conséquent, il nous semble qu'un travail plus poussé sur cette méthode d'évaluation pourrait être mené suivant deux axes :

- étudier comment les résultats de couverture de tests pourraient être exploités dans le but de renforcer la couverture d'un système ;
- étudier quels choix sont les plus adaptés pour construire une abstraction (choix des variables et du partionnement de leurs domaines) en fonction de l'utilisation que l'on souhaite en faire (mesure de complémentarité entre ensembles de tests et/ou identification de parties du système peu couvertes par les tests).

8.3 Publications associées à la thèse

Dans le cadre de nos travaux, nous avons réalisé six publications de recherche et deux publications concernant l'enseignement des méthodes formelles. Nous présentons un classement de ces publications en trois catégories :

- les publications présentant notre approche de génération de tests :
  - [JMTB09] – Revue ;

  - [JMT08b] – Conférence internationale ;
  - [DT09] – Workshop international.
- les publications plus spécifiquement orientées vers les cas d'études ;
  - [DdKT08] – Conférence internationale ;
  - [DPT08] – Conférence internationale ;
  - [JMT08a] – Workshop international.
- les publications concernant à l'enseignement.
  - [DT08] – Workshop international ;
  - [DJT08] – Workshop international.

Génération de tests à partir de schémas de test

Trois publications présentent différents aspects de nos travaux concernant la génération de tests à partir de schémas de test. L'article [JMT08b] présente le langage de description des schémas de test et la génération de tests à partir du modèle obtenu par produit synchronisé du modèle du système à valider et du schéma de test. Cet article a été étendu pour faire l'objet d'une publication en revue [JMTB09]. Les extensions réalisées concernent principalement :

- la présentation de la méthode d'évaluation de la couverture des tests sur une abstraction du modèle du système à valider ;
- l'application de cette méthode pour l'évaluation de la couverture des tests générés pour le cas d'étude IAS ;
- l'utilisation de ces résultats pour évaluer la complémentarité des ensembles de tests générés à partir des schémas et ceux générés à partir d'un critère de couverture structurelle des opérations du modèle.

Enfin, l'article [DT09] présente l'implémentation de nos travaux dans l'outil de génération de test jSynoPSys réalisé en exploitant les fonctionnalités de l'outil BZ-TT.

Cas d'études

Trois autres publications sont axées sur les différents cas d'études que nous avons traités. L'article court [JMT08a] présente les travaux réalisés autour de la validation de l'application IAS au cours du projet POSÉ. L'article [DPT08] présente également les travaux effectués sur IAS, mais le sujet présenté concerne principalement la formalisation de politiques de sécurité (utilisation de l'outil MECA), la formalisation des liens entre le modèle abstrait et le système concret, et la définition d'une relation de conformité. Enfin, l'article [DdKT08] présente l'application de nos travaux au cas d'étude POSIX qui concerne la validation d'un système de fichiers (c.f. 7.2).

Enseignement

De manière annexe, nous référençons deux publications dont le sujet porte sur l'enseignement. L'article [DT08] a pour sujet l'enseignement des méthodes formelles. Dans cet article, nous présentons, de manière générale, l'introduction des méthodes formelles à des étudiants de première année de master, au travers du langage B et de ses différentes utilisations. Nous présentons également le déroulement d'un projet portant sur la validation du porte-monnaie électronique Demoney, où les étudiants ont dû :

- modéliser l'application Demoney en langage B ;
- générer des tests avec l'outil LTG ;

– concrétiser et exécuter les tests sur plusieurs implémentations de l'application Demoney dans le but de détecter les fautes que nous y avions introduites.

L'article [DJT08] concerne également l'enseignement, mais le sujet est centré d'une part sur le langage B et d'autre part sur les transferts de connaissance entre recherche et enseignement.

Annexes

Chapitre 1

Porte-monnaie électronique "Demoney"

1.1 Spécification informelle

Le système considéré est inspiré de la spécification de *Demoney* (Demonstrative Electronic Purse), un porte-monnaie électronique développé par Trusted Logics à des fins de recherche. Aucune implantation de Demoney n'est embarquée sur une carte à puce utilisée au quotidien. Néanmoins, cette spécification comprend des propriétés et un mode de fonctionnement très réaliste qui lui donne tout son intérêt.

Demoney, comme tous les porte-monnaies, gère un solde qui évolue au gré des crédits/débits effectués sur la carte. Il est régi par deux codes PIN, l'un identifiant le porteur, l'autre identifiant la banque. Le solde du porte-monnaie est plafonné, tout comme le montant maximal d'un débit. Ce porte-monnaie fonctionne comme une carte à puce standard, notamment, il suit un *cycle de vie*. La vie de l'application commence par une phase de personnalisation permettant de fixer les codes PIN, et les plafonds autorisés de débit et du solde. Une fois la personnalisation correctement effectuée, la carte passe en phase d'utilisation.

Durant cette phase, l'utilisateur peut utiliser le porte-monnaie en lui-même, c'est-à-dire payer des achats d'un certain montant ou créditer son solde. L'opération de crédit nécessite au préalable une authentification de l'utilisateur par l'intermédiaire de son code PIN[16].

Lorsque l'utilisateur échoue toutes ses tentatives d'authentification, la carte est bloquée. Seule la banque peut la débloquer après s'être authentifiée avec son code PIN banque. Si la banque échoue toutes ses tentatives d'authentification, la carte est définitivement perdue. Si la carte est bloquée, la banque peut débloquer le code PIN utilisateur, en réinitialisant le code PIN utilisateur (compteur d'essais et valeur du PIN).

