Génération de tests à partir de modèle UML/OCL pour les systèmes critiques évolutifs

Elizabeta Fourneret

SPIM

Thèse de Doctorat

école doctorale sciences pour l'ingénieur et microtechniques

UFC UNIVERSITÉ DE FRANCHE-COMTÉ

Génération de tests à partir de modèles UML/OCL pour les systèmes critiques évolutifs

THÈSE

Soutenance le 5 Décembre 2012

pour l'obtention du grade de

Docteur de l'Université de Franche-Comté

(Spécialité Informatique)

par

Elizabeta FOURNERET

Composition du jury

Président :	Bruno Legeard, Professeur, Université de Franche-Comté
Directeur :	Fabrice Bouquet, Professeur, Université de Franche-Comté
Rapporteurs :	Lionel Briand, Professeur, Université du Luxembourg Ioannis Parissis, Professeur, Institut Polytechnique de Grenoble

Département d'Informatique des Systèmes Complexes femto-st SCIENCES & TECHNOLOGIES

Remerciements

Je tiens en premier lieu à remercier mon directeur de thèse, Fabrice Bouquet, pour son encadrement et ses conseils, pour le temps consacré à m'accompagner et son soutien constant durant ces trois années. C'était un défi de choisir une petite macédonienne comme thésarde, mais je pense que cela n'a pas été un échec. J'ai passé trois ans de thèse dans un projet d'envergure internationale, rempli de rencontres et de résultats intéressants.

Je remercie chaleureusement Lionel Briand et Ioannis Parissis, mes rapporteurs, d'avoir accepté de se plonger dans le manuscrit, pour comprendre et évaluer mes travaux. Je remercie vivement Bruno Legeard d'avoir accepté le rôle de Président et d'avoir examiné mes travaux. Je tiens également à remercier Jean-Christophe Lapayre et Olga Kouchnarenko de m'avoir bien accueillie au sein du LIFC, puis du DISC.

Je remercie Frédéric Dadeau, pour ses idées fructueuses et ses conseils précieux. Je tiens à le remercier de m'avoir ouvert les portes de l'école internationale de test TAROT et de m'avoir associée à l'organisation de l'édition 2012 au sein d'une équipe efficace et conviviale. "Frédéric, travailler avec toi a été un plaisir. Tout est facile et tout est toujours fini à la perfection. Je te souhaite beaucoup de réussite dans ta vie professionnelle future, tu le mérites !"

Je remercie Bruno Legeard pour sa collaboration et son suivi dans le cadre du projet SecureChange, ainsi que d'avoir apporté sa grande expertise dans la recherche scientifique appliquée dans le monde industriel. Je remercie également Julien Botella pour les heures de travail consacrées au développement du prototype et pour la bonne ambiance des déplacements durant le projet SecureChange, à travers toute l'Europe.

Je tiens à remercier Anne Bouquet pour la relecture complète de mon manuscrit de thèse. Je remercie également Fabien Peureux, d'avoir apporté son savoir et sa maîtrise de la langue française lors de la relecture de mon manuscrit. "Fabien, tu as le don d'écouter les personnes et de les conseiller, toujours en communion avec la Force, tel maître Yoda."

Merci à Jacques Julliand, Frédéric Dadeau, Fabrice Bouquet et Fabrice Ambert, Pierre-Alain Masson, Hassan Mountassir et Anne Heam, de m'avoir fait confiance et de m'avoir offert la possibilité d'effectuer des enseignements. Grâce à vous, j'ai pu transmettre mes connaissances aux étudiants. C'est dans ces moments qu'on se rend compte que le rôle d'enseignant n'est pas simple. L'enseignant porte sur ses épaules la responsabilité de transmettre et de faire passer son savoir. J'exprime ma gratitude envers tous les enseignants que j'ai eu tout au long de mes études à l'université pour le savoir qu'ils m'ont apporté.

Je souhaite apporter un affection particulière à Brigitte et Emmanuelle, sans qui l'administration serait un vrai puits noir, et à Laurent, toujours à notre secours quand les ordinateurs rendent l'âme.

Je ne remercierai jamais assez John - le canadien -, Stéphane, Kalou, PC et Jérôme de

m'avoir accueillie et permis de m'intégrer dans le monde des doctorants au laboratoire. Merci pour votre amitié. Merci à mes co-buraux : Julien, Thomas et Ivan d'avoir supporté mes hauts et mes bas pendant la rédaction, merci pour les traductions et synonymes lorsque j'en avais besoin : oui, le macédonien surprend toujours. Merci à tous mes amis et collègues : Elena ("la Russie est splendide, merci pour ton aide et le cours accéléré de Russe"), Seb, Rami, Aydée, Oscar, Laloi, Régis, Sékou, Jean-Baptiste et tous ceux que j'oublie, merci pour vos discussions, votre aide, votre soutien et votre jovialité ... Éléments vitaux qui mènent vers une thèse réussie. J'espère que nous pourrons fêter d'autres réussites ensemble.

La fin de cette thèse ferme l'anneau de mes études supérieures, pour lesquelles je ne pourrai jamais cesser de dire merci à mes parents, Ratko et Silvana et à ma soeur, Hristina. D'avoir eu le courage de me laisser partir à 1964km de mon pays, où j'ai été accueilli les bras grands ouverts par deux personnes merveilleuses et uniques Monique et Gilbert. Je ne pourrai jamais assez vous remercier pour tout l'amour parental, pour tous les conseils et soutiens que vous m'avez apportés, comme si vous étiez mes parents. J'exprime mes affections vers ma formidable famille et belle-famille, qui ont fait l'impossible pour me soutenir inconditionnellement pendant ma thèse. Mais tout cela, n'aurait pas été possible sans ma professeur de français et amie Dobrila. "Je te remercie de m'avoir lancée dans les défis de la langue française, de me l'avoir enseignée ; maintenant j'ai appris un peu plus que de chanter l'alphabet.

Je souhaite profiter de cette occasion et adresser ma reconnaissance à l'Institut Français de Skopje, et plus particulièrement à Suzana Pesic, pour son aide et ses conseils concernant ma venue en France. Par cette même occasion, j'exprime ma gratitude à tout le personnel de la bibliothèque Tane Gerogievski de Kumanovo, notamment la directrice Nada Ivanovska, vos livres ont étés mon inspiration.

Je tiens à ajouter une pensée affectueuse pour mes amis de Macédoine : Dani, Diki, Eki, Frosi, Kate, Chaki, Dasha, Tanja et Maja, et de France : Solaine, Arnaud, Amélie, Tof, Cé, Chris, Guigui, Laeti, Dooley et Christelle, qui ont toujours été à mes côtés et ont contribué à ce que je n'oublie pas le macédonien, tout en continuant d'améliorer mon français.

Enfin, avec ses dernières lignes, d'une grande importance, je remercie mon cher et tendre époux, de m'avoir supportée, rassurée et guidée tout au long de ces années de thèse. Jacques André Victor Fourneret, je t'aime.

Кога сакаш нешто, сакај го толку многу,
што дури и Вселената ќе скова заговор
да се оствари тоа што го сакаш.

инспирирано од Пауло Коељо

Table des matières

Partie I Contexte et problématiques **1**

Chapitre 1 Introduction **3**

1.1 Contexte 4

1.1.1 Model-Based Testing 4

1.1.2 Projet *SecureChange* 5

1.2 Problématiques 8

1.2.1 Gestion de l'évolution du système 8

1.2.2 Traitement de la problématique de la sécurité 9

1.3 Contributions 10

1.3.1 Génération sélective de tests 10

1.3.2 Tests liés à la sécurité dans les systèmes évolutifs 11

1.4 Plan du mémoire 12

Chapitre 2 Test à partir de modèles **15**

2.1 Le test fonctionnel à partir de modèles 17

2.1.1 Formalismes de modélisation 18

2.1.2 Critères de sélection 19

2.2 Le test de sécurité 22

2.2.1 Formalismes pour l'expression des exigences de sécurité 23

2.2.2 Techniques de test de sécurité 25

2.3 Outillage Smartesting pour la génération de tests 27
2.3.1 Diagrammes UML4MBT 27
2.3.2 Traçabilité des exigences 29
2.3.3 La prise en compte des exigences de sécurité 29
2.4 Synthèse 31

Chapitre 3 Test de non-régression **33**

3.1 Classification des techniques de non-régression 34
3.1.1 La classification générale 34
3.1.2 Classification des techniques sélectives 35
3.2 Panorama des techniques sélectives 35
3.2.1 À partir de modèles 36
3.2.2 Cadre d'évaluation des techniques 37
3.3 La prise en compte de l'aspect sécurité 38
3.4 Synthèse 39

Chapitre 4 Exemple fil rouge **41**

4.1 Introduction à l'application eCinema 42
4.1.1 Exigences fonctionnelles 42
4.1.2 Exigences de sécurité pour eCinema 45
4.2 Génération de tests pour eCinema 47
4.2.1 Tests fonctionnels 47
4.2.2 Tests dédiés à la sécurité 49
4.3 Évolution de la spécification d'eCinema 51
4.3.1 Description de l'évolution 51
4.3.2 Évolution de modèle 52
4.4 Synthèse 53

Partie II Contributions **55**

Chapitre 5 Le cycle de vie et suites de tests **57**

5.1 L'évolution du cycle de vie des tests 59

5.2 La gestion des suites de tests 61

5.2.1 Suite de tests 61

5.2.2 Composition des suites de tests 62

5.3 Gestion du référentiel de tests 63

5.4 Synthèse 64

Chapitre 6 SeTGaM pour les diagrammes d'états-transitions **65**

6.1 Différences dans un diagramme d'états/transitions 66

6.2 Calcul des dépendances 69

6.2.1 Dépendances de données 70

6.2.2 Dépendances de contrôle 72

6.3 Analyse des exigences impactées par l'évolution 76

6.3.1 Calcul des impacts 76

6.3.2 Règles de catégorisation des tests 78

6.4 Synthèse 80

Chapitre 7 SeTGaM pour des comportements UML/OCL **81**

7.1 Définition des comportements en UML/OCL 82

7.2 Différences des comportements dans les opérations 83

7.3 Dépendances entre comportements 85

7.4 Analyse des impacts des comportements 88

7.4.1 Calcul des comportements impactés 88

7.4.2 La catégorisation de tests à partir des comportements 89

7.5 Synthèse 91

Chapitre 8 SeTGaM pour le test de sécurité **93**

8.1 Comparaison des Spécifications des Cas de Tests 94
8.2 Statuts des tests et des suites de tests pour la prise en compte des TCS 96
8.2.1 Statuts des suites de tests de sécurité 96
8.2.2 Composition des suites de tests de sécurité 97
8.3 Génération sélective de tests à partir des TCS 98
8.3.1 Règles de catégorisation des tests 98
8.3.2 Extension du processus pour les TCS 100
8.4 Synthèse 102

Partie III Réalisations et expérimentations 103

Chapitre 9 Le prototype 105
9.1 Composants du prototype 105
9.2 Générateur de tests pour les systèmes évolutifs 107
9.3 Publication des tests dans le référentiel de tests 111
9.4 Création du plug-in pour IBM Rational Software Architect 112
9.5 Synthèse 114

Chapitre 10 Étude de cas : Card Life Cycle et bilan 115
10.1 Présentation du scope : Card Life Cycle 116
10.1.1 Exigences fonctionnelles 118
10.1.2 Modèles de test 118
10.1.3 Exigences de sécurité 120
10.2 Évolution du scope : Card Life Cycle 122
10.2.1 Les exigences fonctionnelles 122
10.2.2 Modèles de Test 123
10.3 Résultats obtenus pour Card Life Cycle 126
10.3.1 SeTGaM pour les diagrammes d'états/transitions 126

10.3.2 SeTGaM pour les comportements d'opérations UML/OCL . . . 126
10.3.3 SeTGaM pour les TCS . . . 127
10.4 Bilan : Card Life Cycle . . . 128
10.5 Bilan : eCinema . . . 130
10.6 Synthèse . . . 133

Chapitre 11 Évaluation de SeTGaM **135**
11.1 Sûreté . . . 135
11.2 Précision . . . 137
11.3 Efficacité . . . 138
11.3.1 Le temps . . . 138
11.3.2 L'automatisation . . . 139
11.3.3 Calculer la modification . . . 139
11.3.4 Traitement de plusieurs modifications . . . 139
11.4 Généralité . . . 139

Partie IV Conclusion et perspectives **141**

Chapitre 12 Conclusion **143**

Chapitre 13 Perspectives **145**
13.1 Langages . . . 145
13.2 Amélioration de SeTGaM . . . 146
13.3 Validation et extensions . . . 147

Annexes **149**

Annexe A Détails sur les exigences fonctionnelles pour les modèles d'eCinema **151**

Annexe B Relations de dépendances pour les modèles d'eCinema **155**

Annexe C Catégorisation des tests pour les modèles d'eCinema **171**

Annexe D Le langage de schémas de Smartesting **173**

Bibliographie **177**

Résumé **185**

Abstract **185**

Table des figures

1.1 Model-Based Testing 5
1.2 Processus général du projet SecureChange 7

2.1 Processus général de Model-Based Testing 17
2.2 Processus MBST à partir de schémas [Sec12b] 30

4.1 Diagramme de classes d'eCinema 43
4.2 Diagramme d'états/transitions d'eCinema 43
4.3 buy_ticket-success 44
4.4 login-success 44
4.5 register-success 45
4.6 Opération buy_ticket 48
4.7 Diagramme de classes d'eCinema - évolué 52

5.1 Processus global de SeTGaM 58
5.2 Cycle de vie de tests 60
5.3 Référentiel de tests 64

6.1 Extrait du diagramme d'états/transitions d'eCinema 72
6.2 Extrait du diagramme d'états/transitions d'eCinema 75

7.1 Transformation d'une opération en comportements 82
7.2 Transformation d'un fragment de l'opération "buyTicket" 83
7.3 Calcul de différences comportementales 85
7.4 Calcul de dépendances comportementales 86
7.5 Dépendance fausse-positive dans le cas d'eCinema 87
7.6 Dépendance positive dans le cas d' eCinema 87

7.7 Calcul d'un comportement impacté sur eCinema 89

8.1 Définition du statut des tests par rapport à la comparaison des TCSs . . . 99
8.2 Extension de SeTGaM pour les TCS 101

9.1 Processus de génération et de gestion des tests 106
9.2 Diagramme d'activité du processus associé au prototype 107
9.3 Composants de SeTGaM 107
9.4 Diagramme d'activité pour SeTGaM 108
9.5 Diagramme d'activité du calcul de dépendances 109
9.6 Diagramme d'activité de classification de tests 110
9.7 Calcul des tests re-testable 111
9.8 Export via Smart Publisher 112
9.9 L'intégration du prototype en plugin Eclipse 113
9.10 Résumé de l'architecture du prototype 114

10.1 Card Life Cycle v2.1.1 117
10.2 Diagramme d'états/transitions Card Life Cycle v2.1.1 119
10.3 Card Life Cycle évolution vers v2.2 123
10.4 Diagramme d'états/transition Card Life Cycle v2.2 125

13.1 Liens entre modèle, test et code 148

Liste des tableaux

4.1 Tests Générés pour l'exigence de sécurité 1, eCinema diagramme d'états/-transitions 50

4.2 Tests Générés pour l'exigence de sécurité 1, eCinema sans diagramme d'états/transitions 50

4.3 Tests Générés pour l'exigence de sécurité 2 50

6.1 Différence de transitions pour un extrait d'eCinema 69

6.2 Def/Use pour un extrait d'eCinema 72

6.3 Post-dominance et dépendances de contrôle sur un fragment d'eCinema . . 76

6.4 Calcul des impacts sur un fragment d'eCinema 77

7.1 Tableau résumant les impacts sur un fragment d'eCinema 89

8.1 TCS classification pour eCinema (Modèle 1 - avec et Modèle 2 - sans diagramme d'états/transitions) 96

8.2 Classification de tests pour les exigences de sécurité pour eCinema 101

10.1 Comparaison résultats Card Life Cycle v2.1.1 et v2.2 pour SeTGaM - test fonctionnel 129

10.2 Comparaison résultats Card Life Cycle v2.1.1 et v2.2 pour SeTGaM - test de sécurité 129

10.3 Comparaison résultats eCinema v1 et v2 pour SeTGaM - test fonctionnel . 132

10.4 Comparaison résultats eCinema v1 et v2 pour SeTGaM - test de sécurité . 132

A.1 Exigences pour eCinema 151

A.2 Comportement/Transitions pour eCinema. 152

A.3 Nouvelles exigences - eCinema évolution 153

A.4 Comportements/Transitions modifiés et supprimés - eCinema 153

A.5 Nouveaux comportements/transitions - eCinema évolution 154

B.1 Dépendances de données et de contrôle pour le diagramme d'états/transitions pour eCinema 155
B.2 Dépendances comportementales pour eCinema 156
B.3 Dépendances de contrôle pour le diagramme d'états/transitions d'eCinema - évolution 157
B.4 Dépendances de données pour le diagramme d'états/transitions eCinema - évolution 157
B.5 Dépendances comportementales pour eCinema -évolution 169

C.1 Résultats de SeTGaM pour le test de sécurité - eCinema 171
C.2 Résultats de SeTGaM pour le test fonctionnel - eCinema 172

D.1 Syntaxe des schémas : règles de grammaire 174
D.2 Syntaxe des schémas : symboles terminaux 175

Première partie

Contexte et problématiques

Chapitre 1

Introduction

Sommaire

1.1 Contexte 4
1.1.1 Model-Based Testing 4
1.1.2 Projet *SecureChange* 5
1.2 Problématiques 8
1.2.1 Gestion de l'évolution du système 8
1.2.2 Traitement de la problématique de la sécurité 9
1.3 Contributions 10
1.3.1 Génération sélective de tests 10
1.3.2 Tests liés à la sécurité dans les systèmes évolutifs 11
1.4 Plan du mémoire 12

Aujourd'hui, les systèmes doivent évoluer rapidement pour satisfaire les exigences commerciales ou simplement pour répondre aux besoins toujours croissants du marché. Il faut pouvoir corriger les bugs, créer de nouvelles fonctionnalités ou des produits face à un public volatil. Cela n'est pas sans risque. En effet, en plus des évolutions des produits, il faut faire évoluer la partie "back-office" pour répondre et adapter l'infrastructure à ces évolutions. Malgré tout, cette frénésie ne doit pas s'exercer au détriment de la qualité ou de la sécurité.

Cela est devenu une véritable course contre le temps qui ne laisse pas la place aux erreurs, notamment lorsqu'il s'agit de systèmes critiques.

Un problème récent dans les télécommunications en est une parfaite illustration. Le 11 juillet 2012, Orange a été victime d'une panne qui a affecté 26 millions d'abonnés en France. D'après son PDG, qui s'est exprimé dans le journal Ouest-France, la panne a concerné le réseau mobile suite à une mise à jour. Cela a entrainé une perte de plusieurs

dizaines de millions d'euros. Rappelons nous également, le stress et la frayeur provoqués par le passage à l'an 2000 et à l'idée de son bug. Ma conviction personnelle est que ce changement de millénaire a été un facteur important pour mettre en exergue la nécessité de tester de tels systèmes. Cet électrochoc a permis d'augmenter l'exploration des problématiques associées, d'améliorer les solutions existantes et d'inciter à la mise en pratique dans l'industrie d'un processus de validation continue.

Je souhaite dans cette introduction rappeler les motivations de ce travail ainsi que les raisons de mon implication dans cette thématique. Pour ces raisons, je souhaitais et j'ai fait partie du projet *SecureChange*, qui bien dirigé, a apporté des réponses à ce défi d'évolution et de sécurité.

Ce chapitre en section 1.1 positionne le contexte du travail qui est le test à partir de modèles (ou *Model-Based Testing* (MBT)), effectué dans le cadre du projet *SecureChange*. Les problématiques qui en découlent vont être détaillées dans la section 1.2. Les contributions que nous allons présenter plus en détails dans la suite de ce cette thèse sont introduites dans la section 1.3. Nous conclurons ce chapitre en donnant le plan général de ce mémoire.

1.1 Contexte

Dans cette section nous positionnons le contexte de cette thèse dont les deux éléments principaux sont le Model-Based Testing et le projet européen *SecureChange*.

1.1.1 Model-Based Testing

Nous décrivons le processus du Model-Based Testing (MBT) sur la figure 1.1. Il commence par la création d'un modèle de test par l'ingénieur de test, à partir des exigences décrites dans le cahier des charges (spécification). La création des tests peut être manuelle ou automatique. Dans le cadre du MBT automatisé, le générateur de tests prend en entrée le modèle de test dans le but de produire des cas de tests et une matrice de couverture. Cette dernière illustre le lien entre les tests et les éléments du modèle couverts par les tests. Enfin, les tests sont exportés, ou publiés, dans un référentiel de tests. À partir de ce point, il est possible de générer des scripts pour exécuter les tests sur le système sous test (ou *System Under Test* (SUT)). L'exécution peut être également manuelle ou automatique. Elle peut être automatisée via une couche d'adaptation, en se basant sur le principe de test à partir de mot-clefs (ou *keyword-based testing*). Celle-ci permet d'associer chaque pas du test à une commande, ou opération, du SUT. Après l'exécution, les résultats des tests et les métriques sont fournis à l'utilisateur.

Le processus du MBT tel que décrit, ne prend pas en compte l'évolution du système.

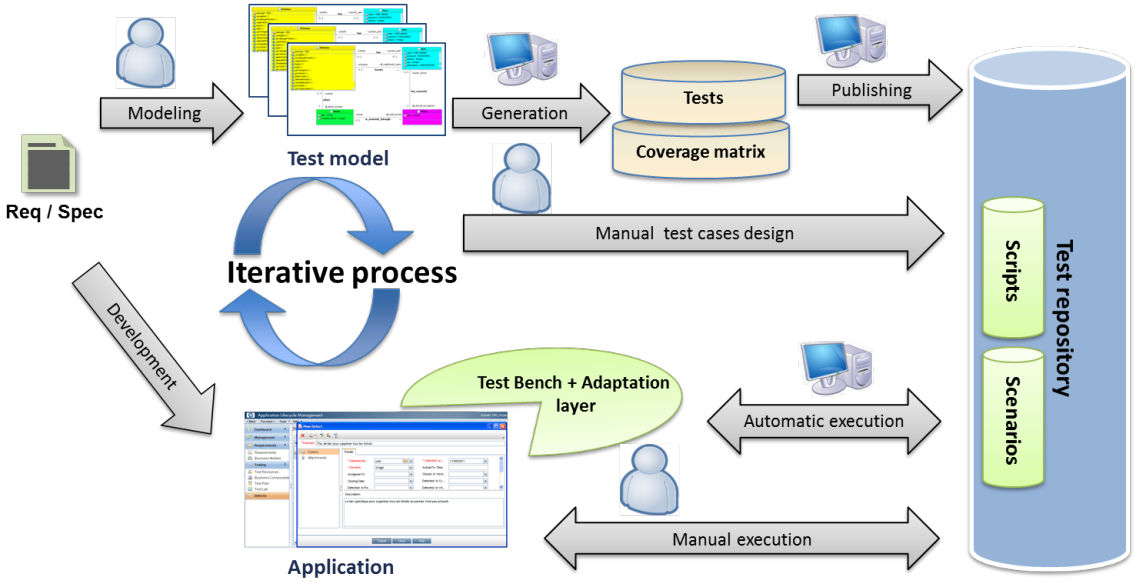

FIGURE 1.1 – Model-Based Testing

En se basant sur la figure 1.1, nous pouvons définir les entités qui peuvent évoluer lors d'évolution du système :

- les exigences,
- le modèle de test,
- les cas de tests.

En conséquence trois autres entités sont modifiées :

- la matrice de couverture,
- le référentiel de test,
- les scripts exécutables.

Chaque évolution des exigences est décrite de manière informelle dans la spécification. L'ingénieur de test effectue cette modification dans le modèle de test, qui est envoyé au générateur de tests. Ce dernier propage l'évolution dans la matrice de couverture et dans les tests générés. L'export de ses tests modifie le référentiel de tests. Enfin, les scripts sont mis à jour afin de permettre l'exécution sur le SUT. De cette façon le processus peut permettre le test à partir de modèles des systèmes évolutifs.

1.1.2 Projet *SecureChange*

Le projet *SecureChange* est un projet Européen de type FET de l'appel FP7 [1]. Il a été coordonné par Fabio Massacci de l'Università degli Studi di Trento en Italie. Ce projet consiste à créer des techniques et des outils pour les systèmes critiques évolutifs, afin de les accompagner tout au long de leur vie sur toutes les étapes de création d'un logiciel, en

1. ICT-FET-231101 (http ://securechange.eu)

commençant par sa conception, à travers des analyses des risques jusqu'à sa vérification et sa validation par le test. Ce sujet induit plusieurs défis pour les chercheurs qui sont en lien direct avec les enjeux du contexte industriel actuel. Les challenges principaux de ce projet sont :

- Comment redéfinir les étapes du processus d'ingénierie des systèmes classiques pour prendre en compte la réalisation de nouveaux logiciels et la conception de systèmes qui devront être pérennes ?
- Comment se convaincre et être confiant dans quelque chose qui va évoluer ? Est-il possible de vérifier et tester un système qui évolue sans cesse ?
- Comment évaluer et garantir la sécurité d'un système lorsque celui-ci change ? Comment redéfinir les méthodologies d'évaluation de risque dans le but de faire face aux changements ?

Ce projet était composé de cinq partenaires industriels et huit partenaires académiques. Les partenaires industriels étaient Gemalto (FR), Smartesting (FR), Thales (FR), Telefonica Investigacion y Desarrollo Sociedad Anonima Unipersonal (ES) et Deep Blue (IT). Les partenaires académiques étaient Università degli Studi di Trento (IT), Budapest University of Technology and Economics (HU), Institut national de Recherche en Informatique et en Automatique (FR), Katholieke Universiteit Leuven (BE), Open University (UK), Stiftelsen for industiell og teknisk forskning ved Norges Tekniske Hogskole (NO), University of Innsbruck (A) et Technische Universität Dortmund (D).

Ces équipes ont travaillé pour répondre aux défis du projet à travers sept groupes de travail (en anglais *Workpackages* (WP)).

WP1 - Industrial Use Cases Ce groupe de travail propose des scénarios et des études de cas industriels afin de guider les recherches vers le besoin industriel d'une part et d'évaluer les méthodologies et outils d'autres part.

WP2 - Design process Ce groupe de travail se concentre sur deux aspects principaux :

- développer des concepts de modélisation et gestion de processus pour des composants de sécurité configurables.
- développer des techniques basées sur les modèles pour couvrir les exigences par une analyse de risque.

WP3 - Evolving requirements engineering L'objectif du WP3 est de permettre de manipuler les concepts associés aux exigences.

WP4 - Model Design L'objectif de ce groupe de travail est de développer une approche basée sur les modèles.

WP5 - Risk assessment Le groupe de travail de *Risk Assessment* cible le développement de méthodes et d'outils pour évaluer la sécurité à travers la notion de risque.

WP6 - Verification Le travail du WP6 se situe au niveau du code. Son but est de vérifier le code qui évolue.

WP7 - Model-Based Testing (MBT) for evolution L'objectif de ce groupe de travail est de valider la sécurité du système par le biais de tests générés à partir de modèles.

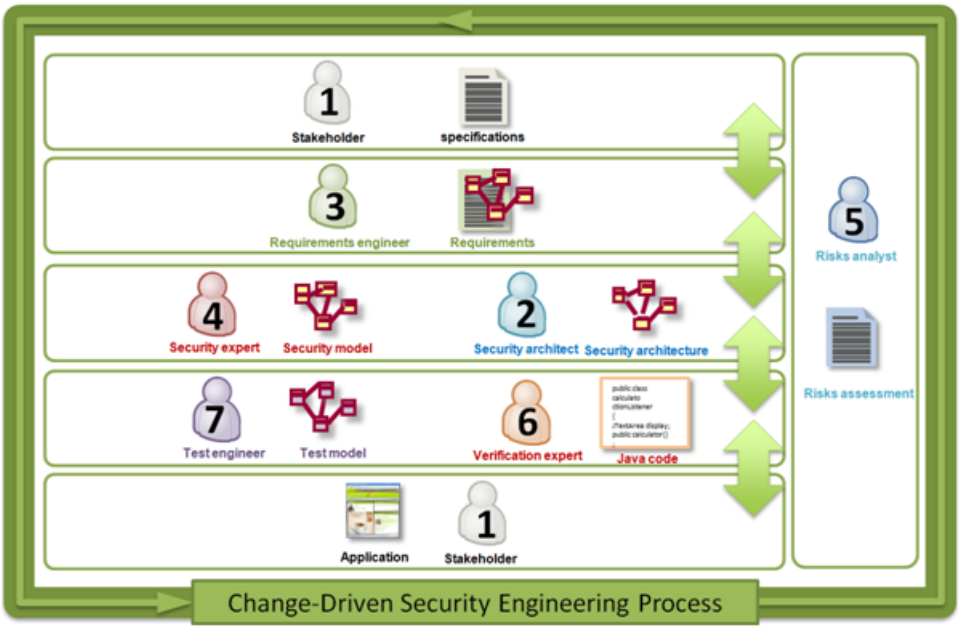

FIGURE 1.2 – Processus général du projet SecureChange

Je suis intervenue dans le projet dans le cadre des actions menées par le groupe WP7. Dans ce projet, plus précisément, nous avons travaillé sur le test de non-régression et la prise en compte des aspects de sécurité lors d'évolutions.

Concernant les collaborations dans le cadre du projet, nous avons pu interagir avec les autres WP comme illustré sur la figure 1.2.

Plus précisément, pour le lien avec le WP4, nous avons travaillé sur la complémentarité entre la validation et la vérification. Dans le processus MBT, les modèles pour la génération de tests de sécurité sont considérés comme vérifiés. Néanmoins pour effectuer cela, il faut prouver que le modèle préserve les propriétés de sécurité issues des exigences. Les résultats de ces travaux avec le WP4 ont donné lieu à 2 publications ([FOB+11] et [FBO+12]) qui concernent les techniques de validation des exigences de sécurité sur le modèle en prenant en compte ses évolutions.

Dans la gestion des évolutions des systèmes, il est particulièrement difficile de propager le changement au travers des différents modèles (modèles d'exigences, de test, etc). Afin de répondre à cette problématique, nous avons travaillé de concert avec le WP3. Nous avons étudié, d'une part, l'analyse des évolutions des exigences, et d'autre part, des règles de propagation des changements entre les exigences et le modèle de test via la traçabilité des exigences. Les résultats de ce travail sont publiés dans [PMBD12].

1.2 Problématiques

Dans cette partie, nous allons détailler les problématiques du test à partir de modèles (MBT) issus des systèmes évoluant et plus particulièrement le test de non-régression à partir de modèles.

Pour répondre à cela, nous avons déterminé deux défis pour le domaine du test à partir de modèles : la gestion de l'évolution et de la sécurité. En se basant sur ces deux grands défis, nous allons d'abord détailler comment faire pour gérer l'évolution pour une technique MBT, ensuite comment prendre en compte l'aspect sécurité et à quel degré il est possible de le prendre en compte dans le processus de l'évolution.

1.2.1 Gestion de l'évolution du système

Pour pouvoir prendre en compte la gestion des évolutions de bout en bout dans le processus MBT, il faut pouvoir répondre à ces sept points :

1. **Modéliser** l'évolution du système en prenant compte les exigences fonctionnelles.
2. **Identifier** les modifications apportées.
3. **Sélectionner** un sous-ensemble de tests à partir de la suite de tests avant l'évolution.
4. **Exécuter** *les tests sélectionnés* sur le système modifié.
5. Si nécessaire **créer** des nouveaux cas de tests.
6. **Exécuter** *les nouveaux tests* sur le système modifié.
7. **Construire** la nouvelle suite de tests pour le système modifié.

Notre approche s'inscrit dans une démarche **MBT**. C'est pourquoi, nous avons besoin de définir le *formalisme* que nous avons retenu pour représenter le système et ses évolutions. Il faut aussi expliciter les *critères de sélection de tests*. Ces critères de sélection de tests vont permettre de couvrir les comportements du système et générer un nombre suffisant de tests pour couvrir un maximum de situations possibles du système testé. Ainsi, nous ferons un rapide état des lieux pour choisir la technique que nous utiliserons. Ce travail est présenté dans la section 2.1.

Notre travail trouve ses fondements dans le cadre du projet *SecureChange*. *Smartesting* est l'un des partenaires industriels du WP7. Nous nous sommes donc naturellement orientés vers leur solution pour l'outil de génération de tests à partir de modèles. Cet outil a été le point de départ pour définir notre méthodologie pour le test des systèmes évolutifs.

Pour ce faire, nous utiliserons les techniques de test de non-régression à partir de modèles, détaillés en chapitre 3. Classiquement, toutes les techniques de non-régression basées sur les modèles font une comparaison des modèles afin de définir précisément les évolutions introduites. À partir du formalisme choisi, une **identification des modifications** doit être effectuée. Ensuite, une classification des tests est effectuée pour permettre la gestion du *cycle de vie* des tests. Tout cela sera le point de départ de notre approche.

1.2.2 Traitement de la problématique de la sécurité

Le **test de sécurité** du système est défini comme une réaction correcte du système lors de la présence d'attaques ou d'actions mal intentionnées [MP04]. Durant ces dernières années, une grande attention a été portée aux exigences de sécurité.

Ils existent plusieurs types de *formalisations* des exigences de sécurité utilisées pour le test à partir de modèles. Ces formalismes sont utilisés par différents *groupes* de techniques de test de sécurité : Vulnerability Testing, Static Application Security Testing et Dynamic Application Security Testing. Nous allons nous intéresser plus particulièrement au groupe des techniques nommées *Dynamic Application Security Testing* (DAST), dont le test de sécurité à partir de modèles fait partie.

Cette approche servira de base pour répondre à la problématique **d'évolution des exigences de sécurité et de test de non-régression orienté sécurité et basé sur les modèles**. Ainsi, il est indispensable de définir quelles entités évoluent : *les exigences de sécurité, le modèle de test ou les deux* et d'étudier l'impact sur les tests associés.

En effet, les *exigences de sécurité* peuvent évoluer de deux manières. La première concerne l'évolution due à un changement de spécification pour étendre le périmètre du système. La seconde vient de l'ingénieur de sécurité pour valider l'absence de nouvelles vulnérabilités. Dans les deux cas, il est nécessaire de *détecter l'évolution dans le formalisme* qui représente la propriété.

Dans la section suivante, nous présentons les contributions de cette thèse qui viennent répondre aux problématiques présentées ici.

1.3 Contributions

Cette section résume nos contributions. Elles se composent d'un apport théorique et d'un apport pratique.

Notre première contribution a été d'étendre les travaux existants sur la non-régression et permettre une classification fine des tests. La seconde est d'avoir étendu ce travail pour prendre en compte des variations de modélisation et l'utilisation de schémas de test.

D'un point de vue pratique, nous avons implémenté l'ensemble de ces algorithmes dans un outil appelé *EvoTest*. Cet outil a été mis à disposition et évalué par les partenaires du projet européen *SecureChange*.

1.3.1 Génération sélective de tests

Pour décrire les comportements du système, nous nous basons sur les modèles de test UML/OCL. Les notations UML/OCL utilisées au sein des gardes/actions des transitions ou des pré/post conditions des opérations permettent de formaliser les comportements du système. Afin de rester le plus général possible dans le processus de modélisation supporté, nous prenons en compte ces deux possibilités pour gérer l'évolution et créer deux axes possibles de la méthodologie.

En utilisant les *comportements décrits dans les transitions du diagramme d'états/transitions et les opérations*, nous pouvons identifier **les évolutions** de modèles. À partir des comportements identifiés comme ayant évolué, nous calculons l'ensemble des comportements impactés de façon indirecte par ces évolutions. Pour cela, nous utilisons *l'analyse de dépendances* entre ces comportements. Nous allons proposer deux approches complémentaires, une pour effectuer une analyse à partir des diagrammes d'états/transitions et l'autre seulement à partir des comportements des opérations. Pour la première, nous appliquons l'analyse des dépendances de données et de contrôle adaptées à nos modèles. Pour la deuxième, nous appliquons une analyse des dépendances basée sur l'utilisation et la définition des variables à laquelle est ajoutée une vérification de consistance. Nous utilisons le lien de **traçabilité** entre les comportements présents dans le modèle et les tests pour déterminer les tests impactés.

Ces derniers sont les tests qui vont voir leur statut de **cycle de vie des tests** évoluer. Nous avons affiné les différents statuts que peuvent prendre les tests durant leur vie, pour mieux cibler les éléments à valider. Typiquement, la classification en *obsolète, réutilisable et retestable* ne permet pas de prendre en compte le fait qu'un test soit impacté par l'évolution ou non. Effectivement, il peut y avoir deux raisons pour que le test soit obsolète. Par exemple, un test *obsolète* contient par définition une exécution seulement de la séquence. Néanmoins, il peut être obsolète parce que la séquence de test n'a plus les mêmes données d'entrée/sortie suite aux modifications dans le système ou encore en

raison de la suppression d'un comportement qu'il couvrait. Il en est de même pour les deux autres catégories. Pour cela, nous proposons d'étendre ces trois catégories en sept catégories : *outdated, failed, adapted, updated, reexecuted, unimpacted and removed.*

Cette distinction est très importante lors de l'analyse de l'exécution des tests. En effet, il est d'autant plus facile de comprendre d'où peut venir l'erreur si cela concerne un test mis à jour. Nous nous retrouvons ensuite face au triptyque qui est une erreur dans le modèle, une erreur dans la couche d'adaptation ou finalement dans l'implémentation. Cette dernière peut être contrôlée en rejouant la version précédente du test.

Afin de tester les nouveaux comportements, nous ajoutons la catégorie *new*, qui est l'état de départ de chaque test.

Pour aider le travail de sélection des tests, nous les regroupons suivant leur statut en quatre suites de tests : *Evolution, Regression, Stagnation et Deletion*, qui vont permettre d'obtenir une **priorité d'exécution** naturelle et selon le besoin de l'utilisateur. Ces suites de tests vont permettre de tester l'évolution, la non-régression, la stagnation et la dernière suite va permettre de garder l'historique des tests obsolètes d'une version à une autre. L'utilisateur peut choisir de tester les nouvelles fonctionnalités (Evolution) ou la non-régression, mais il peut également tester la stagnation. C'est-à-dire valider que ce qui aurait du être supprimé du système a effectivement disparu des fonctionnalités du système. Ceci peut être très important lors des modifications des droits d'accès dans des systèmes traitant des données critiques et confidentielles, par exemple.

Tout cela est pris en compte par notre méthodologie appelée *génération sélective de tests* (en anglais Selective Test Generation Method (SeTGaM)). Afin de permettre une gestion de l'**historique des tests**, sujet à notre connaissance très peu discuté par les techniques de non-régression, nous utilisons un gestionnaire de **référentiel de tests**.

1.3.2 Tests liés à la sécurité dans les systèmes évolutifs

Il existe beaucoup de travaux de recherche sur le **test des exigences de sécurité** mais ils ne prennent pas en compte l'aspect évolution. Il est cependant important que ces techniques de test restent sûres, précises et efficaces, lors de l'évolution du système.

Pour formaliser les exigences de sécurité, nous travaillons avec des **schémas**. Ces derniers sont ensuite utilisés en combinaison avec le modèle pour définir des spécifications de cas de test. En se basant sur ces spécifications, nous sommes capables d'**identifier la ou les entités qui évoluent**. Ainsi, nous prenons en compte les trois cas de changements possibles :

- le schéma peut évoluer suite aux changements des objectifs de test de sécurité de l'ingénieur de sécurité ou de la spécification.
- le schéma ne change pas, mais des modifications fonctionnelles sont apportées dans

le modèle.
- le schéma peut évoluer et le modèle également.

Dans les trois cas, le changement se propage au niveau de la production des spécifications de cas de tests. Ensuite, nous allons montrer comment lier cette approche aux principes fonctionnels introduits pour la génération sélective de tests.

De cette façon, il est possible de commencer à convaincre la communauté d'aller au-delà du test de non-régression fonctionnel et d'appliquer des approches guidées par les objectifs de sécurité.

1.4 Plan du mémoire

Cette section présente l'organisation de ce mémoire. Après ce chapitre d'introduction, nous organisons la suite du manuscrit en quatre parties.

La première partie présente les éléments de l'état de l'art permettant de répondre à notre problématique dans le contexte de notre travail. Elle contient aussi la présentation de l'exemple fil rouge de cette thèse. Plus précisément :

Le **Chapitre 2** donne une vue détaillée sur l'état de l'art du test à partir de modèles fonctionnels et de sécurité. Il est suivi d'une description de la technique outillée par Smartesting, utilisée comme base pour les travaux menés pendant cette thèse.

Le **Chapitre 3** présente l'état de l'art du test de non-régression, les principes, les catégories ainsi que le cadre d'évaluation de telles techniques. Ce chapitre introduit également les travaux courants du test de non-régression basé sur le test de sécurité.

Le **Chapitre 4** introduit l'exemple *eCinema* et son évolution utilisés pour illustrer les travaux de cette thèse.

La deuxième partie détaille nos contributions, comme suit dans les chapitres :

Le **Chapitre 5** introduit le cycle de vie des tests et leur regroupement sous forme de suites de tests pour s'assurer de l'évolution, de la non-régression, de la stagnation et de l'historique du système évolué. Ce chapitre pose les bases des trois axes de l'approche.

Le **Chapitre 6** présente la méthodologie de génération sélective de tests pour les systèmes évolutifs, en se basant sur les comportements issus des gardes/actions des transitions du diagramme d'états/transitions en UML/OCL.

Le **Chapitre 7** présente la méthodologie de génération sélective de tests pour les systèmes évolutifs, en se basant sur les comportements issus des pré/post conditions des opérations du diagramme de classes en UML/OCL.

Le **Chapitre 8** se base sur les deux précédents pour prendre en compte le test de sécurité des systèmes évolutifs.

La troisième partie détaille le prototype et les expérimentations menées pour valider nos travaux.

Le **Chapitre 9** présente l'architecture et la vue du prototype créé.

Le **Chapitre 10** donne les résultats sur le cas d'étude industriel et fait un bilan sur les expérimentations.

Le **Chapitre 11** évalue les travaux de cette thèse selon le cadre d'évaluation des techniques de non-régression.

La dernière partie conclut ce manuscrit et propose des perspectives de recherches sur les thèmes développés dans ce mémoire.

Le **Chapitre 12** donne une rétrospective du travail effectué pendant cette thèse.

Le **Chapitre 13** introduit des nouveaux challenges pour le domaine du test.

Enfin, nous joignons quatre annexes pour compléter et expliciter certains éléments présentés dans les chapitres précédents :

L'**Annexe A** liste toutes les exigences fonctionnelles pour les deux versions d'eCinema.

L'**Annexe B** liste les relations des dépendances des modèles d'eCinema.

L'**Annexe C** liste les tests pour les modèles d'eCinema et leur statut pour la version évoluée.

L'**Annexe D** donne la grammaire du langage de schémas de Smartesting.

Chapitre 2

Test à partir de modèles

Sommaire

2.1 Le test fonctionnel à partir de modèles 17
2.1.1 Formalismes de modélisation 18
2.1.2 Critères de sélection 19
2.2 Le test de sécurité 22
2.2.1 Formalismes pour l'expression des exigences de sécurité . . 23
2.2.2 Techniques de test de sécurité 25
2.3 Outillage Smartesting pour la génération de tests 27
2.3.1 Diagrammes UML4MBT 27
2.3.2 Traçabilité des exigences 29
2.3.3 La prise en compte des exigences de sécurité 29
2.4 Synthèse 31

Dans ce chapitre, nous présentons les différents concepts sur le test à partir de modèles. Nous commençons avec quelques rappels de terminologie sur le test.

Le test est l'évaluation du système en observant ses points d'exécution [AO08]. Le système exécuté est appelé système sous test (en anglais *system under test* (SUT)). Le test est également l'exécution d'un système ou d'un composant par des moyens automatiques ou manuels, pour vérifier s'il répond à ses spécifications. Ceci permet d'identifier les différences entre les résultats attendus et les résultats obtenus [BD04]. Dans un cas de test, le comportement courant et le comportement attendu du SUT sont comparés, le résultat obtenu est appelé un *verdict*. Le verdict est la décision associée à un test et peut prendre trois valeurs *pass* (pour une réussite), *fail* (pour un échec) et *inconclusif* (pour un état inattendu) [ISO94]. Un *oracle de test* est la fonction qui permet d'assigner le verdict. Les valeurs d'entrée de cette fonction peuvent provenir d'un modèle, du cahier des charges ou

d'un autre applicatif. Dans ce manuscrit lorsque nous parlons de changement ou non de l'oracle nous considérons le changement ou non de ses valeurs d'entrée.

Le test peut être qualifié en fonction : du cycle de développement du logiciel, de l'accessibilité du SUT ou des caractéristiques testées [Tre04, UL07].

Dans le cycle de développement du logiciel, nous distinguons le test unitaire i.e. le test des unités du programme, le test d'intégration i.e. le test s'applique sur le résultat de l'assemblage des modules, et le test système i.e. le test est réalisé sur l'ensemble du système comme une unité, en se basant sur les exigences du système.

Suivant l'accessibilité du système testé, deux catégories existent : le test boite noire (en anglais *black box testing*) i.e. les tests sont issus d'une description externe (indépendante de l'implémentation) comme une spécification du système et le test boite blanche (en anglais *white box testing*) i.e. les tests sont issus d'une description interne du système, tel que le code.

Selon les caractéristiques testées dans le système, nous distinguons le test fonctionnel, qui permet de tester le comportement opérationnel du SUT, et le test des caractéristiques non-fonctionnelles comme la sécurité, la performance, la robustesse, ou la facilité d'utilisation.

Le test à partir de modèles (en anglais *Model-Based Testing (MBT)*), sur lequel nous nous basons pour nos recherches, repose sur les principes du test fonctionnel. Le modèle de test est la représentation externe formelle ou semi-formelle du système testé. Ce modèle est créé dans le but de s'en servir pour la génération de tests [LBP09].

Il y a beaucoup d'approches et de définitions pour le MBT. Cependant, elles ont toujours deux aspects communs : la spécification et la génération de tests à partir d'un modèle [RBGW10]. Le MBT est un sujet de recherche très actif [UPL12], avec des résultats pratiques indéniables [UL07, Sch12]. De plus, la mise en place d'un processus outillé permet d'assurer à la fois la reproductibilité des activités de test, de mesurer leur qualité et de mieux maîtriser les coûts [LBP09].

C'est pourquoi nous nous sommes intéressés à proposer une approche basée sur le MBT. Dans la suite de ce chapitre, nous détaillons les formalismes associés aux techniques de MBT, les différents critères de sélection de tests ainsi que la prise en compte des exigences fonctionnelles en section 2.1 et de sécurité en section 2.2. Avant de conclure ce chapitre dans la section 2.4, nous présentons dans la section 2.3 l'outil de génération de tests qui sera utilisé en tant que base de notre outillage pour le test des systèmes évolutifs. Il est basé sur la méthodologie MBT et créé par Smartesting.

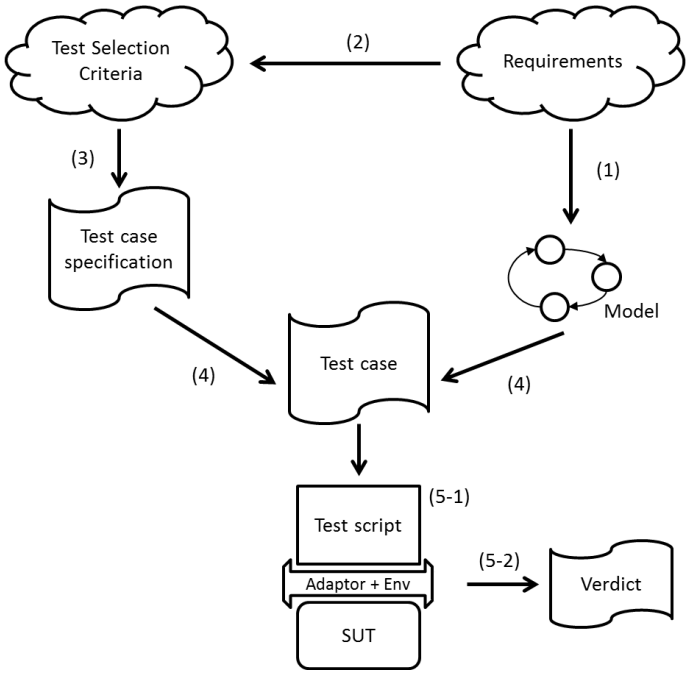

FIGURE 2.1 – Processus général de Model-Based Testing

2.1 Le test fonctionnel à partir de modèles

Dans cette section, l'approche fonctionnelle est abordée avec une approche basée sur les modèles. Nous présentons la notion de processus MBT. Pour ce faire, nous l'illustrons sur la figure 2.1. Les 5 étapes sont les suivantes :

1. Définition du modèle du système sous test. Le modèle est construit à partir des spécifications (exigences) du système sous test et du plan de tests. Ce modèle est une vue abstraite du système sous test, défini dans un formalisme particulier. Une présentation des formalismes possibles est faite en section 2.1.1.
2. Choix du critère de sélection des tests. Sur la base du plan de tests, l'ingénieur de validation choisit les éléments de modèles et les critères de couverture qu'il souhaite. Nous présentons en section 2.1.2 les différentes catégories de critères de sélection présents dans la littérature.
3. Extraction des spécifications de tests. Sur la base des critères de sélection définies dans l'étape précédente, un ensemble d'informations est extrait du modèle pour définir les objectifs de tests à couvrir pour obtenir le critère de couverture recherché.
4. Établissement de l'ensemble de cas de tests. En utilisant le modèle du système sous test et des spécifications des cas de tests, les cas de tests sont générés (manuellement ou automatiquement). Un cas de test est une séquence d'activation d'opérations du modèle, munies de leurs paramètres.
5. Exécution des cas de tests sur le système sous test. Premièrement, en $(5-1)$ les cas

de tests abstraits sont concrétisés par le biais d'une couche d'adaptation. La couche d'adaptation permet d'associer à chaque élément du modèle le ou les éléments de l'implémentation qu'il modélise. À l'étape $(5-2)$, un verdict est associé au cas de tests, définissant ainsi le résultat de l'exécution de chaque test sur l'implémentation. Ainsi, ceci permet de révéler les non-conformités entre les résultats prévus par le modèle et les résultats obtenus de l'implémentation.

2.1.1 Formalismes de modélisation

Les auteurs dans [UL07] proposent de regrouper en 7 catégories les différents formalismes de modélisation pour la génération de tests. Ces catégories sont les suivantes :

1. *Modèles pré/post* : l'état interne du système sous test est présenté comme une collection de variables. Les états du système sont définis sur la base de ces variables et l'évolution du système est décrite au sein des opérations. Au lieu de définir les opérations par un langage de programmation, chacune est définie par une pré et une post condition. Nous pouvons citer par exemple les langages de spécification : OCL [WK03], B [Lan96], Z [Smi00] et JML [LBR06].
2. *Modèles de transitions* : ils décrivent les *transitions* entre les états du système sous test. Nous retrouvons typiquement les notations comme : les machines à états finis (en anglais *Finite State Machine* (FSMs)), les machines à états/transitions écrites en *Unified Modeling Language* (UML) [2], les systèmes de transitions étiquetées (en anglais *Labelled Transition Systems* (LTS)), les systèmes de transitions étiquetées à entrées et sorties (en anglais *Input/Output Labelled Transition Systems* (IOLTS)) et Simulink [3], notation utilisée pour les systèmes embarqués.
3. *Modèles basés sur l'historique* : ce formalisme décrit le système à travers des possibles traces et il permet de répondre aux problématiques du temps. Un exemple type sont les diagrammes de séquences à messages (en anglais *Message sequence charts* (MSC)).
4. *Modèles basés sur les formules* : ce formalisme décrit le système comme un ensemble de formules mathématiques (formules de premier ordre (en anglais *First Order Logic*(FOL)) ou d'ordre supérieur (en anglais *Higher Order Logic* (HOL)).
5. *Modèles basés sur les processus* : permettent d'exprimer le système comme un ensemble de processus exécutables en parallèle (CPS, CSS ou les réseaux de Petri).
6. *Modèles stochastiques* : le système est exprimé via un modèle probabiliste d'événements et de données d'entrée, modélisant plutôt l'environnement du système que le système lui même (par exemple les chaines de Markov).

2. www.omg.org
3. http ://www.mathworks.com/products/simulink

7. *Modèles basés sur les flots de données* : les modèles se basent sur le flot de données, qui décrit comment chaque donnée (représentée par une variable ou assimilable) est modifiée dans le système. Ce sont par exemple les formalismes Lustre [HCRP91] et Simulink.

La plupart des formalismes se situent dans plusieurs catégories. Par exemple UML, qui est devenu un standard de modélisation dans le domaine des systèmes d'information est très répandu dans le domaine industriel [LBP09]. UML utilisé avec OCL permet d'exprimer des formalismes en pré/post (via OCL) et avec des transitions (via les diagrammes d'états/transitions).

2.1.2 Critères de sélection

La génération de tests repose sur la sélection des critères qui peuvent être appliqués et la technologie de génération de tests. Nous détaillons ici les différents critères de sélection de tests catégorisés en deux types : **statiques et dynamiques**.

Les critères statiques

Les approches sur critères statiques sont majoritairement basées sur les critères de flot de données et de contrôle. Ils peuvent être séparés en quatre types de couverture :

1. *couverture des données*,
2. *couverture du graphe de flot de contrôle*,
3. *couverture sur les graphes d'états/transitions*,
4. *couverture aléatoire ou probabilistes.*

Dans [CDKP94], les auteurs proposent une technique *pair-wise* [4] en utilisant le **critère couverture de données**, implémentée dans l'outil *Automatic Efficient Test Generator* (AETG). Plus exactement, AETG génère des tests concrets en se basant sur une couverture de flot de données, comme *tous les couples*. Il utilise un algorithme *pair-wise* qui permet de vérifier que chaque couple de données associé à une paire de variables est testé.

Les critères de **couverture du graphe de flot de contrôle** permettent de définir des critères de sélection de tests basés sur l'analyse statique des instructions de branchement d'un programme ou d'un modèle. Les conditions sont des expressions booléennes élémentaires ne pouvant être décomposées. Les décisions sont des expressions booléennes composées par une ou plusieurs conditions liées par des connecteurs logiques. La valeur d'une condition ou d'une décision est donc soit vraie, soit fausse. Les principaux critères

4. La technique de test pair-wise permet de tester toutes les combinaisons possibles pour chaque paire de paramètres d'entrée d'une opération ou d'un algorithme

de couverture structurelle d'un graphe de contrôle, comme discuté dans [UL07], sont aux nombres de six :

1. *Statement Coverage (SC)* : chaque instruction doit être exécutée au moins une fois,
2. *Decision Coverage (DC)* : chaque décision doit prendre toutes les valeurs possibles au moins une fois,
3. *Condition Coverage (CC)* : chaque condition doit prendre toutes les valeurs possibles au moins une fois,
4. *Condition Decision Coverage (C/DC)* : chaque décision et chaque condition doivent prendre toutes les valeurs possibles au moins une fois,
5. *Modified Condition/Decision Coverage (MC/DC)* : chaque décision prend toutes les valeurs 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,
6. *Multiple Condition Coverage (MCC)* : toutes les combinaisons possibles des valeurs des conditions à l'intérieur des décisions doivent être exécutées au moins une fois.

Dans [LBP09], les auteurs proposent une méthode de génération se basant sur le critère de flot de contrôle (critère CC) pour calculer les objectifs de tests et les cas de tests correspondants, à partir du modèle UML/OCL [BGLP08]. Cette approche est implémentée dans l'outil *Test Designer* de Smartesting.

Afin d'obtenir une couverture exhaustive d'un système représenté sous la forme d'un **graphe états/transitions**, il est nécessaire de générer toutes les traces d'exécutions possibles depuis l'ensemble d'états initiaux du graphe. Or ces traces d'exécutions et leur nombre sont potentiellement infinis, dans le cas de l'existence d'un cycle par exemple. D'autres critères de sélection de cas de tests pour ces graphes sont donc nécessaires. Parmi les critères de sélection usuels, on trouve la couverture : d'états, des transitions, des paires de transitions, des k-chemins, des chemins, des chemins indépendants et de la couverture aléatoire.

Ces critères de sélection peuvent être mis en œuvre de différentes manières. Par exemple, le critère de couverture de toutes les transitions peut conduire à la sélection d'un ensemble de tests ayant chacun pour but de couvrir une transition particulière du système d'états/transitions. Mais cette sélection de cas de tests peut également conduire à sélectionner un ensemble de tests minimaux en termes de nombre de pas de tests couvrant l'ensemble des 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 du nombre de pas de tests des tests produits, ou une limitation à un paramètre k du nombre maximum 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].

Le hasard est également considéré **comme un critère de sélection de tests**. Les auteurs dans [Pro03] proposent l'outil *Java Usage Model Builder Library* (JUMBL) qui utilise le critère de sélection de tests aléatoires et probabilistes. Les modèles pour JUMBL utilisent les chaines de Markov, qui possèdent un état unique de départ et de fin. Les arcs des transitions sont labellisés par l'événement qu'ils couvrent et la probabilité d'utilisation dans le système sous test. Ainsi, les premiers tests générés sont ceux qui ont la plus grande probabilité.

Les critères dynamiques

Contrairement aux critères statiques de sélection des tests, les critères dynamiques ne reposent pas uniquement sur l'analyse syntaxique du modèle du système sous test, mais également sur des informations des exécutions visées par l'ingénieur de validation exprimant son objectif de validation en termes d'exécutions particulières :

- la description d'objectifs de test,
- l'approche combinatoire,
- les modèles de fautes.

Les auteurs proposent donc une méthode permettant de formaliser un **objectif de test** sous la forme d'un IOLTS, implémentée dans l'outil *TGV* [JJ05]. L'objectif de test décrit la 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'états *Accept* et *Refuse*. Les états de refus (*Refuse*) servent à couper au plus tôt les traces que nous ne souhaitons pas voir apparaître dans les cas de tests générés. Un nouvel IOLTS est produit par le produit synchronisé effectué entre le modèle du système sous test et l'objectif de test ainsi formalisé. C'est à partir de ce nouveau modèle que les cas de tests sont générés.

Sur ce principe, les auteurs dans [BFG+03] proposent une méthode implémentée dans l'outil *AGATHA*. Deux critères de sélection des chemins d'exécution symbolique sont proposés. Le premier repose sur l'extraction de tous les chemins d'exécution symbolique de taille n, fourni par l'utilisateur. Le second, repose sur la détection d'inclusion d'états permettant de déterminer le plus long chemin d'exécution symbolique sans redondance entre les états parcourus.

Dans [VCG+08], les auteurs proposent une démarche de sélection de tests assez proche des objectifs de test. Le modèle de départ, formalisé dans le langage AsmL ou Spec♯, est transformé en une machine à états finis, servant de base à la génération de tests. Cette approche est implémentée dans l'outil *SpecExplorer*.

Dans [LdBMB04], les auteurs proposent une méthode de génération de cas de tests **basée sur le dépliage combinatoire de schémas de test**, implémentée dans l'outil *TOBIAS*. La méthode propose de décrire des *patterns* de tests sous la forme d'expressions

régulières, ayant comme vocabulaire les appels d'opérations et sous forme de contraintes sur les paramètres de ces appels d'opérations. L'outil TOBIAS assure le dépliage de ces expressions régulières sur les appels d'opérations et les différentes valuations des paramètres dans les domaines définis par le schéma. Du fait de l'absence de modèle pour la génération de tests, un nombre potentiellement grand de tests générés peuvent ne pas être exécutables. Pour cela, dans [MLdB03], les auteurs étudient la connexion entre les outils UCASTING et TOBIAS. L'outil *UCASTING* [VAJ03], qui génère des tests à partir de spécifications en UML, permet d'animer des spécifications et de valuer des séquences d'opérations, non ou partiellement instanciées, à l'aide d'un solveur de contraintes. Ainsi, trois axes ont été étudiés pour connecter TOBIAS et UCASTING :

- validation des séquences de tests complètement instanciées produites par TOBIAS.
- instanciation des valeurs des paramètres à partir de séquences d'opérations, dont les valeurs des paramètres ne sont pas instanciées, produites par TOBIAS.
- complétion de l'instanciation des valeurs des paramètres à partir de séquences d'opérations, dont les valeurs des paramètres sont partiellement instanciées, produites par TOBIAS.

L'étude menée montre que les trois méthodes éliminent les tests non exécutables par TOBIAS. Cependant, la dernière s'avère la plus intéressante. Elle est plus rapide que la première et permet de guider plus finement la génération de tests que la deuxième.

Les auteurs dans [CCB11], d'une manière similaire, utilisent le dépliage combinatoire des scénarios. Ils se basent sur les expressions régulières dans le but d'enrichir les tests produites par l'outil *Test Designer* de Smartesting, qui n'utilise pas de critères dynamiques pour la génération. Ainsi, ils proposent un langage de scénarios, extension des travaux en [MPJ+10], qui peut être utilisé par l'ingénieur de test dans le but d'exprimer des situations complexes dans le système.

Ils existent aussi des approchés basées sur les **modèles de fautes**. 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 produite à l'aide d'un model-checker. Cette trace d'exécution est ensuite utilisée comme objectif de test. Cette approche est implémentée dans l'outil TGV.

2.2 Le test de sécurité

Dans le cadre de la sécurité, les exigences qui définissent comment le système doit se comporter sont appelées exigences *positives*. Celles qui représentent les vulnérabilités du système, sont appelés *négatives* [FB97].

Ainsi, la validation des exigences de sécurité positives est effectuée par le test des fonctions de sécurité. Contrairement à celui-ci, le test des vulnérabilités s'intéresse à l'identifi-

cation des vulnérabilités connues ou inconnues, introduites par les défauts dans le design ou l'implémentation de l'application [TyYsYy10].

Pour répondre à cette problématique, les techniques de MBT offrent la possibilité de diminuer l'expertise nécessaire pour le test de sécurité [FAZB11] sur plusieurs points :

- le modèle de sécurité aide à améliorer la compréhension des aspects de sécurité, ainsi l'aspect métier de la sécurité est capturé au niveau du modèle.
- le niveau d'abstraction est plus grand, ainsi l'aspect sécurité peut être réutilisé pour différents systèmes sous test.
- le modèle peut être utilisé pour la génération automatique de tests.
- les modèles de sécurité embarquent souvent des éléments de risque. Cette information peut être ensuite utilisée pour piloter la génération des cas de tests ou permettre de définir une priorité d'exécution des tests.

Pour essayer de classer les différents type d'exigences de sécurité, de nombreuses classifications ont été proposées. Nous reviendrons sur les grandes catégories qui les composent dans la section 2.2.1. Ensuite, nous présentons les techniques et outils de test de sécurité basées sur les modèles (en anglais *Model Based Security Testing* (MBST)) dans la section 2.2.2.

2.2.1 Formalismes pour l'expression des exigences de sécurité

La sécurité d'un logiciel est définie comme une réaction correcte du logiciel lors de la présence d'attaques ou d'actions mal intentionnées [MP04]. Ainsi, les logiciels qui n'auraient pas validé ces exigences de sécurité possèderaient des vulnérabilités et représenteraient une cible potentielle pour les intrus. De manière générale, les exigences de sécurité peuvent être classés en six catégories :

- *confidentialité* : l'information est disponible seulement aux personnes autorisées.
- *intégrité* : l'information peut être modifiée seulement par les personnes autorisées.
- *authentification* : l'identité de la personne est déterminée avant l'attribution des droits sur l'application.
- *autorisation* : les personnes sont autorisées ou non à accéder à l'application ou à ses données.
- *disponibilité* : l'application et ses services doivent pouvoir répondre à l'utilisateur lors d'une requête.
- *non-répudiation* : une personne ne peut pas effectuer une action et ensuite la nier.

Nous ne nous intéressons pas à l'organisation hiérarchique des ces catégories mais à comment pouvoir prendre en compte ces exigences sous la forme de besoin de tests.

Les auteurs, dans [SGS12], définissent les besoins du test de sécurité à partir de modèles. Pour cela, il faut définir l'architecture et la fonctionnalité des spécifications des systèmes, trouver des vulnérabilités potentielles, des attaques et calculer les probabili-

tés d'apparition de ces attaques. Les probabilités sont identifiées à partir de l'analyse de risque menée par l'expert en sécurité i.e. l'estimation des risques et des conséquences liés à une menace ou un scénario qui permettrait de violer un ensemble d'exigences de sécurité. Pour cela, les auteurs proposent trois types de modélisation pour le MBST :

- *modèles d'architectures et fonctionnels* : ils correspondent aux modèles d'interaction. Ils permettent de définir le comportement du système. Ils décrivent l'évolution des informations liées aux exigences de sécurité. Ces modèles servent de guide pour la génération de tests.
- *modèles de menaces*, fautes et risques : ces modèles contrairement aux précédents permettent une modélisation des risques et de se focaliser sur les exigences négatives. Ils peuvent être représentés par des arbres de fautes, des arbres d'attaques, etc.
- *modèles de faiblesses* et de vulnérabilités : ces modèles décrivent les faiblesses et les vulnérabilités pour lesquelles les informations sont obtenues par le biais des bases de données comme *National Vulnerability Database* (NVD), *Common Vulnerabilites and Exposures*(CVE) ou dans des domaines plus spécifiques comme les applications web avec *Open Web Application Security Project* (OWASP) ou les malwares (tel que malware.lu).

Concrètement, la mise en œuvre de ces approches reste limitée. Il existe une approche basée sur le langage de modélisation UML proposée par J. Jürjens dans [JÖ5]. Elle définit un profil UML appelé UMLsec. Elle permet la vérification de propriétés décrites sous forme d'annotations ajoutées aux diagrammes UML et de *stéréotypes*. Une implémentation sous forme d'un plugin Eclipse est disponible sous le nom CARiSMA [5].

La majorité des approches reste focalisée sur la description et la gestion des politiques de contrôle d'accès [Mou10, XTK+12]. Il existe un grand nombre de modèles de contrôle d'accès [Mou10] : *Role Based Access Control* (RBAC), *Mandatory Access Control* (MAC), *Discretion Access Control* (DAC) et *Organisation Based Access Control* (OrBAC). Afin de les formaliser, plusieurs langages sont proposés, comme *eXtensible Access Control Markup Language* (XACML), JAAS ou EJB. XACML est un langage textuel pour spécifier les politiques. Il n'a pas été construit pour représenter un type bien particulier des politiques, ce qui l'a rendu très populaire à l'utilisation. Pour revenir sur le langage UML, Xu et al. dans [XP06] proposent de modéliser les menaces de sécurité (intrusions) par des *misuse cases* réalisés avec des diagrammes de séquences. À partir de ces cas, ils évaluent si les architectures sont résistantes aux menaces et quels moyens peuvent être introduits afin de les contourner. Les auteurs d'une autre approche [MPJ+10] proposent de générer des tests à partir de modèles écrits en B. Ces tests permettent de valider la politique de contrôle d'accès. La politique de contrôle d'accès est exprimée par des objectifs. Ces derniers sont définis par des expressions régulières et décrivent une séquence d'appels d'opérations sur le système sous test.

Nous allons maintenant nous intéresser aux techniques de tests utilisés pour la sécurité.

5. `http://carisma.umlsec.de/`

2.2.2 Techniques de test de sécurité

Dans [SGR], les techniques de test de sécurité peuvent être séparées en quatre groupes :
- *Vulnerability scanning* : il consiste à utiliser des ordinateurs, appelés scanneurs de vulnérabilités, pour essayer d'accéder à un programme, un système, un réseau ou des applications dans le but de trouver des faiblesses connues.
- *Static application Security Testing* (SAST) : il consiste à tester les applications en utilisant les informations contenues dans le code source, le byte code ou les binaires.
- *Monitoring* : il consiste à observer (en anglais *monitor*) les entrées et les sorties des événements du système lors son exécution. Ces événements sont enregistrés dans une trace. L'analyse de cette trace permet de décider si l'application est conforme aux exigences de sécurité.
- *Dynamic Application Security Testing* (DAST) : il consiste à tester des vulnérabilités de l'application en observant son comportement au travers des réponses retournées. Les techniques faisant partie de cette catégorie sont le MBST, le *fuzz testing* et le test de pénétration.

Nous nous intéressons plus particulièrement à ce dernier point. Les techniques du MBST incluent le test de sécurité fonctionnel, le test de vulnérabilités qui sont orientés sur le risque et les menaces, le fuzzing basé sur les modèles [6], ainsi que le test à partir de patterns.

Ainsi, nous retrouvons les travaux liés à l'expression des exigences de sécurité. En se basant sur la vérification des exigences par stéréotypes avec UMLsec, J. Jürjens dans [JÖ8] propose de générer des traces d'exécutions par injection de fautes. Ces traces mènent aux fautes introduites et ensuite sont utilisées pour la génération des tests de sécurité.

D'autre part, les travaux sur les exigences comme l'authentification ou le contrôle d'accès sont grands. Les auteurs dans [XTK+12] proposent une technique de test de politiques de contrôle d'accès en modélisant les menaces par des *misuse cases*.

Les auteurs dans [MPJ+10] proposent la stratégie de génération de tests guidée par le critère dynamique objectifs de test (nommée aussi *test purposes*) qui exercent la politique de sécurité, les objectifs étant représentés par des expressions régulières. Les formalismes utilisés par Cabrera et al. [CCB11] et par l'outil de génération de tests Smartesting CertifyIt (cf. section 2.3.3), sont aussi basés sur ces expression régulières.

La mutation de protocoles, représentés par des arbres de fautes, est utilisé pour la génération de traces et la vérification des protocoles comme dans [DHK11]. Les auteurs

6. Le fuzzing basé sur les modèles (en anglais *model-based fuzzing*) utilise les données et comportements du système issus de modèles et applique la mutation (de protocoles, modèles de données) dans le but de réduire le nombre de tests générés.

définissent l'arbre de fautes pour le protocole de sécurité dans le langage HLPSL [7] et utilisent l'outillage de vérification de protocoles d'AVISPA. Ils réalisent une mutation du protocole et le vérifient en produisant des traces d'attaques qui peuvent être utilisées comme des cas de tests, si elles sont déclarées comme dangereuses.

Dans [SGS12], les auteurs illustrent la méthodologie prévue par le projet *Developement and Industrial Application of Multi Domain Security Testing Technologies* (DIAMONDS). Le projet développe des méthodes pour définir les objectifs, les tests de sécurité basés sur les spécifications pour prendre en compte les valeurs de risques. Ainsi, ils ciblent le MBST basés sur l'analyse de risque et le *model-based fuzzing*.

Felderer et al. [FAZB11] proposent une classification de ces approches DAST. Ils prennent en compte l'automatisation des techniques et la notion de risque. Cela donne un découpage en six catégories de ces techniques :

1. *Individual knowledge* : la connaissance est individuelle. Il n'y a pas d'utilisation de modèle pour identifier les valeurs du risque. L'expertise des ingénieurs métiers est la seule considérée pour créer les tests de sécurité.
2. *Adapted risk based testing* : les modèles de sécurité sont inexistants mais le risque est identifié et modélisé. Il est ensuite possible d'ordonner par priorité l'exécution des tests générés [Aml00].
3. *Scenario-based MBT* : les exigences sont exprimées par des scénarios et sont appliquées aux modèles (par example diagrammes UML). La stratégie de génération se base sur le critère de couverture des flots de contrôle. Un tel exemple est *Telling Test sorties* (TTS) [FABA10], une approche partiellement automatisée. TTS propose une approche pour la génération automatisée de tests, mais aussi la définition manuelle des *test stories*, définis par des diagrammes de séquences en UML.
4. *Risk enhanced scenario-based MBT* : les exigences sont également exprimées par des scénarios, comme la catégorie précédente, mais les valeurs de risque sont intégrées dans les modèles. Les auteurs [FAZB11] planifient d'intégrer le risque à la méthode TTS.
5. *Adapted MBT* : les modèles ne sont pas initialement prévus pour gérer la sécurité mais ils sont enrichis pour cela. Nous pouvons citer, par exemple, l'ajout de stéréotype ou de critères dynamiques. Le risque n'est pas intégré dans le modèle, mais il est possible de générer automatiquement des tests à partir des modèles [MPJ+10]. Ce groupe est très proche du groupe scenario-based MBT.
6. *Automated risk-based security test generation* : les valeurs de risque sont annotés dans les modèles et une génération automatique de tests de sécurité est possible. Les auteurs dans [SMP08] proposent l'outil RiteDAP pour la gestion des risques au niveau des modèles et la génération de tests, mais pas pour les tests de sécurité.

7. Le langage de spécification de protocoles *High Level Protocol Specification Language* (HLPSL) est issu du projet européen AVISPA (2003-2006, adresse web : http ://avispa-project/org)

Cependant, l'approche proposée par Zech dans [Zec11] propose de générer automatiquement des tests de sécurité en se basant sur des patterns d'attaques et des profils de menaces.

Nous allons vous présenter l'approche que nous avons utilisé pour le test de sécurité. Elle utilise l'outil Smartesting et des schémas (pour traduire en besoin de tests les exigences de sécurité). Cette approche se classe dans la catégorie intermédiaire *Automated Security Test Generation* puisqu'elle permet une génération automatisée des tests de sécurité. Plus particulièrement, elle se classe en tant que *Adapted MBT* parce qu'elle ajoute la notion de sécurité dans les modèles de test à l'aide des schémas.

2.3 Outillage Smartesting pour la génération de tests

Le processus MBT, tel que nous l'avons décrit précédemment, est implémenté dans l'outil industriel Smartesting CertifyIt, successeur de l'outil Test Designer. Tout d'abord, un ingénieur de test prend en entrée les exigences de la spécification afin de créer le modèle du système sous test. Ce modèle est défini sur un sous-ensemble de UML, nommé UML4MBT. Ce sous ensemble prend en compte les diagrammes de classes, d'objets, d'états/transitions enrichis avec des annotations écrites en OCL. Il est détaillé par Bouquet et al. [BGL+07] et nous en donnerons un aperçu dans la section 2.3.1. Ensuite, le modèle de test est passé au générateur de tests afin de produire automatiquement des cas de tests. Le générateur de tests réalise une couverture systématique des comportements (ou encore des *objectifs de tests*) [LBP09]. Les objectifs de tests sont calculés à partir du modèle comportemental, suivant le critère de sélection de tests *CC*. En parallèle, une matrice de couverture reliant les tests et les cibles de tests aux exigences dans le modèle est établie (cf. section 2.3.2). Les tests peuvent être ensuite exportés, publiés dans un référentiel de tests pour une gestion future comme par exemple exécuter les tests sur le SUT. Après l'exécution des tests, les résultats et les métriques peuvent être renvoyés au référentiel de tests.

2.3.1 Diagrammes UML4MBT

Le sous-ensemble d'UML que nous utilisons est basé sur trois diagrammes : *diagrammes de classes* (permettant de modéliser des points de contrôle et d'observation du système sous test - ou encore SUT, pour System Under Test), *diagrammes d'objets* (permettant de définir l'oracle des tests) et des *diagrammes d'états/transitions*. Ces diagrammes sont enrichis avec le langage OCL permettant de modéliser les comportements dynamiques du SUT.

Diagrammes de classes

Les diagrammes de classes représentent la vue statique du modèle. Ils décrivent les objets abstraits du système et leurs dépendances. Les dépendances sont représentées par des *associations* binaires ou réflexives entre les classes.

Il est à noter que l'héritage d'objets n'est pas pris en compte. Les *attributs* de classe permettent de décrire la structure d'un objet. Ils sont caractérisés par une valeur par défaut et un type (les types supportés sont les entiers, les booléens et les énumérés - une *énumération* étant une classe composée que de littéraux).

Enfin, les opérations modélisent les actions effectuées par l'objet. Elles peuvent être définies avec ou sans paramètre d'entrée ou de sortie. Comme pour les attributs, elles peuvent avoir les types : entier, booléen ou énuméré. Elles ont des pré- et des post-conditions écrites en OCL. Ici, nous pouvons observer une double interprétation du langage OCL, plus précisément, en pré-condition d'une opération en tant que langage de contraintes, exprimant la condition d'exécution de l'opération et en post-condition en tant que langage d'action, exprimant un comportement de l'opération.

Diagrammes d'objets

Le diagramme d'objet donne la liste exacte des objets utilisés pour calculer les séquences des tests et permet de définir l'état initial du système. Chaque objet du diagramme est une instance d'une classe et ne peut pas être créé dynamiquement par les actions décrites dans le modèle lors de la génération des tests. La création des objets du système est simulée par la création des liens entre les objets UML, nommés des instances d'associations.

Diagrammes d'états/transitions

Le diagramme d'états/transitions modélise le comportement dynamique du système. Il est défini en tant qu'automate d'états finis, avec un état initial et un état final optionnel. Il est composé d'*états simples* (représentant d'états différents dans le système) et de *transitions* (utilisées pour modéliser les actions).

Une transition peut être : externe (entre deux états) ou interne (ne change pas d'état). Une transition est la réponse à un événement qui provoque un changement d'état du système. Elle est définie par le triplet : événement, garde et action. L'*événement* ou encore *trigger*, est déclaré comme une opération du diagramme de classes. La *garde* ou encore *guard* est une expression booléenne à satisfaire afin d'activer la transition. Enfin, une *action* est exécutée si la garde est satisfaite. L'action peut agir directement ou indirectement sur les objets visibles à partir de l'objet utilisé.

2.3.2 Traçabilité des exigences

Une des valeurs ajoutées par le MBT est la traçabilité entre les exigences et les tests [BLH09]. Plus exactement, il doit être possible de lier les exigences aux éléments de modélisation. Ces derniers sont eux-mêmes liés aux tests. Dans l'approche Smartesting, ce lien est assurée via les mot clefs *@REQ* et *@AIM*. Ils sont utilisés au sein du code OCL.

Ainsi, le test est défini en tant qu'une séquence de pas. À chaque pas, un appel d'opération issue du modèle est associé. Pour chacun de ces pas, l'ensemble des exigences couvertes est identifié. Un test est défini de la façon suivante.

Définition 1 (Test) *Un test* tc *est un quadruplet* $\langle \mathcal{M}, seq, targets, TAGS \rangle$*, où :*
- $\mathcal{M}$ *est un modèle UML4MBT à partir duquel les tests sont calculés,*
- *seq est l'expression syntaxique du test, comme une séquence de pas, et chaque pas est défini comme :* $\vec{o}, s' \leftarrow op(\vec{i}, s)$ *avec*
  - $op(\vec{i}, s)$ *est l'appel de l'opération op et* $\vec{i}$ *le vecteur des paramètres d'entrée,*
  - s *est l'état courant du système à partir duquel est appelé l'opération,*
  - s' *est l'état obtenu après l'exécution de l'opération,*
  - $\vec{o}$ *est le vecteur des paramètres de sortie de l'opération,*
- *targets sont les cibles de test couvertes par le test (transitions d'un diagramme d'états/transitions ou un comportement d'une opération à activer), et*
- *TAGS est l'ensemble d'exigences couvertes par le test.*

2.3.3 La prise en compte des exigences de sécurité

Durant le projet SecureChange, Smartesting a proposé une extension des critères de sélection de tests pour la prise en compte des exigences de sécurité à partir de schémas. Les schémas permettent de générer des tests de conformité liés aux exigences de sécurité dont la partie fonctionnelle est décrite dans le modèle. Nous présentons ce qui se cache derrière la notion de schéma.

Les schémas

Les schémas sont la définition formelle des objectifs de test pour les exigences de sécurité (en anglais *Security Test Objectives*). Ils permettent de définir des objectifs à partir du langage de définitions des cibles de test (en anglais *Test Purpose Definition Language*). Ce formalisme est basé sur les transitions du système entre différents états. Chaque schéma est déplié en plusieurs spécifications de cas de test (en anglais *Test Case Specification*, notée TCS). Celles-ci sont utilisées en tant que critère dynamique pour guider la génération de tests. Cette approche reprend la démarche MBST que nous avons présentée dans la section précédente.

Le langage de schémas est basé sur les travaux initiés durant le projet RNTL POSE, dédié au test de conformité du système à une politique de sécurité [MPJ+10]. Sa conception a été guidée par l'expérience des ingénieurs de sécurité, pour ainsi permettre d'avoir un langage dédié à la description des besoins de tests associés aux exigences de sécurité.

L'ensemble du langage est présenté en Annexe D, page 173. Nous y exposons les règles de grammaire dans le tableau D.1 et les symboles terminaux dans le tableau D.2. À noter qu'un éditeur dédié à la gestion des schémas a été créé pour IBM Rational Software Architect (RSA) [Sec12b].

Génération de tests à partir de schémas

Dans la figure 2.2, nous illustrons le processus global du MBST à partir de schémas.

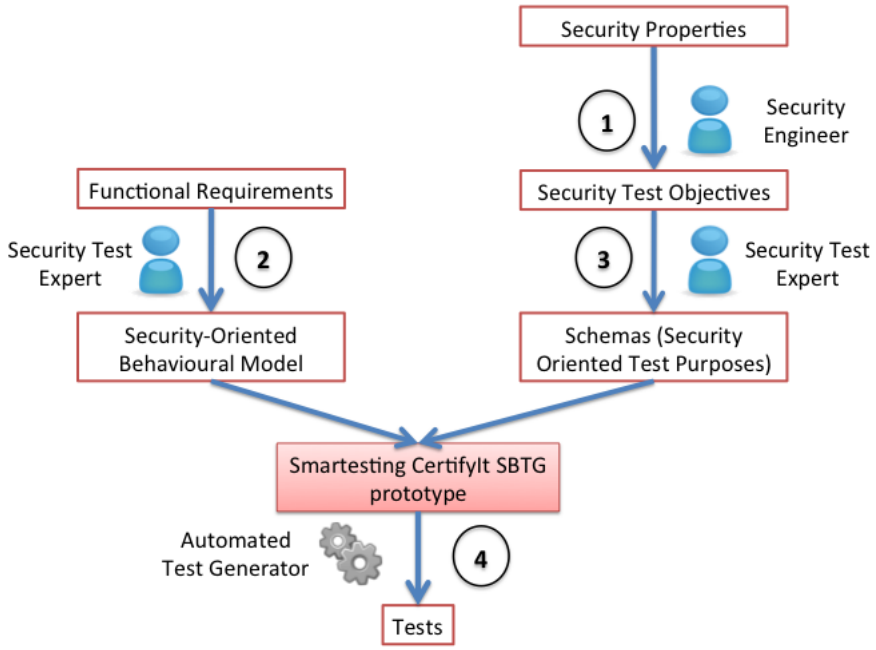

FIGURE 2.2 – Processus MBST à partir de schémas [Sec12b]

.

Ce processus se déroule en quatre étapes principales :

① Définir les objectifs de test de sécurité (prenant en compte les exigences de sécurité). Cette étape est effectuée par l'ingénieur de sécurité.

② Création du modèle comportemental. Cette étape est effectuée par l'ingénieur de test de sécurité.

③ Définir les schémas pour la génération de tests. Cette étape est effectuée par l'ingénieur de test de sécurité.

④ Génération de tests de sécurité automatisée. Cette étape est effectuée par un automaticien de tests.

À partir de l'analyse des exigences de sécurité, l'ingénieur de sécurité définit les objectifs de test nécessaires pour valider les exigences de sécurité. Cette phase prépare des objectifs de tests de sécurité d'une manière détaillée mais informelle. Dans la phase de modélisation, il n'y a plus une mais deux activités. En effet, en plus de la création du modèle pour définir le comportement du système, la prise en compte des objectifs de tests de sécurités sont aussi formalisés pour devenir des besoins de tests sous la forme de schémas.

Une fois cette activité accomplie, le moteur de génération de tests permet de créer des séquences de tests dédiées aux besoins de tests issus des schémas et utilisant les comportements du système exprimé dans le modèle. Plus exactement, les schémas vont être transformés en TCS et ces TCS vont être pris en compte par le moteur de génération de tests. Du point de vue des algorithmes de génération de tests, il ne manipule qu'un seul type d'information le modèle et les TCS à couvrir, le résultat étant la séquence et les éléments de traçabilité.

Nous avons présenté l'approche Smartesting telle qu'elle a été développée dans le cadre du projet *SecureChange*. Dans un cadre plus général, cette méthode peut être utilisée pour un autre usage. En effet, les schémas permettent de définir des enchainements d'états et d'opérations avec des trous à combler par les algorithmes de génération. Cela rend l'approche indépendante de l'aspect sécurité. Nous faisons le choix de rester fidèles à cette première utilisation de l'approche même si nous sommes conscients qu'elle est beaucoup plus générale. Nous reviendrons sur cet aspect dans les perspectives de nos travaux.

2.4 Synthèse

Dans ce chapitre, nous avons introduit la génération de tests à partir de modèles, plus particulièrement UML/OCL. Nous avons également détaillé les différents critères de sélection pour la génération de tests, dans le but de positionner la démarche de génération de tests utilisée par l'outil Smartesting CertifyIt, qui sera le départ de notre méthodologie.

Nous avons vu, que leur méthodologie proposée pour le test de sécurité se positionne au niveau des techniques DAST. Plus exactement, le fait qu'elle apporte les éléments de risque par le biais des schémas, permet de la classifier comme une technique *Automated Security Test Generation et Adapted MBT*.

Dans le chapitre suivant, nous nous intéressons à la problématique liée à l'évolution des exigences ou du système sous test (implémentation, technologie, environnement). Plus particulièrement, comment faire évoluer et/ou sélectionner les cas de tests de façon efficace. Cette problématique est traitée par des approches de test de non-régression mais elle est plus large.

Chapitre 3

Test de non-régression

Sommaire

3.1 Classification des techniques de non-régression 34
3.1.1 La classification générale 34
3.1.2 Classification des techniques sélectives 35
3.2 Panorama des techniques sélectives 35
3.2.1 À partir de modèles 36
3.2.2 Cadre d'évaluation des techniques 37
3.3 La prise en compte de l'aspect sécurité 38
3.4 Synthèse 39

Les applications qui évoluent doivent être validées avant leur redéploiement sur le marché et d'autant plus si ces dernières sont des applications critiques. Le processus de validation prend en compte le test du code avant la livraison du produit. Le processus permet d'assurer que les changements dans le code, suite à l'évolution des exigences décrites dans le cahier des charges, n'ont apporté aucun impact sur le code dit *sain*, s'appelle le **test de non-régression**. Le test de non-régression peut être utile aux différents niveaux du test : unitaire, intégration ou système.

Cependant, il est reconnu en tant qu'activité très coûteuse dans la maintenance des logiciels [8]. Les coûts sont estimés à presque la moitié des coûts totaux de la maintenance [LW89, Kap04, BMSS11]. Les auteurs dans [DMTR10] donnent des chiffres de coût du test de non-régression mené sur leur étude. Ils l'ont estimé à 1 000 machines/heure pour exécuter 30 000 tests fonctionnels et une centaine d'hommes/heure dépensés par les ingénieurs de tests pour effectuer le test de non-régression i.e. préparation des exécutions,

8. La maintenance logiciel consiste en l'amélioration d'un logiciel dans le but d'apporter des nouvelles fonctionnalités ou de corriger des erreurs.

monitorage de l'exécution, analyse des résultats et maintenance des ressources des tests.

Afin de se positionner par rapport aux challenges dans le test de non-régression vis à vis de l'état de l'art courant et en même temps avoir une vue globale des techniques, nous allons les détailler dans la section 3.1. D'autre part, le MBT offre un très grand potentiel d'automatisation et d'adaptation du processus de test. Il est très bien adapté pour tester dynamiquement les systèmes qui évoluent. Ceci provient du fait qu'il est d'autant plus facile de prédire l'impact de l'évolution au niveau modèle qu'au niveau code. Ainsi, dans la section 3.2, nous allons détailler les techniques de test de non-régression basées sur les modèles. Enfin, dans la section 3.3, nous allons discuter dans quelle mesure les exigences de sécurité sont prises en compte dans l'industrie et lors d'une évolution du système.

3.1 Classification des techniques de non-régression

Nous allons présenter la classification générale dans la section 3.1.1 et la classification des techniques de non-régression sélectives dans la section 3.1.2.

3.1.1 La classification générale

Les techniques de test de non régression peuvent être basées sur l'analyse directement du code ou au niveau d'abstraction, basées sur le modèle. Cette dernière catégorie, est considérée comme *propre* en terme des modifications, puisqu'elles sont explicitement définies à partir du cahier des charges. Il existe une complémentarité entre ses deux types. Néanmoins, Yoo and Harman [YH10], Biswas et al. [BMSS11], ainsi que Briand et al. [BLH09] soulignent les bénéfices d'une méthode basée sur les modèles :

- ils permettent la traçabilité entre les exigences/la spécification et l'activité de test.
- en terme de scalabilité, les techniques basées sur les modèles permettent de mieux travailler avec des systèmes de très grande taille.
- ils sont indépendants du langage de programmation. Il est très difficile d'implémenter une technique générale pour différents langages de programmation. Cependant, l'utilisation de modèles permet de surpasser ce problème.

Indépendamment des sources sur lesquelles les techniques se basent, Harrold et al. dans [RHG+01] les classifient en deux grandes familles de test de non-régression :

- *tout sélectionner* (en anglais *retest-all*) : qui demande l'application de toute la suite de tests avant l'évolution. Dans certaines classifications, nous retrouvons cette stratégie en tant que technique de sélection.
- *sélective* : qui demande de sélectionner un sous-ensemble de tests de la suite de tests avant l'évolution selon différentes techniques.