Similairement à toutes les applications carte à puce, toutes les commandes peuvent toujours être invoquées. Un code de retour (appelé *status word*) est renvoyé par la commande, indiquant si celle-ci a réussi (code 9000), ou si elle a échoué, précisant l'erreur qui s'est produite. Les codes d'erreur et le modèle de données de l'application sont détaillés dans la partie suivante.

[16] En effet, on suppose qu'un crédit du porte-monnaie implique le débit du compte bancaire associé à la carte –non modélisé ici– d'où la nécéssité d'une authentification.

1.2 Spécification technique

1.2.1 Le modèle de données

Les codes PIN sont des nombres à 4 digits, allant de 0000 à 9999, qui seront codés sur des entiers courts (`short`).

Les différents montants sont codés sur des entiers courts (`short`).

Les constantes sont codées sur des octets (`byte`).

Les status words sont codés sur des entiers (`int`).

1.2.2 Commandes de Demoney

Pour chaque commande nous donnons les valeurs possibles des paramètres et les status words donnés en réponse à l'exécution des commandes.

La commande PUT_DATA

Signature : `int PUT_DATA(byte p, short data)`

Cette commande permet de personnaliser le porte-monnaie. Elle ne peut être invoquée que lorsque la carte est en phase de personnalisation. Le paramètre `p` contient le type de paramétrage effectué, la sémantique du paramètre `data` en dépend.

- `SET_MAX_BALANCE` (`p` = 0) fixe comme solde maximal la valeur de `data`
- `SET_MAX_DEBIT` (`p` = 1) fixe comme débit maximal la valeur de `data`
- `SET_HOLDER_PIN` (`p` = 2) fixe le code pin utilisateur à la valeur de `data`. Le nombre d'essais associé à ce code est 3 (valeur fixe).
- `SET_BANK_PIN` (`p` = 3) fixe le code pin banque à la valeur de `data`. Le nombre d'essais associé à ce code est 4 (valeur fixe).

Status words retournés par cette commande :

- `9000` la commande a réussi.
- `9100` les valeurs des paramètres sont invalides (`p` diffère des 4 valeurs ci-dessus, `data` est strictement négatif ou alors `data` ne représente pas un code pin).
- `9101` la carte n'est pas dans le bon état du cycle de vie.

La commande STORE_DATA

Signature : `int STORE_DATA()`

Cette commande permet de terminer la personnalisation et de passer en phase d'utilisation. Elle ne peut être invoquée que si la personnalisation a été effectuée sur toutes les données et si elle a été correctement effectuée (i.e. si la personnalisation permet de respecter les propriétés données dans la partie 1.3).

Status words retournés par cette commande :

- `9000` la commande a réussi.
- `9101` la carte n'est pas dans le bon état du cycle de vie.
- `9401` tous les différents éléments personnalisables n'ont pas été personnalisés.
- `9402` la personnalisation n'a pas été correctement effectuée.

La commande VERIFY_PIN

Signature : `int VERIFY_PIN(byte p, short data)`

Cette commande permet de vérifier un des codes PIN du porte-monnaie. Elle ne peut être invoquée qu'en mode utilisation pour vérifier le code PIN du porteur ou lorsque la carte est bloquée pour vérifier le code PIN de la banque. Le paramètre `data` contient la proposition faite pour le code PIN. Le paramètre `p` peut avoir une des deux valeurs suivantes :

- `BANK` (`p` = 0) pour vérifier le code PIN banque
- `HOLDER` (`p` = 1) pour vérifier le code PIN utilisateur

Status words retournés par cette commande :

- `9000` la commande a réussi, la demande d'authentification est satisfaite.
- `9100` les valeurs des paramètres sont invalides.
- `9101` la carte n'est pas dans le bon état du cycle de vie.
- `9501` le code PIN proposé est incorrect, mais il reste encore des essais.
- `9502` le code PIN proposé (pour l'utilisateur) est incorrect, il n'y a plus d'essai, la carte est bloquée.
- `9503` le code PIN proposé (pour la banque) est incorrect, il n'y a plus d'essai, la carte est définitivement perdue.

La commande INITIALIZE_TRANSACTION

Signature : `int INITIALIZE_TRANSACTION(byte p, short data)`

Cette commande permet d'initialiser une transaction (qui sera ensuite à valider immédiatement par la commande COMMIT_TRANSACTION). Le type de la transaction, défini par la valeur de `p`, peut être soit un crédit (valeur `TRANSACTION_CREDIT` / `p` =0) soit un débit (valeur `TRANSACTION_DEBIT` / `p` =1). Le paramètre `data`, désignant le montant du crédit ou du débit, doit toujours être strictement positif, sinon il est invalide. Si une transaction de crédit est initiée, l'utilisateur devra être authentifié. Status words retournés par cette commande :

- `9000` la commande a réussi, la transaction est initiée.
- `9100` les valeurs des paramètres sont invalides.
- `9101` la carte n'est pas dans le bon état du cycle de vie.
- `9601` le contexte du crédit n'est pas correct (la somme à créditer trop importante).
- `9602` le contexte du débit n'est pas correct (somme à débiter trop importante).

La commande COMMIT_TRANSACTION

Signature : `int COMMIT_TRANSACTION()`

Cette commande permet de finaliser (effectuer concrètement) la transaction précédemment initiée. Si un crédit était initié, alors le solde est augmenté ; si un débit était initié, alors

le solde est diminué.

Status words retournés par cette commande :
- `9000` la commande a réussi, la transaction est effectuée.
- `9101` la carte n'est pas dans le bon état du cycle de vie.
- `9701` aucune transaction n'était initiée.