Nous pouvons constater que la première est une technique assez naïve. Même si elle est souvent utilisée dans l'industrie, les limites lors de l'analyse des régressions sont très vite atteintes. Pour cela, une grande attention est portée au techniques sélectives.

3.1.2 Classification des techniques sélectives

Dans la littérature, plusieurs auteurs ont produit des études de techniques sélectives de non-régression en les catégorisant de différentes manières [RHG+01, ESR08, YH10, BMSS11]. Cependant, la plupart d'entre elles se réfèrent aux catégories définies par Harrold et al. [RHG+01] :

1. *techniques de minimisation* : choisir le nombre minimal de tests depuis la suite de tests d'origine, ceux qui passent par les instructions modifiées ou affectées du programme. Par exemple, Fischer et al. [KFA81] utilisent un système d'équations linéaires pour exprimer le lien entre les tests pour que chaque élément du programme soit exercé par au moins un test.

2. *techniques basées sur la couverture* : choisir les tests en tenant compte d'une part de la couverture des transitions et des chemins dans le programme par les tests et d'autre part de la couverture en utilisant le graphe de dépendances.

3. *techniques sûres* (en anglais *safe techniques*) : une technique est dite sûre si elle a sélectionné tous les tests de la suite de tests initiale qui peuvent révéler des fautes dans le système. La plupart des techniques de minimisation et de flots de données ne sont pas sûres. Cependant, il existe plusieurs travaux dans ce domaine : Laski et Szermer [JW92], Rothermel et Harrold [RH97] et Vokolos and Frankl [VF98]. Cette technique est utilisée et détaillée par Rothermel et Harrold dans [RH96], et implémentée dans l'outil nommé *DejaVu*.

4. *techniques aléatoires* (en anglais *ad-hoc random techniques*) : sélection au hasard d'un nombre de tests prédéfinis de la suite de tests du programme. Cette technique est souvent utilisée quand il manque du temps pour les tests.

3.2 Panorama des techniques sélectives

Dans cette section, d'après la catégorisation des techniques sélectives, nous allons donner un panel des techniques sélectives basées sur les modèles dans la section 3.2.1 et nous allons présenter le cadre d'évaluation des techniques de non-régression dans la section 3.2.2.

3.2.1 À partir de modèles

Les solutions MBT pour le test de non-régression dépendent du type de modèle et de ou des techniques choisies. Dans Tahat et al. [TBVK01], nous retrouvons une étude sur les SDL [9] systèmes et la notion d'exigence pour la génération des tests de non régression. Le but est de couvrir tout le système avec ces tests.
Donc, dans le but d'exprimer les exigences, nous pouvons utiliser des fragments SDL individuels du système. Ensuite, nous pouvons les rassembler en un seul système SDL. À partir de là, nous pouvons le transformer en modèle EFSM [10], qui va être l'entrée pour la génération des tests boîte-noire. Le langage descriptif des *EFSM* est donc utilisé pour repérer plus facilement l'évolution du système et mapper les exigences qu'ils utilisent. De cette manière ils ont défini des règles pour l'addition, la suppression et la modification d'une exigence, qui peut être interne ou suite à un changement d'une autre transition.

Dans Korel et al. [KHV02], nous retrouvons une méthode qui est un complément de la précédente mais plus précise car elle utilise l'analyse des dépendances (Data et Control Dependencies). La sélection de tests de la suite d'origine est effectuée selon des *Patterns*. Les *Patterns* sont créés selon les différents effets : l'effet du modèle sur la modification, la modification sur le modèle et autres effets secondaires causés par la modification. Un test est inclus dans la RTS (suite de test de non régression) si au moins un des patterns d'interaction n'existe pas dans la sélection des tests.

D'autre part, Chen et al. [UPC07] décrivent une analyse des dépendances beaucoup plus approfondie définissant les règles et les dépendances pour chaque modification élémentaire. Au total, nous retrouvons douze dépendances différentes. Ici, le mot *dépendance* est très proche de la signification du *Pattern* défini par Korel et al. La suite de test réduite contient les tests vérifiant les effets directs et indirects du modèle modifié, ayant pour but de couvrir toute les nouvelles dépendances par modification.

Récemment, nous retrouvons des études qui portent en partie sur le langage unifié de modélisation *UML*. Plus exactement, Briand et al.[BLH09] font une étude des tests de non régression sur les diagrammes de classes, les cas d'utilisation et les diagrammes de séquences. La méthode propose dans un premier temps de définir les différences entre les deux modèles. Ensuite, ils classifient les tests d'après la classification de Leung et White [LW89] en trois catégories : tests obsolètes, tests re-testables et tests réutilisables. Finalement, ils ont créé un outil pour des programmes en langage **C** qui automatisent cette méthode, nommé *RTSTool*. C'est ici que nous retrouvons l'un des nos objectifs prenant en compte le cycle de vie des tests, qui va nous servir par la suite dans le développement de notre méthode.

9. Specification Description Language
10. Extended Finite State Machines

La modélisation pour le test en UML, est très répandue et trouve son utilité dans l'industrie [UPL12]. Ainsi, plusieurs techniques de non-régression sont créées pour les modèles UML. Gorthi et al. [GPCL08] ont proposé une technique basée sur les diagrammes d'activités. Ils utilisent le *slicing* de comportements afin d'identifier les changements dans le diagramme. À chaque fois qu'une exigence est modifiée le diagramme d'activités est modifié dans le but de refléter l'évolution (le changement). Ainsi, chaque chemin dans le diagramme qui contient des noeuds modifiés est considéré comme chemin affecté par le changement. Ainsi, les tests qui exécutent ces chemins affectés sont sélectionnés pour le test de non-régression.

Naslavsky et Richardson [NR07], en se basant sur le flot de contrôle du diagramme de séquences, créent une traçabilité entre les cas de tests et le diagramme de séquences, ce qui permet d'identifier quels éléments du flot de contrôle sont exécutés par un test donné. Les flots de contrôle des deux systèmes sont analysés pour identifier les éléments affectés par le changement et la traçabilité est utilisée pour sélectionner des tests.

Farooq et al.[FIMN07, FIMR10] présentent une technique basée sur le UML 2.1, en utilisant des diagrammes d'états/transitions comportementales et des diagrammes de classes pour la sélection de tests. Leur technique permet de retrouver directement ou indirectement l'élément affecté en UML, en utilisant l'information de la classe ou du diagramme d'états/transitions modifié i.e une transition est considérée comme modifiée si elle utilise un attribut ou une méthode modifiée dans son action, garde ou événement. Les tests qui couvrent ces transitions sont considérés comme des tests *retestables*.

3.2.2 Cadre d'évaluation des techniques

Rothermel et Harrold dans [RH96] définissent un cadre pour évaluer les techniques de test de non-régression :

- *inclusive ou sûre* : une technique est sûre si elle a sélectionné tous les tests dont les résultats (ou encore l'oracle) d'exécution sur le nouveau système sont différents. Ces tests sont aussi appelés *modification-revealing*.
- *précise* : une technique est précise lorsqu'elle ne sélectionne que des tests qui produisent un oracle différent après l'exécution sur le nouveau système.
- *efficace* : pour ce critère il y a plusieurs facteurs à prendre en compte. Le premier facteur est la sélection de tests lors du cycle de vie de la technique elle-même. Elle peut être appliquée lors d'une phase préliminaire i.e. correction du logiciel après une livraison logiciel intermédiaire. La deuxième c'est la phase critique, qui commence après les corrections du logiciels. Ici le test de non-régression est limité en temps, souvent borné par le temps de livraison du logiciel. La minimisation du coût du test de non-régression est très important à ce moment là. Ensuite le deuxième facteur est l'automatisation de la technique. Le troisième facteur est la capacité de la technique

à calculer les modifications vis à vis de la nouvelle version. Enfin, le dernier facteur est que la technique soit applicable sur plusieurs modifications à un instant donné.
– *générale* : une technique est dite générale quand elle est applicable dans plusieurs situations. Plusieurs facteurs doivent être pris en compte tel que le fonctionnement sur plusieurs cas pratiques, prise en compte des cas réalistes, un nombre minimal d'assomptions et indépendance d'outils d'analyse de programmes.

Les techniques [FIMN07, FIMR10, BLH09], sont des techniques sûres d'après la structure d'évaluation de Rothermel et Harrold [RH96].

Enfin, beaucoup de ces techniques, ainsi que d'autres basées sur les modèles ont été détaillées dans les études [RHG+01, ESR08, YH10, BMSS11] où les auteurs montrent l'efficacité, la sûreté, la précision de telles techniques et soulignent le besoin de modéliser pour tester la non-régression.

3.3 La prise en compte de l'aspect sécurité

Durant ces dernières années, nous avons pu constater que la recherche dans le domaine de la validation et de la vérification des exigences de sécurité des applications critiques montre une très forte augmentation. Une grande attention est portée aux propriétés de sécurités, les politiques de contrôle d'accès, les vulnérabilités etc, que nous appelons exigences de sécurité, tel que nous l'avons détaillé en section 2.2. Nous sommes arrivés à un stade dans le développement de systèmes critiques matures et stables et lors d'une évolution qu'il est important de garantir la continuité de cette stabilité. Nous nous rendons compte que nous souhaitons préserver ces exigences de sécurité lors de l'évolution du système.

Beaucoup de travaux portent sur le test de non-régression pour les fonctionnalités du système, ainsi, Yoo et Harman [YH10] mettent l'accent sur le besoin de continuer l'investigation du domaine de test de non-régression pour des exigences non fonctionnelles (dont les exigences de sécurité font partie). D'après nos connaissances, très peu de travaux, sont dirigés dans le sens de la re-validation des exigences de sécurité lors d'une évolution du système. Ce processus nous l'appelons test de non-régression pour la prise en compte de la sécurité ou encore *Security Regression Testing (SRT)*.

Nous retrouvons l'évocation du besoin de test de non-régression pour la sécurité dans les travaux sur les méthodologies agiles [AGA10, Kon06]. Dans [Kon06], l'auteur propose d'utiliser des *misuse stories*, contraires aux *user stories* et les considère comme une possibilité de prendre en compte les exigences liées à la sécurité. D'autre part, il propose d'utiliser les tests dédiés à ces *misuse stories* pour s'assurer que des régressions de sécurité n'ont pas été introduites. Mais, aucune technique en particulier à utiliser n'est précisée à ces propos. Les études menées dans les projets sur le *System Developpement Life Cycle*

(SDLC) de OWASP [11], tel que détaillé en [Meh], surlignent l'importance du test de non-régression lors de la vérification des vulnérabilités du système. Ces études permettront de mieux analyser les problématiques de sécurité pour les créations des futures systèmes.

D'autre part, nous retrouvons des travaux récents par Hwang et al. [HXEK+12]. Ils proposent trois techniques pour les politiques de contrôle d'accès, spécifiés en XACML et la sélection de tests de non-régression lors de l'évolution des politiques basées sur : (1) la mutation des politiques de contrôle d'accès, (2) leur couverture et (3) évaluation des enregistrements de demandes.

La première technique sélectionne des règles r_i, issues d'une politique P et crée des mutants de la police, notés $M(r_i)$ en changeant la décision sur r_i. Cette technique sélectionne des tests qui révèlent des comportements différents de la politique en exécutant les tests sur le programme en interaction avec la politique P et ses mutants $M(r_i)$. Elle est très coûteuse parce qu'elle exécute un test $2xn$ fois, n étant le nombre de règles dans une politique.

La deuxième technique observe les règles de la politique qui sont couverts par les tests en exécutant les tests sur le programme, lié à la politique de contrôle d'accès, et établie une corrélation, comme la technique précédente, entre les règles et les tests.

Enfin, la dernière technique enregistre les demandes des points de vérifications de la politique lors de l'exécution du test sur le programme. Ensuite, les tests qui encapsulent les différentes décisions pour la politique P et la politique modifiée P' sont sélectionnés.

3.4 Synthèse

Dans le processus MBT que nous considérons (cf. section 2.3), l'évolution n'est pas prise en compte i.e. lors de l'évolution du modèle l'ingénieur de validation est obligé de régénérer tous les tests à partir du nouveau modèle. Le référentiel de tests est ensuite mis à jour avec les nouveaux tests. Ceci est un problème d'efficacité et de coût dans l'analyse des tests de non-régression de très grands systèmes industriels. Tel que précisé par Jiang et al. dans [JTG+10] la régénération complète de la suite de tests d'un modèle du projet Microsoft de test de documentations de protocoles *(protocol documentation test project)* peut prendre plusieurs heures voire même une journée. Ensuite, l'ingénieur vérifie toutes les parties non affectées. Cette étape peut prendre plusieurs jours, voire même semaines en fonction de la grandeur du modèle.

Une autre possibilité est d'utiliser la stratégie *retest all*, ce qui présume que les fautes peuvent être introduites n'importe où dans le système. Néanmoins, cette stratégie ne fait aucune différence entre une faute classique ou une faute suite à l'exécution d'un cas de test obsolète. Le juste milieu entre ces deux approches est de ne pas considérer que toutes

11. www.owasp.org

les parties dans le système sont impactées i.e. d'utiliser une approche sélective.

Ainsi, notre objectif est d'éviter la régénération complète des tests, en effectuant une analyse des évolutions dans le modèle et ensuite classifier les tests issus de la suite de tests initiale. D'après Leung et White dans [LW89], le processus de maintenance des systèmes qui évoluent demande une catégorisation des tests en : *obsolete, reusable and re-testable* par rapport aux changements et ensuite mise à jour du référentiel de tests. Suite à cela, nous souhaitons distinguer : (*i*) les éléments qui n'ont pas changé (ceci ne demande à priori aucune régénération de tests), (*ii*) les éléments qui ont changé (ceci demande une nouvelle génération de tests spécifiques pour ces éléments ou cela demande une modification de la version précédente du test) ;

Pour cela, nous proposons une méthodologie et un outil qui permet la gestion d'une ou plusieurs évolutions du système, pour les exigences fonctionnelles (cf. chapitres 6 et 7) et de sécurité (cf. chapitre 8). Dans ce cas, l'ingénieur de validation met à jour le modèle à partir des exigences issues de la nouvelle spécification. De nouveaux tests sont produits en appliquant la génération sélective de tests (*Selective test generation method*, notée aussi SeTGaM). Ensuite, un composant nommé *Smart Publisher* est utilisé pour stocker et organiser les tests. Le référentiel de tests est mis à jour avec ces tests ainsi que leur statut pour la nouvelle version, permettant une traçabilité des cas de tests entre les différentes versions. La méthode que nous proposons est indépendante de l'outil de génération de tests, même si l'outillage que nous proposons est associé à l'outil Smartesting CertifyIt.

Chapitre 4

Exemple fil rouge

Sommaire

4.1 Introduction à l'application eCinema 42

4.1.1 Exigences fonctionnelles 42

4.1.2 Exigences de sécurité pour eCinema 45

4.2 Génération de tests pour eCinema 47

4.2.1 Tests fonctionnels 47

4.2.2 Tests dédiés à la sécurité 49

4.3 Évolution de la spécification d'eCinema 51

4.3.1 Description de l'évolution 51

4.3.2 Évolution de modèle 52

4.4 Synthèse 53

Dans ce chapitre, nous présentons l'exemple "fil rouge" qui sera utilisé dans le manuscrit pour expliquer et illustrer nos algorithmes, méthodes et outils. Cet exemple s'appelle *eCinema* et possédera deux versions. Pour chacune d'elles, nous aurons le cahier des charges et une implémentation. Nous avons décomposé ce chapitre en trois parties. La première décrit la spécification pour la 1ère version d'*eCinema* avec ses exigences fonctionnelles et de sécurité. Pour permettre d'illustrer l'ensemble des approches proposées dans cette thèse, nous avons réalisé deux modèles de tests, un avec un diagramme d'états/-transitions et un sans. Ces deux modèles de l'application *eCinema* couvrent les mêmes éléments de la spécification. Ces deux modèles sont présentés dans cette première section. La deuxième section détaille les tests obtenus par générations de tests pour couvrir les exigences décrites précédemment. Une troisième section présente une évolution de la spécification et les modèles correspondants.

4.1 Introduction à l'application eCinema

L'application *eCinema* est un site-web dont l'objectif est la réservation/l'achat de tickets pour des séances de cinéma. Plus exactement, au travers de cette application l'utilisateur a la possibilité d'acheter des tickets, d'afficher les tickets achetés. Pour ce faire, il doit être inscrit et s'être authentifié dans l'application. Dans cette section, nous allons identifier les exigences fonctionnelles et de sécurité et nous allons voir comment nous les avons modélisées (avec et sans diagramme d'états/transitions) pour la première version de l'application.

4.1.1 Exigences fonctionnelles

Plusieurs exigences fonctionnelles ont été identifiées pour l'application :
(1) l'utilisateur authentifié sur le site doit pouvoir accéder aux services proposés,
(2) le système doit permettre l'enregistrement de plusieurs utilisateurs seulement s'ils remplissent correctement les données concernant leur nom, prénom et qu'ils ne soient pas déjà enregistrés,
(3) il peut acheter des tickets de cinéma avec un tarif unique et s'il reste encore des tickets disponibles,
(4) il doit pouvoir afficher les tickets achetés,
(5) il doit pouvoir supprimer un, plusieurs, voire même tous les tickets achetés (avant le début de la séance),
(6) il doit pouvoir quitter l'application,
(7) l'utilisateur authentifié doit pouvoir naviguer entre les pages du site web et les services proposés.

Nous travaillons dans un sous-ensemble de UML, nommé UML4MBT (UML for Model-Based Testing) que nous avons présenté dans la section 2.3.1. Dans ce sous-ensemble, nous considérons quatre types de diagrammes pour la génération de tests. Nous n'avons pas traduit le modèle qui a été réalisé initialement en anglais pour les besoins du projet *SecureChange*.

Pour la gestion de notre application nous avons créé un diagramme de classe en UML/OCL, représenté sur la figure 4.1, contenant les classes suivantes :
- eCinema : représente le modèle du cinéma i.e le système sous tests, ainsi que les opérations,
- movies : correspond aux informations sur les films projetés dans le cinéma,
- tickets : correspond aux informations sur les tickets,
- users : correspond aux informations sur l'utilisateur : nom, prénom.

Les exigences fonctionnelles sont marquées dans le modèle par le mot-clef **@REQ**. Chaque exigence couvre plusieurs comportements dans le modèle, qui sont identifiés par le mot-clef **@AIM**. Ce dernier peut être considéré comme un raffinement d'une exigence

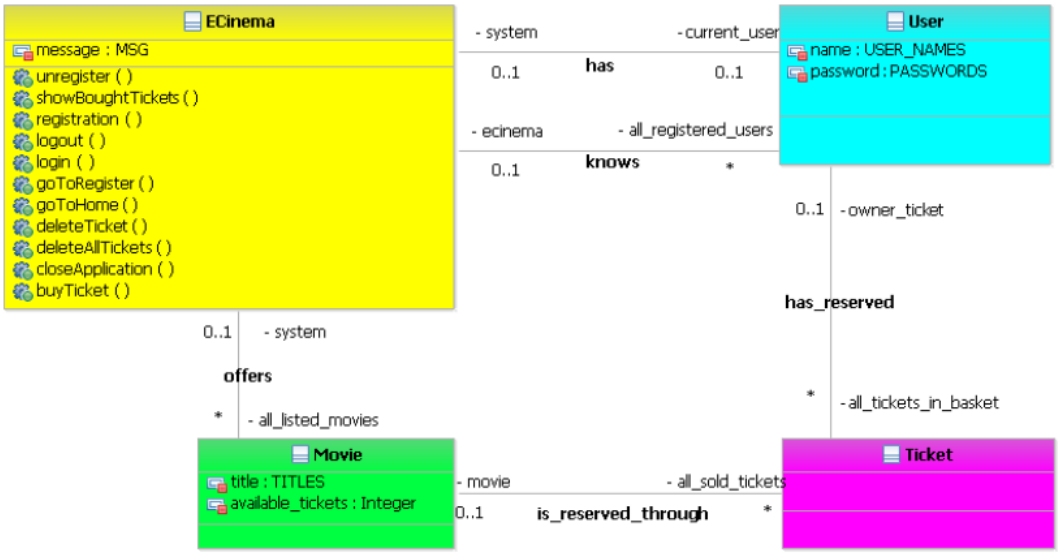

FIGURE 4.1 – Diagramme de classes d'eCinema

sous forme d'un comportement fonctionnel. La couverture visée est celle des exigences. Ainsi, une cible de test est définie par le comportement associé à une exigence à travers le code OCL situé au sein d'une transition ou d'une opération. Afin d'avoir une gestion optimale entre les exigences, les comportements et les cibles de tests, nous associons à chaque cible de test un couple de *tags* **@REQ/@AIM**.

Pour représenter la dynamique de notre système dans un cycle d'utilisation, nous avons tout d'abord créé le modèle d'*eCinema* avec un diagramme d'états/transitions. Ce diagramme est représenté dans la figure 4.2.

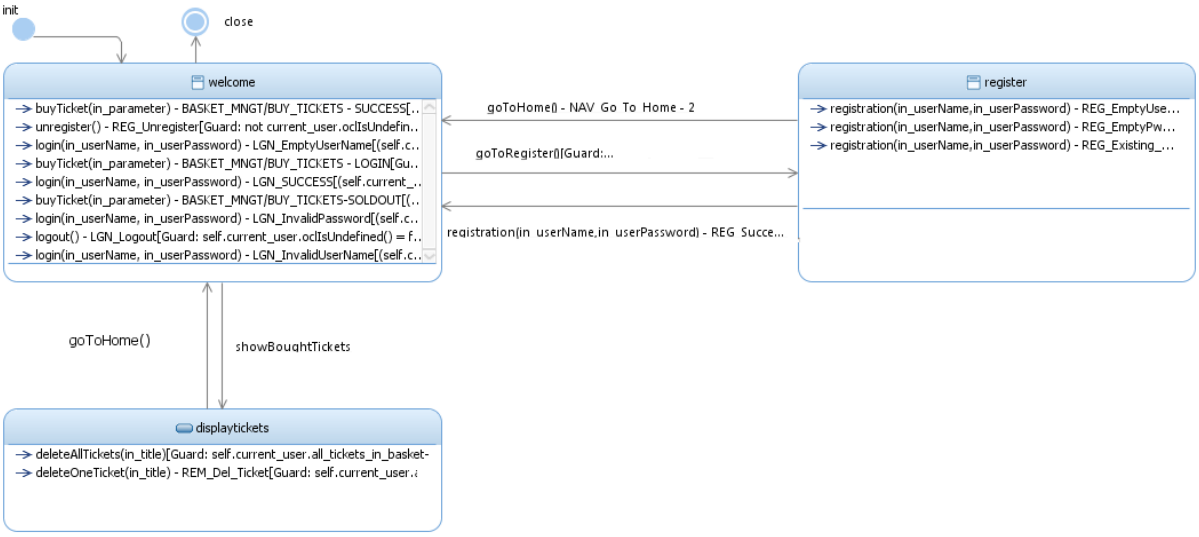

FIGURE 4.2 – Diagramme d'états/transitions d'eCinema

Le système est composé de trois états principaux :

- *welcome* : l'état d'accueil de l'utilisateur, représentant le menu d'accueil à partir duquel il peut s'authentifier, s'enregistrer. Une fois authentifié, l'utilisateur peut acheter des tickets ou demander l'affichage des tickets achetés,
- *registrer* : l'état d'enregistrement d'un nouvel utilisateur, représentant le menu d'enregistrement,
- *displaytickets* : l'état d'affichage de tickets achetés, représentant le menu à partir duquel il est possible d'afficher les tickets, mais aussi de les supprimer.

Dans ces trois états, nous retrouvons les différentes opérations définies dans les **sept** exigences présentées au début de cette section. Pour couvrir notre critère, nous obtenons **vingt** transitions, qui sont également des cibles de tests. Dans ce manuscrit, nous allons utiliser particulièrement les transitions représentant les cas positifs des opérations *buy_ticket, login et register*. Pour cela nous donnons les détails des transitions correspondantes.

La transition *buy_ticket-success* est présentée par la figure 4.3. L'état *welcome* est son état source et destination. Son action 4.3(b) est activée si la garde 4.3(a) est satisfaite i.e. un utilisateur est connecté avec succès.

```
((not self.current_user.oclIsUndefined()) and
self.all_listed_movies
    ->any(t : Movie | t.title = in_title)
    .available_tickets >=1)=true
```

(a) garde

```
---@REQ: BASKET_MNGT/BUY_TICKETS
---@AIM: BUY_Success
self.current_user.all_tickets_in_basket
                ->includes(unallocated_ticket) and
targeted_movie.all_sold_tickets
                ->includes(unallocated_ticket) and
targeted_movie.available_tickets =
          targeted_movie.available_tickets - 1 and
message= MSG::NONE
```

(b) action

FIGURE 4.3 – buy_ticket-success

La transition *login-success* est présentée par la figure 4.4. L'état *welcome* est son état source et destination. Son action 4.4(b) est activée si la garde 4.4(a) est satisfaite i.e. un utilisateur enregistré tente de se connecter à l'application.

```
(self.current_user.oclIsUndefined() and
all_registered_users->exists(name = in_userName
           and password=in_userPassword))=true
```

(a) garde

```
---@REQ: ACCOUNT_MNGT/LOG
---@AIM: LOG_Success
let user_found:User = all_registered_users
              ->any(name = in_userName) in
self.current_user = user_found and
message= MSG::WELCOME
```

(b) action

FIGURE 4.4 – login-success

La transition *register-success* est présentée par la figure 4.5. L'état *register* est son état source et l'état *welcome* son état destination. Son action 4.5(b) est activée si la

garde 4.5(a) est satisfaite i.e. un utilisateur ayant accédé à la page d'enregistrement et n'étant pas déjà enregistré.

```
in_userName<>USER_NAMES::INVALID_USER and
not all_registered_users
   ->exists(user | user.name=in_userName)
```

(a) garde

```
---@REQ: ACCOUNT_MNGT/REGISTRATION
---@AIM: REG_Success
let user_found:User = User.allInstances()->
 any(name = in_userName and password= in_userPassword) in
    all_registered_users->includes(user_found) and
    self.current_user = user_found and
    message= MSG::WELCOME
```

(b) action

FIGURE 4.5 – register-success

Pour avoir plus de détails sur les transitions et les exigences qu'elles représentent, vous pouvez vous référer aux tableaux A.1 et A.2 de l'annexe A.

Maintenant, nous présentons le modèle sans le diagramme d'états/transitions. Nous avons ainsi géré l'état du système directement dans le code OCL. Pour se faire, nous avons ajouté une classe énumération *SystemState* (État du système) modélisant les états : *welcome, register et displaytickets*. Ensuite, nous avons remplacé les comportements exprimés dans chaque transition par l'équivalent dans la post-condition de l'opération correspondant au déclencheur de la transition. Ainsi, le diagramme de classe présenté dans la figure 4.1 reste inchangé puisque les classes, attributs, liens et énumérations précédents n'ont pas été impactés.

Néanmoins, la mise en place dans le code OCL des états du système a pour effet d'ajouter des pré-conditions aux opérations. Pour ce modèle, sans diagramme d'états/-transitions, nous retrouvons les **sept** exigences fonctionnelles précédentes et les **vingt** cibles de tests. Pour avoir plus le détail sur les comportements obtenus, vous pouvez consulter l'annexe A.

4.1.2 Exigences de sécurité pour eCinema

Pour compléter nos exigences fonctionnelles, nous avons aussi pris en compte des exigences de sécurité. Pour cela, nous avons utilisé l'outil de génération de tests de Smartesting pour compléter les tests générés précédemment avec des tests dédiés au test de sécurité. Ces tests ont pour vocation de couvrir les exigences de sécurité. Ces exigences sont différentes par nature des exigences fonctionnelles et elles nécessitent l'écriture de schémas de tests. Le schéma de test est un langage proposé par Smartesting pour définir un besoin de test associé à une exigence. Ces schémas sont dépilés par le générateur de tests en un ensemble de spécifications de cas de tests (*Test Case Specification* noté **TCS**), qui sont utilisées en entrée du générateur de tests, tel que détaillé dans la section 2.3.3. Avant d'écrire les schémas, nous donnons les exigences de sécurité pour l'application *eCi-*

nema. Elles sont au nombre de deux.

Exigence de sécurité 1 (nommée *enregistrement*) **:** *Si la base de données utilisateur d'eCinema est vide, la seule opération possible est de s'enregistrer.*

L'élément important ici est qu'il soit impossible pour tout utilisateur désabonné d'accéder aux services. Si nous sommes dans le cas d'une application critique l'ingénieur de validation peut définir plusieurs sous-objectifs. Par exemple, ajouter puis supprimer un utilisateur ou vérifier ce qui se passe au sein de l'application si la base de données est complètement vidée et voir s'il reste des éléments résiduels qui permettraient une connexion au système non-autorisée.

Afin d'écrire le schéma, nous allons donner les étapes correspondant à notre intention de test associée à cette exigence de sécurité :

1. enregistrer plusieurs utilisateurs dans la base de données,
2. authentifier un utilisateur et accéder à plusieurs services, par exemple achat d'un ticket,
3. vider la base de données et ensuite essayer la seule opération possible "s'enregistrer".

Ensuite, nous pouvons écrire un schéma correspondant :

```
  for_each operation $X from any_operation_but goToRegister
or unregister or logout or closeApplication,
use any_operation any_number_of_times to_reach_state
"self.all_registered_users->size() >= 1" on_instance sut then
use any_operation any_number_of_times to_reach_state
"not self.current_user.oclIsUndefined()" on_instance sut then
use $X to_reach_state "not self.current_user.oclIsUndefined()"
on_instance sut then
use any_operation any_number_of_times to_reach_state
"self.all_registered_users->size() <= 0" on_instance sut then
use any_operation any_number_of_times to_reach_state
"not self.current_user.oclIsUndefined()" on_instance sut
```

Exigence de sécurité 2 (nommée *authentification*) **:** *Un achat peut être effectué si et seulement si l'utilisateur est existant et authentifié avec succès.*
Pour cette exigence, un ingénieur de validation sera intéressé de tester si le système autorise l'achat de tickets par des utilisateurs dits invalides i.e. non-enregistrés, non-authentifiés. Comme précédemment, nous allons donner d'une manière informelle les intentions de test pour cette exigence :

1. enregistrer et authentifier un utilisateur dans l'application,

2. vérifier que l'utilisateur peut acheter un ticket,
3. dés-inscrire/déconnecter l'utilisateur,
4. essayer d'acheter un ticket et vérifier que le résultat est une erreur.

En accord, avec les intentions de tests décrites, nous définissons le schéma correspondant.

```
 for_each operation $X from registration or login,
for_each operation $Y from unregister or logout,
for_each operation $Z from goToRegister,
use $Z 0..1 on_instance sut then
use $X at_least_once to_reach_state
"self.message = MSG : :WELCOME" on_instance sut then
use buyTicket at_least_once to_reach_state
"self.message = MSG : :NONE" on_instance sut then
use $Y at_least_once to_reach_state
"self.message = MSG : :BYE" on_instance sut then
use buyTicket at_least_once to_activate_behavior
without_tags {AIM :BUY_Success}
```

Dans ce schéma, nous avons utilisé une variable supplémentaire $Z pour permettre d'aller sur la page d'enregistrement s'il n'y a pas d'utilisateur enregistré dans le système en passant par l'étape *"goToRegister"* avant d'appeler l'opération *"registration"*. Cette étape peut ne pas être nécessaire lors de la génération de tests car le système pourra faire directement un *"login"* avec un utilisateur déjà enregistré.

4.2 Génération de tests pour eCinema

Dans cette section, nous présentons la génération de tests pour l'application *eCinema*. Nous verrons pour la partie fonctionnelle comment extraire les informations à partir du modèle. Nous présentons pour la partie sécurité les tests générés en utilisant les schémas associés et les deux modèles (avec et sans diagramme d'états/transitions).

4.2.1 Tests fonctionnels

Pour assurer la traçabilité des exigences, nous utilisons dans le code OCL des opérations les mots-clefs **@REQ** et **@AIM**. Ces derniers sont ensuite utilisés pour déterminer les cibles des tests. Les exigences peuvent être utilisées pour annoter les comportements dans le diagramme d'états/transitions ou dans la post-condition d'une opération du diagramme de classes.

Pour comprendre comment cela fonctionne, nous prenons l'opération `buyTicket`. Cette opération possède trois cibles de test :

- acheter un ticket avec succès,
- erreur lors d'achat des tickets, dont le stock est épuisé,
- erreur lors d'une tentative d'achat d'un utilisateur non connecté.

Suivant le modèle choisi, l'élément à partir duquel est produite la cible n'est pas le même. Pour le modèle avec le diagramme d'états/transitions, nous allons chercher à cibler la transition où apparait l'opération `buyTicket`. Dans la figure 4.3 nous aurons :

(1) cible : @REQ : BASKET_MNGT/BUY_TICKETS et @AIM : BUY_Success,

(2) transition : *buyTicket(in_ticket) - BASKET_MNGT/BUY_TICKETS*,

(3) opération : buyTicket.

```
---@REQ: BASKET_MNGT/BUY_TICKETS
if self.all_listed_movies
    ->any(t : Movie | t.title = in_title)
    .available_tickets >=1 then
    if self.current_user.oclIsUndefined()then
        ---@AIM: BUY_Login_Mandatory
        message = MSG::LOGIN_FIRST
    else
        ---@AIM: BUY_Success
        let targeted_movie : Movie =
            self.all_listed_movies
            ->any(t : Movie | t.title = in_title) in
        let unallocated_ticket : Ticket =
            (Ticket.allInstances())
            ->any(owner_ticket.oclIsUndefined()) in

        self.current_user.all_tickets_in_basket
            ->includes(unallocated_ticket) and
        targeted_movie.all_sold_tickets
            ->includes(unallocated_ticket) and
        targeted_movie.available_tickets =
            targeted_movie.available_tickets - 1 and
        message= MSG::NONE
    endif
else
    ---@AIM: BUY_Sold_Out
    message = MSG::ALL_MOVIES_SOLD_OUT
endif
```

FIGURE 4.6 – Opération buy_ticket

Pour le modèle sans le diagramme d'états/transitions, tel que décrit sur la figure 4.6, les comportements sont exprimés dans la post condition de l'opération dans chacune des branches de l'instruction *if*. Pour notre exemple, à l'équivalent de celui avec le diagramme d'états/transitions, nous retrouvons la couverture des éléments suivants :

(1) cible : @REQ : BASKET_MNGT/BUY_TICKETS et @AIM : BUY_Success,

(2) opération : buyTicket.

Pour couvrir la cible "*acheter un ticket avec succès*", nous utiliserons le test suivant :

```
ECinema::sut.login(REGISTERED_USER, REGISTERED_PWD);
ECinema::sut.buyTicket(TITLE1);
```

Il est important de noter que les deux cas de test couvrent aussi l'opération `login` et son comportement identifié par l'exigence : @REQ : ACCOUNT_MNGT/LOG et @AIM : LOG_Success, qui dans le cas d'un diagramme d'états/transition est déclenchée par la transition *login(in_userName, in_userPassword) - LGN_SUCCESS*.

Ainsi, pour le modèle avec le diagramme d'états/transitions afin de couvrir toutes les exigences fonctionnelles, **seize** tests ont été générés à partir des vingt cibles. Pour le modèle sans le diagramme d'états/transitions, nous avons aussi obtenu **seize** tests pour les vingt cibles. Vous pouvez retrouver la liste complète des tests générés pour les deux modèles dans l'annexe C.

4.2.2 Tests dédiés à la sécurité

Afin de compléter la suite de tests et valider les aspects de sécurité dans l'application, nous avions défini des schémas dans la section 4.1.2. Nous allons par la suite détailler la production des TCS pour les schémas donnés en vue de génération de tests, afin de donner les résultats sur les tests produits. Nous rappelons que dans ce cas les TCS sont les cibles de tests qui doivent être couverts par les tests générés (pour plus de détails voir section 2.2). Nous allons donner les TCS obtenus pour les deux modèles d'eCinema (avec et sans diagramme d'états/transitions) et les tests générés à partir de ceux-ci.

Exigence de sécurité 1 - *enregistrement* Le premier schéma pour le modèle diagramme d'états/transition est déplié en **sept** TCS, un pour chaque $\$X$ opération, définies par le opérateur *any_operation_but*. Nous listons les TCS, respectivement via la notation :

{Exigence+Numéro}"."{Numéro TCS}.

Le générateur pour ces huit cibles a produit **trois tests**. Ceci vient du fait que le moteur de génération tend vers la minimisation du nombre de tests créés. La minimisation du nombre des tests est très importante dans le coût du processus de validation logiciel, mais à condition de ne pas diminuer la couverture des cibles. Dans notre cas lors de la création des tests pour couvrir les cibles, le moteur retrouve des séquences de tests identiques. Ensuite, afin de minimiser le nombre de tests générés il mutualise ceux qui le permettent. Cependant, afin d'atteindre une cible il est nécessaire de passer par une autre. Ainsi nous pouvons avoir un test qui couvre plusieurs cibles et qu'une cible est couverte par plusieurs

tests. Plus de détails sur les opérations couvertes par les TCS et les tests sont donnés dans le tableau 4.1, ci-dessous. Dans ce tableau nous allons utiliser les identifiants pour les tests attribués par l'outil de génération de *Smartesting*, les tests produits respectent la notation des identifiants {NomSchéma}+{Identifiant}.

TCS	**$X**	**Test - modèle 1**
TCS 1.1,1.2,1.4,1.5,1.7	buyTicket, deleteAllTickets, goToHome, login, showBoughtTickets	requirement11(bb-25-c2)
TCS 1.1,1.3,1.4,1.5,1.7	buyTicket, deleteTicket, goToHome, login, showBoughtTickets	requirement11(bb-d3-b1)
TCS 1.5,1.6	login, registration	requirement15(bb-91-ae)

TABLE 4.1 – Tests Générés pour l'exigence de sécurité 1, eCinema diagramme d'états/-transitions

Le même schéma pour le modèle sans diagramme d'états/transitions est déplié de même que le modèle précédent en **sept** TCS, vu qu'ils sont équivalents. Cependant, le moteur de génération de tests est moins contraint qu'avec le diagramme d'états/transitions. Les tests produits couvrent différents chemins pour les différentes TCS. Dans le Tableau 4.2, nous donnons les détails sur les tests générés :

TCS	**$X**	**Test - modèle 2**
TCS 1.1,1.3,1.4,1.5,1.7	buyTicket, deleteTickets, goToHome, login, showBoughtTickets	requirement11(bb-7c-93)
TCS 1.2,1.4,1.5,1.7	deleteAllTickets, goToHome, login, showBoughtTickets	requirement12(bb-fd-f9)
TCS 1.5,1.6	login, registration	requirement15(bb-8f-fe)

TABLE 4.2 – Tests Générés pour l'exigence de sécurité 1, eCinema sans diagramme d'états/transitions

Exigence de sécurité 2 - *authentification* Le deuxième schéma est déplié en **quatre** TCS. Elles sont obtenues par le produit Cartésien des opérations *registration, login* d'une part et *unregister, logout* d'autre part.

TCS	**$Z**	**$X**	**$Y**	**Test - modèle 1**	**Test - modèle 2**
TCS 2.1	goToRegister	registration	unregister	requirement28(bb-0c-3d)	requirement28(bb-f1-e3)
TCS 2.2	goToRegister	registration	logout	requirement210(bb-ca-6c)	requirement210(bb-72-5c)
TCS 2.3	-	login	unregister	requirement29(bb-c0-2c)	requirement29(bb-5c-3f)
TCS 2.4	-	login	logout	requirement211(bb-24-b0)	requirement211(bb-be-43)

TABLE 4.3 – Tests Générés pour l'exigence de sécurité 2

Dans le tableau 4.3, nous détaillons les résultats de la génération pour les deux modèles d'*eCinema*, avec et sans diagramme d'états/transitions et nous les nommons *Test - modèle 1* et *Test - modèle 2* respectivement. Nous utilisons aussi les identifiants attribués par l'outil de génération pour noter les tests. Cependant, suite à la différence des deux modèles, l'outil génère des identifiants différents pour nos tests, mais les pas de tests sont identiques. En résumé pour cette exigence, nous avons obtenu le même nombre de tests

pour les deux modèles. Nous pouvons aussi constater que l'étape "goToRegister" a été utilisée seulement dans le cas de l'enregistrement de l'utilisateur.

Nous venons de générer les tests pour couvrir les exigences fonctionnelles et de sécurités. Dans la section suivante, nous présenterons l'évolution de la spécification et les changements induits dans les modèles.

4.3 Évolution de la spécification d'eCinema

Dans cette section, nous présentons l'évolution de la spécification d'*eCinema*. La nouvelle version de l'application prend en compte les types d'abonnement proposés aux utilisateurs (par exemple étudiants, seniors etc.) et aussi la gestion du solde, ce qui n'a pas été traité par la spécification précédente.

4.3.1 Description de l'évolution

L'évolution est faite en deux directions : (*i*) inclusion de type, influençant le prix du ticket et (*ii*) la gestion du solde de l'utilisateur voir plus de détails techniques dans les tableaux A.3, A.4 et A.5 de l'annexe A. Plus généralement, les évolutions considérées sont les suivantes :

- lors de l'enregistrement l'utilisateur choisit son type d'abonnement,
- l'abonnement peut être mis à jour à tout moment,
- les abonnements proposés sont : enfant, étudiant, normal, sénior et un ticket offert pour dix tickets achetés,
- le tarif du billet est associé à l'abonnement sélectionné,
- l'utilisateur peut gérer son solde.

Les abonnements sont attribués aux utilisateurs sous réserve qu'ils remplissent les conditions spécifiques à chaque type d'abonnement :

- enfant : tarif pour les enfants moins de 12 ans,
- étudiant : tarif pour les étudiants sur présentation de la carte étudiante,
- normal : tarif normal i.e. plein tarif,
- sénior : tarif pour les séniors, personnes âgées de plus de 64 ans,
- un gratuit : un ticket offert pour dix tickets achetés, au plein tarif.

Lors de l'achat du ticket, l'utilisateur paye le tarif associé à son type d'abonnement. De même lors du suppression du ticket de son panier, le tarif payé est remboursé sur le compte de l'utilisateur. Lors de la suppression de tickets, avec l'abonnement *un gratuit*, l'utilisateur doit pouvoir réutiliser son offre lors du prochain achat. Les tarifs des tickets selon l'abonnement sont à définir par le cinéma.

Ci-dessous nous allons décrire les changements dans le modèle pour prendre en compte les nouvelles fonctionnalités.

4.3.2 Évolution de modèle

L'évolution du modèle se passe en deux étapes. En premier lieu, nous adaptons les éléments du diagramme de classes : ajout des opérations, attributs, liens nécessaires afin d'accéder aux nouvelles entités.

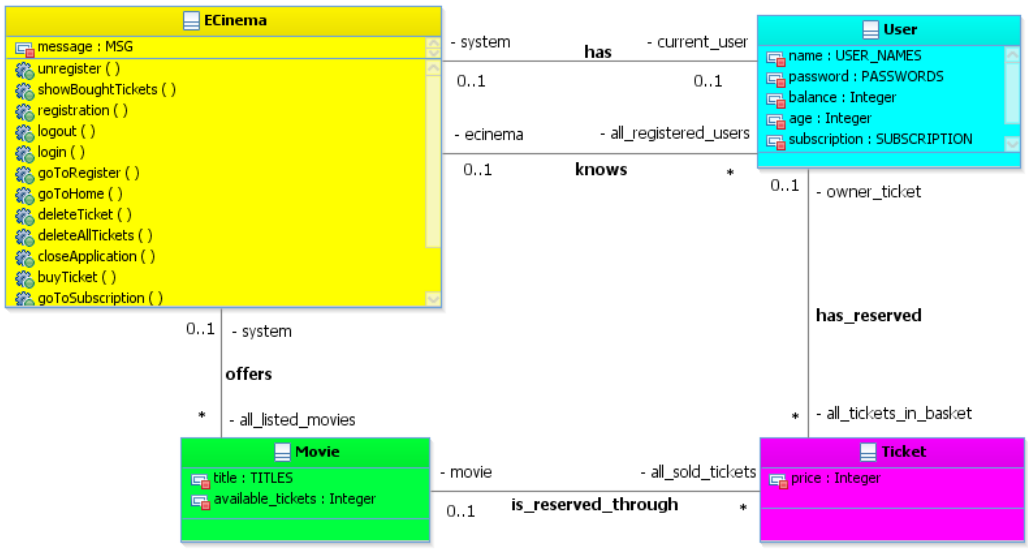

FIGURE 4.7 – Diagramme de classes d'eCinema - évolué

Dans le diagramme de classes dans la figure 4.7, nous pouvons distinguer les différentes évolutions des classes. Ces dernières sont valables pour les deux modèles, avec et sans diagramme d'états/transitions. Pour permettre d'avoir différents tarifs associés à un ticket, l'attribut *price (prix)* est ajouté à la classe *Ticket*. Le tarif du ticket est attribué lors de son achat en fonction de l'abonnement de l'utilisateur. Pour la classe *User(utilisateur)* les attributs suivants ont été ajoutés :

- *balance* : permettant de stocker l'état du solde actuel de l'utilisateur,
- *age* : l'âge de l'utilisateur, qui permettra d'associer ou pas certains types d'abonnements,
- *subscription* : indique le type d'abonnement que l'utilisateur a souscrit lors de son enregistrement.