La commande PIN_CHANGE_UNBLOCK

Signature : `int PIN_CHANGE_UNBLOCK(short data)`

Cette commande permet, lorsque la carte est bloquée, de changer le code PIN utilisateur et de réinitialiser son nombre d'essais. Le paramètre `data` contient la valeur du nouveau code PIN utilisateur. Le contexte d'utilisation de cette commande requiert l'authentification préalable de la banque.

Status words retournés par cette commande :
- `9000` la commande a réussi, le code PIN utilisateur est réinitialisé avec sa nouvelle valeur.
- `9100` les valeurs des paramètres sont invalides.
- `9101` la carte n'est pas dans le cycle de vie correct.
- `9801` le contexte du changement de code PIN n'est pas valide.

Opérations d'observations

Voici les opérations d'observations admises par le système :
- `short getBalance()` retourne la valeur du solde (un solde non initialisé retourne -1)
- `short getMaxBalance()` retourne la valeur du solde maximal autorisé (retourne -1 si non initialisé).
- `short getMaxDebit()` retourne la valeur du débit maximal autorisé (retourne -1 si non initialisé).
- `boolean isPinAuthenticated(byte p)` indique si le PIN passé en paramètre (valeur `HOLDER` –`p` = 1– pour le PIN utilisateur ou `BANK` –`p` = 0– pour le PIN banque) est vérifié (retourne "false" si non initialisé).

1.2.3 Constantes

Les constantes utilisées comme valeurs pour les paramètres `p` des commandes décrites précédemment sont stockées dans une interface nommée `Constants`. Leurs noms et les valeurs sont identiques à ceux donnés dans les descriptions.

1.2.4 Vérifications inhérentes aux commandes

Les vérifications de la bonne invocation des commandes s'effectuent dans l'ordre suivant :

1. Valeurs des paramètres : les paramètres doivent avoir des valeurs admises par la commande.
2. Cycle de vie de la carte : la commande doit être invoquée dans un état spécifique de son cycle de vie.

3. Contexte des commandes : les variables d'état doivent avoir des valeurs correctes pour que la commande puisse s'exécuter.

En cas de doute, l'ordre donné pour les status words dans le sujet fait office de référence.

1.3 Propriétés de l'application

Voici une liste non-exhaustive des propriétés qui doivent être vérifiées par l'application.
- Le solde maximal du porte-monnaie est strictement positif.
- Le débit maximal du porte-monnaie est strictement positif.
- Le solde du porte-monnaie ne doit jamais être négatif, ni dépasser le plafond maximal.
- Une fois la carte personnalisée, les codes PIN doivent avoir des valeurs différentes.
- La banque et l'utilisateur ne peuvent pas être authentifiés en même temps.
- L'utilisateur ne peut être authentifié que lorsque la carte est en mode utilisation.
- La banque ne peut être authentifiée que lorsque la carte est bloquée.
- Lorsque la carte est bloquée, l'utilisateur n'est pas authentifié.
- Lorsque l'utilisateur n'a plus d'essais pour s'authentifier, la carte est bloquée.
- Lorsque la banque n'a plus d'essais pour s'authentifier, la carte est définitivement perdue.
- Toute initiation d'une transaction doit être immédiatement suivie d'une confirmation de la transaction, toute autre commande différente d'une observation, même erronée, annule la transaction.
- Toute authentification acquise n'est valable que pour une seule utilisation correcte.
- Si une authentification échoue (status words 95XX) toute authentification déjà acquise est perdue.

1.4 Modèle B

MACHINE dmoney

SETS

```
/*Etats de cycle de vie de la carte*/
ETAT_CARTE = {perso, use, invalid, dead};

/* STATUS_WORD */
sw ={sw_Success, /*9000*/
sw_Error_life_cycle, /*9101*/
sw_Error_parameter,/*9100*/
sw_Error_perso_not_correct,/*9402*/
sw_Error_perso_not_completed, /*9401*/
sw_Error_pin_value, /*9501*/
sw_Error_pin_value_no_more_try_user, /*9502*/
sw_Error_pin_value_no_more_try_bank, /*9503*/
sw_Error_invalid_credit,/*9601*/
sw_Error_invalid_debit,/*9602*/
sw_Error_invalid_transaction, /*9701*/
sw_Error_invalid_pin_value /*9801*/
}
```

CONSTANTS

```
  /* PIN */
  PIN_BANK,
  PIN_USER,
  PIN_NONE,

  /*Type de code pin (erroné, banque ou utilisateur)*/
  TYPE_PIN_WITH_NONE,
  /*Type de code pin (banque ou utilisateur)*/
  TYPE_PIN,

  /*Compteur d'essais d'autentification des PIN*/
  MAX_RETRY,

  /* Identifiants de données */
  SET_MAX_BALANCE,
  SET_MAX_DEBIT,
  SET_HOLDER_PIN,
  SET_BANK_PIN,

  /*Type de transaction (débit ou crédit)*/
  TYPE_TRANSACTION,
  TRANSACTION_CREDIT,
  TRANSACTION_DEBIT
```

PROPERTIES

```
  /* PIN */
  PIN_BANK = 0
& PIN_USER = 1
& PIN_NONE = 13
& TYPE_PIN = {PIN_BANK, PIN_USER}
& TYPE_PIN_WITH_NONE = TYPE_PIN \/ {PIN_NONE}

& MAX_RETRY : TYPE_PIN --> NAT1
& MAX_RETRY = {PIN_USER |-> 3, PIN_BANK |-> 4}

  /* PUT_DATA */
& SET_MAX_BALANCE    = 0
& SET_MAX_DEBIT      = 1
& SET_HOLDER_PIN     = 2
& SET_BANK_PIN       = 3

  /*INITIALIZE_TRANSACTION*/
& TRANSACTION_CREDIT = 0
& TRANSACTION_DEBIT = 1
& TYPE_TRANSACTION = {TRANSACTION_CREDIT, TRANSACTION_DEBIT}


DEFINITIONS
  SHORT == -32768..32767;
  BYTE == -128..127;
```

```
  PIN_SET == 0..9999
```

VARIABLES

```
  /* perso */
  max_solde,
  max_debit,

  /* use */
  etat_carte,
  solde,
  debit,
  pin_2_value,
  authenticated_pin,
  retry_counter
```

INVARIANT

```
  /* Typage */
  etat_carte : ETAT_CARTE
& max_solde : SHORT
& max_debit : SHORT
& solde : SHORT
& debit : SHORT
& pin_2_value : TYPE_PIN --> SHORT
& authenticated_pin : TYPE_PIN_WITH_NONE

& retry_counter : TYPE_PIN --> SHORT
& retry_counter(PIN_BANK) <= MAX_RETRY(PIN_BANK)
& retry_counter(PIN_USER) <= MAX_RETRY(PIN_USER)
& retry_counter(PIN_BANK) >= 0
& retry_counter(PIN_USER) >= 0

  /* Propriétés invariantes */
& (etat_carte /= use => debit = 0)


  /* propriétés valables uniquement sur la phase d'utilisation */
& (etat_carte /= perso => solde - debit <= max_solde)
& (etat_carte /= perso
    => pin_2_value(PIN_USER) /= pin_2_value(PIN_BANK))
& (etat_carte /= perso => max_debit <= max_solde)
& (etat_carte /= perso => debit <= solde)
& (etat_carte /= perso => debit <= max_debit)
& (etat_carte /= perso => max_debit>=0)
& (etat_carte /= perso => solde >= 0)
& (etat_carte /= perso => solde <= max_solde)
  /* propriétés sur la perso */
& (etat_carte = perso
    => (authenticated_pin = PIN_NONE & solde = -1 & debit = 0))

  /* propriétés de cycle de vie */
```

```
& (debit /= 0 => etat_carte = use)
&   ((retry_counter(PIN_BANK)=0) => retry_counter(PIN_USER)=0)
& ((retry_counter(PIN_BANK)=0) <=> (etat_carte = dead))
& ((retry_counter(PIN_USER)=0 & retry_counter(PIN_BANK)/=0)
    <=> (etat_carte = invalid))
& (authenticated_pin = PIN_USER => etat_carte = use)
& (authenticated_pin = PIN_BANK => etat_carte = invalid)

& (PIN_BANK = authenticated_pin
    => retry_counter(PIN_BANK) = MAX_RETRY(PIN_BANK))
& (PIN_USER = authenticated_pin
    => retry_counter(PIN_USER) = MAX_RETRY(PIN_USER))


INITIALISATION
  /* perso */
  max_solde := -1
||  max_debit := -1

  /* use */
||  etat_carte := perso
||  solde := -1
||  debit := 0
||  pin_2_value := {PIN_USER |-> -1, PIN_BANK |-> -1}
||  authenticated_pin := PIN_NONE
||  retry_counter := {PIN_USER |-> MAX_RETRY(PIN_USER),
            PIN_BANK |-> MAX_RETRY(PIN_BANK)}

OPERATIONS
  /*******************************/
  /* Commande de personnalisation */
  /*******************************/
  out <-- PUT_DATA(type_data, data) =
  PRE
    /*Identifiant de la donnée initialisée*/
    type_data : BYTE
    /*Valeurs affectée*/
  & data : SHORT
  THEN
    IF type_data = SET_MAX_BALANCE & data >= 0 THEN
      /*Initialisation du solde maximum*/
      IF etat_carte = perso THEN
        /*Etat du cycle de vie cohérent*/
        max_solde := data
        ||out := sw_Success
      ELSE
        /*Etat de cycle de vie incohérent*/
        out := sw_Error_life_cycle
      END

    ELSE
      IF type_data = SET_MAX_DEBIT & data >= 0 THEN
```

```
        /*Initialisation du débit maximum*/
        IF etat_carte = perso THEN
          /*Etat du cycle de vie cohérent*/
          max_debit := data
          ||out := sw_Success
        ELSE
          /*Etat de cycle de vie incohérent*/
          out := sw_Error_life_cycle
        END
      ELSE
        IF type_data = SET_HOLDER_PIN & data : PIN_SET THEN
          /*Initialisation du code PIN utilisateur*/
          IF etat_carte = perso THEN
            /*Etat du cycle de vie cohérent*/
            pin_2_value(PIN_USER) := data
            || out := sw_Success
          ELSE
            /*Etat de cycle de vie incohérent*/
            out := sw_Error_life_cycle
          END
        ELSE
          IF type_data = SET_BANK_PIN & data : PIN_SET THEN
            /*Initialisation du code PIN banque*/
            IF etat_carte = perso THEN
              /*Etat du cycle de vie cohérent*/
              pin_2_value(PIN_BANK) := data
              || out := sw_Success
            ELSE
              /*Etat de cycle de vie incohérent*/
              out := sw_Error_life_cycle
            END
          ELSE
            /*Valeurs des paramètres hors de leurs domaines*/
            out := sw_Error_parameter
          END
        END
      END
    END
    /*Réinitialisation de la transaction*/
    || debit := 0
END;


/************************************/
/* Commande de fin de personalisation */
/************************************/
out <-- STORE_DATA =
BEGIN
  IF etat_carte = perso THEN
  /*Etat de cycle de vie cohérent*/
    IF (max_solde >= 0) & (max_debit >= 0) &
       (pin_2_value(PIN_USER)>=0) & (pin_2_value(PIN_BANK)>=0)
    THEN
```

```
        /*Perso complète*/
        IF (pin_2_value(PIN_USER) /= pin_2_value(PIN_BANK))
           & (max_solde >=  max_debit)
        THEN
          /*Perso cohérente*/
          etat_carte := use
          ||solde := 0
          ||out := sw_Success
        ELSE
          /*Perso incohérente*/
          out := sw_Error_perso_not_correct
        END
      ELSE
        /*Perso incomplète*/
        out := sw_Error_perso_not_completed
      END
    ELSE
      /*Etat de cycle de vie incohérent*/
      out := sw_Error_life_cycle
    END
    /*Réinitialisation de la transaction en cours*/
    || debit := 0
END;


/*****************************/
/* Commande d'authentification */
/*****************************/
out <--VERIFY_PIN(pin, value) =
PRE
   /*identifiant du PIN (utilisateur, banque ou erroné)*/
   pin : BYTE
   /*code PIN*/
& value : SHORT
THEN
   IF pin : TYPE_PIN & value : PIN_SET THEN
     /* Valeurs des paramètres dans leurs domaines de définition*/
     IF (etat_carte = use & pin = PIN_USER)
     or (etat_carte = invalid & pin = PIN_BANK) THEN