Afin de permettre l'ajout des différents abonnements, nous avons défini une classe de type énumération :

- *YOUNG* : tarif pour les enfants de moins de 12 ans,
- *STUDENT* : tarif étudiant, sur présentation de la carte étudiante,

– *NORMAL* : plein tarif,
– *SENIOR* : tarif pour les séniors, personnes âgées de plus de 64 ans,
– *ONEFREE* : tarif spécial, un ticket offert pour dix tickets achetés au plein tarif.

Afin de gérer l'abonnement et le solde de l'utilisateur deux états du système ont été ajoutés : *BalanceManagment* et *SubscriptionManagment*, qui correspondent aux menus de gestion du solde du compte et à l'abonnement de l'utilisateur. Pour permettre la gestion, les opérations suivantes ont été ajoutées à la classe principale du système :

– *goToBalanceMenagement* : permettant d'aller au menu de gestion du solde,
– *addUnits* : permettant de créditer le solde afin d'acheter des tickets,
– *retrieveUnits* - permettant de retirer l'ensemble ou une partie de l'argent déposé sur le compte,
– *goToSubscription* : permettant d'aller au menu de gestion de l'abonnement,
– *setSubscription* : permettant de changer l'abonnement de l'utilisateur.

Nous donnerons plus de détails sur le calcul des différences des comportements dans les sections dédiées aux calculs des différences entre modèles dans les sections 6.1 et 7.2.

4.4 Synthèse

Ce chapitre introduit l'application *eCinema*. Nous avons défini les exigences fonctionnelles et de sécurité. Nous avons proposé deux modélisations pour l'application : une avec un diagramme d'états/transitions et une autre sans. Cette dernière est basée seulement sur l'utilisation d'opérations des classes écrites en OCL. De plus, nous avons défini les intentions de tests associés aux exigences de sécurité à l'aide du langage de schéma de *Smartesting* afin de compléter la génération de tests. Ces éléments de modélisation ont servi à la génération de tests avec comme critère de couverture celui des exigences fonctionnelles et de sécurité pour les deux modèles d'*eCinema*. Pour terminer, nous avons défini un ensemble d'évolutions pour l'application *eCinema*. Ces évolutions portent sur la gestion de différents abonnements (enfant, étudiant, sénior, normal et un ticket offert pour dix tickets achetés) ainsi que la gestion du solde associé au compte de fidélité de l'utilisateur.

Nous allons voir dans la seconde partie comment les algorithmes proposés dans cette thèse permettent de gérer l'évolution.

Deuxième partie

Contributions

Chapitre 5

Le cycle de vie et suites de tests

Sommaire

5.1 L'évolution du cycle de vie des tests 59
5.2 La gestion des suites de tests 61
5.2.1 Suite de tests 61
5.2.2 Composition des suites de tests 62
5.3 Gestion du référentiel de tests 63
5.4 Synthèse 64

Dans la première partie, nous venons de voir les principes du MBT et du test de non-régression. Les approches associées au test de non-régression ont pour vocation de sélectionner les tests pour vérifier le bon fonctionnement des parties inchangées dans le système sous test après modification. Cependant, il n'existe pas d'équivalent pour s'occuper de valider la suppression des fonctionnalités du système devenues obsolètes lors de l'évolution, ni de tester les nouveautés apparues.

Dans cette partie, nous traitons la classification des tests et définition de leur cycle de vie durant l'évolution des exigences. Pour ce faire, le chapitre 5 défini le cycle de vie d'un test, introduit les statuts et permet de regrouper les tests en des catégories utilisables pour la validation du SUT. Pour établir le statut des tests, nous devons déterminer ce qui a évolué dans le modèle et l'impact de cette évolution sur les tests.

Le processus global de la méthode que nous proposons, nommée SeTGaM, est illustré sur la figure 5.1. Elle est basée sur deux versions de modèles, celui de référence M et son évolution M'. Tout d'abord dans l'étape ①, elle compare les exigences, exprimées par le code OCL du diagramme d'états/transitions ou celui des opérations du diagramme de classes. Nous proposons un algorithme qui permet de déduire les impacts de l'évolution du modèle sur les tests existants. Ainsi, à l'étape ② les résultats de la comparaison d'une

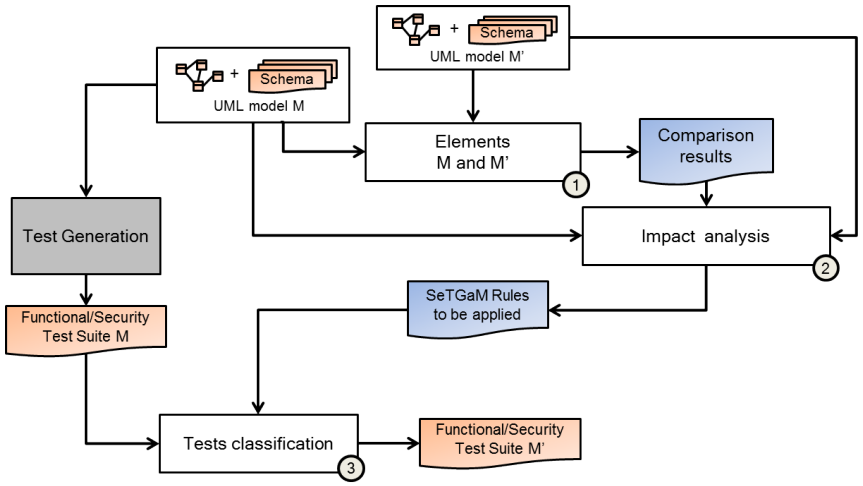

FIGURE 5.1 – Processus global de SeTGaM

part et les calculs des graphes de dépendances d'autre part, permettent d'effectuer une analyse des tests impactés. À l'étape ③, cet impact est pris en entrée d'un calcul qui détermine le statut du test comme décrit dans la section 5.1.

Ceci permet de le classifier dans une des suites données, comme expliqué dans la section 5.2, afin de valider le nouvel état du système. Enfin, la suite de tests pour le modèle *M'* est construite à partir des tests valides pour le modèle *M'*.

Dans les chapitres qui suivent, nous allons détailler les différents briques nécessaires à cette méthode comme illustré sur la figure.

Les chapitres 6 et 7 présentent la méthodologie qui permet de déterminer les évolutions de modèles UML/OCL et son impact sur les exigences et sur les tests. Ils traitent respectivement les deux approches retenus par la définition de comportements au niveau des modèles UML/OCL, plus précisément UML4MBT. Ce sont respectivement l'utilisation de diagramme d'états/transitions et des opérations.

Le dernier chapitre de cette partie définit l'approche SeTGaM pour la prise en compte de la dimension sécurité.

Dans ce chapitre, nous allons poser les bases d'une approche qui permet de répondre à ce problème. Nous allons tout d'abord introduire le cycle de vie d'un test à travers l'évolution des exigences dans le système, qui sera plus précisément raffiné par chacune des méthodes axes de SeTGaM. Pour ce faire, nous proposons de compléter la nomenclature du cycle de vie d'un test, telle que proposée dans [BLH09, LW89] entre les différentes versions du système. Ensuite, nous allons regrouper les tests selon leur statut en **quatre** différentes suites de tests : *Regression, Stagnation, Evolution* et *Deletion*. Chacune d'elle a pour vocation de tester une spécificité lors de l'évolution i.e. les éléments inchangés, les supprimés, les nouveautés et la dernière pour sauvegarder les tests qui étaient précédemment dans la suite de tests *Stagnation*.

Ce chapitre est organisé de la manière suivante. Dans le section 5.1, nous allons définir le cycle de vie des tests. Ensuite dans le section 5.2, nous allons identifier les règles pour classifier les tests dans une suite de tests en fonction de leurs statut dans le cycle. Nous verrons en section 5.3 comment le référentiel de test est géré pour aider à la validation de la nouvelle version du système, avant donner une synthèse de ce travail en section 5.4.

5.1 L'évolution du cycle de vie des tests

Dans cette section, nous introduisons la notion d'évolution pour le test. Ainsi, chaque test possède une version qui est associée à un *statut*, celui-ci indique l'état du test dans le cycle de vie.

Le cycle de vie d'un test est décrit dans la figure 5.2. Nous avons étendu la classification des tests par Leung et White [LW89], qui proposent les statuts obsolete, reusable, retestable, new specification et new structural. À la différence de Leung et White, nous avons défini huit statuts dans le cycle de vie du test pour raffiner leur classification. De plus, nous considérons qu'il doit exister un autre statut temporaire (non présent dans le cycle de vie) *Involved*. Les tests de cette catégorie *Updated* et *Adapted* sont ceux qui sont impactés par l'évolution des exigences.

Dans le cycle de vie de tests, décrit dans la figure 5.2, un test lors de sa création couvre une nouvelle exigence dans le système et possède le statut *new*, ce qui est donc le statut initial pour chaque test. Lors d'une évolution des exigences, un test ne peut pas revenir à cet état initial et il évolue en trois directions : impacté, supprimé ou non-impacté (ou inchangé). Dans le premier cas, son statut change en *updated* ou *adapted* parce son oracle ou ses pas sont modifiés suite à l'évolution. Dans le deuxième cas, il est *obsolète* et peut prendre deux statuts : *outdated*, si l'exigence qu'il couvre est supprimée ou *failed*, s'il couvre une exigence impactée ou modifiée. La version précédente d'un test *adapted* est mise en *failed*. Un test *obsolète* peut devenir seulement *removed* et il sera gardé seulement pour l'historique. Enfin, un test ni impacté ni supprimé, peut prendre deux statuts : *reexecuted*, s'il couvre une exigence impactée mais ne relève aucun changement dans l'oracle du test ou *unimpacted*, s'il couvre une exigence non modifiée et non impactée. Avec l'aide de ces statuts nous construisons quatre suites de tests : *Evolution, Regression, Stagnation et Deletion* pour tester respectivement l'évolution, la non-régression et la stagnation et garder une trace des tests obsolètes pour une stabilité du référentiel de tests.

Plus précisément, l'évolution du statut est définie en considérant deux versions de modèles, $\mathcal{M}$ et $\mathcal{M}'$, pour lesquels il y a eu un ensemble d'évolutions du type d'ajout, de modification ou de suppression des éléments du modèle (opérations, comportements, transitions etc.). Comme déjà expliqué dans la section 2.3, chaque test a pour but la couverture d'une cible de test. Une cible peut être l'activation d'une transition dans la machine à états/transitions, ou encore la couverture des comportements issus du code

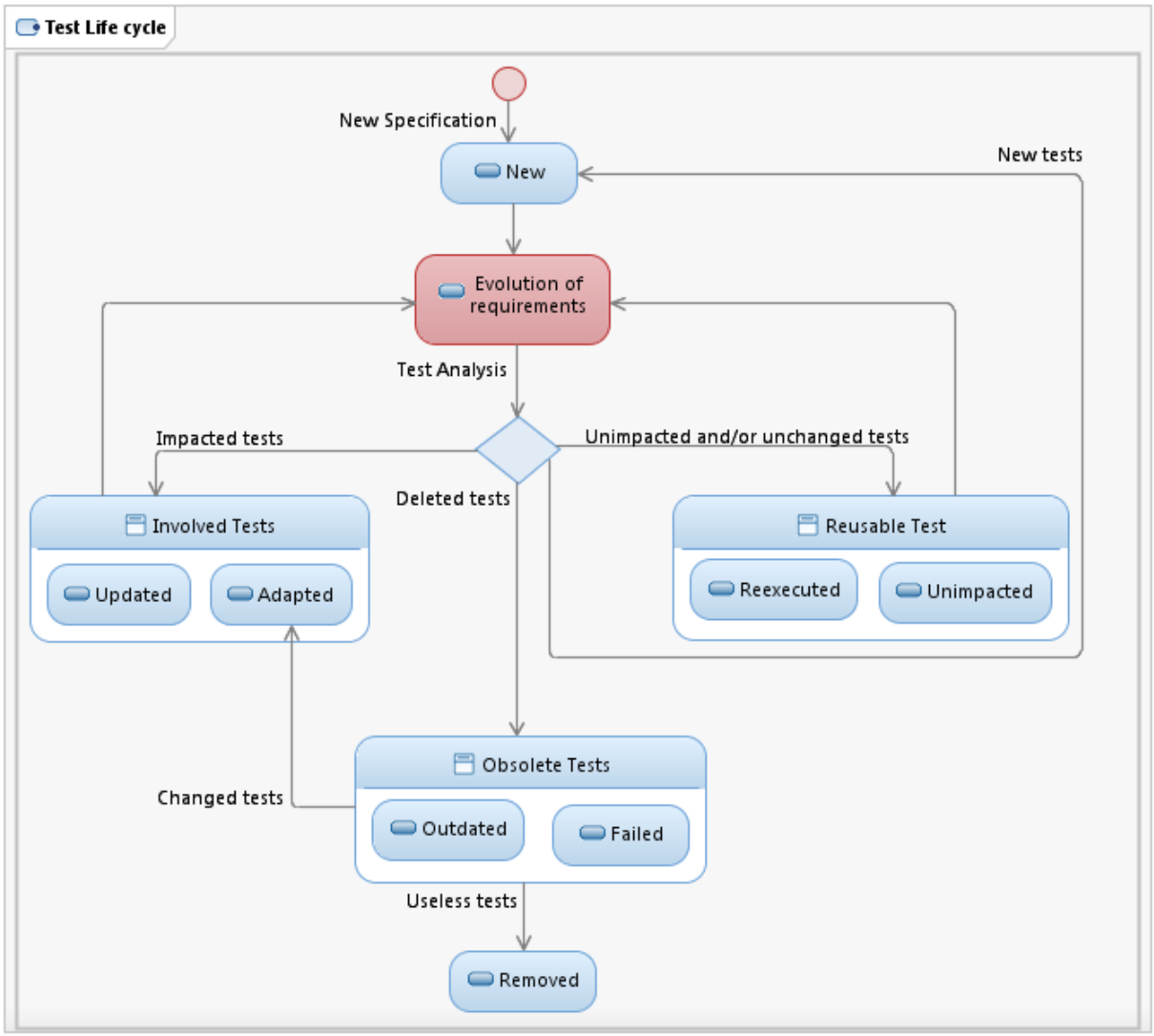

FIGURE 5.2 – Cycle de vie de tests

OCL d'une opération.

Nous définissons un cas de test évolutif comme un cas de test pouvant changer de statut à chacune de ses versions

Définition 2 (Cas de test évolutif) *Un* cas de test évolutif *tc est un cas de test caractérisé par son statut et la version considérée. On le note* tc^n *où* n *est la version du modèle pour lequel le test tc couvre une cible de test. On note* $status(tc^n)$ *le statut associé à* tc^n.

$status(tc^n) \in \{new, adapted, updated, unimpacted, reexecuted, failed, outdated, removed\}$

Remarque : $status(tc^1) = new$

5.2 La gestion des suites de tests

Dans cette section, nous définissons les différentes suites de tests et les règles qui permettent d'associer un test à l'aide de son statut.

5.2.1 Suite de tests

Nous avons défini ces quatre suites de tests pour permettre de regrouper les tests suivant un usage de validation du système spécifique. Chaque suite est dénotée Γ_X, où Γ est la notation pour une suite de tests et X est son type : *Evolution, Regression, Stagnation* et *Deletion*. Pour bien comprendre cette spécificité, nous donnons pour chacune une description informelle.

Suite de tests *Evolution* Γ_E contient les tests qui exercent les évolutions dans le système (nouvelles exigences, comportements et opérations).

Suite de tests *Regression* Γ_R contient les tests qui exercent les parties non modifiées dans le système. Ces tests ont pour but d'assurer que l'évolution du système n'a pas impacté les parties du SUT qui sont supposées ne pas avoir été modifiées. La spécificité de ces tests est qu'ils ont été calculés à partir d'une version précédente du modèle.

Nous parlons d'une version précédente et non de l'avant-dernière (la dernière étant la version courante) car un test ne change pas obligatoirement de statut entre deux versions de modèle.

Suite de tests *Stagnation* Γ_S contient les tests dits *non-valides* sur la version courante du modèle. Ces tests permettent d'assurer que l'évolution qui a été réalisée sur le système et que le comportement de celui-ci a changé suite à cette évolution. Il est important de souligner ici le fait que ces tests ont été générés à partir d'une version précédente du modèle, mais contrairement aux tests de non-régression $\in \Gamma_R$, ils doivent être invalides pour la version courante. Leur nature prévoit qu'ils doivent échouer lors de l'exécution sur le SUT (soit parce qu'ils ne peuvent pas être exécutés, soit parce qu'ils détectent une non-conformité dans le SUT vis-à-vis du résultat attendu).

Suite de tests *Deletion* Γ_D contient les tests provenant de la suite de tests de *Stagnation* du modèle de la version précédente du système. Néanmoins, nous pouvons les considérer en tant que tests invalides, puisqu'ils sont déjà obsolètes pour la version précédente et qu'il n'y a plus de sens d'en parler dans cette nouvelle version. Un ingénieur de validation par nature est toujours vigilant, nous gardons l'historique de l'ensemble des tests. Ainsi, il a le choix de garder ou pas cette dernière suite de tests car elle ne fera que

s'accroître.

Nous avons défini la nature de nos quatre suites de tests. Maintenant, nous définissons les règles qui permettent de peupler ces suites de tests pour une version du système donnée.

5.2.2 Composition des suites de tests

Nous décrivons ici les règles qui permettent de classer les tests de la version précédente suivant leurs statuts (ancien et nouveau) pour être ensuite utilisés pour la validation du système courant. Pour les définitions suivantes, nous notons *tc* un cas de test et nous considérons son évolution d'une version n vers $n+1$.

En général pour la version courante, la suite de tests *Evolution* est composée de tests nouveaux et adaptés, la suite de *Regression* contient les tests réutilisables, la suite de *Stagnation* contient les tests obsolètes et ceux qui ont échoué et enfin, la suite de tests *Deletion* est composée des tests provenant de la suite de tests *Stagnation* de la version précédente.

Règle 1 (New tests) *Un nouveau test existe seulement à la tc^{n+1} version. Tous les nouveaux tests sont ajoutés à la suite de tests Evolution :*

$status(tc^{n+1}) = new \quad \rightsquigarrow \quad tc^{n+1} \in \Gamma_E^{n+1}$

Règle 2 (Reusable tests) *Un test est dit reusable lorsqu'il n'a pas été impacté par l'évolution ou qu'il n'a pas subi de changements après animation sur le nouveau modèle. Il est soit unimpacted soit reexecuted. Pour cela, il est issu d'une suite de tests existante $tc^n \in \Gamma_E^n \cup \Gamma_R^n$ et il est resté inchangé pour la version actuelle $tc^{n+1} = tc^n$. Tous les tests reusable sont ajoutés dans la suite de tests Regression :*

$status(tc^{n+1}) \in \{unimpacted, reexecuted\} \rightsquigarrow tc^{n+1} \in \Gamma_R^{n+1}$

Règle 3 (Obsolete tests) *Un test obsolète provient d'une suite de tests existante (qui peut être obsolète aussi si maintenu par l'ingénieur de validation) $tc^n \in \Gamma_E^n \cup \Gamma_R^n \cup \Gamma_S^n$. Tous les tests qui sont désignés en tant que obsolètes sont ajoutés à la suite de tests Stagnation.*

$status(tc^{n+1}) = \{outdated, failed\} \rightsquigarrow tc^{n+1} \in \Gamma_S^{n+1}$

Remarque : Il est à noter que les tests dits *failed*, afin de garantir la couverture des exigences, sont recalculés pour la version courante. Ces tests produits sont appelés *adapted*.

Règle 4 (Involved tests) *Un test involved est issu d'une suite de tests déjà existante et il a été impacté par l'évolution $tc^n \in \Gamma_E^n \cup \Gamma_R^n$. Tous les tests de ce type sont ajoutés à la suite de tests Evolution.*

$status(tc^{n+1}) = \{updated, adapted\} \rightsquigarrow \quad tc^{n+1} \in \Gamma_E^{n+1}$

Remarque : Le test *updated* après animation sur le nouveau modèle a subit un changement de l'oracle par rapport à sa version précédente n. Cependant, contrairement à celui-ci, le test *adapted* est créé afin de maintenir la couverture des exigences. Dans sa version précédente n, il est *obsolete* pour la nouvelle version du modèle.

Règle 5 (Removed tests) *Un test est removed (i.e. enlevé) s'il provient de la suite de tests Stagnation de la version précédente* ($tc^n \in \Gamma_S^n$). *Tous les tests déclarés en tant removed par l'ingénieur de validation pour la version* $(n+1)$ *sont ajoutés dans la suite de tests Deletion.*

$status(tc^{n+1}) = \{removed\} \quad \rightsquigarrow \quad tc^{n+1} \in \Gamma_D^{n+1}$

Dans la section suivante, nous donnons une vue synthétique de la gestion du référentiel de tests et de la gestion de l'évolution des suites de tests ainsi que de la prise en compte de l'évolution du cycle de vie des tests.

5.3 Gestion du référentiel de tests

Le processus de validation doit être fait de façon méthodique. Une bonne pratique est d'utiliser un référentiel de tests pour gérer les tests. Il permet la réutilisation et la capitalisation des tests tout en permettant de garder le lien avec les exigences. Comme nous l'avons dit dans la section 1.1.1, ces outils permettent de construire des campagnes en lien avec des configurations spécifiques du SUT [12]. Ainsi, dans le cadre du MBT, nous pensons qu'il serait normal qu'ils puissent être directement gérés dans le référentiel.
Pour cela, il faut pouvoir effectuer la classification. Celle-ci va nécessiter trois éléments. Les deux premiers sont les modèles : la version $\mathcal{M}$ et son évolution $\mathcal{M}'$. Le troisième est la suite de tests créée à partir de la version $\mathcal{M}$ du modèle. A la fin du processus de sélection et après avoir appliqué les règles de classification associées aux suites de tests, nous pouvons mettre à jour les tests dans le référentiel.

Dans la figure 5.3, nous retrouvons un résumé des règles définies dans la section 5.2. Nous retrouvons la composition de chaque suite de tests. Ainsi, dans le référentiel, nous retrouvons le projet pour lequel nous avons généré les tests, les quatre types de suites où sont classés les tests avec l'historique des statuts attribués par version.

12. http ://www.teamst.org

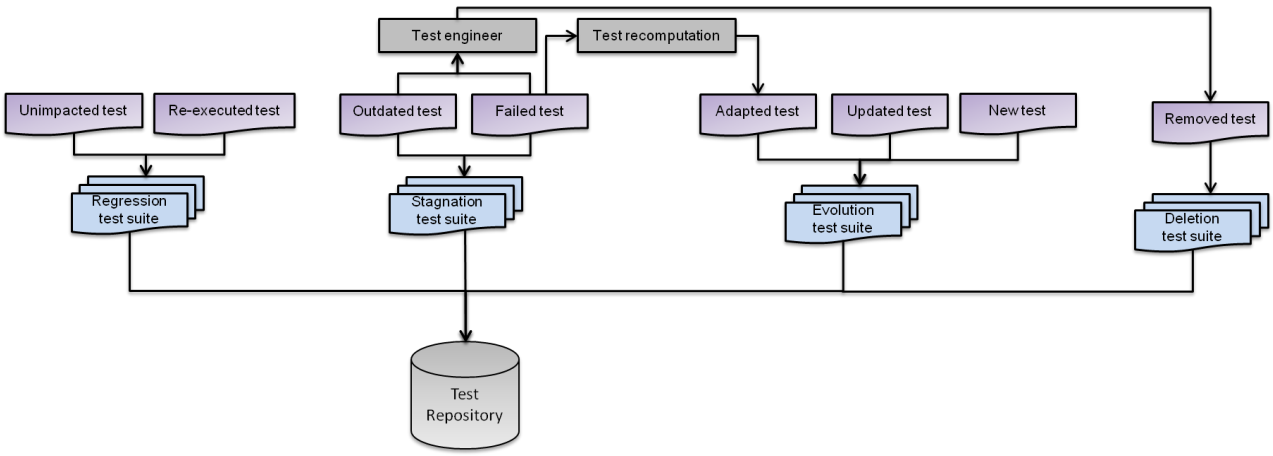

FIGURE 5.3 – Référentiel de tests

5.4 Synthèse

Dans ce chapitre, nous avons défini comment compléter l'approche, classique, proposée par [BLH09, LW89], de *Model-Based Regression Testing* pour pouvoir prendre en compte des aspects insuffisamment traités qui sont les évolutions d'exigences lorsqu'elles sont supprimées ou modifiées.

Pour cela, nous avons proposé de partir des tests existants pour ainsi factoriser les connaissances sur ceux-ci. Puis, nous les classifions selon leurs changements induit par la nouvelle version du modèle. Enfin, nous complétons les suites existantes avec des nouveaux tests pour garantir la couverture demandée. Nous avons proposé de classifier les tests en **quatre** suites de tests : *Regression, Stagnation, Evolution* et *Deletion* qui permettent de tester respectivement le bon fonctionnement des exigences déjà existantes, leur suppression effective, la prise en compte des nouvelles exigences, et la conservation de l'historique des tests invalides des versions précédentes.

Pour ce faire, nous calculons pour chaque test un statut en regard de l'évolution. Ceci permet de définir des règles qui permettront d'établir les conditions qui mènent au changement de statut d'un test et de classer les tests dans chacune des quatre suites. Pour cela, nous proposons une nouvelle méthode de génération sélective de tests à partir de modèles appelée *SeTGaM (Selective Test Generation Method)*.

Suite aux différentes possibilités de modélisations proposées par UML/OCL, le chapitre suivant va définir la méthode pour des modèles de tests avec un diagramme d'états/-transitions. Dans un second temps dans les chapitres 7 et 8, nous allons étendre cette méthode pour les modèles sans diagramme d'états/transitions et pour les exigences de sécurité.

Chapitre 6

SeTGaM pour les diagrammes d'états-transitions

Sommaire

6.1 Différences dans un diagramme d'états/transitions 66
6.2 Calcul des dépendances 69
6.2.1 Dépendances de données 70
6.2.2 Dépendances de contrôle 72
6.3 Analyse des exigences impactées par l'évolution 76
6.3.1 Calcul des impacts 76
6.3.2 Règles de catégorisation des tests 78
6.4 Synthèse 80

Dans le chapitre précédent, nous avons défini les statuts des tests et leur classification dans les suites de tests. Nous présentons dans ce chapitre la déclinaison pour les diagrammes d'états/transitions de l'approche *SeTGaM (Selective Test Generation Method)*. Plus précisément, dans ce chapitre nous allons définir comment le statut de chaque test est attribué à partir des diagrammes d'états/transitions.

Nous travaillons avec des systèmes critiques et nous ne pouvons pas nous permettre d'omettre des tests. Il est très important de pouvoir identifier toutes les erreurs éventuellement introduites dans le code lors de l'évolution du système. D'une part, le calcul de différences nous permet d'identifier les éléments directement liés aux changements (modifications, suppressions, ajouts). D'autre part, par le biais des dépendances entre les différents éléments, il est nécessaire de capturer les éléments impactés, ceux qui sont indirectement affectés par l'évolution. À ce propos, les dépendances de données et de contrôle entre les transitions sont utilisées sur les diagrammes d'états/transitions. Prendre

en compte les deux types de dépendances est très important, Harrold et al. dans [RH96] confirment que les approches qui se basent seulement sur les dépendances de données ne sont pas considérées comme *sûres*. Ainsi, les deux types de dépendances sont complémentaires et permettent de prendre en compte le plus d'impacts possibles. Suite à cela, nous allons étendre les définitions pour les *Extended Finite State Machines*(EFSM) par Korel et al. dans [KHV02] et Chen et al. dans [UPC07] et les adapter pour les diagrammes d'états/transitions en UML/OCL. Enfin, nous allons utiliser les résultats obtenus par le calcul des différences et l'analyse des dépendances pour définir les règles d'attribution des statuts aux tests, dont la notion a été introduite dans le chapitre précédent.

L'organisation de ce chapitre est la suivante. Nous allons d'abord présenter l'algorithme de calcul de différence de comportements dans un diagramme d'états/transition en section 6.1, utilisé pour le calcul des exigences et tests impactés. L'extension des définitions de dépendance et leur adaptation pour la calcul de dépendances pour des diagrammes d'états/transitions et l'analyse du code OCL dans les transitions, sont détaillés dans la section 6.2. Enfin, nous allons donner les algorithmes et règles permettant de calculer les impacts et classifier les tests dans la section 6.3, avant de donner la synthèse en section 6.4.

6.1 Différences dans un diagramme d'états/transitions

Le premier élément dans le processus de sélection de tests est de définir les différences dans les exigences du système exprimées dans le diagramme d'états/transitions.

Une bonne pratique est de considérer qu'une transition représente une exigence définie par un comportement et est un ensemble de tags, identifiée par le mot-clef **TAGS**.

Définition 3 (Transition) *Une transition T appartenant au diagramme d'états/transitions DET est définie par un sept-uplet composé de : son nom N, l'opération déclencheur de la transition ou encore appelé Trigger, état de départ Source, état d'arrivée Destination, une garde G, une action A et une exigence TAGS, ainsi :*

$$\forall\, T \in DET.T = < N, Trigger, Source, Destination, G, A, TAGS >$$

Remarque : Pour accéder à un élément de T, on utilisera l'opérateur ".". Par exemple le nom est obtenu par $T.N$

Lors du calcul de différence, s'il n'existe qu'une version de modèle, alors toutes les exigences fonctionnelles sont définies en tant que nouvelles. Sinon le calcul de différence entre les machines à états est effectué pour calculer l'ensemble des exigences modifiées. D'après la définition d'une transition, nous avons un ensemble de tags par transition, noté *TAGS*. Rechercher les exigences modifiées revient à rechercher les différences des transitions dans le diagramme d'états/transitions. Pour cela, l'algorithme compare deux machines à états

dans le but d'extraire plus exactement les transitions : ajoutées, supprimées ou modifiées. Nous définissons ci-dessous les modifications autorisées pour une évolution du diagramme à états/transitions DET en DET'.

Définition 4 (Transition Ajoutée) *Une transition T' est dite ajoutée dans le diagramme d'états/transition DET' si et seulement si l'exigence TAGS de T' n'existe pas dans les autres transitions de DET i.e. la transition T' n'existe pas dans DET :*

$$T' \in Ajoutee \text{ si et seulement si } T' \in DET' \wedge (\forall\, T \in DET, T'.TAGS \neq T.TAGS)$$

Remarque : L'ensemble de transitions ajoutées est défini de la manière suivante :
$Ajoutee = \{\, T' \mid T' \in DET', \forall\, T \in DET$ tel que $T \neq T'\}$

Définition 5 (Transition Supprimée) *Une transition T est dite supprimée du diagramme d'états/transition DET' si et seulement si l'exigence TAGS de T n'existe pas dans les transitions de DET' i.e. la transition T' n'existe pas dans DET' :*

$$T \in Supprimee \text{ si et seulement si } T \in DET \wedge (\forall\, T' \in DET', T.TAGS \neq T'.TAGS)$$

Remarque : L'ensemble de transitions supprimées est défini de la manière suivante :
$Supprimee = \{\, T \mid T \in DET \wedge T \notin DET'\}$

Définition 6 (Transition Modifiée) *Une transition T' est dite modifiée dans le diagramme d'états/transition DET' si et seulement si l'exigence TAGS de T' existe dans une transition T de DET et que la garde G ou l'action A écrite en OCL est modifiée :*

$$T' \in Modifiee \text{ si et seulement si}$$
$$T' \in DET' \wedge (\exists\, T \in DET, T.TAGS = T'.TAGS \wedge (T.G \neq T'.G \vee T.A \neq T'.A))$$

Remarque : L'ensemble de transitions modifiées est défini de la manière suivante :
$Modifiee = \{\, T' \mid T' \in DET', \forall\, T \in DET$ tel que $T.TAGS = T'.TAGS$
$\wedge (T.G \neq T'.G \vee T.A \neq T'.A)\}$

Sur la base de ces définitions, nous proposons l'algorithme 1. Celui-ci calcule pour un diagramme d'états/transitions les différences entre deux versions de modèles. Nous avons défini les changements par un type énuméré TypeModification $\{NEW, DEL, MODIF\}$, qui dénote respectivement les transitions ajoutées, supprimées et modifiées. Nous utilisons dans l'algorithme des fonctions basiques qui nous permettent d'accéder aux éléments des modèles.

- *markAllTransitionAs(TypeModification)* : retourne toutes les transitions du diagramme d'états/transitions comme nouvelles, parce que le modèle vient d'être créé et il est nécessaire de générer des tests pour cette première version.

```
Entrées :
model_N : le modèle référence;
model_{N+1} : le modèle évolué;
Sorties : Set<Transition,TypeModification> listeDifferences : liste des différences entre les machines à
          états/transitions ;
si (model_{N+1}) = NULL alors
    return markAllTransitionAs(NEW);
fin
pour chaque Transition T ∈ model_{N+1} faire
    si (T ∉ model_N) alors
        listeDifferences.add(T,NEW);
    sinon
        si T ∈ model_N alors
            T' = (model_{N+1}).getTransitionSameAs(T);
            si T'.sameActionGuardAs(T) alors
                listeDifferences.add(T',MODIF);
            fin
        fin
    fin
fin
pour chaque Transition T ∈ model_N faire
    si (T ∉ model_{N+1}) alors
        listeDifferences.add(T,DEL);
    fin
fin
retourner listeDifferences;
```

Algorithme 1: Comparaison des machines à états/transitions

- *getStatechart* : s'applique au modèle UML et permet de récupérer le diagramme d'états/transitions.
- *add(Transition, TypeModification)* : s'applique sur la liste de résultats et permet d'ajouter une transition et le type de modification apporté à cette dernière.
- *getTransitionSameAs(Transition)* : s'applique sur un diagramme d'états/transitions et permet de trouver une transition selon le nom et les tags de la transition passée en paramètre.
- *sameActionGuardAs(Transition)* : s'applique sur une transition et retourne vrai si la transition passée en paramètre possède la même action ou la même garde, faux sinon.

Dans le cas d'eCinema et son évolution, nous allons illustrer le calcul des dépendances sur un fragment du modèle i.e. l'achat d'un ticket par un utilisateur, représenté par l'opération *buyTicket* et par la transition $buyTicket - success$ et enregistrement d'un nouvel utilisateur, représenté par l'opération *registration* et la transition $registration - success$. Une des évolutions d'eCinema est d'ajouter des abonnements : jeune, étudiant, senior, dix tickets achetés un gratuit et normal.

Le tableau 6.1, ci-dessous, liste les différences des transitions que nous considérons pour illustrer ce principe. Plus en détails, une transition est supprimée, cinq nouvelles, une modifiée et une inchangée.

Notre but est de générer des tests correspondant à *l'achat d'un ticket avec succès* par chaque type utilisateur et d'abonnement. Pour cela, nous ajoutons cinq nouvelles tran-

sitions, une pour chaque type d'utilisateur. Cependant, la transition initialement créée *byuTicket − success* est inutile, puisqu'elle est remplacée par les cinq nouvellement ajoutées, tel que synthétisé sur le tableau 6.1.

Opération	Transition	TAGS	Évolution
buyTicket	buyTicket-success	BUY_TICKET Buy_success	Supprimée
buyTicket	buyTicket-young	BUY_TICKET Buy_success,Young	Ajoutée
buyTicket	buyTicket-student	BUY_TICKET Buy_success,Student	Ajoutée
buyTicket	buyTicket-senior	BUY_TICKET Buy_success,Senior	Ajoutée
buyTicket	buyTicket-oneFree	BUY_TICKET Buy_success,OneFree	Ajoutée
buyTicket	buyTicket-normal	BUY_TICKET Buy_success,Normal	Ajoutée
registration	registration-success	ACCOUNT_MNGT/REG Reg_Success	Modifiée
login	login-success	ACCOUNT_MNGT/LOG Log_Success	Inchangée

TABLE 6.1 – Différence de transitions pour un extrait d'eCinema

De plus, l'évolution impose une modification du processus d'enregistrement. Nous devons ajouter un attribut et l'opération *registration* doit prendre en compte le type d'abonnement de l'utilisateur lors de son enregistrement. Pour cela, nous modifions la transition *registration − success* afin de prendre en compte cette modification. Cependant une fois enregistré, pour se connecter au système l'opération *login* et son transition *login − success* se comportent comme à la version précédente et restent inchangées. Dans la section suivante, nous allons définir des dépendances de données et de contrôle entre ces transitions dans un diagramme d'états/transitions.

6.2 Calcul des dépendances

Les dépendances entre les transitions dans le diagramme d'états/transitions nous permettent d'identifier les transitions dont le comportement peut être impacté par les changements. Pour connaitre les comportements impactés par les changements, nous calculons les dépendances entres les transitions du diagramme d'états/transitions. Dans cette partie, nous donnons les définitions et les algorithmes permettant de calculer les dépendances de données et de contrôle. Ces travaux se basent sur l'analyse des dépendances de Korel et al. [KHV02] et Chen et al. [UPC07] établis pour les EFSM. Nous étendons leur démarche aux diagrammes d'états/transitions en UML/OCL et nous proposons une nouvelle démarche sur l'analyse des dépendances entre exigences basées sur ces diagrammes dans l'objectif de sélectionner et classifier des tests.

6.2.1 Dépendances de données

Les dépendances de données sont calculées à partir des couples de variables définies et utilisées dans un diagramme à états/transitions en UML/OCL, qui respectent la définition de dépendances de données.

Nous définissons d'abord ce qu'est une transition de définition et transition utilisatrice.

Définition 7 (Transition de définition) *La transition qui modifie la valeur d'une variable v lors d'une instruction OCL (affectation, déclaration) est appelée transition de définition.*

Remarque : La modification d'une variables dans une instruction OCL d'une transition est notée *Def*.

Définition 8 (Transition utilisatrice) *Une transition qui utilise la valeur d'une variable v dans son code OCL, par rapport à sa définition dans une autre transition (sans aucune redéfinition) est appelée transition utilisatrice.*

Remarque : L'utilisation d'une variable dans le code OCL d'une transition est notée *Use*.

En nous basant sur cette propriété, nous donnons la définition d'une transition data dépendante par rapport à une autre.

Définition 9 (Transition data dépendante) *Une transition T' est data dépendante d'une autre transition T par rapport à la variable v si et seulement si T est transition de définition de v et T' transition utilisatrice de v.*

L'ensemble de triplets $\langle T, T', v\rangle$ est appelé ***graphe de dépendances de données****.*

Ces définitions sont le point de départ pour proposer l'algorithme 2 de calcul des dépendances de données pour les diagrammes d'états/transitions en UML/OCL. Le calcul prend en compte les définitions et les utilisations des variables dans le code OCL, mais aussi les attributs des classes. Plus exactement les définitions sont collectées à partir du code OCL dans l'action ou de la post condition de l'opération événement déclencheur de la transition *(trigger)*. Les utilisations sont déterminées dans la garde/action d'une transition ou dans la pre/post condition d'une opération. La spécificité du code OCL en UML4MBT nous autorise à faire des appels d'opérations dans l'action d'une transition ou dans la post condition d'une opération. Pour cela, nous devons prendre en compte ces appels afin de calculer toutes les définitions et les utilisations des variables.

Dans l'algorithme 2, nous proposons tout d'abord d'établir toutes les définitions et utilisations des variables dans la machine à états/transitions. Ensuite, nous proposons

de lier chaque transition utilisatrice d'une variable à sa transition de définition, si et seulement si il n'y aucune redéfinition. Pour identifier cette condition dans l'algorithme, nous utilisons le nom *def-clear*.

Entrées : model : le modèle avec un diagramme d'états/transitions
Sorties : dataDependence(X,Y,v) : Tableau tri-dimensionnel de type booléenne, qui indique *vrai* si Y est data dépendent de X pour la variable v, et *faux* sinon
Données :

```
Set⟨Transition,Variable⟩ definitions = model->gatherAllDefinitions();
Set⟨Transition,Variable⟩ uses = model->gatherAllUses();
pour chaque <Transition,Variable> Def ∈ definitions faire
    Transition defTransition = Def.getTransition();
    Variable variable = Def.getVariable();
    pour chaque <Transition,Variable> Use ∈ uses faire
        Transition useTransition = Use.getTransition;
        si def-clear(defTransition,useTransition,variable) alors
            dataDependence(defTransition,useTransition,variable) = true;
        fin
    fin
fin
```

Algorithme 2: Dépendance de données entre transitions

L'algorithme ainsi proposé utilise cinq opérations dont nous allons donner l'explication ci-dessous :

- *gatherAllDefinitions* : s'applique sur le modèle de test et renvoie en résultat l'ensemble de couples <Transition,Variable> définies dans l'action de la transition ou encore dans la condition du trigger (variables locales ou attributs de classes ou associations).
- *gatherAllUses* : s'applique sur le modèle de test et en résultat renvoie l'ensemble de couples <Transition,Variable>, variables utilisées au sein de l'action/garde de la transition (attributs de classes ou associations) ou dans la pré/post condition du trigger. A noter que les appels d'opérations sont pris en compte.
- *getTransition* : s'applique sur le couple <Transition,Variable> et permet de recouper le premier élément du couple.
- *getVariable* : s'applique sur le couple <Transition,Variable> et permet de recouper le deuxième élément du couple.
- *def-clear* : opération permettant de calculer si un chemin entre la transition utilisatrice et la transition de définition par rapport à une variable est unique i.e. sans aucune redéfinition de la variable dans une autre transition faisant partie du chemin.

Afin de représenter les liens de dépendances de données nous allons prendre les cas de succès des opérations : *buyTicket, login et register*, représentés par les transitions : *buyTicket-success, login-success et register-success*. Ces transitions sont représentées dans le diagramme d'états/transitions simplifié de la figure 6.1.

Dans le tableau 6.2, partie de gauche du tableau, nous récapitulons les variables définies et utilisées dans chaque transition.

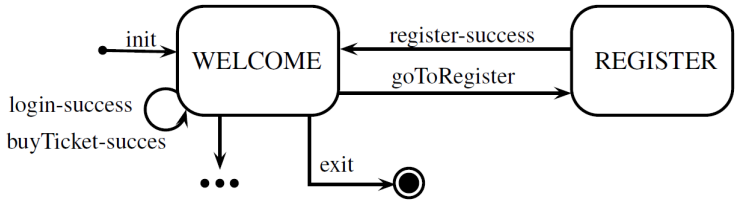

FIGURE 6.1 – Extrait du diagramme d'états/transitions d'eCinema

	Variables		**Dépendances de données**		
	Def	Use	buy-success	register-success	login-success
buy-success	all_bought_tickets	current_user		x	x
register-success	current_user				
	all_registered_users	all_registered_users		x	
login-success	current_user	current_user		x	x
		all_registered_users		x	

TABLE 6.2 – Def/Use pour un extrait d'eCinema

Sur la partie de droite du tableau 6.2 nous définissons les relations de dépendance de données. La condition de chemin def-clear est satisfaite par tous les couples def/use. Nous pouvons constater que *buy-success* et *login-success* sont dépendantes de *register-success* et de *login-success* par rapport à l'utilisation/définition de la variable *current_user*. Les transitions *register-success* et *login-success* sont dépendantes de *register-success* par rapport à l'utilisation/définition de la variable *all_registered_users*.

Nous allons voir dans la section 6.3 comment les dépendances de données en mêlant les dépendances de contrôle, définies dans la sous-section suivante 6.2.2, contribuent à l'identification des exigences impactées par les changements.

6.2.2 Dépendances de contrôle

Les dépendances de contrôle sont basées sur la post-dominance entre état et état et entre état et transition. Pour ce faire, nous allons proposer trois algorithmes, associés à chaque définition donnée. Puis nous proposons l'algorithme global qui s'appuie sur ces dernières.

Définition 10 (Post-Dominance entre États) *Un état S post-domine un autre état V du diagramme d'états/transitions si et seulement si tous les chemins partant de V jusqu'à l'état de final F passent pas S.*

Pour construire l'algorithme correspondant à la définition de *Post-dominance entre États* nous nous sommes retrouvés face au problème de *cycle*, comme la présence de

transitions réflexives dans les diagrammes. Ceci autorise un nombre infini de passages par une transition, donc un parcours entre deux états possiblement infini. Afin de résoudre ce problème, à chaque passage par une transition, nous la marquons comme visitée, et nous continuons le parcours tant qu'il reste des transitions non-visitées. Pour chaque nouveau parcours, les transitions sont remises à l'état initial i.e non-visités.

```
   Entrées : S,V : State - états du diagramme d'états/transitions
   Sorties : res : Boolean - variable booléen
   Données : model : le modèle avec un diagramme d'états/transitions;
 1 res = true;
 2 si V = S alors
 3     return true;
 4 sinon
 5     si V.isFinalState() alors
 6         return false;
 7     sinon
 8         si V ∈ model.getVisistedStates() alors
 9             return true;
10         sinon
11             tant que (V.getNbNonVisitedNeighbours()>0 & res = true) faire
12                 State B = V.getFirstNonVisitedNeighbour();
13                 V.visitNeighbour(B);
14                 res = res & totalAccessFromStateToState (S,B);
15             fin
16             model.visit(V);
17             return res;
18         fin
19     fin
20 fin
```

Procédure totalAccessFromStateToState(S,V)

La procédure totalAccessFromStateToState est un parcours en profondeur *(first-depth)* qui détermine si un état V du diagramme d'états/transitions est post-dominé par un autre état S. Il renvoie vrai si tout chemin de l'état V passe par S, faux sinon. Cette idée est exprimée dans la procédure récursive totalAccessFromStateToState dont la condition d'arrêt est quand :

- les états S et V sont les mêmes (ligne 2)
- l'état final est atteint sans passer par l'état S (ligne 5)
- l'état V a été déjà visité (ligne 8)
- il n'y a plus de voisins de l'état V à visiter ou qu'il a pu conclure sur la post-dominance (ligne 11)

Il est à noter, que nous considérons six opérations qui seront seulement décrites :

- *getNbNonVisitedNeighbours()* : pour un état, il donne le nombre d'états voisins non-visités.
- *getFirstNonVisitedNeighbour()* : pour un état, il renvoie le premier voisin non-visité d'un état donné.
- *getVisistedStates()* : renvoie l'ensemble d'états dont tous les états voisins ont été visité lors du parcours. Cet état est dit *complètement visité*.
- *visitNeighbour(State)* : s'applique sur un état et permet de marquer l'état voisin *State* en tant que visité.

– *visit(State)* : permet de marquer un état *State* comme complètement visité.
– *isFinalState()* : permet de vérifier si un état est l'état final.

```
Entrées :
S, S' : de type State qui représente les états du diagramme d'états/transitions
Sorties : postDominance : martice [State]x[State] où [S][S'] est vrai si l'état S post-domine S', faux sinon;
Données : model : le modèle;
pour chaque State S ∈ model faire
    pour chaque State S' ∈ model où S ≠ S' faire
        postDominanceStateState[S][S'] = totalAccessFromStateToState (S,S');
        model.ReInitVisit() ;
    fin
fin
```

Algorithme 3: Post-dominance entre États ⇝ post-domine(S,S')

L'algorithme 3 de *Post-dominance entre États* détermine pour chaque état s'il post-domine un autre état en complétant par *vrai* ou *faux* le tableau de post-dominance. À chaque boucle dans l'algorithme, l'appel à la procédure totalAccessFromStateToState est fait sur les deux états sélectionnés. Après l'exécution de la procédure totalAccessFromStateToState, les états visités sont remis à leur état initial i.e. non-visité par la fonction *ReInitVisit*.

En respectant la définition 11, nous recherchons s'il existe un état qui post-domine une transition, défini par l'algorithme 4.

```
Entrées :
S : état du diagramme d'états/transitions;
T : transition du diagramme d'états/transitions;
Sorties :
postDominanceStateTransition : matrice [State][Transition] où [S][T] est vrais si l'état S post-domine la transition
T;
Données : model : le modèle;
pour chaque State S ∈ model faire
    pour chaque Transition T ∈ model faire
        si T.arriving()=S & T.departure() ≠ S alors
            postDominanceStateTransition[S][T] = true;
        sinon
            si (postDominanceStateState[S][T.arriving()]) alors
                postDominanceStateTransition[S][T] = true;
            fin
        fin
    fin
fin
```

Algorithme 4: Post-dominance État/Transition ⇝ post-domine(S,T)

Définition 11 (Post-Dominance État/Transition) *Un état S post-domine une transition T du diagramme d'états/transitions si et seulement si tout chemin passant par la transition T passent obligatoirement par S pour atteindre l'état final F.*

Dans l'algorithme 4, nous utilisons deux fonctions qui permettent de récupérer l'état cible ou l'état source d'une transition :

– arriving() : s'applique sur une transition et renvoie l'état d'arrivée d'une transition i.e l'état cible.
– departure() : s'applique sur une transition et renvoie l'état de départ d'une transition i.e l'état source.

En disposant des définitions 10 et 11 et les algorithmes 3 et 4 respectifs nous pouvons constituer la définition sur les dépendances de contrôle 12 et son algorithme 5.

```
Données : model : le modèle;
T, T' : transitions du diagramme d'états/transitions;
Sorties : controlDependance : matrice de type [Transition]x[Transition] où [T][T'] est vrai si T' est contrôle
          dépendante de T;
pour chaque Transition T ∈ model faire
    State sourceT = T.departure();
    pour chaque Transition T' ∈ model faire
        State sourceT' = T'.departure();
        si !post-domine(sourceT',sourceT) & post-domine(sourceT',T) alors
            controlDependance[T][T']=true;
        fin
    fin
fin
```

Algorithme 5: Dépendance de contrôle

Définition 12 (Dépendance de contrôle) *Une transition T contrôle une autre transition T' si et seulement si : (i) l'état parent de T' ne post-domine pas l'état parent de la transition T, et (ii) l'état parent de T' post-domine la transition T.*

Remarque : L'ensemble de couples $\langle T, T' \rangle$ est appelé **graphe de dépendances de contrôle**.

Nous allons illustrer le calcul de dépendances de contrôle sur la transition *register − success*, représentée sur la figure 6.2.

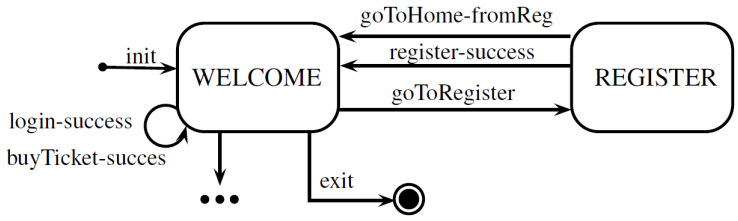

FIGURE 6.2 – Extrait du diagramme d'états/transitions d'eCinema

Sur la partie gauche *Post-dominance* du tableau 6.3, nous donnons les résultats sur les post-dominances entre état/état et état/transition, issus du diagramme d'états/transitions de la figure 6.2.

Sur la partie droite *Dépendance de contrôle* du tableau 6.3, nous récapitulons les dépendances de contrôle, issues du diagramme d'états/transitions et les post-dominances

	Post-dominance		Dépendance de contrôle		
	welcome	register	register-success	goToHome-fromReg	goToRegister
register-success	x				x
goToHome-fromReg	x				x
goToRegister		x			
welcome					
register	x				

TABLE 6.3 – Post-dominance et dépendances de contrôle sur un fragment d'eCinema

état/transition et état/état. En appliquant l'algorithme 5, nous constatons que l'état *register*, qui est l'état de départ de *register-success*, ne post-domine pas l'état *welcome* qui est l'état de départ de *goToRegister* et que la transition *goToRegister* est post-dominée par l'état *register*.

Suite à ces résultats, issus du tableau 6.3, nous pouvons conclure que la transition *register-success* est contrôle dépendante de la transition *goToRegister*. De même, nous pouvons conclure que la transition *goToHome-fromReg* est contrôle dépendante de la transition *goToRegister*.

Ces dépendances entre les transitions ajoutées au résultat du calcul des différences vont ensuite permettre de définir les transitions indirectement impactées par l'évolution. L'ensemble de tous ces résultats sera utilisé pour la gestion du cycle de vie des tests.

6.3 Analyse des exigences impactées par l'évolution

Dans cette section, nous décrivons l'analyse de dépendances de données et de contrôle dans le but d'identifier et classifier les tests existants de la version précédente du modèle. Nous détaillons la façon dont les exigences impactées sont calculées et ensuite les règles de classification des tests vis-à-vis de la nouvelle version du modèle.

6.3.1 Calcul des impacts

Afin de calculer les transitions impactées dans une version évoluée d'un diagramme d'états/transitions, nous nous basons sur le calcul de la différence entre les deux versions de modèles et les dépendances obtenues pour les deux versions (version évoluée et courante). L'algorithme 6 de calcul d'impacts prend ainsi en compte les dépendances des deux versions et les différences entre les versions. Nous allons considérer la fonction *merge* qui prend en compte deux diagrammes de dépendances D et D' en entrée et elle renvoie un diagramme *dependenceDiff* composée des deux diagrammes en résultat. La composition est faite en utilisant le principe de marquage des graphes de façon à ce que :

– les dépendances qui sont supprimées dans D' par rapport au D sont ajoutées à

l'ensemble *dependenceDiff* et marquées par le signe distinctif *deleted*.
- les dépendances qui sont ajoutées dans D' par rapport au D sont ajoutées à l'ensemble *dependenceDiff* et marquées par le signe distinctif *added*.
- toutes les autres sont ajoutés à *dependenceDiff* et par le signe distinctif *none*.

Pour le calcul des impacts, nous considérons qu'une arrête dans le graphe de dépendances est représentée par le quadruple (t, t', v, m) où t' est la transition dépendante de t, et v est soit la variable (dans le cas d'une dépendance de données), soit vide si dépendance de contrôle, m est la marque mise à jour lors de l'appel de la fonction *merge*. Nous dénotons l'ensemble de transitions impactées *Impactee*. Les transitions qui ne sont ni impactées, ni modifiées, ni ajoutées, ni supprimées du diagramme d'états/transitions sont appelées des transitions non-impactées. Leurs ensemble est dénoté *Nonimpactee*.

```
Données : diff : ensemble de différences entre les modèles;
D, D' : dépendances pour les deux modèles d'états/transitions M et M' respectivement;
Sorties : impacts : ensemble de transitions impactées par le changement;
dependeceDiff = merge(D,D');
pour chaque Dépendance Q(t,t',v,m) ∈ dependenceDiff faire
    si t,t' ∉ diff & m ≠ none alors
        impacts.add(t);
        impacts.add(t');
    sinon
        si t ∈ diff & t' ∉ diff alors
            impacts.add(t');
        fin
    fin
fin
```

Algorithme 6: Calcul des transitions impactées lors d'évolution de la spécification

Lors du calcul des impacts dans le diagramme d'états/transitions, chaque transition est associée à une exigence. De cette manière, nous pouvons remonter les exigences impactées à l'utilisateur.

Transition	**Dépendances de données**			**Dépendances de contrôle**	**Impactée**
	Variable	login-success	register-success *	goToRegister	
buy-success *	current_user	x	x		
login-success		x	x		x
	all_registered_users		x		
register-success *			x	x	
goToHome-fromReg				x	

TABLE 6.4 – Calcul des impacts sur un fragment d'eCinema

Pour illustrer le calcul de transitions/exigences impactés, dans le tableau 6.4 nous présentons un récapitulatif des dépendances entre les transitions et le résultat du calcul des impacts. Sur le tableau nous marquons par $*$ les transitions changées suite à l'évolution. Pour rappel, $buyTicket - success$ est supprimée et cinq nouvelles transitions sont ajoutées et la transition $register - success$ est modifiée. Nous pouvons constater que la transition $login - success$ est impactée suite aux modifications dans les dépendances avec $buyTicket - success$ et $register - success$. Ces dernières sont aussi impactées par l'évolution, mais elles

font partie de l'évolution, pour cela nous ne les considérons pas dans l'ensemble des impactées.

6.3.2 Règles de catégorisation des tests

Les règles de classification des tests dans les suites de tests ont été définies dans le chapitre précédent. Dans cette section, nous donnerons les règles de catégorisation d'un test ou encore d'attribution de statut d'un test, identifié à partir des diagrammes d'états/transitions.

Nous pouvons dire que le principe de la catégorisation des tests se base sur deux versions de modèles et se déroule en trois étapes :

1. calculer les exigences impactées,
2. définir les statuts des tests existants,
3. générer des tests *new* pour maintenir la couverture des exigences.

La première étape a été détaillée en section 6.3.1. Par la suite, nous prendrons en compte les changements et les éléments impactés et nous allons les utiliser pour déterminer le statut des tests. De manière informelle, le processus va pour chaque exigence supprimée classifier les tests correspondants en tant qu'*outdated*, pour chaque exigence modifiée ou impactée, classifie les tests correspondants dans la catégorie intermédiaire *retestables*. Les tests de cette catégorie seront animés sur le nouveau modèle afin de déterminer leur statut définitif et les classifier en tant que : *updated, adapted, failed, reexecuted*. Pour les nouvelles exigences, nous allons utiliser l'outil de génération pour créer de nouveaux tests. Enfin pour toutes les autres exigences, l'algorithme classifie les tests correspondants (s'ils n'ont pas été déjà sélectionnées précédemment) en tant qu'*unimpacted*. Nous rappelons qu'un test peut couvrir plusieurs exigences. Afin d'avoir un classement déterministe et non-ambigüe de chaque test, nous donnons des critères de supériorité dans la détermination de statut d'un test existant du plus fort vers le moins fort :

$$\textbf{outdated} \rightarrow \textbf{retestable} \rightarrow \textbf{unimpacted}$$

En tenant compte de la priorité de cet ordonnancement, nous formalisons la classification par les règles définies ci-dessous. Afin de maintenir la couverture sur les nouvelles cibles ainsi que les existantes, nous définissons les règles suivantes. Pour construire nos règles, nous utilisons la définition du test donnée dans la section 2.3, les définitions de transition ajoutée, supprimée, modifiée données en section 6.1, ainsi que les notations de transition impactée et non-impactée, section 6.3.1.

Règle 6 (RT-New) *Le test correspondant à une nouvelle transition qui n'a été couverte par aucun test existant.*
$\forall\, T \in Ajoutee \rightsquigarrow\ \ status(tc^{n+1}) = new$

Règle 7 (RT-Unimpacted) *Le test correspondant à une transition non impactée et non modifiée i.e non impactée est classifié en tant qu'Unimpacted.*
$\forall\ T \in Nonimpactee\ and\ T.TAGS \in tags(tc^{n+1}) \rightsquigarrow\ status(tc^{n+1}) = unimpacted$

Règle 8 (RT-Outdated) *Le test correspondant à une transition supprimée est classifié en tant qu'Obsolete.*
$\forall\ T \in Supprimee\ and\ T.TAGS \in tags(tc^{n+1}) \rightsquigarrow\ status(tc^{n+1}) = outdated$

Règle 9 (RT-Retestable) *Le test correspondant à une transition modifiée ou à une exigence impactée est classifié en tant que Retestable.*
$\forall\ T \in \{Modifiee\ \cup Impactee\}\ and\ T.TAGS \in tags(tc^{n+1})$
$\rightsquigarrow\ status(tc^{n+1}) = retestable$

Les tests sont ensuite animés sur le modèle afin d'obtenir leur statut définitif : reexecuted, updated, failed.

Toujours pour maintenir la couverture des cibles, il faut prendre en compte les tests qui n'ont pas pu être animés sur le nouveau modèle ou les tests *outdated* qui couvrent des exigences qui peuvent toujours exister dans la nouvelle version. Pour ce faire, nous devons générer des tests pour les couvrir. Ces tests ne peuvent pas être désignés en tant que nouveaux, puisqu'ils ne couvrent pas de nouvelles exigences, nous les appelons *adapted*.

Règle 10 (RT-Adapted) *Le test correspond à une exigence existante et non couverte par les tests actuels. Elle devient une cible de tests pour laquelle un test adapted est créé.*
$\forall\ T \in\ uncovered \rightsquigarrow\ status(tc^{n+1}) = adapted$

Nous considérons l'évolution de *buyTicket* et *register* présentés dans la Section 6.1, le détail sur le calcul de dépendances et d'impacts est respectivement donné en Section 6.2 et Section 6.3.1. Afin d'illustrer la classification des tests, nous considérons les trois tests suivants :

```
(1) ECinema::sut.login(REGISTERED_USER, REGISTERED_PWD);
    ECinema::sut.buyTicket(TITLE1);
(2) ECinema::sut.goToRegister();
    ECinema::sut.register(UNREGISTERED_USER, UNREGISTERED_PWD);
    ECinema::sut.buyTicket(TITLE1);
(3) ECinema::sut.login(REGISTERED_USER, REGISTERED_PWD);
    ECinema::sut.logout();
```

Les tests (1) et (2) sont classifiés en tant que *outdated*, suite aux calculs de différences et la suppression de l'exigence :

$TAGS : BUY\_TICKET, Buy\_Success$

Cependant, le test (2) couvre une exigence qui est toujours présente, même si elle est modifiée. Le générateur de tests est appelé et crée un test **adapted** afin de la couvrir :

```
(2) Adapted ECinema::sut.goToRegister();
```

```
    ECinema::sut.register(UNREGISTERED_USER, UNREGISTERED_PWD, SUB_NORMAL);
    ECinema::sut.buyTicket(TITLE1);
```

Enfin, le dernier test (3) est classifié d'abord en tant que *retestable*, suite à l'impact sur la transition *login-succes*. Après exécution sur le modèle, il est classifié en tant que *updated*, parce que son oracle a changé i.e. l'instance de l'utilisateur est modifié et il contient un abonnement. Il est à noter que l'exécution du test sur le système se comporte comme prévu précédemment.

```
(3) Updated ECinema::sut.login(REGISTERED_USER, REGISTERED_PWD);
    ECinema::sut.logout();
```

Finalement cinq nouveaux tests sont créés par le générateur pour couvrir les nouvelles exigences.

```
(1) ECinema::sut.login(YOUNG_USER, YOUNG_PWD);
    ECinema::sut.buyTicket(TITLE1);
(2) ECinema::sut.login(STUDENT_USER, STUDENT_PWD);
    ECinema::sut.buyTicket(TITLE1);
(3) ECinema::sut.login(SENIOR_USER, SENIOR_PWD);
    ECinema::sut.buyTicket(TITLE1);
(4) ECinema::sut.login(ONEFREE_USER, ONEFREE_PWD);
    ECinema::sut.buyTicket(TITLE1);
(5) ECinema::sut.login(NORMAL_USER, NORMAL_PWD);
    ECinema::sut.buyTicket(TITLE1);
```

6.4 Synthèse

Nous avons proposé une technique basée sur les modèles UML/OCL, et plus particulièrement les diagrammes à états/transitions, pour prendre en compte le test des systèmes évolutifs. Notre but est d'éviter la re-génération complète de la suite de tests pour chaque évolution du système. Nous avons proposé d'analyser les dépendances extraites du modèle en fonction de l'évolution faite, et ainsi d'identifier automatiquement les tests qui ont été impactés par l'évolution, et ceux qui ne l'ont pas été. Nous avons introduit dans le chapitre 5 la notion de cycle de vie du test.

Dans ce chapitre, nous avons défini les règles qui permettent l'identification de chaque statut de test, s'il est toujours valide pour la nouvelle version ou s'il est obsolète ou s'il doit être recalculé. L'exemple fil rouge illustre bien l'utilité de la méthode, qui permet d'avoir une génération et une gestion de tests sophistiquée, ce qui est une des préoccupations principales dans l'industrie. Une gestion de la traçabilité des exigences est garantie par les tags. Finalement, l'historique est gardé et assuré d'une version à une autre, ce qui facilite le travail de l'utilisateur. Dans le chapitre suivant, nous présentons la méthode appliquée aux modèles n'utilisant pas le diagramme d'états/transitions.

Chapitre 7

SeTGaM pour des comportements UML/OCL

Sommaire

7.1 Définition des comportements en UML/OCL 82

7.2 Différences des comportements dans les opérations 83

7.3 Dépendances entre comportements 85

7.4 Analyse des impacts des comportements 88

7.4.1 Calcul des comportements impactés 88

7.4.2 La catégorisation de tests à partir des comportements . . . 89

7.5 Synthèse 91

Ce chapitre complète les définitions pour la sélection de tests avec la prise en compte d'autres façons d'exprimer l'aspect comportemental lors de la phase de modélisation. Ces comportements sont exprimés dans les pré et post conditions OCL des opérations dans le diagramme de classes.

L'utilisation de diagrammes d'états/transitions aide à l'identification de dépendances entre les exigences. Ces relations de dépendance sont utilisées pour définir les exigences impactées dans le système, comme cela est détaillé dans le chapitre 6. Cependant, pour des raisons de complexité ou de modélisation, il est souvent difficile, voire impossible, d'exprimer le système par le biais d'un diagramme d'états/transitions.

Afin de pouvoir effectuer une analyse des comportements impactés par l'évolution et de classifier les tests, nous allons donner la définition : d'un comportement, des différences entre comportements et des dépendances entre comportements.

7.1 Définition des comportements en UML/OCL

Une opération OCL est définie par une pré-condition (notée *PRE*) et une post-condition (notée *POST*). Généralement, elle possède différents comportements, identifiés par des exigences et notés par l'ensemble de tags (noté *TAGS*), selon le contexte dans lequel l'opération est appelée. Ainsi, un comportement est défini par le triplet pré-condition, post-condition et exigences. Nous définissons le comportement d'un modèle de la façon suivante :

Définition 13 (Comportement d'un modèle) *Un comportement B correspond à tout ou partie d'une opération du diagramme de classe d'un modèle en UML/OCL et il est défini par le triplet pré-condition, post-condition et tags :*

$$B =< PRE, POST, TAGS >$$

Nous faisons apparaître les différents comportements lors de la présence des conditions (if, then, else) dans la post-condition d'une opération. Les actions décrites par le "then" correspondent à un comportement et les actions dans le "else" à un autre, comme le décrit la Figure 7.1.

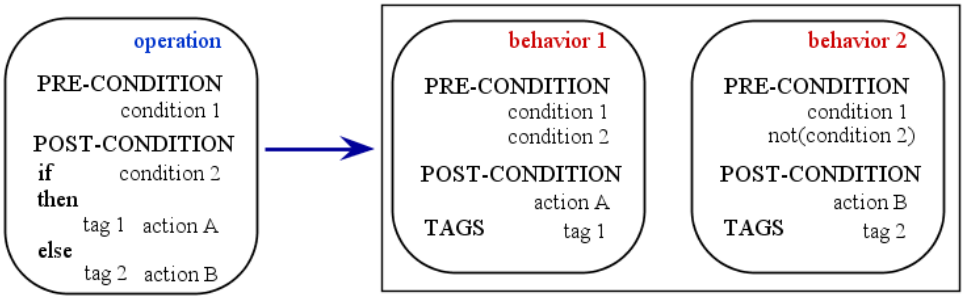

FIGURE 7.1 – Transformation d'une opération en comportements

La post-condition correspond aux actions exécutées par le comportement. La pré-condition est l'union des pré-conditions de l'opération et des conditions nécessaires pour atteindre l'action de la post-condition. Lorsqu'une post-condition du comportement fait partie du "else" de la condition, la négation de la condition est sauvegardée en tant que pré-condition du comportement. Les tags du comportement réfèrent aux exigences couvertes par la post-condition du comportement.

Dans les conditions OCL nous acceptons des opérateur logiques unaires, tels que *not* et binaires, tels que *<, >, =, >=, <=, <>, and, xor, or*. Lors de la présence de *or* dans une condition, nous la décomposons en trois comportements qui correspondent aux trois valeurs satisfaisant l'expression, par exemple l'expression *"A or B"* est dépliée en trois cas possibles : *"A and B"*, *"not A and B"* et *"A and not B"*. Ainsi, chaque comportement

contenant un opérateur *or* dans sa post-condition est séparé en trois comportements avec les mêmes tags et post-conditions mais les pré-conditions sont différentes. Ainsi, une fois que les comportements sont extraits des opérations, le but est de faire disparaitre cet opérateur. Il est à noter que lors d'un très grand nombre d'expressions avec l'opérateur logique *or* nous pouvons arriver à une explosion combinatoire dans le nombre des comportements obtenus. Pour éviter cela, nous avons autorisé l'utilisation de la syntaxe pour bloquer la réécriture, ainsi *"(A or B) = true"*est considéré comme un seul comportement.

Sur la figure 7.2, nous détaillons la transformation d'une partie de l'opération *buyTicket* en comportements. Pour cet exemple, nous avons construit deux comportements $buy - logErr1$ et $buy - success$. Le premier représente le cas d'échec lors de l'achat d'un ticket par un utilisateur non identifié. Le deuxième représente le cas d'un achat réussi par un utilisateur connecté.

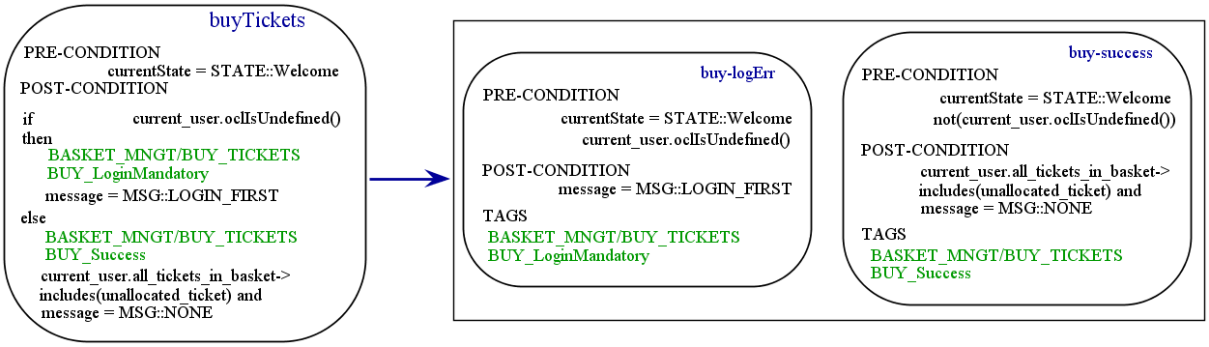

FIGURE 7.2 – Transformation d'un fragment de l'opération "buyTicket"

Dans la section suivante, nous allons montrer comment l'évolution entre deux modèles est calculée et l'illustrer sur le fragment d'eCinema.

7.2 Différences des comportements dans les opérations

Nous avons défini qu'une transition dans un diagramme d'états/transitions représente un comportement (Section 6.1). Actuellement, nous avons extrait les comportements à partir du code OCL des opérations dans un diagramme de classe. De ce fait, la définition des différences des comportements repose sur la définition des transitions ajoutées, supprimées ou modifiées. Donc, nous pouvons dire qu'un comportement est identique à un autre s'il a les mêmes tags, pré- et post-conditions. Dans le cas où la pré- ou la post-condition diffère, nous considérons le comportement comme modifié. Cependant, deux comportements ayant des tags différents sont considérés comme deux comportements distincts. Nous donnons formellement les définitions ci-dessous :

Définition 14 (Comportement Ajouté) *Un comportement B' est considéré comme ajouté dans le diagramme de classes DC' si et seulement s'il existe l'exigence TAGS de B' différente des exigences TAGS de tous les autres comportements B du DC :*

$$B' \in Ajoute \text{ si et seulement si } B' \in DC' \wedge (\forall\, B \in DC, B'.TAGS \neq B.TAGS)$$

Remarque : L'ensemble de comportements ajoutés est défini de la manière suivante :
$Ajoute = \{B' \mid B' \in DC', \forall\, B \in DC$ tel que $B \neq B'\}$

Définition 15 (Comportement Supprimé) *Un comportement B est considéré comme supprimé du diagramme de classes DC' si et seulement si l'exigence TAGS de B n'existe pas dans les comportements du diagramme de classes DC' :*

$$B \in Supprime \text{ si et seulement si } B \in DC \wedge (\forall\, B' \in DC', B.TAGS \neq B'.TAGS)$$

Remarque : L'ensemble de comportements supprimés est défini de la manière suivante :
$Supprime = \{B \mid B \in DC \wedge B \notin DC'\}$

Définition 16 (Comportement Modifié) *Un comportement B' est considéré comme modifié dans le diagramme de classes DC' si et seulement si l'exigence TAGS de B' est équivalente à l'exigence TAGS de B dans DC et que leur pré-condition PRE ou post-condition POST écrite en OCL sont modifiées :*

$$B' \in Modifie \text{ si et seulement si } B' \in DC' \wedge$$
$$(\exists\, B \in DC, B.TAGS = B'.TAGS \wedge (B.PRE \neq B'.PRE \vee B.POST \neq B'.POST)$$

Remarque : L'ensemble de comportements modifiés est défini de la manière suivante :
$Modifie = \{B' \mid B' \in DC', \forall\, B \in DC$ tel que $B.TAGS = B'.TAGS$
$\wedge (B.PRE \neq B'.PRE \vee B.POST \neq B'.POST)\}$

Considérons le comportement d'achat d'un ticket avec succès et son évolution, décrit sur la figure 7.3. L'évolution consiste à ajouter des types d'abonnements pour l'utilisateur. Cela implique de prendre en compte les différents tarifs des tickets en fonction de chaque abonnement. Afin de générer des tests pour valider les achats en fonction de l'utilisateur connecté, nous créons cinq comportements i.e. un pour chaque type d'abonnement, qui est la transformation du comportement d'achat d'un ticket par un utilisateur connecté, donné ci dessous :

$$(TAGS : BASKET\_MNGT/BUY\_TICKETS,\ Buy\_Success)$$

Pour la nouvelle version d'eCinema, nous considérons que le comportement *buy-success* est supprimé et les comportements pour tous les comportement de *buy-success-young* à *buy-success-onefree*, sont ajoutés.

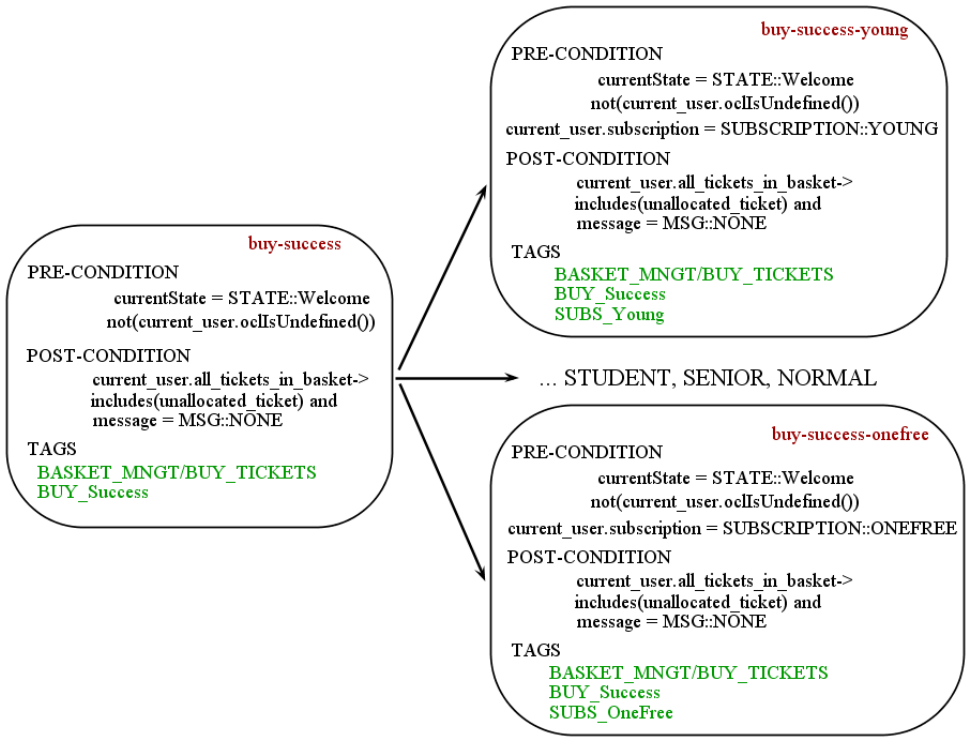

FIGURE 7.3 – Calcul de différences comportementales

Notre algorithme les considère comme nouveaux. Ceci est illustré dans la figure 7.3.

Il est important de noter que toute réécriture du code OCL, afin de mieux le présenter ou de l'améliorer, est considérée comme une modification. Il serait intéressant de pouvoir comparer les arbres syntaxiques du code OCL et de décider s'ils sont équivalents. Néanmoins, pour nos besoins et les cas d'études que nous avons menés, nous n'avons pas cherché à optimiser ce point.

7.3 Dépendances entre comportements

Nous définissons les dépendances entre comportements en tant que sous-ensemble des couples def/use. Dans le cas où un diagramme d'états/transitions n'est pas connu, nous ne considérons pas les dépendances de contrôle parce que chaque opération peut être appelée à n'importe quel moment. Le résultat dépend des conditions dans lesquelles l'appel est fait. Afin de calculer les dépendances de contrôle, il faut évaluer les conditions. Notre but est d'optimiser le temps de calcul. Pour cela, nous proposons de construire un graphe de dépendances de données basé sur les comportements issus du modèle, tel que nous l'avons détaillé dans la section 7.1.

En effet, le graphe basé sur la dépendance comportementale permet d'obtenir des in-

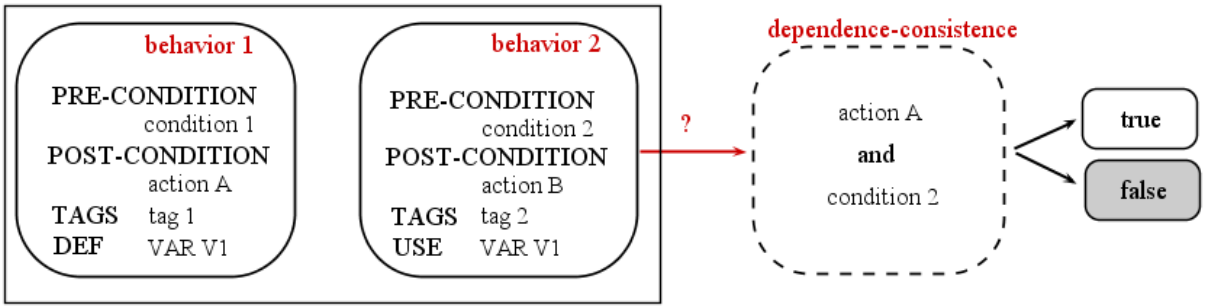

FIGURE 7.4 – Calcul de dépendances comportementales

formations plus pertinentes que les dépendances entre opérations. Plus exactement, nous proposons une technique qui réduit cet ensemble à des dépendances *positives*. Qu'est-ce qu'une dépendance *positive* ? Comme l'illustre la figure 7.4, le comportement *behavior* 1 pourrait dépendre d'un autre comportement *behavior* 2 par rapport à une variable $V1$ puisqu'elle est définie dans *behavior* 1 et utilisée dans *behavior* 2. De plus, si la condition composée de la post-condition *action A* et de la pré-condition *condition* 2 est consistante alors la condition de *dependance-consistance* est satisfaite. Nous aurons ainsi établi la dépendance entre ces deux comportements. Ci-dessous, nous donnons les définitions formelles de *dépendance-consistance*, définition 17 et *dépendance comportementale*, définition 18.

Définition 17 (Dépendance-consistance) *Il existe une dépendance-consistance entre un comportement B' et un autre comportement B si et seulement si (B'.PRE & B.POST) est consistant. Dans ce cas nous disons que B' est dépendant-consistant de B.*

À partir de cette définition nous pourrons définir la *dépendance-comportementale*.

Définition 18 (Dépendance comportementale) *Il existe une dépendance comportementale entre des comportements B' et B par rapport à une variable v si et seulement si v est définie dans B et utilisée dans B' et B' est dépendant-consistant de B.*

Remarque : Une variable v, dont la valeur est modifiée dans une instruction OCL (affectation, déclaration) dans un comportement B, est dite définie dans B. Si sa valeur est utilisée dans le code OCL dans un comportement B', nous disons qu'elle est utilisée dans B'. Lorsqu'une variable v est définie dans un comportement B et utilisée dans une comportement B', nous disons que ces deux comportements créent un couple def/use.

Nous illustrons l'approche sur deux extraits de l'exemple fil rouge. Nous prenons trois comportements : achat d'un ticket avec succès (opération *buyTicket*, comportement *buy-success*), connexion sur le site avec succès (opération *login*, comportement *login-success*) et de déconnexion (opération *logout*, comportement *logout-success*). Sur la figure 7.5, nous

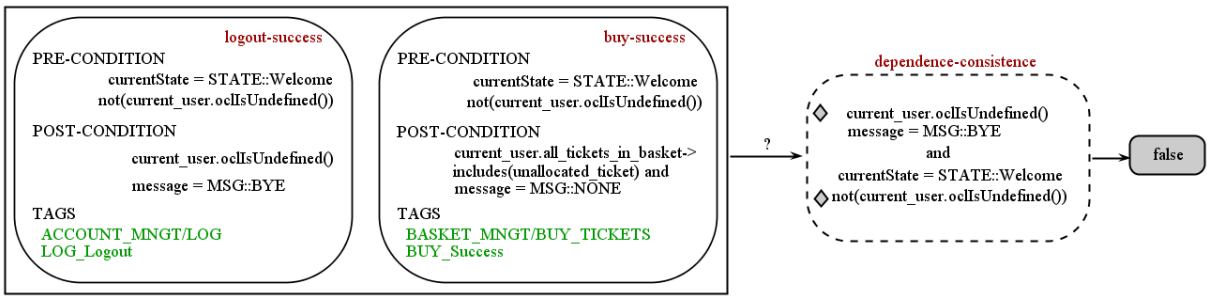

FIGURE 7.5 – Dépendance fausse-positive dans le cas d'eCinema

pouvons voir que les deux comportements *logout-success* et *buy-success* créent un couple *def/use* par rapport à la variable *current_user*. Cependant, il n'y a pas de dépendance entre ces deux comportements ; une fois le premier comportement activé, nous ne pouvons pas effectuer un achat avec succès. Avec notre approche, nous trouvons que l'expression composée de la post-condition de *logout-success* et de la pré-condition de *buy-success* est inconsistante. Ainsi, nous ne la considérons pas comme une dépendance comportementale.

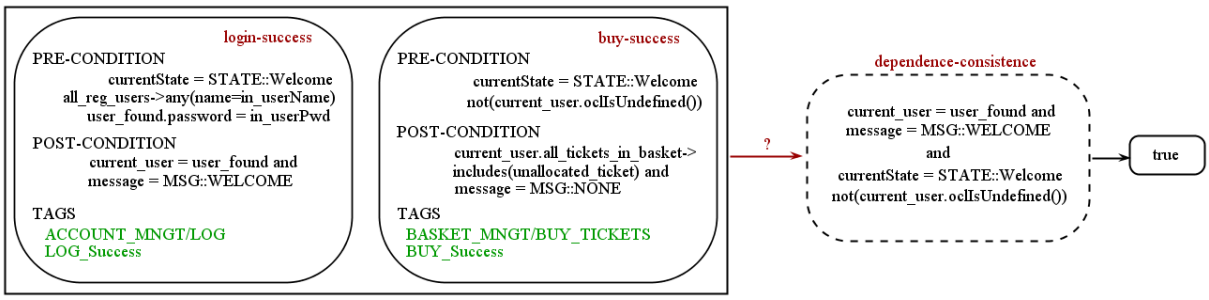

FIGURE 7.6 – Dépendance positive dans le cas d' eCinema

Cependant, dans le cas représenté sur la figure 7.6, un cas similaire de création d'un couple *def/use* pour *login-success* et *buy-success* par rapport à la variable *current_user*. L'expression composée est consistante et nous confirmons la dépendance entre ces deux comportements. En effet, afin d'effectuer un achat, l'utilisateur doit tout d'abord se connecter.

Afin de vérifier la consistance d'une condition OCL dans un modèle UML, nous utilisons un *Satisfying Modulo Theory* solveur sur des formules écrites en logique de premier ordre (FOL - *First Order Logical Formulas*). Il existe plusieurs outils pour résoudre des contraintes tels que CVC3 [DHBT07] et Z3 [DBr08]. Les auteurs dans [CAB12] proposent une approche outillée de transformation des contraintes issues des modèles UML/OCL en modèle SMT [13]. Nous avons décidé d'utiliser leur approche et le solveur Z3 pour notre évaluation. Z3 est intégré dans notre chaîne outillée décrite dans le chapitre 9.

13. Le modèle SMT contient des informations pour la ou les contraintes que l'on souhaite résoudre et il est écrit dans le langage, nommé SMT-lib et utilisé en entrée des solveurs, tel que CVC3 et Z3.

7.4 Analyse des impacts des comportements

Dans cette section, nous allons définir le calcul des comportements impactés et ensuite la classification des tests à partir des différences et des impacts des comportements. Nous allons voir que le calcul est très similaire à celui basé sur les diagrammes d'états/transitions.

7.4.1 Calcul des comportements impactés

Pour calculer les comportements impactés, nous nous basons sur l'algorithme 6 décrit en section 6.3.1. À la place des dépendances de données et de contrôle pour un diagramme d'états/transition, nous considérons des dépendances comportementales respectant la contrainte de dépendance-consistance par rapport à la définition et à l'utilisation d'une variable, tel que décrit précédemment.

```
Données : diff : ensemble de différences des comportements entre les modèles;
D, D' : dépendances comportementales pour les deux modèles M et M' respectivement;
Sorties : Impacte : ensemble de comportements impactés par le changement;
dependeceDiff = merge(D,D');
pour chaque Dépendance Q(c,c',v,m) ∈ dependenceDiff faire
    si c,c' ∉ diff & m ≠ none alors
        Impacte.add(c);
        Impacte.add(c');
    sinon
        si c ∈ diff & c' ∉ diff alors
            Impacte.add(c');
        fin
    fin
fin
```

Algorithme 7: Calcul des comportements impactés lors d'une évolution

Nous décrivons, dans l'algorithme 7, le calcul d'impacts de manière similaire à celui des diagrammes d'états/transitions. Il prend en entrée l'ensemble de différences des comportements *diff* et les diagrammes de dépendances comportementales pour les deux modèles respectivement D et D'. Nous identifions également les différences entre les diagrammes de dépendance en utilisant la fonction *merge*, telle que définie en section 6.3.1.

Nous considérons les ensembles : *Impacte* comme ensemble des comportements impactés. Tous les autres comportements (non-impactés) font partie de l'ensemble *Nonimpacte*.

Pour illustrer le calcul des impacts, prenons l'exemple de suppression et d'ajout de comportements pour l'opération *buyTickets*, telle que définie sur la figure 7.3.

Suite à cette évolution, les dépendances concernant ces comportements seront modifiées, ainsi les comportements dépendants seront considérés comme impactés. Afin de mieux illustrer le principe sur les comportements, nous allons étendre le fragment d'eCinema pour permettre à l'utilisateur de supprimer le ticket acheté.

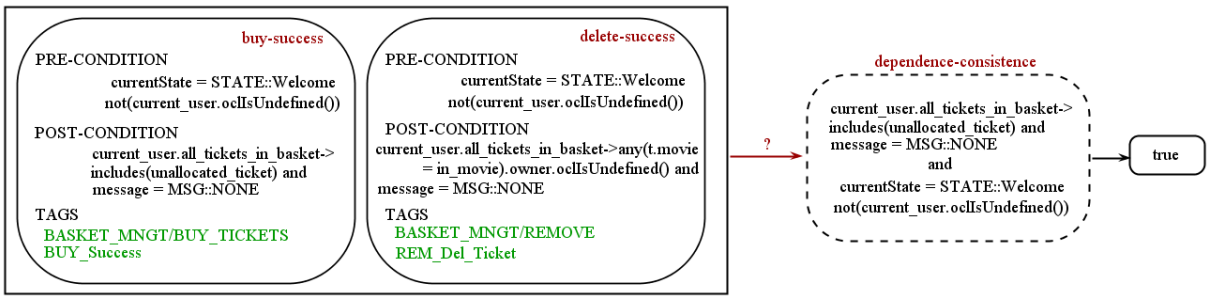

FIGURE 7.7 – Calcul d'un comportement impacté sur eCinema

Nous donnons une partie du code OCL simplifié pour une meilleure compréhension de l'illustration, sur la figure 7.7. Comme le décrit la figure, la variable permettant ce lien est *all_tickets_in_basket*. Elle est définie dans *buy-success* et utilisée dans *delete-success*. La condition de dépendance-consistance est satisfaite, ainsi nous considérons comme vrai la dépendance entre ces comportements. Suite au processus de calcul d'exigences impactées, le fait que le comportement d'achat d'un ticket avec succès ait été supprimé, déclenche une réaction en chaine i.e. les comportements de suppression d'un ticket qui sont dépendants d'une modification sont dit impactés.

Le tableau 7.4.1 résume le calcul d'impacts sur l'ensemble des comportements considérés du fragment d'eCinema.