       /*Etat du cycle de vie de la carte cohérent*/
       IF pin_2_value(pin) = value THEN
         /*Code pin correct*/
         authenticated_pin := pin
         || retry_counter(pin) := MAX_RETRY(pin)
         || out := sw_Success
       ELSE
         /*Code pin erroné*/
         IF retry_counter(pin) <= 1 THEN
           /*Compteur d'essais épuisé*/
           IF pin = PIN_USER THEN
             etat_carte := invalid
             || out := sw_Error_pin_value_no_more_try_user
           ELSE
```

```
            etat_carte := dead
            || out := sw_Error_pin_value_no_more_try_bank
          END
          || retry_counter(pin) := 0
        ELSE
          /*Compteur d'essais décrémenté*/
          retry_counter(pin) := retry_counter(pin) - 1
          || out := sw_Error_pin_value
        END
        /* A VIRER ? erreur*/
        || authenticated_pin := PIN_NONE
      END
    ELSE
      /*Etat du cycle de vie de la carte incohérent*/
      out := sw_Error_life_cycle
    END
  ELSE
    /*Valeurs des paramètres hors de leurs domaines de définition*/
    out := sw_Error_parameter
  END
  /*Réinitialisation de la transaction en cours*/
  || debit := 0
END;


/*****************************************/
/* Commande d'initialisation de transaction */
/*****************************************/
out <-- INITIALIZE_TRANSACTION(transaction, data) =
PRE
  /*Type de transaction (débit, crédit, ou erroné)*/
  transaction : BYTE
  /*Montant de la transaction*/
& data : SHORT
THEN
  IF data > 0 & transaction : TYPE_TRANSACTION THEN
    /*Paramètres dans leurs domaines de définition*/
    IF etat_carte = use THEN
      /*Etat du cycle de vie de la carte cohérent*/
      IF transaction = TRANSACTION_CREDIT THEN
        /*Credit*/
        IF data + solde <= max_solde
           & authenticated_pin = PIN_USER
        THEN
          /*contexte et paramètres correctes*/
          debit := -data
          || authenticated_pin := PIN_NONE
          || out := sw_Success
        ELSE
          /*contexte ou paramètres incorrectes*/
          out := sw_Error_invalid_credit
        END
      ELSE
```

```
        /*Debit*/
        IF data <= solde & data <= max_debit THEN
          /*contexte et paramètres correctes*/
          debit := data
          || out := sw_Success
        ELSE
          /*contexte ou paramètres incorrectes*/
          out := sw_Error_invalid_debit
        END
      END
    ELSE
      /*Etat du cycle de vie de la carte incohérent*/
      out := sw_Error_life_cycle
    END
  ELSE
    /*Valeurs des paramètres en dehors de leurs domaines*/
    out := sw_Error_parameter
  END

END;


/*****************************************/
/* Commande de finalisation de transaction */
/*****************************************/
out <-- COMMIT_TRANSACTION =
BEGIN
  IF etat_carte = use THEN
  /*Etat du cycle de vie de la carte cohérent*/
    IF debit /= 0 THEN
      /*Transaction initiée*/
      solde := solde - debit
      || out := sw_Success
    ELSE
      /*Aucune transaction inititiée*/
      out := sw_Error_invalid_transaction
    END
  ELSE
    /*Etat de cycle de vie de la carte incohérent*/
    out := sw_Error_life_cycle
  END
  /*Réinitialisation de la transaction en cours*/
  || debit := 0
END;


/*************************************************************/
/*Commande de déblocage et modification du code PIN utilisateur */
/*************************************************************/
out <-- PIN_CHANGE_UNBLOCK(data) =
PRE
  /*nouvelle valeur pour le code pin utilisateur*/
  data : SHORT
```

```
  THEN
    IF data : PIN_SET THEN
    /*Valeurs des paramètres dans leurs domaines de définition*/
      IF data /= pin_2_value(PIN_BANK) THEN
        /*Nouveau code PIN utilisateur différent du code PIN banque*/
        IF etat_carte = invalid THEN
          /*Etat de cycle de vie cohérent*/
          IF authenticated_pin = PIN_BANK THEN
            /*Etat d'authentification correcte*/
            pin_2_value(PIN_USER) := data
            ||retry_counter(PIN_USER) := MAX_RETRY(PIN_USER)
            ||etat_carte := use
            ||out := sw_Success
          ELSE
            /*Contexte de changement de code pin invalide*/
            out := sw_Error_invalid_pin_value
          END
        ELSE
          /*Etat de cycle de vie de la carte incohérent*/
          out := sw_Error_life_cycle
        END
      ELSE
        /*Contraintes non respectées sur les valeurs des paramètres*/
        out := sw_Error_parameter
      END
    ELSE
      /*Valeurs des paramètres hors de leurs domaines de définition*/
      out := sw_Error_parameter
    END
    /*Réinitialisation de la transaction en cours et de l'authentification*/
    ||debit := 0
    || authenticated_pin := PIN_NONE
  END;


  /*******************************/
  /* Commande de reset de la carte */
  /*******************************/
  RESET =
  BEGIN
    debit := 0
    ||authenticated_pin := PIN_NONE
  END
/*************************************************************************/
/**************** Commandes d'observation ****************************/
/*************************************************************************/

  out <-- GET_BALANCE =
  BEGIN
    out := solde
  END;
```

```
  out <-- GET_MAX_BALANCE =
  BEGIN
    out := max_solde
  END;


  out <-- GET_MAX_DEBIT =
  BEGIN
    out := max_debit
  END;


  out <-- IS_PIN_AUTHENTICATED(pin) =
  PRE
    pin : TYPE_PIN
  THEN
    IF pin = authenticated_pin THEN
      out := TRUE
    ELSE
      out := FALSE
    END
  END;


END
```

Table des figures

Liste des tableaux

Bibliographie

[Abr96] J. R. Abrial. *The B-book : assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[AWW08] Bernhard K. Aichernig, Martin Weiglhofer, and Franz Wotawa. Improving fault-based conformance testing. *Electronic Notes in Theoretical Computer Science*, 220(1) :63 – 77, 2008. Proceedings of the Fourth Workshop on Model Based Testing (MBT 2008).

[BBM02] Francesca Basanieri, Antonia Bertolino, and Eda Marchetti. The cow_suite approach to planning and deriving test suites in UML projects. In *UML'02 : Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 383–397, London, UK, 2002. Springer-Verlag.

[BDL06] F. Bouquet, F. Dadeau, and B. Legeard. Automated boundary test generation from JML specifications. In *FM'06, 14th Int. Conf. on Formal Methods*, volume 4085 of *LNCS*, pages 428–443, Hamilton, Canada, August 2006. Springer-Verlag.

[BDLN06] F. Bouquet, S. Debricon, B. Legeard, and J.-D. Nicolet. Extending the unified process with model-based testing. In *MoDeVa'06, 3rd Int. Workshop on Model Development, Validation and Verification*, pages 2–15, Genova, Italy, October 2006.

[Ber03] A. Bertolino. Software testing research and practice. In *Abstract State Machines 2003*, pages 1–21, Taormina , ITALY, March 2003. Springer Berlin / Heidelberg.

[BFS05] A. F. E. Belinfante, L. Frantzen, and C. Schallhart. Tools for test case generation. In M. Broy, B. Jonsson, J. P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems : Advanced Lectures*, volume 3472 of *Lecture Notes in Computer Science*, pages 391–438. Springer Verlag, 2005.

[BGN+03] Mike Barnett, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Model-based testing with AsmL.NET. In *1st European Conference on Model-Driven Software Engineering*, pages 12–19, December 2003.

[BGN+04] M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Towards a tool environment for model-based testing with AsmL. In Springer Berlin / Heidelberg, editor, *FATES'03, Formal Approaches to Software Testing*, volume 2931/2004, pages 1102–1103, 2004.

[BLPT04] F. Bouquet, B. Legeard, F. Peureux, and E. Torreborre. Mastering Test Generation from Smart Card Software Formal Models. In *Procs. of the Int. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices*

(CASSIS'04), volume 3362 of *LNCS*, pages 70–85, Marseille, France, March 2004. Springer. Selected papers from the CASSIS'04 workshop.

[BMM05] A. Bertolino, E. Marchetti, and H. Muccini. Introducing a reasonably complete and coherent approach for model-based testing. *Electronic Notes in Theoretical Computer Science*, 116 :85–97, January 2005. Proceedings of the International Workshop on Test and Analysis of Component Based Systems (TACoS 2004).

[CLP04] S. Colin, B. Legeard, and F. Peureux. Preamble computation in automated test case generation using Constraint Logic Programming. *The Journal of Software Testing, Verification and Reliability*, 14(3) :213–235, 2004. Selected papers from the 2003 UK-Test Workshop.

[Dad06] F. Dadeau. *Évaluation symbolique à contraintes pour la validation - Application à Java/JML*. Thèse de Doctorat, LIFC, Université de Franche-Comté, 19 juillet 2006.

[DdKT08] F. Dadeau, A. de Kermadec, and R. Tissot. Combining scenario and model-based testing to ensure POSIX compliance. In *ABZ'2008, Int. Conf. on ASM, B and Z*, volume 5238 of *LNCS*, pages 153–166, London, UK, September 2008. Springer.

[DF93] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *FME '93 : Proceedings of the First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, pages 268–284, London, UK, 1993. Springer-Verlag.

[DHM07] F. Dadeau, A. Haddad, and T. Moutet. Test fonctionnel de conformité vis-à-vis d'une politique de contrôle d'accès. In *AFADL'07, Approches Formelles dans l'Assistance au Développement de Logiciels*, Namur, Belgique, 2007.

[DJT08] F. Dadeau, J. Julliand, and R. Tissot. Leirios test generator : from research to teaching, through industry. In *Int. Workshop on the B Method : from Research to Teaching*, pages 1–16, Nantes, France, June 2008. APCB.