Comportements		**Dépendance comportementale**			**Comportement impacté**
id	TAGS	login-success	logout-success	buy-success	
buy-logErr	BUY_TICKETS, Buy_LoginMandatory		x		
buy-success	BUY_TICKETS, Buy_Success	x			
login-success	LOG, LOG_Login				
logout-success	LOG, LOG_Logout	x			
delete-success	REMOVE, REM_Del_Ticket	x		x	x

TABLE 7.1 – Tableau résumant les impacts sur un fragment d'eCinema

Pour l'évolution dans *eCinema*, nous avons remarqué une diminution des comportements impactés à trois, au lieu de quatre comportements impactés obtenus en considérant seulement les couples *def*/*use* entre comportements. Ce nombre peut être très important et le fait d'utiliser la dépendance-consistance peut permettre un gain de temps conséquent dans l'analyse des données par les ingénieurs de validation des systèmes.

7.4.2 La catégorisation de tests à partir des comportements

Cette nouvelle extension de l'approche pour les comportements issus du code OCL des opérations permet de remplir la boîte d'analyse d'impact, étape ②, de la figure 5.1. Le but de l'approche SeTGaM est de déterminer le statut d'un test d'une version à

une autre. Elle repose sur l'utilisation de l'analyse des comportements modifiés, ajoutés, supprimés, impactés ou non. La plupart des techniques de non régression ne considèrent pas la génération de nouveaux tests. SeTGaM prend les nouveaux comportements comme des cibles de tests et le moteur de génération produit des tests pour couvrir ces cibles.

Dans cette section, nous donnons l'extension des règles pour l'attribution de statut aux tests pour les comportements issus des opérations.

Pour cela, nous utilisons la classification des comportements que nous venons de voir. Ce sont les cinq ensembles : *Ajoute, Supprime, Modifie, Impacte* et *Nonimpacte*. Dans les définitions qui suivent nous notons C un comportement et tc un test.

Règle 11 (RC-New) *Le test tc correspond à un nouveau comportement qui n'a pas été couvert par aucun test existant.*

$$\forall\, C \in Ajoute \rightsquigarrow\ status(tc^{n+1}) = new$$

Règle 12 (RC-Unimpacted) *Le test tc correspondant à un comportement non impactée et non modifié i.e non impactée est classifié en tant qu'unimpacted.*

$$\forall\, C \in Nonimpacte \wedge C.TAGS \in tags(tc^{n+1}) \rightsquigarrow\ status(tc^{n+1}) = unimpacted$$

Règle 13 (RC-Outdated) *Le test tc correspondant à un comportement supprimé est classifié en tant qu'oboslete.*

$$\forall\, C \in Supprime \wedge C.TAGS \in tags(tc^{n+1}) \rightsquigarrow\ status(tc^{n+1}) = outdated$$

Règle 14 (RC-Retestable) *Le test tc correspondant à un comportement modifié ou à un comportement impacté est classifié en tant que retestable.*

$$\forall\, C \in \{Modifie\ \cup Impacte\} \wedge C.TAGS \in tags(tc^{n+1})$$

$$\rightsquigarrow\ status(tc^{n+1}) = retestable$$

Remarque : Les tests *retestable* sont ensuite animés sur le modèle afin d'obtenir leur statut définitif : *reexecuted, updated, failed.*

Toujours pour maintenir la couverture des cibles, il faut prendre en compte les tests qui n'ont pas pu être animés sur le nouveau modèle ou les tests *outdated*. Pour cela, nous devons calculons les tests *adapted*. L'ensemble des comportements qui ne sont pas couverts par les tests s'appelle *Uncovered*.

Règle 15 (RC-Adapted) *Le test tc correspond à un comportement existant et non couvert par les tests actuels. Le comportement C devient une cible de tests pour laquelle le*

test adapted tc est créé.

$$\forall\, C \in \ \mathit{Uncovered} \wedge C \neq \mathit{Ajoute} \rightsquigarrow \ \mathit{status}(tc^{n+1}) = \mathit{adapted}$$

Pour illustrer cette partie, nous considérons les trois tests suivants :

```
(1) ECinema::sut.login(REGISTERED_USER, REGISTERED_PWD);
    ECinema::sut.buyTicket(TITLE1);
(2) ECinema::sut.login(REGISTERED_USER, REGISTERED_PWD);
    ECinema::sut.buyTicket(TITLE1);
    ECinema::sut.deleteTicket(TITLE1);
(3) ECinema::sut.buyTicket(TITLE1);
```

Nous les classifions en prenant comme évolution les types d'abonnements utilisateur lors de l'achat de tickets, figure 7.3. Nous tenons compte des résultats obtenus sur les dépendances comportementales et les comportements impactés donnés par le tableau 7.4.1. Les règles de classification donnent comme résultat pour les tests (1) et (2) le statut *outdated*. Cependant, le comportement de suppression d'un ticket n'est plus couvert. Pour cela, le moteur de génération de tests produit un autre test qui sera classifié en tant que *adapted*. Pour le dernier test (3), il couvre le comportement d'achat d'un ticket par un utilisateur non identifié. Ce comportement n'est pas modifié et n'est pas impacté. Ainsi ce test sera classifié en tant qu'*unimpacted*.

7.5 Synthèse

Dans ce chapitre, nous avons présenté notre travail sur les comportements extraits à partir des diagrammes UML/OCL. Nous avons défini une analyse de dépendances entre comportements et nous nous sommes concentrés sur la réduction de ces comportements en utilisation des solveurs SMT. Nous avons ainsi réalisé une extension de l'approche SeTGaM pour une classification des tests existants et une génération sélective pour préserver la couverture des exigences dans le but de valider le système évolutif. Nous avons illustré les bénéfices d'une telle méthode via l'exemple fil rouge.

Ainsi, la méthode SeTGaM permet de couvrir les comportements des modèles avec et sans diagrammes d'états/transitions pour tester l'évolution du système par rapport aux exigences de sécurité. Cette extension de nos travaux est présentée dans le chapitre suivant.

Chapitre 8

SeTGaM pour le test de sécurité

Sommaire

8.1 Comparaison des Spécifications des Cas de Tests 94
8.2 Statuts des tests et des suites de tests pour la prise en compte des TCS 96
8.2.1 Statuts des suites de tests de sécurité 96
8.2.2 Composition des suites de tests de sécurité 97
8.3 Génération sélective de tests à partir des TCS 98
8.3.1 Règles de catégorisation des tests 98
8.3.2 Extension du processus pour les TCS 100
8.4 Synthèse 102

Durant les études de cas que nous avons menées, nous avons pu constater qu'il est souvent nécessaire pour les systèmes critiques de compléter les tests fonctionnels générés à partir d'autres critères de couverture de modèle. En particulier, dans le cas où ces systèmes contiennent un aspect sécurité. Ainsi, le modèle utilisé avec l'approche MBT représente l'aspect de sécurité et la couverture du modèle ne permet pas toujours de valider l'ensemble des exigences de sécurité. Pour compléter les tests, il faut pouvoir utiliser un langage pour exprimer les besoins de tests associés.

Nous considérons l'approche de génération de tests à partir de schémas dédiée à la sécurité détaillée en section 2.3.3. Les schémas permettent d'exprimer des intentions de tests qui sont issus de l'expérience de l'utilisateur afin de garantir la sécurité de l'application. Leur but n'est pas de permettre l'expression de toutes les propriétés, pour rappel, classées généralement en six catégories :

– confidentialité : l'information est disponible seulement aux personnes autorisées.
– intégrité : l'information peut être modifiée seulement par les personnes autorisées.

- authentification : l'identité de la personne est déterminée avant l'attribution des droits sur l'application.
- autorisation : les personnes sont autorisées ou non à accéder à l'application ou à ses données.
- disponibilité : l'application et ses services doivent pouvoir répondre à l'utilisateur lors d'une requête.
- non-répudiation : une personne ne peut pas effectuer une action et ensuite la nier.

Comme défini dans la section 3.4, nous nous basons sur la génération de tests à partir des exigences de sécurité, exprimées par des schémas. Même si nous sommes conscients que son utilisation peut être plus générale, nous allons rester fidèles à son première utilisation dans le projet *SecureChange*.

Un schéma a pour but d'exprimer des états, des opérations ou un enchainement des deux pour permettre d'exprimer le besoin de test associé aux exigences de sécurité. Ce travail effectué grâce à l'expérience de l'ingénieur de sécurité permet d'exprimer, de manière informelle, ces exigences dans le cahier des charges par le biais d'un langage proche du langage naturel. Pour cela, nous ne prétendons pas englober toutes les propriétés de sécurité. Nous proposons de les traduire en besoins de test dédiés pour la validation des exigences de sécurité sur le système.

Ce chapitre va traiter la problématique d'étendre notre approche de génération sélective de tests pour prendre en compte ce langage pour les systèmes évolutifs critiques.

8.1 Comparaison des Spécifications des Cas de Tests

Nous considérons une comparaison des TCS issues du dépliage des schémas (voir section 2.3.3). Afin de prendre en compte l'évolution, il est important de savoir quelles parties des artefacts évoluent. Est-ce le modèle, représenté par les exigences fonctionnelles ou les exigences de sécurité, ou les deux ?

Comme discuté dans la section 1.3, nous proposons d'**identifier la ou les entité(s) qui évoluent**. Nous prenons en compte trois cas de changements qui peuvent se produire :

1. le schéma peut évoluer suite aux changements des objectifs de test de sécurité de l'ingénieur de sécurité ou de la spécification.
2. le schéma ne change pas, mais des modifications fonctionnelles sont apportés dans le modèle.
3. le schéma peut évoluer et le modèle également (l'évolution des exigences fonctionnelles a déjà été discuté dans les chapitres 6 et 7.).

Les changements des schémas peuvent se ramener à l'*ajout* ou à la *suppression de schémas*. La *modification d'un schéma* est considérée comme une suppression, suivi d'un ajout d'un schéma. Nous justifions ce choix par le fait que comparer à posteriori les TCS

de schémas différents n'a pas de sens. Dans le dernière cas, il faut prendre en compte l'*évolution fonctionnelle* de modèle et calculer l'impact sur les TCS et les tests associés. Pour conclure, dans les trois cas *le changement se propage au niveau de la production des TCS.*

Nous avons défini trois statuts pour les TCS, pour lesquelles nous utilisons la notation anglaise : **unchanged** - pour une TCS inchangée, **new** - pour une TCS nouvelle et **deleted** - pour une TCS supprimée. En fonction de l'évolution, un schéma peut être déplié d'une manière différente d'une version de modèle à une autre. Pour définir formellement des statuts des TCS, nous allons utiliser la fonction $tcs\_unfolding(M)$, qui renvoie l'ensemble des TCS dépliées à partir de tous les schémas d'un modèle M.

Définition 19 (TCS New) *Une TCS tcs' est ajoutée dans l'ensemble des TCS dépliées pour le modèle M' si et seulement si tcs' n'existe pas dans l'ensemble des TCS dépliées pour le modèle M :*

$tcs' = new$ *si et seulement si* $tcs' \in tcs\_unfolding(M') \wedge tcs' \notin tcs\_unfolding(M)$

Définition 20 (TCS Deleted) *Un TCS tcs est supprimeé de l'ensemble des TCS dépliées pour le modèle M' si et seulement si la TCS tcs n'existe pas dans l'ensemble des TCS dépliées pour le modèle M' mais existe dans l'ensemble des TCS dépliées pour le modèle M :*

$tcs = deleted$ *si et seulement si* $tcs \notin tcs\_unfolding(M') \wedge tcs \in tcs\_unfolding(M)$

Définition 21 (TCS Unchanged) *Un TCS tcs est dite inchangée si elle existe dans l'ensemble des TCS dépliées pour le modèle M' et pour le modèle M :*

$tcs = unchanged$ *si et seulement si* $tcs \in tcs\_unfolding(M') \wedge tcs \in tcs\_unfolding(M)$

Pour mieux comprendre, nous illustrons la comparaison des TCS sur un fragment d'eCinema. Nous prenons les exigences de sécurité liées à l'*enregistrement* et l'*authentification* présentées dans la section 4.1.2. Dans le tableau 8.1, nous présentons la classification des TCS pour le modèle d'*eCinema* évolué, pour les deux types de modèles (avec et sans diagramme d'états/transitions). Les deux types sont marqués *Modèle 1* et *Modèle 2*.

D'une part, nous constatons que le dépliage pour l'exigence de sécurité *enregistrement* produit cinq nouveaux TCS, un pour chaque nouvelle opération : `addUnits`, `retrieveUnits`, `setSubscription`, `goToBlanceMngt` et `goToSubscription`. Les anciens TCS pour cette exigence restent inchangés.

D'autre part, le dépliage de l'exigence de sécurité *authentification* pour les deux modèles ne produit aucun changement dans les TCS.

Dans la section suivante, nous détaillons la classification des tests pour la validation des exigences de sécurité pour l'évolution du système.

Exigence de sécurité	**TCS**	**Classification**	
		Modèle 1	Modèle 2
enregistrement	TCS 1.1 - 1.7	unchanged	unchanged
	TCS 1.8 $_{\$X=addUnits}$	new	new
	TCS 1.9 $_{\$X=retrieveUnits}$	new	new
	TCS 1.10 $_{\$X=setSubscription}$	new	new
	TCS 1.11 $_{\$X=goToBalanceMngt}$	new	new
	TCS 1.12 $_{\$X=goToSubscription}$	new	new
authentification	TCS 2.1 - 2.4	unchanged	unchanged

TABLE 8.1 – TCS classification pour eCinema (Modèle 1 - avec et Modèle 2 - sans diagramme d'états/transitions)

8.2 Statuts des tests et des suites de tests pour la prise en compte des TCS

Comme pour la gestion de l'évolution, nous considérons aussi les quatre suites de tests : Evolution, Regression, Stagnation et Deletion et les statuts qui les composent (cf. figure 5.3).

Nous donnons leur extension pour les TCS dans les deux sections suivantes, respectivement.

8.2.1 Statuts des suites de tests de sécurité

Les suite de tests sont notées Γ_X, où Γ est la notation pour la suite de tests et X est son type. Nous donnons ici les descriptions informelles de leur but dans le domaine de test à partir et pour les exigences de sécurité. Nous présentons cette extension des règles, en section 5.2.2, ci-dessous.

Suite de tests *Evolution* Γ_E contient des tests produits à partir des **New** TCS, qui représentent les nouveautés dans le système, concernant une exigence de sécurité pour laquelle nous générons des tests, comme par exemple de nouvelles exigences fonctionnelles, opérations, comportements etc.

Suite de tests *Regression* Γ_R contient des tests produits à partir des TCS **inchangés** qui exercent les parties non-modifiées du système. Ces tests ont pour but de s'assurer que l'évolution n'a pas impacté des éléments du SUT, qui ne sont pas censés être modifiés et que ses éléments préservent les exigences de sécurité.

Suite de tests *Stagnation* Γ_S contient des tests invalides produits à partir des TCS inchangés par rapport à la version courante du système. Un échec lors de leur exécution

sur le SUT est le comportement attendu (soit parce qu'ils ne peuvent pas être exécutés, soit parce qu'ils détectent une non-conformité du SUT par rapport aux valeurs attendues).

Suite de tests *Deletion* Γ_D contient des tests produits à partir des TCS **supprimés**, issus des schémas supprimés par rapport à la version courante.

8.2.2 Composition des suites de tests de sécurité

Nous donnons ci-dessous les règles utilisées pour distribuer les tests de la suite de tests, dédiée aux exigences de sécurité. Ces règles sont l'adaptation des règles correspondantes définies pour les exigences fonctionnelles. Nous donnons formellement la définition d'un test de sécurité issus d'un TCS suite à une évolution du système.

Définition 22 (Cas de test de sécurité évolutif) *Un* test évolutif pour la sécurité tc^n *est caractérisé par le triplet* $\langle tc, \{tcs\}, statut\rangle$ *où* n *est la version du modèle pour lequel le cas de test tc est appliqué, tcs est l'ensemble de spécifications de cas de tests à la quelle il est associé et statut est son statut associé :*

$statut \in \{new, adapted, updated, unimpacted, reexecuted, failed, outdated, removed\}$

Nous rappelons la notation, tc^n est un test de sécurité valide pour la version initiale du modèle et tc^{n+1} est le test pour la version évoluée. $statut(tc^n)$ représente le statut associé à tc^n.

Règle 16 (New tests) *Un test nouveau existe seulement à la version* tc^{n+1}*, produit à partir d'un nouvel TCS. Tous les nouveaux tests sont ajoutés à la suite de tests Evolution :*

$$status(tc^{n+1}) = new \quad \rightsquigarrow \quad tc^{n+1} \in \Gamma_E^{n+1}$$

Règle 17 (Reusable tests) *Un test réutilisable est issu d'un TCS inchangé et d'une suite de tests existante* $tc^n \in \Gamma_E^n \cup \Gamma_R^n$*. Il reste inchangé à la version* $tc^{n+1} = tc^n$*. Les tests réutilisables sont ajoutés à la suite de tests Regression :*

$$status(tc^{n+1}) \in \{unimpacted, reexecuted\} \rightsquigarrow tc^{n+1} \in \Gamma_R^{n+1}$$

Un test *réutilisable* est soit *unimpacted*, soit *reexecuted*. Un test *unimpacted* couvre des exigences qui n'ont pas été impactées par l'évolution. Un test *reexecuted* couvre des exigences qui ont été impactées par l'évolution.

Règle 18 (Obsolete tests) *Un test obsolète est issu d'un TCS inchangé et d'une suite de tests existante (qui peut être obsolète)* $tc^n \in \Gamma_E^n \cup \Gamma_R^n \cup \Gamma_S^n$*. Tous ces tests qui sont déclarés obsolètes sont ajoutés à la suite de tests Stagnation :*

$status(tc^{n+1}) = \{outdated, failed\} \rightsquigarrow tc^{n+1} \in \Gamma_S^{n+1}$

Il est à noter qu'un test *failed* couvre des exigences de sécurité existantes et un test *adapted* est créé pour la nouvelle version afin de garantir une couverture maximale.

Règle 19 (Involved tests) *Un test involved est issu d'un TCS inchangé et d'une suite de tests déjà existante et il a été impacté par l'évolution* $tc^n \in \Gamma_E^n \cup \Gamma_R^n$. *Tous les tests de ce type sont ajoutés à la suite de tests Evolution.*

$status(tc^{n+1}) = \{updated, adapted\} \rightsquigarrow tc^{n+1} \in \Gamma_E^{n+1}$

Le test *updated* après animation sur le nouveau modèle a subi un changement de l'oracle par rapport à sa version précédente n. Cependant, contrairement à celui-ci, le test *adapted* est créé afin de maintenir la couverture des TCS. Dans sa version précédente n il est *obsolète* pour la nouvelle version du modèle.

Règle 20 (Removed tests) *Un test removed est issu d'un TCS supprimé et d'une suite de tests existante. Ils sont attribués à la suite de tests Deletion.*

$status(tc^n) = \{removed\} \rightsquigarrow tc^n \in \Gamma_D^n$

Le test dont le statut est *removed*, dans le processus de gestion de l'évolution pour les exigences de sécurité, est gardé afin de garantir la traçabilité des tests entre les différentes versions.

8.3 Génération sélective de tests à partir des TCS

Dans cette section nous allons donner l'extension des règles qui catégorisent le statut du test (cf. section 8.3.1). Ensuite, nous allons présenter l'effet de l'évolution de ses règles sur le processus de la méthode SeTGaM lors d'évolution du système.

8.3.1 Règles de catégorisation des tests

Nous proposons une extension de l'approche SeTGaM en se basant sur la comparaison des TCS produites pour les deux modèles.

Comme cela est décrit sur la figure 8.1, lors de l'étape ①, les TCS sont comparés et classés en : unchanged, deleted et new. À l'étape ② tous les tests inchangés sont envoyés à SeTGaM, pour une classification en fonction des évolutions dans le modèle. Pour un schéma supprimé les TCS et leurs tests associés sont inutiles pour la nouvelle version. Cependant, pour assurer la traçabilité des tests d'une version à une autre, à l'étape ③

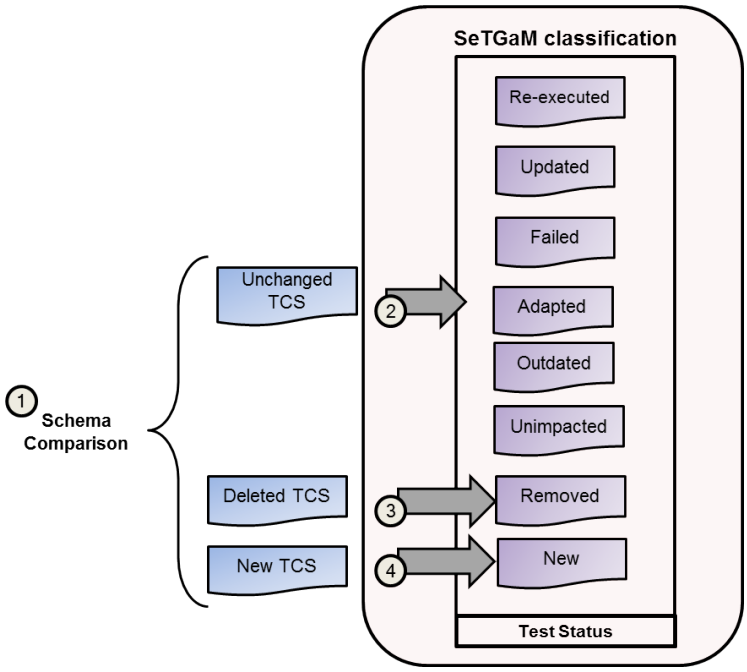

FIGURE 8.1 – Définition du statut des tests par rapport à la comparaison des TCSs

nous leur attribuons le statut *removed* et nous les classons ensuite dans la suite de tests correspondante. Enfin, un même schéma pour une version différente peut produire de nouveaux TCS. Pour chaque nouveau TCS, à l'étape ④, le générateur produit 0 ou 1 test. Nous présentons ci dessous, l'extension des règles d'attribution de statut aux tests pour le test de sécurité. L'approche peut s'appliquer sur les deux types de formalisation en UML que nous sommes en mesure de prendre en compte. Ainsi, le comportement que nous considérons ici désigne les deux façon de formalisation, par biais de la garde/action une transition ou pre/post condition d'une opération. Nous utilisons les définitions de comportement ajouté, supprimé, modifié, impacté et non-impacté, selon les définitions données dans les deux chapitres précédents.

Règle 21 (RS-New) *Le test correspond à une nouvelle TCS qui n'a pas été couverte par aucun test existant.*

$\forall\ TCS \in New \rightsquigarrow\ \ status(tc^{n+1}) = new$

Règle 22 (RS-Unimpacted) *Le test correspondant à une TCS inchangée et un comportement non impacté et non modifié i.e non impactée est classifié en tant qu'unimpacted.*

$\forall\ TCS \in Unchanged \wedge C \in Nonimpacte \wedge C.TAGS \in tags(tc^{n+1}) \rightsquigarrow\ \ status(tc^{n+1}) = unimpacted$

Règle 23 (RS-Outdated) *Le test correspondant à une TCS inchangée et à un comportement supprimé est classifié en tant qu'oboslete.*

$\forall\ TCS \in Unchanged \wedge C \in Supprime \wedge C.TAGS \in tags(tc^{n+1}) \rightsquigarrow\ status(tc^{n+1}) = outdated$

Règle 24 (RS-Removed) *Le test correspondant à une TCS supprimée est classifié en tant que removed.*

$\forall\ TCS \in Deleted\ \rightsquigarrow\ status(tc^{n+1}) = removed$

Règle 25 (RS-Retestable) *Le test correspondant à une TCS inchangée et soit à un comportement modifié soit à un comportement impacté, est classifié en tant que retestable.*

$\forall\ TCS \in Unchanged \wedge C \in \{Modifie\ \cup Impacte\} \wedge E.TAGS \in tags(tc^{n+1})$
$\rightsquigarrow\ status(tc^{n+1}) = retestable$

Remarque : Les tests sont ensuite animés sur le modèle afin d'obtenir leur statut définitif : *reexecuted, updated, failed.*

Toujours pour maintenir la couverture des cibles et de même que pour l'approches basées sur comportements issus des diagramme d'états/transitions ou des opérations, il faut prendre en compte les tests qui n'ont pas pu être animés sur le nouveau modèle. Pour cela, nous devons générer des tests *adapted* pour couvrir la TCS existante.

Règle 26 (RS-Adapted) *Le test correspond à une TCS inchangée mais non couverte par les tests actuels. Elle devienne une cible de tests pour laquelle un test adapted est créé.*

$\forall\ TCS \in Unchanged \wedge TCS \in\ uncovered \rightsquigarrow\ status(tc^{n+1}) = adapted$

Nous verrons dans la section suivante comment ces règles sont prise en compte pour l'extensionde la catégorisation des tests.

8.3.2 Extension du processus pour les TCS

À partir des règles précédemment définies, nous illustrons sur la figure 8.2 plus en détails et sur l'aspect comment le changement fonctionnel implique le changement des tests issus des TCS *Unchanged*. À l'étape ① les tests sont classifiés en quatre statuts :*outdated, unimpacted, removed (pour les TCS supprimés)* et le statut intermédiaire *retestable*. Chaque test issu d'un comportement supprimé est classifié comme *outdated*, s'il couvre un comportement non-impacté il est classifié comme *unimpacted* ou s'il couvre un comportement impacté ou modifié il est classifié comme *retestable*.

Ces derniers sont animés sur le modèle évolué M', à l'étape ②. Quand l'animation produit un oracle identique à la version précédente du test, le statut *reexecuted* lui est attribué. Sinon, le statut *updated* lui est donné. S'il est impossible d'animer un pas du test, alors son statut est *failed*. Si ces derniers tests couvrent des exigences (fonctionnelles)

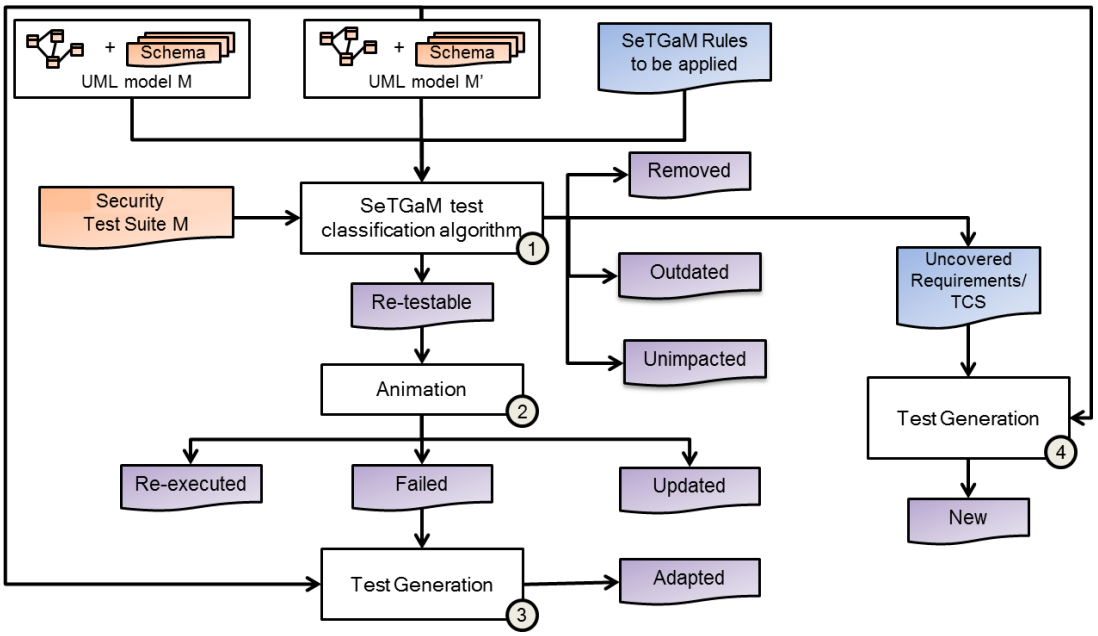

FIGURE 8.2 – Extension de SeTGaM pour les TCS

ou des TCS existants et non-couverts par la nouvelle suite de tests, pour préserver la couverture des cibles le générateur crée des tests, étape ③. Ceux-ci sont appelés *adapted*. Pour toutes les nouvelles TCS, à l'étape ④ le moteur de génération produit de nouveaux tests, nommés *new*.

Pour illustrer l'approche nous prenons les deux exigences de sécurité, définies dans le chapitre 4. Nous prenons en compte l'évolution complète du système d'*eCinema*, ses dépendances d'exigences fonctionnelles sont détaillés dans les tableaux B.2 et B.5, de l'annexe B. Dans le tableau 8.2, nous donnons les résultats de classification des tests existants après l'évolution du système, représentés par les deux modèles avec et sans diagramme d'états/transitions, notés respectivement *Modèle 1* et *Modèle 2*.

Modèle 1		**Modèle 2**	
Test	Statut	Test	Statut
requirement11(bb-25-c2)	outdated	requirement11(bb-7c-93)	outdated
requirement11(bb-d3-b1)	outdated	requirement12(bb-fd-f9)	failed
requirement15(bb-91-ae)	failed	requirement15(bb-8f-fe)	failed
requirement28(bb-0c-3d)	outdated	requirement28(bb-f1-e3)	outdated
requirement210(bb-ca-6c)	outdated	requirement210(bb-72-5c)	outdated
requirement29(bb-c0-2c)	outdated	requirement29(bb-5c-3f)	outdated
requirement211(bb-24-b0)	outdated	requirement211(bb-be-43)	outdated

TABLE 8.2 – Classification de tests pour les exigences de sécurité pour eCinema

Pour le modèle avec diagramme d'états/transitions, colonne *Modèle 1* sur le tableau 8.2, six tests ont le statut *outdated*, puisqu'ils couvrent l'exigence *buy-success* qui a été supprimée. Et un test a le statut *failed*, parce qu'il couvre l'exigence impactée *login-success* et l'exigence modifiée *register-success*. Afin de garantir la couverture des TCS existants

sept tests *adapted* sont générés. Pour couvrir les nouveaux TCS, quatre nouveaux tests sont générés, dont le statut est *new*.

Une légère différence apparait lors de la catégorisation des tests existants pour le deuxième modèle, colonne *Modèle 2*, suite à l'absence d'un diagramme d'états/transitions (discuté dans le chapitre 4). Ainsi, cinq tests sont classifiés *outdated* et deux sont *failed*. De même, afin de garantir la couverture des TCS existants et nouveaux, respectivement sept tests *adapted* et cinq tests *new* sont générés.

8.4 Synthèse

Dans ce chapitre, nous avons présenté le dernier cas pris en compte par l'approche SeTGaM. Nous avons comparé les dépliages des schémas afin de prendre en compte leur évolution due au changement des objectifs de test de sécurité, d'une part ou due au changement des fonctions dans la spécification du système.

Le processus de *SeTGaM*, tel que nous l'avons décrit, prend en entrée deux modèles (celui de l'origine M et son évolution M'). Ces modèles contiennent l'information sur les exigences fonctionnelles, dans les diagrammes d'états/transitions ou dans les opérations d'un diagramme de classes et l'aspect sécurité avec les schémas.

Nous avons proposé une approche complète nommée *SeTGaM* pour le test des systèmes évolutifs critiques. Elle répond aux problématiques :

- d'**identification des différences**,
- de **sélection de tests**,
- de gestion de leurs **cycle de vie** au travers de huit statuts (new, unimpacted, reexecuted, updated, adapted, failed, outdated et removed),
- de **priorité d'exécution** sur le système en fonction de la suite de tests dans laquelle ils sont classés (Evolution, Regression, Stagnation et Deletion),
- de **couverture des nouvelles exigences**,
- de centralisation des tests dans un **référentiel de tests** pour une gestion de l'historique.

L'ensemble de ces contributions ont mené à la création d'un **prototype**, qui a été utilisé sur un cas industriel et dans le projet européen *SecureChange*. Ceci nous a permis de donner une **évaluation détaillée**, basée sur les critères d'évaluation définis par Rothermel et Harrold.

Troisième partie

Réalisations et expérimentations

Chapitre 9

Le prototype

Sommaire

9.1 Composants du prototype 105
9.2 Générateur de tests pour les systèmes évolutifs 107
9.3 Publication des tests dans le référentiel de tests 111
9.4 Création du plug-in pour IBM Rational Software Architect 112
9.5 Synthèse 114

Dans ce chapitre, nous décrivons le prototype EvoTest. Ce prototype intègre l'ensemble des développements effectués dont SeTGaM. Nous présentons chacun de ses composants et leur intégration en tant que plugin de l'environnement IBM Rational Software Architect.

9.1 Composants du prototype

Le prototype développé, SeTGaM compris, s'intègre dans le processus général décrit dans la figure 9.1. Cette figure est séparée en deux grandes phases décrivant l'application de l'approche sur différentes versions d'un système. La première phase (Version 1) représente le processus général du MBT. Un ingénieur de test crée un modèle pour le système à tester à partir des exigences décrites dans la spécification informelle. Ce modèle prend en compte la vue statique et dynamique du système à l'aide de la notation UML/OCL. Il est utilisé pour la génération de tests avec la technologie de Smartesting CertifyIt. Un test est considéré comme une séquence d'actions associée à un résultat à chaque pas du test. Le verdict de test est créé à partir du code OCL issu de chaque pas de test.

Pour compléter ce premier ensemble de tests et prendre en compte le test de sécurité par exemple, il est possible d'ajouter un deuxième artefact de modélisation avec les sché-

mas de tests. Ce dernier, en coordination avec le modèle UML/OCL, est pris en entrée du module *SBTG (Schema Based Test Generator)* pour produire des tests dédiés aux schémas, soit indépendamment des éléments de couverture du modèle. Un référentiel de tests est utilisé pour stocker les tests et les résultats d'exécution sur le SUT. L'approche MBT que nous considérons est basée sur l'animation du modèle [14].

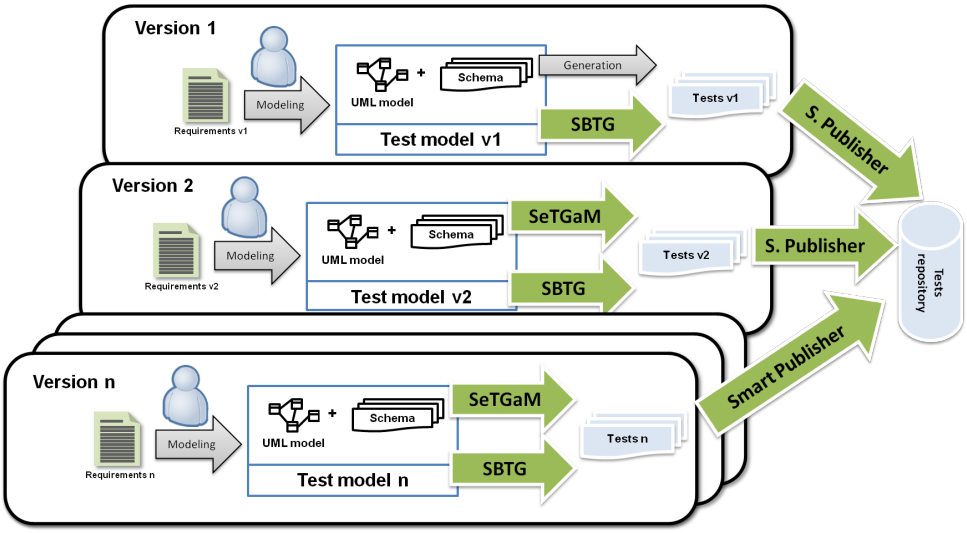

FIGURE 9.1 – Processus de génération et de gestion des tests

La phase deux (Version 2 à n, figure 9.1) décrit la suite du processus pour deux évolutions du système. Dans ce cas, l'ingénieur de test met à jour le modèle de test à partir des évolutions issues du cahier des charges. De nouveaux tests sont produits par SeTGaM et le module SBTG, pour les exigences de sécurité. Après cette étape, le composant - *Smart Publisher* est utilisé pour maintenir la cohérence au niveau de la gestion des tests. Pour cela, le statut des tests est mis à jour dans le référentiel de tests et sa version précédente (statut, exécution précédente et valeurs...) est sauvegardée dans l'historique. De cette façon, le lien est assuré entre les différentes versions de tests.

La figure 9.2 présente le diagramme d'activité associé au processus géré par le prototype (pour une version courante et son évolution). Les deux modèles d'entrée sont notés respectivement *in_model_n* et *in_model_n+1*. Le processus ainsi architecturé est créé en tant qu'un *Plugin Eclipse* dans *IBM Rational Software Architect* et comprend trois composants :

- génération de tests (pour des exigences fonctionnelles et de sécurité, cf. section 2.3),
- génération sélective de tests - SeTGaM,
- gestion du référentiel de tests - Smart Publisher.

Dans les sections suivantes, nous allons voir dans le détail les deux derniers composants.

14. Une animation du test consiste à exécuter le test sur le modèle, si possible et mettre à jour l'oracle, si besoin.

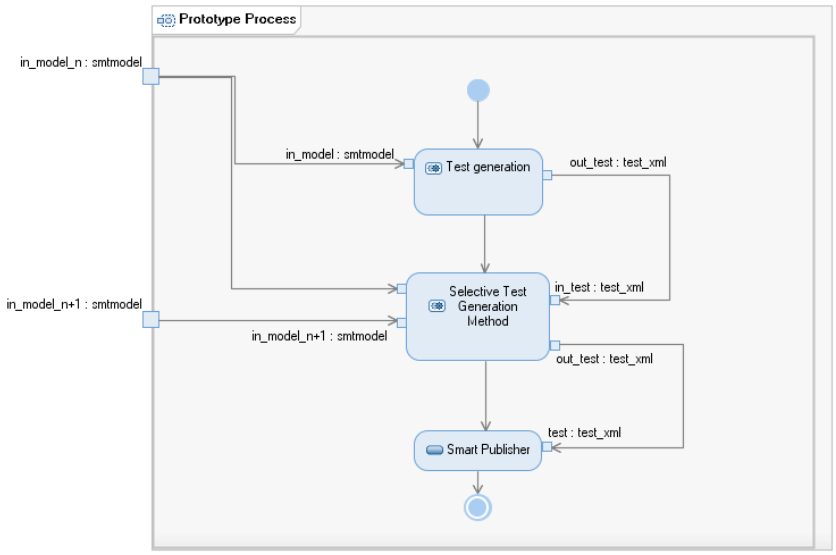

FIGURE 9.2 – Diagramme d'activité du processus associé au prototype

9.2 Générateur de tests pour les systèmes évolutifs

Dans cette section, nous introduisons une vue conceptuelle du processus de SeTGaM et de ses composants internes. Comme l'illustre la figure 9.3, ils sont séparés en analyseur d'impacts (*Impact analyser*) et module de classification de tests (*Test classification*).

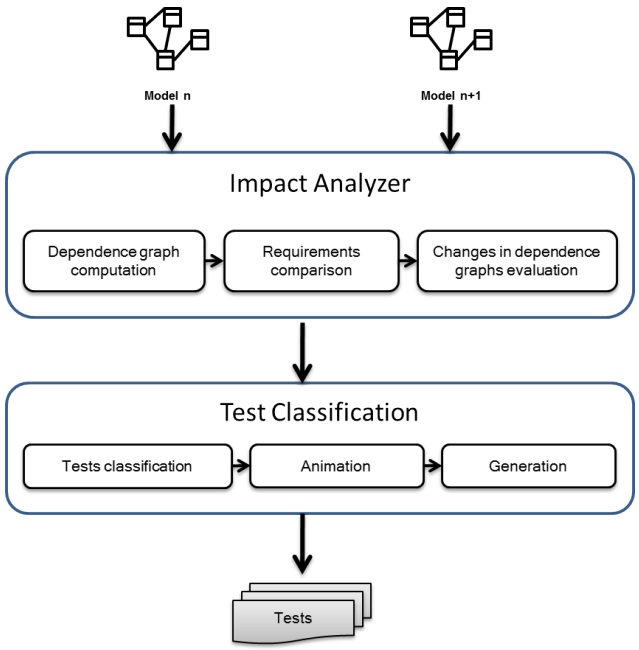

FIGURE 9.3 – Composants de SeTGaM

L'analyseur d'impacts prend en entrée deux versions de modèles UML/OCL afin de :

- calculer les graphes de dépendances,
- comparer les modèles,
- évaluer les changements entre les deux versions i.e. produire des règles de classification.

Cet analyseur produit un objet interne à l'application qui sera utilisé par le module de classification de tests. Ce processus est décrit en détails dans la Partie II Contribution de ce mémoire.

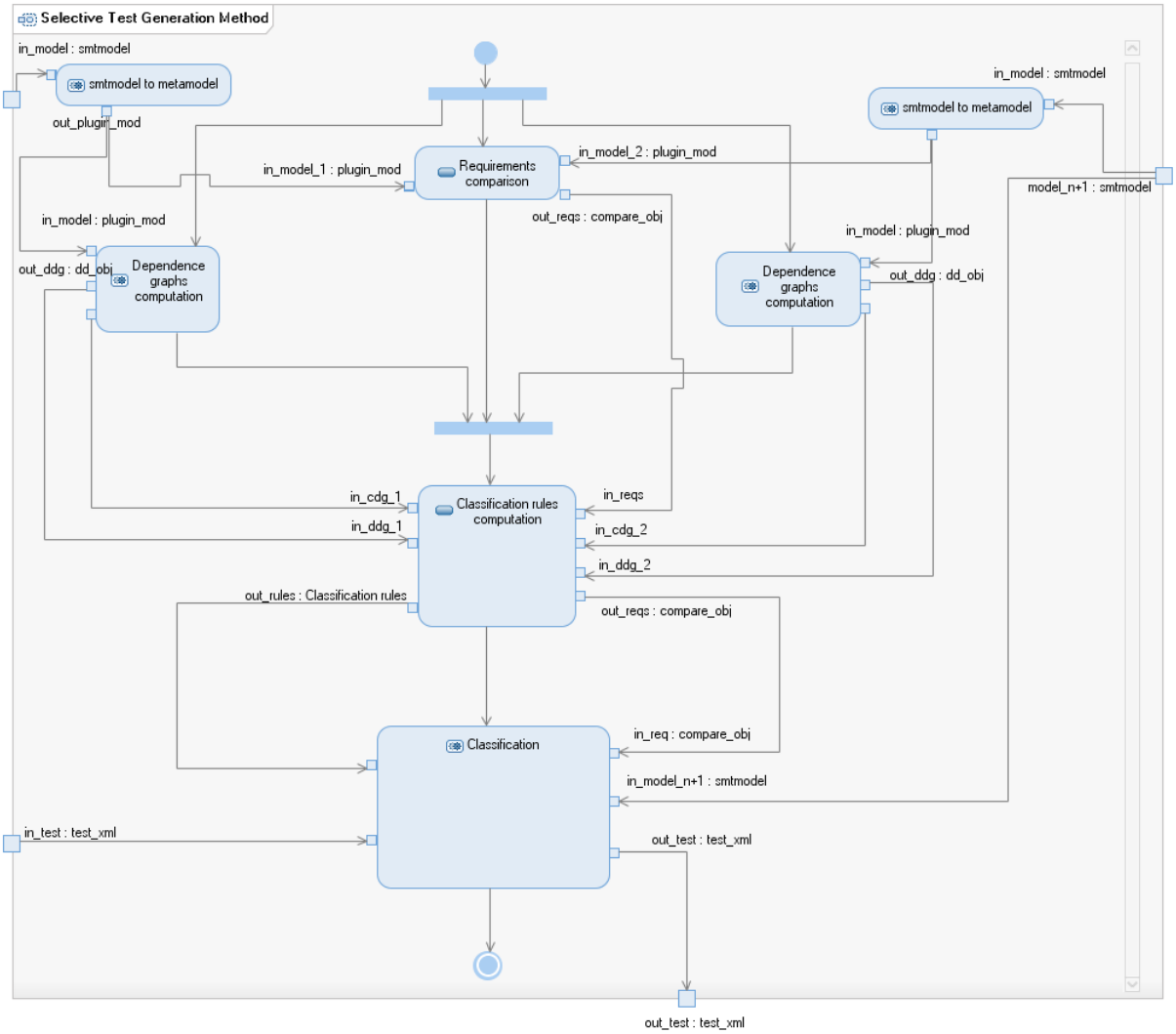

FIGURE 9.4 – Diagramme d'activité pour SeTGaM

Le processus du prototype de SeTGaM est présenté sur la figure 9.4. La première étape est la transformation du modèle en fichier de format spécifique pour le traitement interne. La transformation permet au prototype d'être indépendant du format utilisé dans le modeleur. Ensuite, ces fichiers sont utilisés pour détecter les évolutions entre modèles

(avec ou sans diagramme d'états/transition). Cette étape produit un objet de l'application qui contient tous les changements détectés.

Parallèlement, les dépendances pour chaque version sont calculées comme indiqué sur la figure 9.5. Le moteur détecte la (non)présence d'un diagramme d'états/transitions (élément sur la figure *statechart*) et applique l'algorithme de dépendances soit pour un diagramme d'états/transitions soit entre comportements. Les résultats sont stockés dans un objet interne à l'application.