[DPT08] F. Dadeau, M.-L. Potet, and R. Tissot. A B formal framework for security developments in the domain of smart card applications. In *SEC'2008, 23rd int. Information Security Conference*, volume 278 of *IFIP*, pages 141–155, Milano, Italy, September 2008. Springer.

[DT08] F. Dadeau and R. Tissot. Teaching model-based testing with Leirios Test Generator. In *FORMED'08, Int. Workshop on Formal Methods in Computer Science Education, co-located with ETAPS'2008)*, pages 129–138, Budapest, Hungary, March 2008.

[DT09] Frédéric Dadeau and Régis Tissot. jSynoPSys - a scenario-based testing tool based on the symbolic animation of B machines. *Electronic Notes in Theoretical Computer Science*, 253(2) :117–132, 2009. Proceedings of Fifth Workshop on Model Based Testing (MBT 2009).

[dVT01] R. G. de Vries and J. Tretmans. Towards formal test purposes. In G. J. Tretmans and H. Brinksma, editors, *Formal Approaches to Testing of Software 2001 (FATES'01), Aarhus, Denmark*, volume NS-01-4 of *BRICS Notes Series*, pages 61–76, Aarhus, Denkmark, August 2001.

[FBCO+09] M. Felderer, R. Breu, J. Chimiak-Opoka, M. Breu, and F. Schupp. Concepts for model-based requirements testing of service oriented systems. In *Proceedings of the IASTED Software Engineering*, pages 152–157, 2009.

[FGG07] Alain Faivre, Christophe Gaston, and Pascale Le Gall. Symbolic model based testing for component oriented systems. In *TestCom/FATES*, pages 90–106, 2007.

[FTW05] Lars Frantzen, Jan Tretmans, and Tim A. Willemse. Test generation based on symbolic specifications. In *FATES'04, Formal Approaches to Software Testing*, volume 33-95 of *LNCS*, pages 1–15, 2005.

[GGRT06] Christophe Gaston, Pascale Le Gall, Nicolas Rapin, and Assia Touil. Symbolic execution techniques for test purpose definition. In *TestCom*, pages 1–18, 2006.

[GIX04] GIXEL. *Plateforme commune pour l'eAdministration*, spécification technique, IAS 2.0 edition, 2004.

[Had07] Amal Haddad. Meca : A tool for access control models. In J. Julliand and O. Kouchnarenko, editors, *7th International Conference of B Users (B'2007), Besançon, France, January 17-19, 2007, Proceedings*, volume 4355 of *LNCS*, pages 281–284. Springer, 2007.

[HM07] A. Haddad and T. Moutet. Modélisation et vérification de contrôle d'accès selon une démarche critères communs. In *International conference on risks and security of internet and systems (CRISIS 2007), Marrakech, Maroc, July 2007, Proceedings*, pages 117–124, 2007.

[JH07] R. Joshi and G. Holzmann. A mini challenge : build a verifiable filesystem. *Formal Aspects of Computing*, 19(2) :269–272, June 2007.

[JJ05] Claude Jard and Thierry Jéron. TGV : theory, principles and algorithms : A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf.*, 7(4) :297–315, 2005.

[JJRZ05] B. Jeannet, T. Jéron, V. Rusu, and E. Zinovieva. Symbolic test selection based on approximate analysis. In *Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'05*, volume 3440 of *LNCS*, April 2005.

[JMT08a] J. Julliand, P.-A. Masson, and R. Tissot. Generating security tests in addition to functional tests. In *AST'08, 3rd Int. workshop on Automation of Software Test*, pages 41–44, Leipzig, Germany, May 2008. ACM Press.

[JMT08b] J. Julliand, P.-A. Masson, and R. Tissot. Generating tests from B specifications and test purposes. In *ABZ'2008, Int. Conf. on ASM, B and Z*, volume 5238 of *LNCS*, pages 139–152, London, UK, September 2008. Springer.

[JMTB09] J. Julliand, P.-A. Masson, R. Tissot, and P.-C. Bué. Generating tests from B specifications and dynamic selection criteria. *FAC, Formal Aspects of Computing*, 2009. Accepted Manuscript. Revised and extended version of a paper from the ABZ'08 conference. To appear.

[LB08] Michael Leuschel and Michael Butler. ProB : an automated analysis toolset for the B method. *Software Tools for Technology Transfer*, 10(2) :185–203, 2008.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Jml : A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, chapter 12, pages 175–188. Kluwer, 1999.

[LC06] G. Leavens and Y. Cheon. Design by contract with JML. http ://jmls-pec.org/jmldbc.pdf, 2006.

[LdBMP04] Y. Ledru, L. du Bousquet, O. Maury, and Bontron P. Filtering tobias combinatorial test suites. In *Proceedings of ETAPS/FASE'04 - Fundamental Approaches to Software Engineering*, volume 2984 of *LNCS*, pages 281–294, Barcelona, 2004. Springer-Verlag.

[LPU02] B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from Z and B. In *Proc. of the Int. Conf. on Formal Methods Europe, FME'02*, volume 2391 of *LNCS*, pages 21–40, Copenhaguen, Denmark, July 2002. Springer.

[LPU04] B. Legeard, F. Peureux, and M. Utting. Controlling test case explosion in test generation from B formal models. *Software Testing, Verification and Reliability, STVR*, 14(2) :81–103, 2004.

[MDJ08] H. Marchand, J. Dubreil, and T. Jéron. Automatic test generation for security property. Delivrable, Politess Project, 2008.

[MLdB03] O. Maury, Y. Ledru, and L. du Bousquet. Intégration de TOBIAS et UCASTING pour la génération de tests. In *16th International Conference Software and Systems and their applications-ICSSEA*, Paris, 2003.