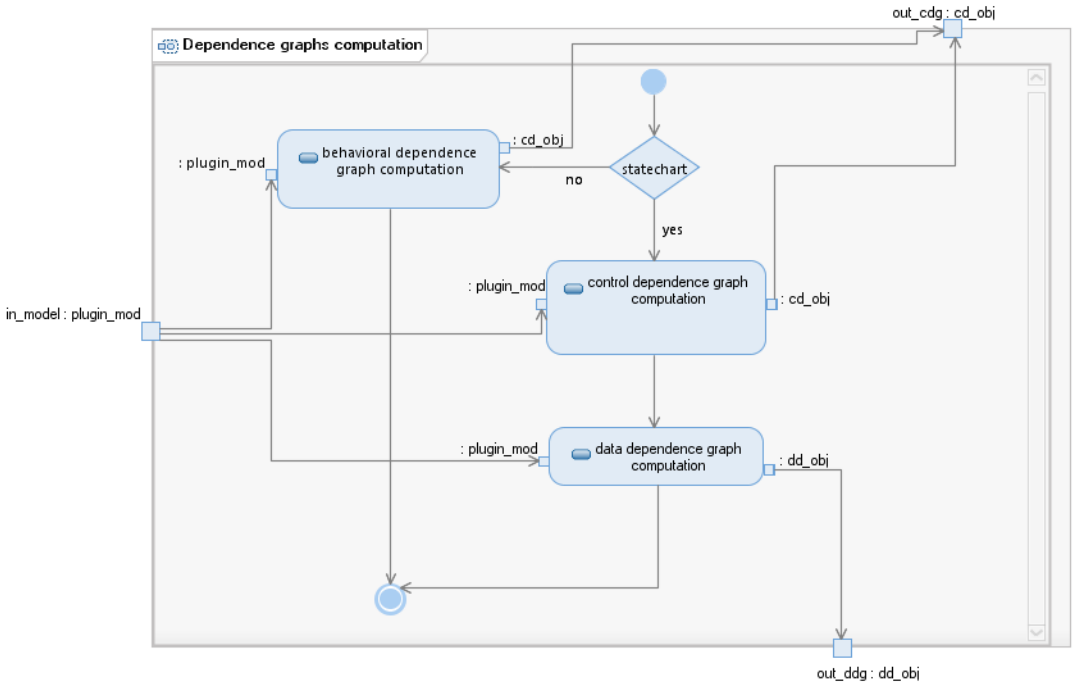

FIGURE 9.5 – Diagramme d'activité du calcul de dépendances

Ainsi, le module de classification de tests va prendre en entrée les structures produites lors de :

- la comparaison des exigences,
- le calcul de dépendances pour le $model\_n$,
- le calcul de dépendances pour le $model\_n + 1$.

La figure 9.6 présente l'activité de la classification. Lors de cette étape, un statut est attribué à chaque test.

Nous identifions quatre processus principaux pour la classification des tests :

- calcul des tests re-testable,
- calcul des tests outdated,
- calcul des tests unimpacted,
- calcul des tests new.

Pour chacun de ces processus, la nouvelle version du modèle et les tests générés par la version précédente sont pris en compte.

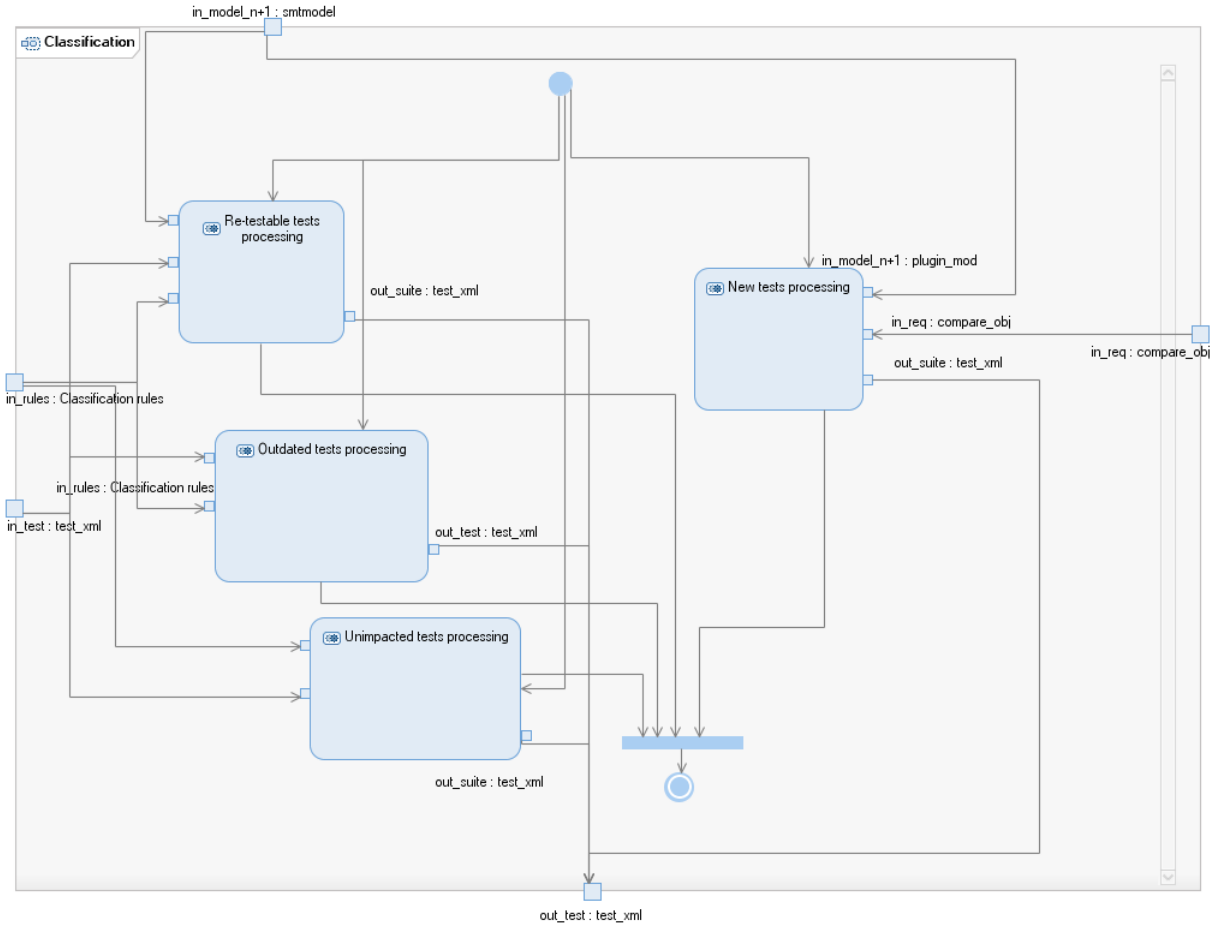

FIGURE 9.6 – Diagramme d'activité de classification de tests

L'activité la plus complexe est le calcul des tests dont le statut est *re-testable*. Nous la décrivons dans la figure 9.7. Le composant d'animation développé par Smartesting est utilisé pour définir leur statut définitif. Si besoin et afin de compléter la couverture des cibles de tests existants, le moteur de génération est utilisé pour créer des tests dont le statut est *adapted*.

Enfin, les nouvelles cibles de tests sont utilisées pour générer de nouveaux tests dont le statut est *new*. Le résultat est écrit dans un nouveau fichier XML, utilisé par le module *Smart Publisher*, afin de mettre à jour le référentiel de tests avec les nouveaux statuts. Ainsi, celui-ci est prêt pour une utilisation avec SeTGaM lors d'une prochaine évolution du système.

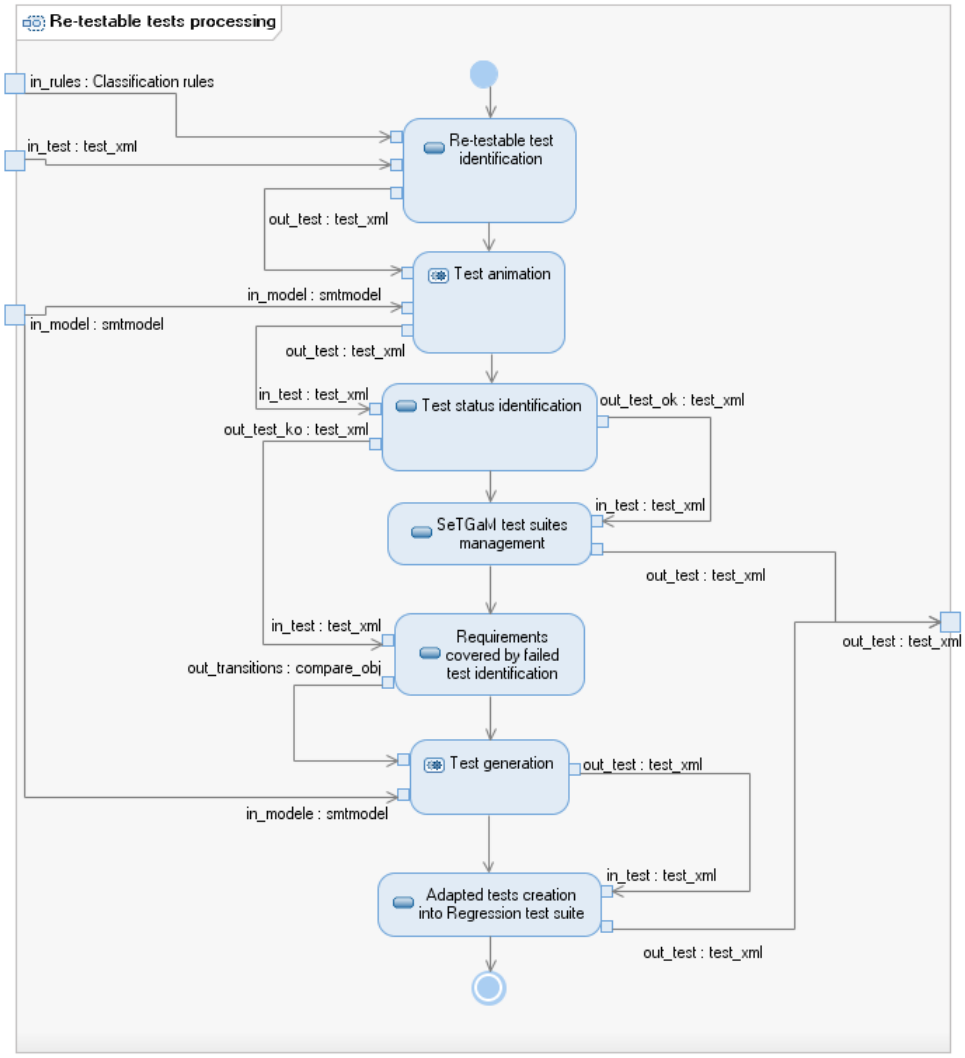

FIGURE 9.7 – Calcul des tests re-testable

9.3 Publication des tests dans le référentiel de tests

Le composant *Smart Publisher* est très important dans la gestion des tests pour les systèmes évolutifs. Il permet de tracer l'exécution des tests sur l'implémentation et d'enregistrer les résultats de l'exécution des tests sur le SUT. Un ingénieur de test a la possibilité de créer un lien via un *bug tracker* (gestionnaire d'anomalies) dans le but de transmettre une information à l'équipe des développeurs. Dans le projet SecureChange, nous avons décidé d'utiliser un référentiel de tests Open Source nommé *TestLink*. *TestLink* est un outil sous licence GPL qui permet la gestion des tests en les organisant dans des plans de tests. Après l'exécution des tests sur l'implémentation, les résultats (succès ou échec) sont sauvegardés et publiés dans des rapports d'exécution. Ils sont ensuite utilisés pour vérifier la couverture des exigences. *TestLink* est un outil indépendant avec une interface en PHP et utilise une base de données en MySQL, laquelle peut être liée à des outils permettant

de tracer les erreurs, tel que Bugzilla.

La première publication dans *TestLink* crée un nouveau projet dans le référentiel de tests. Dans le dossier racine, deux sous-dossiers peuvent être distingués : un pour les tests fonctionnels et un pour les tests dédiés aux exigences de sécurité. En se basant sur la technique SeTGaM proposée, le *Smart Publisher* crée des dossiers dans chacun, pour les quatre suites de tests : evolution, regression, stagnation et deletion (voir figure 9.8).

Durant la vie du système, les ingénieurs peuvent produire plusieurs versions de modèles de tests représentant les évolutions. Ainsi, une nouvelle publication peut être faite pour chaque version. Lors de la publication, si le test a subi un changement de statut, l'ancienne version est désactivée dans le référentiel de tests. Son ancien statut et ses résultats d'exécution sont sauvegardés dans la base de données. De cette manière, l'ingénieur de test peut garder une trace des versions précédentes des tests correspondants à la livraison du système.

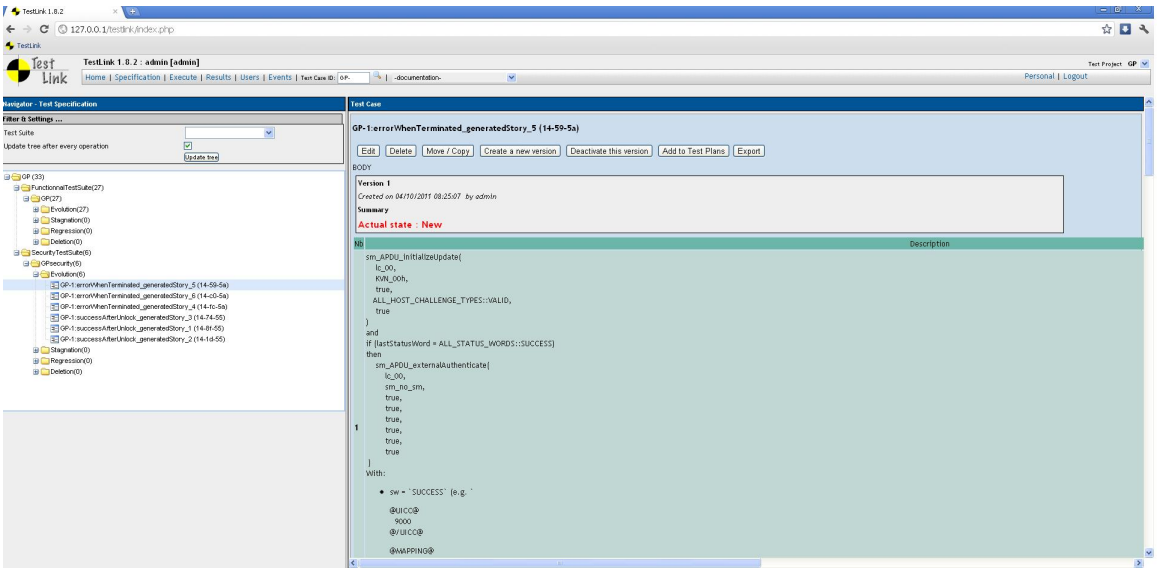

FIGURE 9.8 – Export via Smart Publisher

9.4 Création du plug-in pour IBM Rational Software Architect

Dans cette section, nous présentons l'intégration de l'ensemble des modules. C'est un plugin Eclipse pour IBM Rational Software Architect et il est rendu compatible avec l'outil industriel de Smartesting.

La figure 9.9 est une capture d'écran du plugin nommé *EvoTest*. Il y a deux parties principales dans l'outil. Le premier panel *Test Purposes* est dédié à la rédaction de schémas de tests, produits par Smartesting pour les besoins du projet SecureChange. Le schéma permet d'exprimer des exigences de sécurité pour lesquelles des tests vont être générés. L'utilisateur peut créer plusieurs schémas et les associer à une *smartsuite* donnée. Une *smartsuite* est un fichier xml qui contient les données pour les cibles des tests issus de modèles et utilisés par Smartesting CertifyIt pour la génération de tests.

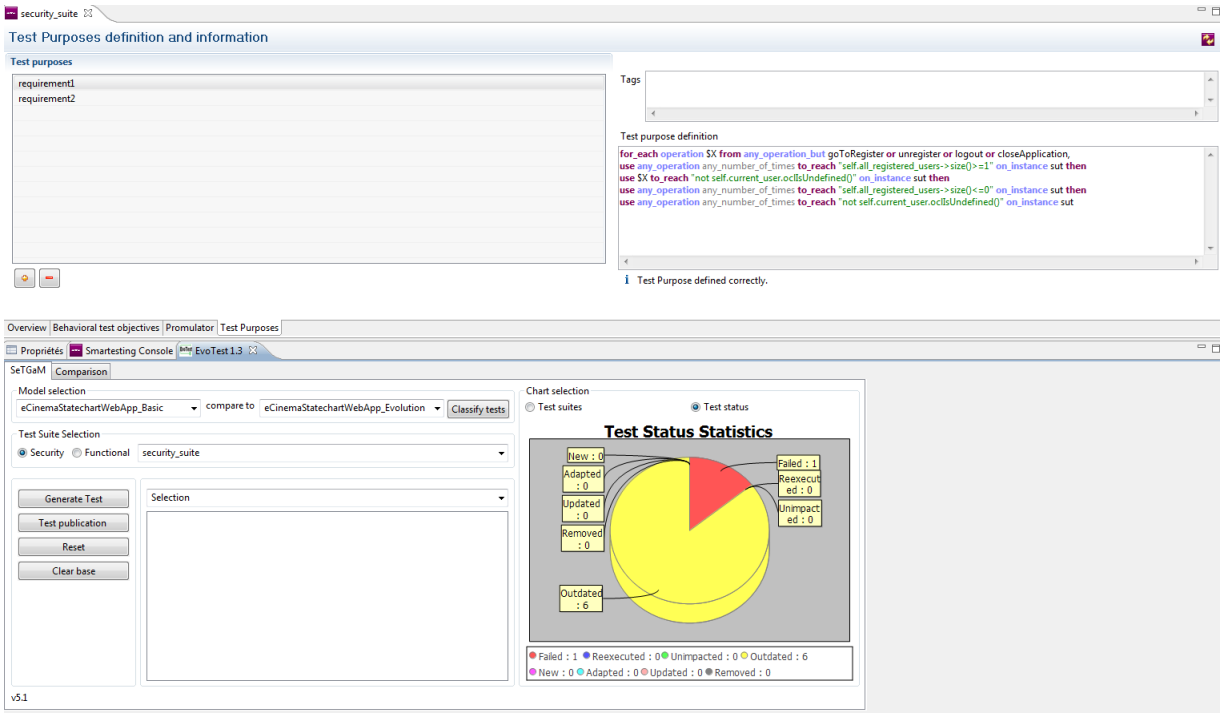

FIGURE 9.9 – L'intégration du prototype en plugin Eclipse

Le deuxième panel est dédié à la solution automatisée de la technique SeTGaM. Il est adapté à la gestion de suites de tests fonctionnelles et aux tests de sécurités, lors des évolutions de système (comme nous l'avons détaillé dans les chapitres 5 à 8). L'interface permet de sélectionner deux versions de modèles et la suite de tests d'entrée (fonctionnelle ou de sécurité). Après l'exécution du processus, les statuts de tests sont affichés dans le camembert. Une couleur est attribuée à chaque statut. Il est possible d'afficher des statistiques concernant les suites de tests : *evolution, regression, stagnation et deletion* en choisissant l'option *test suites*.

Lors de la première génération des tests, les tests sont ajoutés au référentiel directement depuis l'outil Smartesting CertifyIt. Lors de l'utilisation de SeTGaM, l'utilisateur peut exporter les tests dans TestLink et leur statut dans le référentiel de tests (voir Figure 9.8) en cliquant sur le bouton *Test publication*. Ainsi, avec SeTGaM nous assurons l'historique des tests et nous facilitons le processus de maintenance. Ceci est réalisé premièrement en détectant la raison du problème i.e. l'exigence qui a introduit cette erreur et deuxièmement

par la priorité d'exécution des suites de tests. L'expérience nous montre que les tests issus de la suite de *Stagnation* sont les plus importants pour les entreprises traitant des données critiques telles que le partenaire industriel du projet - Gemalto [15]. La raison en est que ces tests adressent les fonctionnalités qui ne doivent plus être autorisées dans le système.

Une fois que les tests sont classifiés (bouton *Classify tests*) et couplés par une génération de tests (bouton *Generate*), l'outil permet l'export vers le référentiel de tests (voir Figure 9.8). L'export est fait dans une base de données *mysql* de *TestLink*, via le *Smart Publisher* que nous avons créé (bouton *Test publication*).

9.5 Synthèse

Dans ce chapitre, nous avons décrit les composants du prototype. Il se compose, d'une part, de la partie développée par Smartesting, dans le cadre projet *SecureChange* : la génération de tests à partir de schémas, l'outil d'animation et de génération de tests et d'autre part de l'implémentation du processus SeTGaM et du Smart Publisher. Les composants sont résumés sur la figure 9.10.

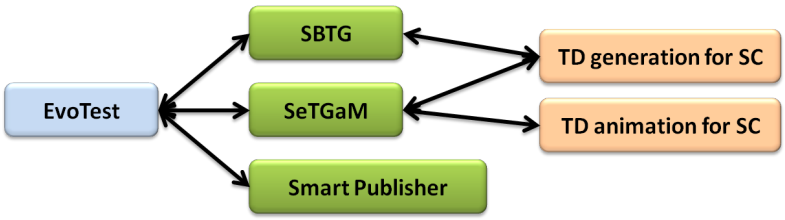

FIGURE 9.10 – Résumé de l'architecture du prototype

- **Schema-Based Test Generation** (SBTG) : ce composant offre deux fonctionnalités : un éditeur de schémas et un générateur de tests à partir de schémas (réalisé par Smartesting).
- **Selective Test Generation Method** (SeTGaM) : ce composant offre la possibilité de calculer les dépendances entre les exigences et permet la classification de tests suite à une évolution du système. Cette méthode peut prendre en compte les aspects de sécurité dans les modèles UML/OCL (indépendamment de la présence ou non d'un diagramme d'états/transitions). Enfin, il est possible de les exporter dans un référentiel de tests via le Smart Publisher.
- **Smart Publisher** : ce composant permet la gestion de tests et la mise à jour de leur cycle de vie, suite aux résultats de SeTGaM dans le référentiel.

Les deux premiers composants utilisent les modules de génération et d'animation de tests développés par Smartesting pour le projet SecureChange, comme cela est présenté sur la figure 9.10.

15. www.gemalto.com

Chapitre 10

Étude de cas : Card Life Cycle et bilan

Sommaire

10.1 Présentation du scope : Card Life Cycle 116
10.1.1 Exigences fonctionnelles 118
10.1.2 Modèles de test 118
10.1.3 Exigences de sécurité 120
10.2 Évolution du scope : Card Life Cycle 122
10.2.1 Les exigences fonctionnelles 122
10.2.2 Modèles de Test 123
10.3 Résultats obtenus pour Card Life Cycle 126
10.3.1 SeTGaM pour les diagrammes d'états/transitions 126
10.3.2 SeTGaM pour les comportements d'opérations UML/OCL 126
10.3.3 SeTGaM pour les TCS 127
10.4 Bilan : Card Life Cycle 128
10.5 Bilan : eCinema 130
10.6 Synthèse 133

Ce cas d'étude porte sur le test des cartes à puce (ou *smart-cards*). Plus exactement, nous effectuons les tests sur la carte nommée *Universal Integrated Circuit Card* (UICC), qui est composée de :

- un circuit intégré embarqué,
- système d'exploitation (en anglais *Operation System* (OS)) qui offre l'accès à la mémoire et à la gestion des différents composants,
- un système *Java Card* (JCS) créé au dessus de l'OS (il offre également des interfaces pour développer des applets [16]),

16. Une Applet Java Card est une application Java, qui respecte la norme ISO 7816. C'est-à-dire qu'elle

– *GlobalPlatform* (GP) qui offre un ensemble de services pour la gestion des applications (comme l'installation des applications, le chargement/exécution d'une application sur la carte, le cycle de vie de la carte, etc),
– des interfaces (U)SIM [17] pour interagir spécifiquement avec les (U)SIM applications [18] [Sec10].

Pour l'étude de cas du projet *SecureChange* (POPS) et selon les besoins identifiés par Gemalto, nous nous intéressons à GP [Con08]. GP offre une interface pour une communication sûre de la carte avec le monde externe, selon les règles décrites dans les spécifications [Sec10, Con08].

La configuration UICC est spécifique à GP et est dédiée aux cartes USIM. Elle associe un ensemble de privilèges aux entités principales de la carte : *Issuer Security Domain* (ISD), *Application Provider Security Domains* (APSD) et *Controlling Authority Security Domain* (CASD).

Nous nous intéressons plus précisément au test du cycle de vie de la carte. Ce dernier, afin qu'il soit disponible d'une manière standardisée en dehors de la carte, est géré par le service proposé par GP. Le cycle de vie permet également de gérer certains risques et certaines politiques de sécurité afin de se protéger des attaques. La carte UICC a été en premier lieu implémentée selon la spécification de GP v2.1.1 et, en tant que telle, a été certifiée par la certification de sécurité des Critères Communs v3.1. Cependant, la carte a été mise à jour suite aux changements apportés dans la version v2.2 de la spécification de GP. Ainsi, le but du WP7 du projet *SecureChange* était de tester cette nouvelle version évoluée contre des régressions et vérifier que les exigences de sécurité sont respectées.

Ce chapitre est organisé de la manière suivante. Nous exposons brièvement dans la section 10.1 le cycle de vie de la carte d'après la spécification GP (en anglais *Card Life Cycle*). Dans la section 10.2, nous présentons l'évolution considérée du périmètre, selon l'évolution de la spécification de v.2.1.1 vers v2.2, configuration Universal Integrated Circuit Card (UICC). Puis, nous allons discuter les résultats obtenus dans la section 10.3. Enfin, nous allons faire le bilan sur le cas d'étude *Card Life Cycle* et sur eCinema, respectivement dans les sections 10.4 et 10.5, avant de conclure ce chapitre en section 10.6.

10.1 Présentation du scope : Card Life Cycle

Dans les documents de GP, les modèles du cycle de vie de la carte sont définis afin de contrôler le comportement des composants de sécurité de GP : la carte, les *executable load files*, les modules exécutables et les applications. Via le cycle de vie de la carte, nous

répond à des requêtes de la même manière qu'elle les reçoit sous la forme de commandes en byte codes.

17. Universal Subscriber Identity Module.

18. Une application (U)SIM est généralement une application Java Card qui stocke et gère les données de l'utilisateur du réseau GSM (ou UMTS) ainsi que sa communication.

allons prendre en compte les étapes de la carte de sa production jusqu'à sa remise au client puis sa destruction.

Pour tester l'implémentation de cette gestion par la carte, nous avons créé deux modèles distincts pour couvrir ce périmètre : un pour GP 2.1.1 et un autre GP 2.2. Il est à noter que nous nous situons au niveau de la configuration particulière de la carte UICC, qui standardise l'interopérabilité minimale entre des applications pour les cartes (U)SIM.

Afin d'introduire les exigences fonctionnelles de *Card Life Cycle*, nous définissons maintenant les entités impliquées :

Issuer Security Domain(ISD) est une composante de la carte qui contient les clefs de sécurité du constructeur de la carte. Il peut les utiliser pour le cryptage des données.

Security Domain(SD) est une application spécifique (ayant le privilège *security domain*) installée sur la carte à puce. Elle est associée à un fournisseur d'applications et elle possède des clefs dédiées.

Application est également une application, mais qui n'a pas le privilège d'un *security domain*. Afin de distinguer cette dernière des *SD*, nous l'appelons *application*.

Les fonctionnalités couvertes et le modèle sont détaillés dans la section 10.1.2, tandis que les exigences de sécurité sont décrites en section 10.1.3.

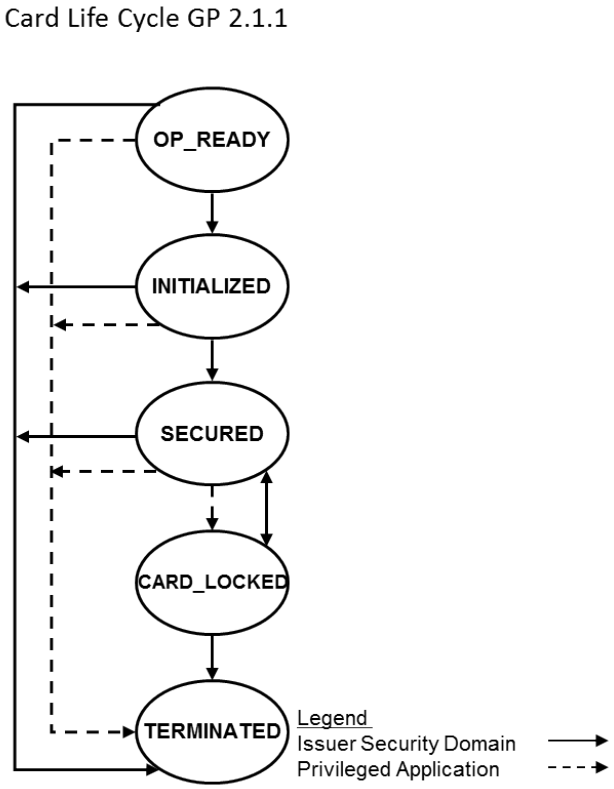

FIGURE 10.1 – Card Life Cycle v2.1.1

10.1.1 Exigences fonctionnelles

La figure 10.1 présente le cycle de vie de la carte. Il commence par l'état *OP_READY*. À l'état *OP_READY*, l'ISD doit être disponible et d'autres applications peuvent être installées. Selon les contraintes définies par la spécification de GP 2.1.1, son cycle évolue jusqu'à l'état *TERMINATED*. Les états autorisés pour son cycle de vie sont :

- OP_READY,
- INITIALIZED,
- SECURED,
- CARD_LOCKED,
- TERMINATED.

Les états de *OP_READY* à *INITIALIZED* sont spécifiquement réservés à la phase de pré-émission de la carte et de livraison chez le fournisseur.

Les trois autres états sont disponibles pour l'utilisation de la carte pendant la phase de post-émission de la carte. Néanmoins, il est possible de rendre la carte inutilisable (état *TERMINATED*) à n'importe quel moment à partir de n'importe quel autre état.

À partir de la description de la figure 10.1, nous pouvons extraire quelques exigences fonctionnelles positives :

1. La transition de *OP_READY* vers *INITIALIZED* est irréversible et seul l'ISD peut effectuer cette action.
2. La transition d'*INITIALIZED* à *SECURED* peut être effectuée uniquement par l'ISD. En arrivant dans cette état, l'ISD doit avoir installé toutes ses clefs de sécurité. Cet état autorise l'utilisation et la modification des applications sur la carte, selon les privilèges attribués aux applications installées.
3. Il est possible de bloquer la carte (état *CARD_LOCKED*). Durant cette étape, la carte n'est plus opérationnelle. Toutefois, l'ISD possède le privilège pour la débloquer et la mettre dans l'état *SECURED*.
4. La carte peut être rendue inutilisable (état *TERMINATED*) par des applications ayant le privilège depuis n'importe quel autre état de la carte.

10.1.2 Modèles de test

Le modèle v2.1.1 du cycle de vie de la carte se focalise sur l'opération, ou encore APDU, *APDU_setStatus* et ses comportements qui permettent de changer le cycle de vie de la carte. Cependant, pour pouvoir générer des tests, le modèle doit contenir suffisamment d'information pour tester chaque cycle de la carte. Ceci demande d'amener le système à une situation donnée. Ainsi, nous modélisons également les comportements nécessaires des autres opérations, c'est-à-dire uniquement ceux permettant d'amener le modèle dans les situations souhaitées.

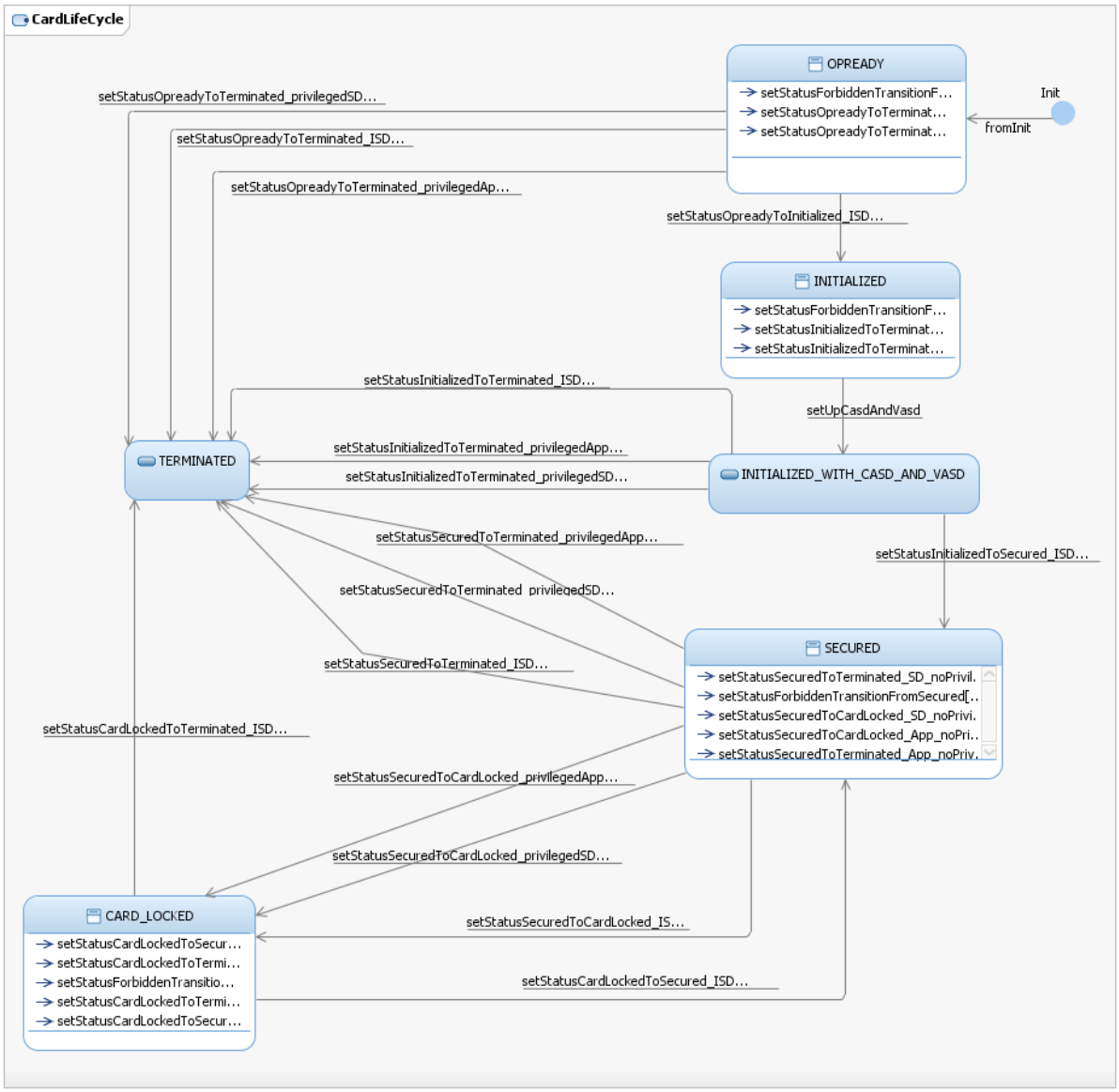

FIGURE 10.2 – Diagramme d'états/transitions Card Life Cycle v2.1.1

Le **diagramme d'états/transitions** pour Card Life Cycle v2.1.1 est présenté sur la figure 10.2. Nous y trouvons les états du cycle de vie de la carte représentés comme des états dans le diagramme d'états/transitions. Les *cas positifs* des exigences fonctionnelles sont représentés respectivement par les *transitions externes* du diagramme.

Les *transitions internes* de chaque état représentent les *cas négatifs* du système suite à une utilisation malveillante. L'état de la carte est représenté par l'état du diagramme d'états/transitions et par la variable *state*, qui contient l'état courant de la carte.

Nous avons obtenu *32 transitions* représentant les comportements de l'*APDU_setStatus*. Ces comportements vont guider la génération de tests.

De manière équivalente au diagramme d'états/transitions, nous avons exprimé les comportements en tant que post condition de l'opération *APDU_setStatus* pour en extraire les **comportements**. Afin de simuler le cycle de la carte, nous utilisons la variable *state* qui contient l'état de la carte en fonction de l'état dans lequel elle se trouve.

Comme déjà discuté précédemment, nous trouvons une différence en nombre de comportements lors de la transformation des transitions en post condition. Pour GP Card Life Cycle, nous avons obtenu *39 comportements* utilisés pour la génération de tests. Les autres opérations utilisées sont les mêmes pour les deux modèles.

10.1.3 Exigences de sécurité

Dans cette section, nous détaillons les exigences de sécurité définies. Pour des raisons de confidentialité, nous avons choisi des exigences de sécurité qui sont valides et intuitives pour toutes les cartes. Dans le Card Life Cycle et l'opération *APDU_setStatus*, chaque application a le droit de changer son cycle selon les privilèges qui lui sont accordées. Les états identifiés pour la carte sont : *OP_READY, INITIALIZED, SECURED, CARD_LOCKED et TERMINATED.*

Pour GP, nous avons ainsi défini deux exigences de sécurité liée à l'état *TERMINATED* : *Card Terminated* et *Terminate Privilege*.

Card Terminated

La première exigence exprimée d'une manière informelle vise à s'assurer qu'une fois la carte rendue inutilisable (mise à l'état *TERMINATED*), par le biais de l'opération APDU_setStatus et une application ayant les privilèges, il est impossible de la ramener à un autre état du cycle de vie de la carte.

Objectifs de test de sécurité Un scénario de test pour cette exigence peut être décrit de manière informelle :

1. sélectionner une application ayant le privilège *Card Terminate*,
2. amener la carte à l'état TERMINATED,
3. essayer n'importe quelle opération décrite dans le modèle et vérifier que le résultat retourné est une erreur.

Schéma de Test Afin d'automatiser la génération de tests, les objectifs sont définis formellement par un schéma. Suite aux objectifs précédemment détaillés pour l'exigence *Card Terminate*, nous avons décrit le schéma suivant :

```
for_each operation $OPERATION from any_operation,
    use any_operation at_least_once
      to_reach "self.selectedApp.privileges.cardTerminate = true" on_instance card then
    use APDU_setStatus
      to_reach "self.state=ALL_STATES : :TERMINATED" on_instance card then
    use $OPERATION
```

Terminate Privilege

La seconde exigence de sécurité consiste à exprimer le fait qu'une application n'ayant pas le privilège *Card Terminate* ne peut en aucun cas changer le cycle de la carte en *TERMINATED* via l'opération APDU_setStatus.

Objectifs de test de sécurité Un scénario de test pour le cas d'échec de cette exigence est décrit informellement de la manière suivante :

1. pour tous les états différents de TERMINATED,
2. sélectionner n'importe quelle application ayant le privilège *Card Terminate*,
3. positionner le cycle de vie de la carte à TERMINATED.

Schéma de Test Nous définissons formellement les objectifs de l'exigence de sécurité *Terminate Privilege* ci-dessous :

```
for_each literal $STATE from
    OP_READY or INITIALIZED or SECURED or CARD_LOCKED,
    use any_operation any_number_of_times
      to_reach "self.selectedApp.privileges.cardTerminate = false
          and self.state = ALL_STATES : :$STATE" on_instance card then
    use card.APDU_setStatus(_, _, _ CARD, TERMINATED)
```

Génération de tests

Tests fonctionnels Pour la génération de tests à partir du diagramme d'états/transitions de GP, nous avons modélisé 9 opérations nécessaires pour la génération de tests, représentant 132 comportements utilisés, dont 32 cibles dédiées au cycle de vie de la carte.

Nous avons utilisé l'outil Smartesting CertifyIt pour couvrir ces cibles. Il a produit 27 tests pour couvrir les 32 cibles.

Pour le modèle sans diagramme d'états/transitions, nous avons utilisé les mêmes opérations, mais nous avons 39 cibles dédiées au cycle de vie de la carte.

Suite à la génération de tests qui n'est pas contrainte par un diagramme d'états/-transitions, l'outil Smartesting CertifyIt a produit 35 tests pour couvrir les 39 cibles de tests.

Tests de sécurité Le générateur de tests crée exactement les mêmes tests pour les deux types de modèles de tests.

Pour la première exigence de sécurité, *9* tests sont générés pour le modèle GP 2.1.1 : un test pour chaque opération. Le schéma ne donne aucune précision quant à l'état qui doit lui permettre d'atteindre l'état *TERMINATED*. Ainsi, il prend le premier état *OP_READY* et change le cycle de vie de la carte à *TERMINATED*, ensuite il fait appel à chaque $*OPERATION*.

La deuxième exigence a produit *4* tests pour le modèle GP 2.1.1 : un pour chaque état donné. Contrairement au premier schéma, celui-ci envoie une précision au générateur de tests, l'obligation d'atteindre l'état de la carte *TERMINATED* à partir de chaque autre état.

10.2 Évolution du scope : Card Life Cycle

L'évolution de GP vers la v2.2 a introduit des changements dans les privilèges des applications qui impactent potentiellement le cycle de vie de la carte.

Il n'y a pas eu d'évolution des exigences de sécurité pour la version 2.2 du Card Life Cycle. Suite à cela, notre objectif est d'identifier l'évolution entre les versions et de la propager vers les modèles de test.

10.2.1 Les exigences fonctionnelles

Sur la figure 10.3, nous présentons l'évolution du graphe, en comparant les deux spécifications. Globalement, l'ISD n'est plus le seul à maitriser le cycle de vie de la carte

dans la phase de pré-émission. Les Security Domains ayant les privilèges peuvent opérer pendant cette phase. Nous pouvons voir deux évolutions principales :

1. Les transitions de *OP_READY* vers *INITIALIZED* et de *INITIALIZED* vers *SECURED* peuvent être effectuées par des Security Domains ayant les privilèges nécessaires.
2. La transition de *SECURED* vers *CARD_LOCKED* peut être faite par une Application ou un Security Domain (l'ISD étant un Security Domain de style "super utilisateur") sous condition qu'il possède le privilège *Card Lock privilege*.
3. La transition de *CARD_LOCKED* vers *SECURED* peut être réalisée par des Security Domains ayant le privilège *Final Application*.
4. Les transitions vers l'état *TERMINATED* sont autorisées pour les Security Domains ou les Applications ayant le privilège *Card Terminate*.

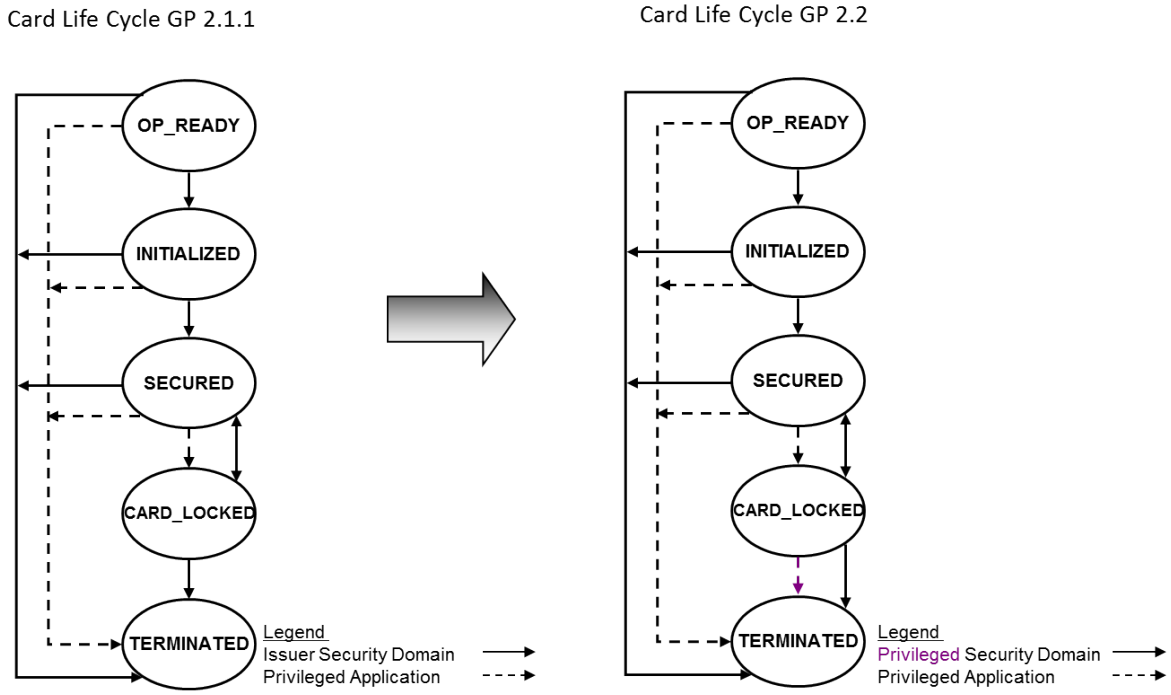

FIGURE 10.3 – Card Life Cycle évolution vers v2.2

10.2.2 Modèles de Test

Nous présentons ici les évolutions des modèles de test (avec et sans diagramme d'états/-transitions).