[MRS+97] Jean R. Moonen, Judi M.T. Romijn, Olaf Sies, Jan G. Springintveld, Loe G.M. Feijs, and Ronald L.C. Koymans. A two-level approach to automated conformance testing of VHDL designs. Technical Report SEN-R9707, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, The Netherlands, 1997.

[Ori05] Catherine Oriat. Jartege : A Tool for Random Generation of Unit Tests for Java Classes. In *2nd International Workshop on Software Quality (SOQUA 2005)*, volume 3712 of *LNCS*, pages 242–256, Erfurt, Germany, September 2005. Springer.

[PP04] Alexander Pretschner and Jan Philipps. Methodological issues in model-based testing. In *Model-Based Testing of Reactive Systems*, pages 281–291, 2004.

[PTJ07] M.-L. Potet, R. Tissot, and E. Jaffuel. Etude des relations de raffinement : modèles de sécurité et conformité d'une application. Livrable L2.5, Projet RNTL POSE, 2007.

[SLB05] Manoranjan Satpathy, Michael Leuschel, and Michael Butler. ProTest : An Automatic Test Environment for B Specifications. *Electronic Notes in Theroretical Computer Science*, (111) :113–136, January 2005.

[Sto07] Nicolas Stouls. *Systèmes de transitions symboliques et hiérarchiques pour la conception et la validation de modèles B raffinés.* Thèse de doctorat, Institut Polytechnique de Grenoble, 2007.

[TB03] G. J. Tretmans and H. Brinksma. Torx : Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering, Nuremberg, Germany*, pages 31–43, December 2003.

[Tou06] Assia Touil. *Exécution Symbolique pour le test de conformité et le test de raffinement.* Thèse de doctorat, laboratoire IBISC, Université d'Évry-Val d'Essonne, 2006.

[Tre96] G. J. Tretmans. Test generation with inputs, outputs, and quiescence. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS '96, Passau, Germany*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146, Berlin, 1996. Springer Verlag.

[TS05] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. *SIGSOFT Softw. Eng. Notes*, 30(5) :253–262, 2005.

[UL06] M. Utting and B. Legeard. *Practical Model-Based Testing - A tools approach.* Elsevier Science, 2006. 550 pages, ISBN 0-12-372501-1.

[VA06] Lionel Van Aertryck. Spécification informelle de la politique de sécurité - cible de sécurité pour la plateforme IAS. Livrable L5.2, Projet RNTL POSE, 2006. Silicomp/AQL.

[VABM97] Lionel Van Aertryck, Marc Benveniste, and Daniel Le Métayer. CASTING : A formally based software test generation method. In *ICFEM'97 : Proceedings of the 1st International Conference on Formal Engineering Methods*, page 101, Washington, DC, USA, 1997. IEEE Computer Society.

[VCG+08] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Nikolai Schulte, Wolfram Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with Spec Explorer. In *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 39–76. Springer, 2008.

Résumé

Cette thèse est une contribution à la conception d'une méthode de génération automatique de tests à partir de modèles (ou MBT — Model Based Testing). Le contexte de nos travaux se situe dans la continuité de ceux effectués autour de l'outil BZ-TT (BZ-Testing Tools) qui permet d'engendrer des tests fonctionnels à partir de modèles décrits en langage B. Les critères de sélection de tests implémentés dans l'outil BZ-TT reposent sur la couverture structurelle du modèle du système à valider en prenant en compte les structures de données et de contrôle de celui-ci.

Cette approche ne permet pas de générer de tests à partir de propriétés liées au comportement dynamique du système, par exemple en tenant compte de propriétés basées sur des enchaînements d'opérations. Afin de répondre à cette problèmatique, un certain nombre de travaux proposent des méthodes où l'expertise humaine est exploitée afin de définir des critères de sélection de tests "dynamiques". De tels critères de sélection permettent, à l'ingénieur validation, de définir des stratégies basées sur des propriétés et des aspects du système qu'il souhaite valider. Nos contributions s'inscrivent dans cette voie, tout en visant la complémentarité par rapport à la génération automatique de tests par couverture structurelle du modèle dans un objectif de valorisation des technologies et ressources déployées à cette fin.

Notre première contribution est la définition d'un langage de formalisation d'objectifs de tests qui permet d'exprimer des ensembles de scénarios de tests inspirés de propriétés à valider sur le système. Ce langage permet de décrire des schémas de tests à partir d'un formalisme, basé sur celui des expressions régulières, qui permet de décrire des ensembles de scénarios principalement par des enchaînements d'appels d'opération et d'états symboliques.

Nous définissons une méthode de génération de tests intégrée à l'outil BZ-TT, afin que celui-ci prenne en compte ce nouveau critère de sélection de tests. Cette méthode permet de réutiliser les technologies d'animation symbolique et de résolution de contraintes de cet outil, ainsi que de conserver les fonctionnalités d'exportation et de concrétisation des tests produits. Dans cette méthode, la seule charge supplémentaire pour l'ingénieur de validation est la définition des schémas de test utilisés comme critère de sélection.

Nos dernières contributions, visent à évaluer la complémentarité de notre méthode avec celle de génération automatique de tests par couverture structurelle du modèle. Nous proposons une méthode d'évaluation de la complémentarité entre deux suites de tests. Cette méthode est basée sur le calcul de la couverture d'états et de transitions des suites de tests sur une abstraction du système. Enfin, nous appliquons cette méthode à trois études de cas (deux applications de type carte à puce et un système de gestion de fichiers Posix), et nous montrons la complémentarité qu'elle apporte.

Mots-clés: Model Based Testing, langage B, génération de tests, critères de sélection de tests