Modèle avec diagramme d'états/transitions Pour le diagramme d'états/transitions spécifiant GP 2.1.1, nous avons identifié *32* transitions. La deuxième version de GP, contient *37* transitions. Nous illustrons le diagramme sur la figure 10.4.

Comme pour la v2.1.1, 9 opérations, comportant 135 comportements au total, au total sont utilisées pour la génération de tests.

Comme nous pouvons le voir sur la figure 10.4, les états du cycle de vie sont inchangés. Néanmoins, des modifications sont faites afin de prendre en compte l'évolution fonctionnelle, telle que nous l'avons décrite précédemment. Comme dans la spécification, le changement consiste à modifier les transitions afin qu'elles prennent en compte la possibilité d'action des *Applications* et *Security Domains* ayant les privilèges pour changer le cycle de la carte.

En utilisant le composant de comparaison de diagramme d'états/transitions, nous avons obtenu la différence des exigences suivante :

- 4 transitions supprimées,
- 9 transitions nouvelles,
- 28 transitions inchangées.

Nous présentons maintenant l'évolution dans le modèle sans diagramme d'états/transitions.

Modèle sans diagramme d'états/transitions Pour le modèle v2.1.1, nous avons identifié *39* comportements. La modèle v2.2 contient *37* comportements.

En utilisant le composant de comparaison de comportements dans un diagramme de classes, nous avons obtenu la différence des exigences suivante :

- 5 comportements supprimés,
- 3 comportements nouveaux,
- 3 comportements modifiés,
- 31 comportements inchangés.

Comme nous pouvons le remarquer il y a une différence avec les résultats du diagramme d'états/transitions. Ceci est dû au fait que, dans le code OCL de l'action de l'opération, il suffit de changer la pré-condition qui autorise le comportement et mettre à jour sa post-condition (plus précisément mettre à jour la variable *state* qui représente l'état de la carte). Cependant, dans le cas d'une transition, tel que le diagramme d'états/transitions a été construit, ceci est traduit comme une suppression d'une transition et l'ajout d'une nouvelle.

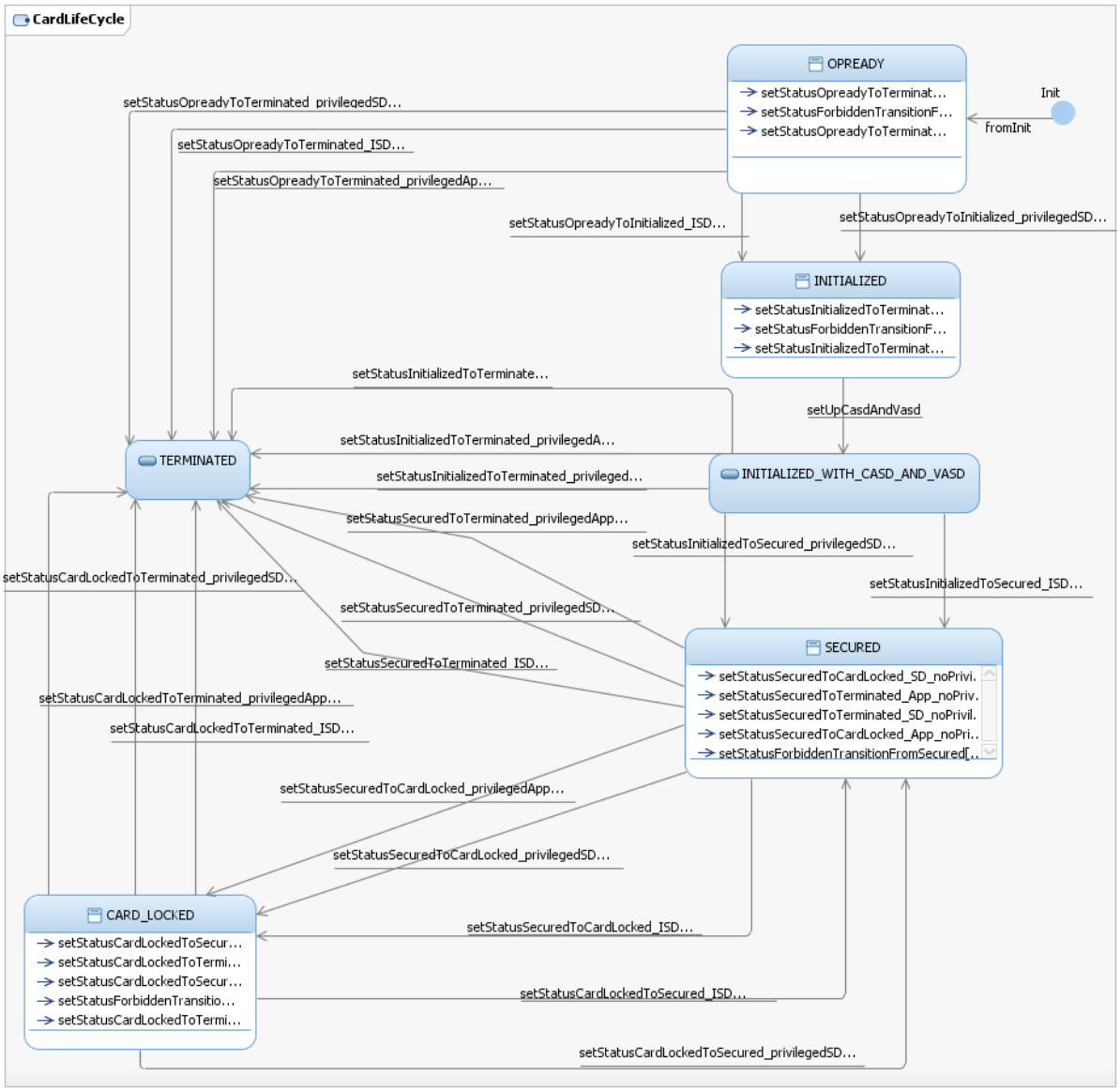

FIGURE 10.4 – Diagramme d'états/transition Card Life Cycle v2.2

10.3 Résultats obtenus pour Card Life Cycle

Nous présentons dans cette partie les étapes de sélection de tests sur le Card Life Cycle. Nous donnons les résultats de la génération sélective des tests pour la version évoluée, pour les deux approches de modélisation (avec et sans diagramme d'états/transitions) et à partir des spécifications des cas de test.

10.3.1 SeTGaM pour les diagrammes d'états/transitions

Le générateur de tests pour la première version a produit *27* tests. Le processus de *SeTGaM*, appliqué à la version évoluée, a classifié les tests de la manière suivante :

- **Outdated** : 4 tests,
- **Unimpacted** : 15 tests,
- **Updated** : 8 tests,
- **New** : 9 tests.

Ainsi, il n'a été nécessaire de ne générer que *9 nouveaux tests*, au lieu de *32*, pour obtenir ainsi une suite de tests d'un total de *32* tests.

10.3.2 SeTGaM pour les comportements d'opérations UML/OCL

Pour la v2.1.1 du modèle sans diagramme d'états/transitions, le générateur a produit **35** tests. Avant de présenter les tests et leurs statuts pour la v2.2 avec SeTGaM, nous allons d'abord donner les métriques obtenues pour la réduction des dépendances et le temps de calcul suite à l'utilisation du solveur :

- Pour la v2.1.1, nous avons réduit le nombre de dépendances à 1325, au lieu de 1729, en environ 5000 secondes (moyenne des temps de calcul de plusieurs exécutions effectuées sur une machine avec un processeur de 2.53GHz et une mémoire installé de 4 Go).
- Pour la v2.2 nous avons réduit le nombre de dépendances à 1425, au lieu de 1857, également en environ 5000 secondes. Le temps de calcul n'est pas négligeable, mais cela est dû à la taille du modèle (un modèle industriel) et à la taille des contraintes à résoudre.

Les impacts calculés suite au très grand nombre de dépendances n'ont pas été réduits. Nous avons obtenu 31 exigences impactées, qui ont été étudiées pour la catégorisation des tests.

L'analyse des tests avec SeTGaM a catégorisé les tests précédemment générés de la façon suivante :

- **Outdated** : 5 tests,
- **Reexecuted** : 16 tests
- **Updated** : 14 tests,
- **New** : 3 tests.

Nous voyons que, suite aux nombreux changements au sein des comportements, même l'approche avec la *dépendance-consistance* ne permet pas d'obtenir l'attribution du statut *unimpacted*, à la place de *reexecuted*. C'est ici que nous trouvons les limites de l'approche en terme de précision, en opposition à l'approche basée sur les diagrammes d'états/transitions. Afin d'avoir une plus grande précision, il serait peut-être judicieux de considérer d'autres contraintes pour réduire le nombre de dépendances.

Dans tous les cas, la valeur ajoutée concerne l'assistance à l'analyse des tests. Pendant l'analyse, le nombre de tests, ayant échoué lors de l'exécution sur le système à étudier, est restreint. Ceci augmente la probabilité de détecter plus facilement et plus rapidement l'erreur dans le système.

10.3.3 SeTGaM pour les TCS

Nous avons appliqué l'approche SeTGaM pour les deux schémas que nous avons présentés précédemment. Les modèles utilisés sont les deux modèles avec et sans diagramme d'états/transitions, et les deux schémas sont ceux donnés précédemment.

Modèle de test avec diagramme d'états/transitions Les deux schémas n'ont produit aucun changement dans les TCS.

Pour le premier schéma, nous avons obtenu *9 unimpacted* et aucun nouveau test à produire.

Cependant, pour le deuxième, nous avons obtenu les catégories suivantes :

- **Outdated** : 1 tests,
- **Unimpacted** : 3 tests,
- **Adapted** : 1 test.

Le test, dont le statut est *adapted*, est issu du test *outdated* du fait qu'il couvre un TCS associé à ce test.

Modèle de test sans diagramme d'états/transitions Comme pour le premier modèle, les deux schémas ont été dépliés de la même manière pour l'évolution du modèle.

Pour le premier schéma, nous avons obtenu *9 reexecuted*. Comme pour le premier modèle, aucun nouveau test n'est à produire.

Cependant, suite à l'évolution dans le modèle, la catégorisation des tests est la suivante :

- **Outdated** : 1 tests,
- **Reexecuted** : 3 tests,
- **Adapted** : 1 test.

De même que précédemment, le test *adapted* est créé pour obtenir une couverture maximale des TCS existants mais plus couverts par les tests courants.

10.4 Bilan : Card Life Cycle

Dans cette section, nous résumons les résultats obtenus par l'approche SeTGaM sur les différents modèles du cycle de vie de la carte à puce.

Afin illustrer les apports d'une méthode comme SeTGaM, nous proposons de la comparer avec deux techniques classiquement utilisées par les industriels pour tester l'absence de régression : retest-all [RHG+01] et regrenerate-all [JTG+10]. Le fait qu'ils optent pour une de ces méthodes est une question de facilité d'implémentation. Cependant, cela a un coût élevé sur l'analyse des résultats d'exécution. Le retest-all, comme nous l'avons dit précédemment, consiste à re-exécuter les tests de l'ancienne suite de tests sur le modèle évolué. D'autre part, comme son nom l'indique, le regenarate-all consiste à regénérer intégralement tous les tests pour le nouveau modèle.

Dans le tableau 10.1, nous synthétisons les résultats obtenus pour le test fonctionnel avec les deux types de modèles, avec et sans diagramme d'états/transitions, et leur évolution. Dans le tableau, nous pouvons voir que nous perdons une information importante sur les tests avec les techniques de retest-all et regenerate-all.

Plus précisément, le retest-all ne permet pas de déterminer les origines du test. Il ne considère que trois statuts généraux : invalide (outdated et failed), réutilisable (unimpacted et reexecuted) et nouveau (new et adapted). Ces deux derniers font partie du groupe des tests valides pour le nouveau système. Cependant, un test *updated* ne peut être identifié. Le regenerate-all ne s'occupe absolument pas des statuts.

Suite à cela, sur le tableau 10.1, nous pouvons voir la différence entre les tests invalides obtenus d'une part avec retest-all (12 et 19 tests respectivement pour les modèles avec et sans diagramme d'états/transitions), et d'autre part avec SeTGaM (4 et 5 tests également pour les deux types de modèles). Ceci est dû à la détection plus efficace des statuts et à la mise à jour de la valeur attendue des tests *updated*. Ces tests, par définition, couvrent des exigences toujours valides et ils subissent une modification de l'attendu afin de prendre en compte l'évolution. Ceci est un écart suffisamment important, qui montre bien les différences de résultats entre les méthodes et la finesse d'attribution des statuts par SeTGaM.

Dans le tableau 10.2, nous donnons également un résumé des résultats pour l'appli-

		Avec diag. d'états/transitions				Sans diag. d'états/transitions			
		v2.1.1	v2.2			v2.1.1	v2.2		
Validité	**Technique** / **Statut**	Gen.	Ret. all	Reg. all	SeTGaM	Gen.	Ret. all	Reg. all	SeTGaM
Tests invalides	Outdated		12		4		19		5
	Failed				0				0
Total invalid	/		12		4		19		5
Tests valides	Reexecuted		15		0		16		16
	Unimpacted				15				0
	Updated				8				14
	Adapted		17		0		17		0
	New	27		32	9	35		33	3
Total valides	/	27	32	32	32	35	33	33	33

TABLE 10.1 – Comparaison résultats Card Life Cycle v2.1.1 et v2.2 pour SeTGaM - test fonctionnel

		Avec diag. d'états/transitions				Sans diag. d'états/transitions			
		v2.1.1	v2.2			v2.1.1	v2.2		
Validité	**Technique** / **Statut**	Gen.	Ret. all	Reg. all	SeTGaM	Gen.	Ret. all	Reg. all	SeTGaM
Tests invalides	Outdated		1		1		1		1
	Failed				0				0
Total invalid	/		1		1		1		1
Tests valides	Reexecuted		12		0		12		12
	Unimpacted				12				0
	Updated				0				0
	Adapted		1		1		1		1
	New	13		13	0	13		13	0
Total valides	/	13	13	13	13	13	13	13	13

TABLE 10.2 – Comparaison résultats Card Life Cycle v2.1.1 et v2.2 pour SeTGaM - test de sécurité

cation de SeTGaM pour le test de sécurité. De même que précédemment, nous pouvons constater, comparé aux deux autres techniques, une plus grande précision de la catégorisation des tests.

Pour conclure, SeTGaM a été évalué sur le *Card Life Cycle* dans le cadre du projet SecureChange par Gemalto/Trusted Labs. Ils notent dans le livrable SecureChange [Sec12a] qu'un fabricant de cartes à puce possède souvent plusieurs plateformes (généralement une par secteur d'activité) ce qui implique des modifications et adaptations du système pour une personnalisation de chaque secteur. Cependant, pour chaque produit certifié, chaque modification doit être soumise à une certification des Critères Communs (CC) du delta en précisant explicitement les changements, les impacts et la manière dont le test est fait.

C'est dans ce contexte de validation des Critères Communs qu'ils voient le plus grand avantage à utiliser une méthodologie comme SeTGaM. Elle a effectivement démontré, qu'en cas de changements de spécification, il est préférable d'apporter des modifications dans le modèle et exploiter les liens des tests avec les objectifs de tests. De plus, ils soulignent que la méthode se trouve extrêmement utile pour la maintenance de la stabilité du référentiel des tests dans le temps et pour plusieurs évolutions.

10.5 Bilan : eCinema

Cette section a pour objectif de synthétiser les résultats obtenus pour l'exemple complet d'eCinema. Nous avons décrit sa version d'origine, son évolution et les exigences de sécurité dans le chapitre 4.

SeTGaM pour les exigences fonctionnelles

Pour la comparaison des modèles avec un diagramme d'états/transitions, nous avons obtenu les résultats suivants :

- 26 transitions nouvelles,
- 6 transitions modifiées,
- 1 transition supprimée.

Ces modifications via les dépendances de données et de contrôle mettent en évidence deux transitions impactées. Elles sont utilisées pour catégoriser les tests. Nous donnons les statuts pour chaque test dans le tableau C.2 (colonne de gauche *Modèle 1*) de l'annexe C.

Pour résumer, nous donnons les données obtenues avec SeTGaM :

- **Outdated** : 3 tests,
- **Failed** : 4 tests,
- **Unimpacted** : 7 tests,

- **Reexecuted** : 2 tests,
- **Adapted** : 6 tests,
- **New** : 19 tests.

D'autre part, pour le modèle sans diagramme d'états/transitions, nous avons calculé les modifications suivantes :

– 26 comportements ajoutés,
– 6 comportements modifiés,
– 1 comportement supprimé,

Avec la *dépendance-consistance*, nous avons obtenu 42 dépendances comportementales, pour la première version d'eCinema et 649 dépendances pour la deuxième version. Ce sont des résultats très satisfaisants vis-à-vis du nombre dépendances de données classiques, respectivement 59 et 739 dépendances. Ceci donne une réduction des dépendances pour les modèles respectivement de 28% et 12%, pour un temps de calcul d'environ 50 secondes (moyenne des temps de calcul de plusieurs exécutions effectués sur une machine avec un processeur de 2.53GHz et une mémoire installée de 4 Go).

Le calcul des dépendances comportementales pour eCinema a réduit le nombre des comportements impactés à trois, au lieu de quatre. Néanmoins, la catégorisation reste la même malgré la réduction d'impacts. Nous donnons le détail des statuts des tests dans le tableau C.2 (colonne de droite *Modèle 2*) de l'annexe C et leur synthèse ci-dessous :

- **Outdated** : 2 tests,
- **Failed** : 4 tests,
- **Unimpacted** : 4 tests,
- **Reexecuted** : 5 tests,
- **Updated** : 1 tests,
- **Adapted** : 4 tests,
- **New** : 21 tests.

Nous pouvons voir qu'il y a une légère différence dans la catégorisation des tests par rapport à l'approche basée sur les diagrammes d'états/transitions. Ceci est dû à la plus petite précision de l'approche basée sur le comportement issu des opérations. De plus, comme dans le cas de l'étude *Card Life Cycle*, ici nous avons une meilleure précision de la part de SeTGaM concernant les tests *obsolètes* qui est due à la détection et à la mise à jour de l'oracle des tests dont le statut est *updated*.

Spécifications de cas de tests

Pour eCinema, nous avons pris en compte deux exigences de sécurité exprimées sous la forme de deux schémas (cf. chapitre 4). Ces schémas ont produit un nombre différent de TCS pour la version de base et pour l'évolution. Les deux modèles (avec et sans diagramme d'états/transitions) ont produit 5 nouvelles TCS, en plus des 7 existantes. Nous avons appliqué SeTGaM pour les 12 TCS et nous avons obtenu les résultats ci-dessous pour les

		Avec diag. d'états/transitions				**Sans diag. d'états/transitions**			
		v1	v2			v1	v2		
Validité	**Technique** / **Statut**	Gen.	Ret. all	Reg. all	SeTGaM	Gen.	Ret. all	Reg. all	SeTGaM
Tests invalides	Outdated		7		3		7		2
	Failed				4				4
Total invalid	/		7		7		7		6
Tests valides	Reexecuted		9		2		9		5
	Unimpacted				7				4
	Updated				0				1
	Adapted		25		6		26		4
	New	16		34	19	16		35	21
Total valides	/	16	34	34	34	16	35	35	35

TABLE 10.3 – Comparaison résultats eCinema v1 et v2 pour SeTGaM - test fonctionnel

		Avec diag. d'états/transitions				**Sans diag. d'états/transitions**			
		v1	v2			v1	v2		
Validité	**Technique** / **Statut**	Gen.	Ret. all	Reg. all	SeTGaM	Gen.	Ret. all	Reg. all	SeTGaM
Tests invalides	Outdated		7		6		7		5
	Failed				1				2
Total invalid	/		7		7		7		7
Tests valides	Reexecuted		0		0		0		0
	Unimpacted				0				0
	Updated				0				0
	Adapted		12		7		12		7
	New	7		12	5	7		12	5
Total valides	/	7	12	12	12	7	12	12	12

TABLE 10.4 – Comparaison résultats eCinema v1 et v2 pour SeTGaM - test de sécurité

deux diagrammes.

Tous les résultats sont identiques à l'exception des outdated et failed. Nous avons respectivement obtenu 6 et 5 tests outdated, et 1 et 2 tests failed, pour le modèle avec et sans diagramme d'états/transitions.

Nous présentons ci-dessous les résultats obtenus :
- **Outdated** : 6 (5) tests,
- **Failed** : 1 (2) tests,
- **Adapted** : 7 tests,
- **New** : 5 tests.

Nous donnons le détail des statuts associés et les tests dans le tableau C.1 de l'annexe C.

Comparaison avec d'autres techniques

Nous avons également comparé les résultats pour eCinema par rapport aux techniques : retest-all et regenerate-all. Nous présentons une synthèse des résultats des exigences fonctionnelles, pour le modèle avec diagramme d'états/transitions et sans, dans le tableau 10.3. Nous donnons également une synthèse pour les deux modèles à propos du traitement des exigences de sécurité (plus exactement en se basant sur les TCS) dans le tableau 10.4.

Comme pour le cas d'étude *Card Life Cycle*, cette comparaison met en évidence l'importance de savoir classer correctement les tests. Même si cela demande un effort de modélisation, souligné dans le livrable D1.3 *SecureChange* [Sec12a] qui fait le bilan sur l'évaluation des outils développés dans le projet, ceci reste abordable et s'apprend comme n'importe quel langage de programmation par les équipes de Recherche et Développement dans les entreprises. Il est évident que l'approche SeTGaM permet de gagner du temps lors du test des systèmes critiques évolutifs, contrairement aux analyses simples ou manuelles.

10.6 Synthèse

Dans ce chapitre nous avons présenté nos travaux sur la partie *Card Life Cycle* de *GlobalPlatform*.

Dans le périmètre du cas d'étude, nous avons appliqué SeTGaM sur quatre cas différents : modèle avec diagramme d'états/transitions, modèle sans diagramme d'états/transitions, et respectivement pour chaque type de modèle (avec et sans diagramme d'états/-transitions), nous avons fait une combinaison pour le test de sécurité par le biais des schémas. Pour cela, nous avons travaillé sur deux exigences de sécurité issues de la spécification GP : *Card Terminated* et *Privileged Application*.

Nous avons fait un bilan sur l'étude menée sur *Card Life Cycle* et sur l'exemple fil rouge *eCinema*. Cette étude nous a permis d'utiliser SeTGaM dans un contexte industriel. Les résultats de l'évaluation faite par Gemalto/Trusted Labs dans le cadre du projet européen *SecureChange*, en tant évaluateur de la technologie du WP7 et particulièrement du prototype *EvoTest* et de la méthodologie SeTGaM ont été très positifs.

Chapitre 11

Évaluation de SeTGaM

Sommaire

11.1 Sûreté 135
11.2 Précision 137
11.3 Efficacité 138
11.3.1 Le temps 138
11.3.2 L'automatisation 139
11.3.3 Calculer la modification 139
11.3.4 Traitement de plusieurs modifications 139
11.4 Généralité 139

Dans ce chapitre, nous évaluons les trois axes de l'approche SeTGaM. Pour cela, nous utilisons le cadre d'évaluation donné par Rothermel et Harrold [RH96]. Cette évaluation est initialement destinée aux techniques basées sur le code, mais la plupart des principes sont applicables sur les techniques basées sur les modèles [BLH09]. L'évaluation de SeTGaM est faite par rapport aux règles de catégorisation que nous avons présentées dans les chapitres précédents, c'est-à-dire les expérimentations et les critères de : sûreté, précision, efficacité et généralité de la méthode. Pour rappel, ces règles permettent d'associer huit statuts du cycle de vie des tests : *new, unimpacted, reexecuted, updated, adapted, failed, outdated* et *removed*. Ces huit statuts sont ensuite classés dans quatre suites de tests : *Evolution, Regression, Stagnation* et *Deletion* (cf. chapitre 5).

11.1 Sûreté

La sûreté d'une technique est en relation avec sa capacité à déterminer les tests dits *modification-revealing*. Ces tests, d'après leur définition, sont des tests non-obsolètes, ne

couvrent pas de nouveaux comportements et produisent un résultat (oracle) différent pour deux versions du programme. Nous donnons ci-dessous la définition initiale d'une technique sûre.

Définition 23 (Technique Sûre) *Une technique est dite sûre si pour tout programme P et son évolution P', elle est capable à 100% de déterminer les tests modification-revealing.*

Vu que nous travaillons avec des comportements issus des modèles UML, nous définissons les tests modification-revealing comme ceux qui produisent un oracle différent pour deux versions du modèle.

La méthode permet également de détecter les tests impactés par le biais de l'analyse des dépendances (de données, de contrôle et de comportements) pour lesquels elle détermine le statut en appliquant les règles de catégorisation. Les tests, dont on ne peut pas conclure immédiatement le statut, sont classifiés dans une catégorie intermédiaire *retestables*. Ces tests sont ensuite animés sur le modèle évolué afin de déterminer leurs statut en fonction de la modification ou non de l'oracle. Ensuite, l'animateur permet de modifier l'oracle et/ou le verdict du test (les séquences du test dans le cas des *adapted*, par le moteur de génération de tests). Ainsi, ces tests exercent les parties modifiées du système.

Ainsi, les tests qui font partie de ce groupe sont issus de la suite de tests *Evolution* et ont les statuts *updated* ou *adapted*. Nous donnons la définition adaptée des tests modification-revealing ci-dessous :

Définition 24 (Modification-revealing) *Un test modification-revealing appartient à la suite de tests, son statut doit être updated ou adapted et il produit forcément un oracle différent dans les deux versions de modèles.*

$$\forall\, tc \in TS^{n+1} \wedge status(tc^{n+1}) = \{updated\ or\ adapted\} \rightsquigarrow oracle(tc^{n}) \neq oracle(tc^{n+1})$$

Par opposition à ces tests, ceux qui sont non-obsolètes et non-modification-revealing sont les *reexecuted* et les *unimpacted*. Les *reexecuted* par définition ont déjà été animés sur le modèle par le moteur d'animation et ont un oracle identique au précédent. Les *unimpacted* sont les tests qui ne sont pas modifiés par la nouvelle version.

Les règles **7**, **12** et **22** (données respectivement dans les sections 6.3.2, 7.4.2 et 8.3.1) détectent les tests non impactés et leur attribuent le statut *unimpacted*.

Enfin, pour dire que SeTGaM est une méthode sûre, il revient à prouver qu'aucun test *unimpacted* ne va produire un oracle différent sur les deux versions de modèle et que les tests *modification-revealing* sont bien détectés. Nous décrivons ainsi le théorème de sûreté de la façon suivante (nous nommons *tc* un cas de test et considérons son évolution d'une version n vers $n+1$).

Théorème 1 (Sûreté) *La technique est sûre si :* $\forall\, tc \in TS^{n+1}$

- $status(tc^{n+1}) = unimpacted \rightsquigarrow oracle(tc^n) = oracle(tc^{n+1})$
- $status(tc^{n+1}) = \{updated\ or\ adapted\} \rightsquigarrow oracle(tc^n) \neq oracle(tc^{n+1})$

La preuve de ce théorème est immédiate de par les règles définissant l'évolution des statuts des tests. Se posent alors la correction de l'implémentation des règles de classification de SeTGaM et la correction de l'algorithme d'animation qui recalcule l'oracle du test.

Pour les cas d'études d'*eCinema* et GlobalPltaform, nous avons utilisé le moteur d'animation des tests sur le modèle et chaque test *unimpacted* a été animé sur le modèle. La méthode attribue le statut *unimpacted* seulement s'il n'y a aucune modification dans les exigences et aucun impact. Pour tous les autres cas, les tests sont animés sur le modèle et désignés comme *reexecuted* si leur oracle ne change pas. De cette façon, 100% des tests non-obsolètes et non modification-revealing sont correctement classés.

11.2 Précision

La précision est importante pour savoir dans quelle mesure la technique de non-régression sélectionne des tests qui vérifient effectivement que le système n'a pas subi de régression.

Définition 25 (Technique précise) *Une technique est dite précise si pour tout programme P et son évolution P', elle ne sélectionne pas des tests non modification-revealing.*

Dans notre cas, prouver la précision de la méthode revient à prouver le théorème qui dit que tous les tests, dont le statut est *updated et adapted*, produisent un oracle différent :

Théorème 2 (Précision) *La technique est précise si :*

$$\forall\, tc \in TS^{n+1}\ and\ status(tc^{n+1}) = \{unimpacted\ or\ adapted\} \rightsquigarrow oracle(tc^n) \neq oracle(tc^{n+1})$$

Pour ce théorème aussi, la preuve est immédiate de par les règles définissant l'évolution des statuts des tests. Se posent alors la correction de l'implémentation des règles de classifcation de SeTGaM et la correction de l'algorithme d'animation qui recalcule l'oracle du test.

Par définition, un test dont le statut est *updated* ou *adapted* est un test modification-revealing et produit un oracle différent quand il est animé sur les deux versions, ce qui a été montré aussi en utilisant le moteur d'animation pour les deux cas d'études.

Cependant, pour cette technique, nous souhaitons exprimer la précision par rapport aux tests dont le statut est *reexecuted* et *unimpacted*. De notre point de vue, une méthode

augmente sa précision si le nombre des tests *reexecuted* est inférieur au nombre de tests *unimpacted*. Une méthode ainsi est précise si l'ensemble des tests *reexecuted* est *vide*.

Sur ce critère, l'axe de SeTGaM basé sur les diagrammes d'états/transitions est plus précise que sans, même si nous avons essayé d'augmenter cette précision en introduisant les dépendances-comportementales. Il s'avère nécessaire d'introduire d'autres contraintes afin de raffiner les dépendances. Néanmoins, cet effort peut nuire à l'efficacité de la technique. Pour conclure sur ce deuxième point de vue, le fait que la technique sélectionne des tests en tant que *reexecuted* montre qu'elle n'est pas suffisamment précise.

11.3 Efficacité

Pour définir si une méthode est efficace, nous ne pouvons pas donner un théorème comme précédemment, il y a plusieurs facteurs à prendre en compte et à détailler : le **temps**, l'**automatisation** de la technique, la **détection des modifications** et la capacité à **traiter plusieurs modifications** dans un même traitement.

11.3.1 Le temps

Le premier facteur est la sélection des tests lors de l'application de la technique elle-même. Elle peut être appliquée lors d'une phase préliminaire, i.e. en parralèle de la correction du logiciel entre deux livraisons.

La deuxième phase, le test de non-régression, constitue la phase critique. Elle débute après les corrections du logiciel. Dans ce contexte particulier, le test de non-régression est limité en temps car souvent borné par les contraintes de livraison du logiciel. La minimisation du coût du test de non-régression est donc très importante.

Afin de savoir si une technique est efficace en temps, il faut déterminer si le temps de sélection des tests est plus long que l'exécution des tests avec la technique *retest-all*. Nous ajoutons à ce facteur la génération des tests pour couvrir les nouveaux comportements et le temps d'analyser les tests de non-régression.

La première approche basée sur les diagramme d'états/transitions (le test de sécurité compris) s'avère très rapide : nous obtenons effectivement une réponse de la classification, même pour GlobalPlatform, en seulement quelques minutes.

La seconde approche basée sur les comportements des opérations est beaucoup plus longue. Ceci est dû à la résolution des contraintes pour détecter la consistance avec le solveur. Ainsi, pour de très grands systèmes, elle demande plus de temps que la première approche. Cependant, ces techniques qui permettent la classification des tests, le calcul des comportements impactés et la gestion du référentiel de tests, apportent un gain de temps lors du processus d'analyse des résultats des tests, qui lui peut prendre des semaines

sans aide automatisée [JTG+10].

Finalement, un avantage majeur de ces techniques basées sur les modèles concerne la classification des tests qui peut être effectuée indépendamment des changements dans le code. Ceci permet alors de gagner plus de temps par rapport à l'efficacité en temps des techniques basées sur le code. Pour cela, nous pouvons la considérer en tant que méthode efficace en temps.

11.3.2 L'automatisation

Le deuxième facteur concerne l'automatisation de la technique. La technique SeTGaM est complètement automatisée. Contrairement à d'autres techniques qui sélectionnent les tests de non-régression, elle calcule également les modifications de l'oracle de test associé. En même temps, elle permet de maintenir la couverture des cibles de tests tout en générant des nouveaux ou des adaptés. De plus, elle permet une gestion du référentiel de test, ce qui facilite la navigation entre les résultats des exécutions des tests, les tests, le modèle et les exigences.

11.3.3 Calculer la modification

Le troisième facteur est la capacité de la technique à calculer les modifications vis-à-vis la nouvelle version. SeTGaM prévoit de calculer les différences, automatiquement entre deux versions, pour toutes les entités considérées dans ses trois axes : les diagrammes d'états/transitions, les comportements des opérations et les spécifications de cas de tests.

11.3.4 Traitement de plusieurs modifications

Le dernier facteur évalue les capacités de la technique à être applicable sur plusieurs modifications à un instant donné. Or, les règles de catégorisation des tests (sections 6.3.2, 7.4.2 et 8.3.1) et le critère de supériorité entre les statuts permettent de gérer plusieurs modifications lors d'une évolution sans aucune ambiguïté.

Nous pouvons donc dire que, d'après ces quatre facteurs, la méthode est efficace lors de l'utilisation des diagrammes d'états/transitions et un peu moins en leur absence.

11.4 Généralité

La généralité consiste à appliquer la technique sur un large panel de situations. Il est impossible de prouver la généralité par un théorème. Nous allons utiliser le principe de

jurisprudence. Les auteurs dans [BLH09] avancent qu'UML est une notation actuellement très utilisée, ce qui rend la solution générale.

Par ailleurs, UML, avec les contraintes OCL, offre plusieurs possibilités pour créer des modèles de test. En proposant une approche qui permet de prendre en compte les comportements issus des gardes/actions des transitions et des pré/post conditions des opérations, nous sommes capables de dire que la technique est suffisamment générale et s'applique à plusieurs situations.

Quatrième partie

Conclusion et perspectives

Chapitre 12

Conclusion

Cette thèse porte sur l'étude de démarches et de techniques pour prendre en compte l'aspect évolution lors de la génération de tests à partir de modèles UML/OCL pour des systèmes sécurisés évolutifs.

Dans ce travail, trois directions ont été étudiées :

1. le cycle de vie des tests pour les systèmes évolutifs critiques,
2. les exigences fonctionnelles,
3. les exigences de sécurité.

Dans un premier temps, nous avons défini la clé de voûte de notre approche qui est le cycle de vie des tests pour les systèmes évolutifs. Nous avons étendu la classification des tests proposée par Leung and White [LW89], et nous avons défini huit statuts dans le cycle de vie du test. Un test, lors de sa création, couvre une nouvelle exigence dans le système et possède alors le statut *new*, qui est le statut initial de chaque test. Lors d'une évolution des exigences, un test peut évoluer de trois façons diffréentes : devenir (*i*) impacté, (*ii*) supprimé ou (*iii*) non-impacté et/ou inchangé. Dans le premier cas, son statut change en *updated* parce que son oracle est modifié suite à l'évolution. Dans le deuxième cas, il est *obsolète* et il peut prendre deux statuts : *outdated*, si l'exigence qu'il couvre est supprimée, ou *failed* s'il couvre une exigence impactée ou modifiée. Afin de couvrir des exigences existantes qui ne sont plus couvertes par un test *failed et outdated*, le statut du test peut évoluer en *adapted*. Mais, dans tout les autres cas du cycle de vie du test, un test *obsolète* ne peut devenir que *removed* et sera uniquement conservé dans l'historique de la base. Enfin, un test ni impacté ni supprimé, peut prendre deux statuts : *reexecuted*, s'il couvre une exigence impactée mais ne relève aucun changement dans l'oracle du test, ou *unimpacted* s'il couvre une exigence non modifiée et non impactée. Ce cycle de vie nous a permis de construire quatre suites de tests : *Evolution, Regression, Stagnation et Deletion* pour tester respectivement l'évolution, la non-régression, la stagnation et garder la trace des tests obsolètes pour une stabilité du référentiel de tests.

À l'issu de ce premier travail, nous avons pu définir la démarche de classification des tests selon les huit statuts du cycle de vie du test, nommée *SeTGaM*, pour tester les systèmes évolutifs. La notation UML, accompagnée du langage de spécification OCL, permet de formaliser des comportements du système. Le langage spécifie ainsi les gardes/actions des transitions et les pré/post conditions des opérations. Le choix de diagramme UML/OCL pour la modélisation, comme nos partenaires industriels nous l'ont confirmé, dépend de la complexité du système ou partie du système modélisé. Afin de garder la généralité de l'approche et prendre en compte des comportements issus de différents types de diagrammes, les objectifs de la méthode ont évolué en deux directions : catégorisation des tests à partir des comportements issus des transitions du diagramme d'états/transitions et ceux issus des opérations du diagramme de classes. Pour atteindre le premier objectif, les statuts des tests sont définis en se basant sur le calcul de différences des transitions des diagrammes d'états/transitions et sur l'analyse des dépendances de données et de contrôle dans ces diagrammes. Pour atteindre le deuxième objectif, nous avons extrait des comportements des opérations du diagramme de classes, et nous avons défini une nouvelle catégorie de dépendances, que nous nommons *dépendances comportementales*, se basant sur le critère de *dépendance-consistance*. Pour catégoriser les tests, la méthodologie se base sur les différences entre les comportements dans les modèles et les dépendances comportementales.

D'autre part, dans le domaine du test des logiciels, une des questions clefs concerne la sécurité. C'est pour cette raison que nous nous basons sur les schémas. Les schémas permettent d'exprimer certaines exigences de sécurité issues des spécifications ou de l'expérience de l'ingénieur de validation. De cette façon, nous avons pu orienter l'objectif fonctionnel de SeTGaM vers un objectif guidé par la sécurité. En se basant sur le dépliage des schémas et sur les comportements dans le modèle, SeTGaM compare les spécifications de cas de tests dépliées et l'évolution fonctionnelle du système, pour classifier les tests selon les statuts définis. Ainsi, avec SeTGaM, nous pouvons couvrir le test de sécurité lors des évolutions du système exprimé à l'aide des schémas.

Finalement, le prototype de SeTGaM a été intégré à l'outil Smartesting CertifyIt comme un plugin d'IBM Rational Software Architect. Cet environnement nous a permis de valider la méthode sur les études de cas fournies par nos partenaires industriels du projet européen *SecureChange*, en particulier Gemalto/Trusted Labs.

Chapitre 13

Perspectives

Ce chapitre présente les perspectives de notre travail. Nous les avons regroupées en trois grandes catégories correspondant aux éléments en amonts et avals de la méthode et sur la méthode SeTGaM elle-même.

13.1 Langages

La partie amont de la méthode SeTGaM concerne les langages de modélisation utilisés pour décrire l'aspect comportemental du système et les scénarios dédiés pour la prise en compte d'exigences particulières comme la sécurité.

Un premier travail serait l'étude systématique des propriétés de sécurité et des besoins de test associés. Cela permettrait de voir comment les prendre en compte avec le langage de schéma. Ainsi, nous pourrions proposer une méthode de description des besoins de test suivant une topographie de propriétés. Cette topographie devrait être affinée suivant les domaines d'application. L'idée serait de définir une méthodologie comme celle proposée dans la certification de critères communs lorsqu'on établit la liste des SFR (Security Functional Requirements) et ainsi d'aider à la rédaction des schémas pour les valider. Cette étude permettrait de connaître réellement les limites du langage et éventuellement d'en proposer une extension.

Comme nous l'avions évoqué, le langage de schéma que nous avons utilisé permet de prendre en compte d'autres besoins de test que ceux liés à la sécurité. En effet, ce langage permet de définir des séquences d'enchainements d'états et de transitions avec la possibilité d'omettre certaines parties. Un travail serait d'étudier de façon théorique l'ensemble des possibilités qu'offre ce langage. Le résultat serait la proposition d'une méthodologie pour aider à la rédaction des schémas, suivant la nature des exigences à valider.

Pour travailler dans un univers complètement UML/OCL, nous pourrions proposer une approche graphique des schémas qui pourraient ainsi être écrits, par exemple, comme

des diagrammes de séquences ou d'activités.

Nous avons travaillé, dans le WP4 du projet *SecureChange*, avec notre partenaire sur la vérification des exigences de sécurité lors de l'évolution du modèle [FBO+12]. Pour effectuer cette vérification, le WP4 doit aussi calculer la différence entre les modules. Nous pourrions utiliser cette information pour la génération sélective des tests par SeTGaM.

L'étape suivante est de mieux valider ces travaux. Pour cela, nous allons nous intéresser aux éléments suivants :

1. dans quelle mesure cette approche permet d'augmenter vraiment la confiance dans le modèle pour le test dans le cas d'évolution du système,
2. est-ce que cette démarche peut être mise en œuvre dans un contexte industriel, et quels sont les verrous qui peuvent en empêcher l'adoption ?

De plus, UMLsec/Ch propose de créer des scénarios d'attaques en utilisant les propriétés de sécurité et des contre-exemples, qui permettent de les satisfaire sur le modèle. Il serait très intéressant de coupler cela dans le processus de SeTGaM afin d'obtenir d'autres types de tests dédiés à la sécurité.

L'élément qui est pertinent à étudier (et qui constitue un des challenges définis par Felderer et al. [FAZB11]) concerne l'intégration de la notion de risque. En effet, dans le cadre d'une approche pour la sécurité, l'analyse de risque est incontournable. Elle peut être faite de façon informelle mais aujourd'hui il existe des méthodes outillées comme celle proposée par CORAS [LSS11]. Les vulnérabilités ou les attaques décrites dans le modèle pourraient être utilisées comme le point d'entrée des schémas voire être traduites automatiquement dans des schémas.

Le dernier élément qui nous semble pertinent à étudier pour cette partie amont est l'intégration des changements qui ne sont pas basés sur l'aspect comportemental, mais sur les changements du code. Les auteurs, dans [NMS+11], soulignent que très peu de techniques se focalisent sur le changement des éléments en dehors du code comme les fichiers de configuration, les bases de données etc. Une possibilité serait donc de prendre en compte l'évolution des configurations pour une même version. Par exemple, ceci aurait été pertinent dans le cadre de GP lors de changement de configuration. Nous avons travaillé dans le cadre de la configuration UICC, mais il en existe d'autres pour la version 2.2. Dans un tel cadre, il serait possible d'analyser les exigences impactées, les tests impactés et l'évolution de leur statut.

13.2 Amélioration de SeTGaM

Cette deuxième catégorie de perspectives regroupe l'ensemble des améliorations qui pourront être faites pour rendre encore plus performante l'approche SeTGaM, qui permet

aujourd'hui de prendre en compte différentes méthodes de modélisation basées sur la notation UML/OCL et les schémas.

Le premier aspect sur la performance concerne le filtrage de la consistance. Les améliorations possibles ne concernent pas la partie génération de formules mais leur traitement qui pourrait être parallélisé. En effet, l'outil Z3 permet nativement d'utiliser plusieurs processeurs ou cœurs mais il est aussi possible de faire traiter chaque recherche de consistance de façon parallèle.

Concernant la phase de génération de tests, nous avons utilisé la démarche outillée proposée par Smartesting. Ce choix nous a imposé les critères de couverture mis à disposition par l'environnement. Nous ne pensons pas que cela ait eu un impact sur notre démarche, mais il serait pertinent de mener des expérimentations en utilisant d'autres environnements de génération de tests pour valider cela, puisque les tests générés dépendent fortement de ces critères.

Il serait aussi intéressant de pouvoir prendre en compte des tests qui ne sont pas issus de la démarche MBT développée dans ce mémoire. Ainsi, ces tests pourraient être issus de méthodes de génération autres ou encore de recette "maison". Se pose alors la question de l'intégration de ces tests pour la classification. Naïvement, nous pouvons considérer qu'ils activent des comportements déjà présents dans le modèle et ainsi imaginer reconstruire le lien avec les exigences. Mais, dans un cadre général, cela aura sûrement des impacts sur le modèle qui devrait être étendu pour prendre en compte les comportements non modélisés. Cela peut rejoindre la question de la synthèse du diagramme d'états/transitions à partir d'une suite de tests. L'intérêt de la présence de ce diagramme est d'avoir des résultats de façon plus efficace. Nous pouvons alors nous poser la question de sa maintenance ou de sa reconstruction en cas d'évolution.

De plus, il existe des méthodes de minimisation des tests pour en réduire le nombre. En effet, il y a toujours la question de réduire le nombre de pas de test à devoir exécuter lors de nouvelle livraison. Il faut donc pouvoir couvrir un maximum de comportements (mis en contexte) en un minimum de pas de test. Dans ce cas, il faudrait pouvoir maintenir le lien entre les tests factorisés et la version initiale complète. En effet, si nous ne gardions que le groupe réduit cela pourrait rendre caduque notre approche car les tests ainsi minimisés risqueraient d'être systématiquement impactés lors d'une évolution.

13.3 Validation et extensions

Cette dernière catégorie regroupe les aspects avals de la méthode. Pour accroître la validation de notre démarche, il semble intéressant de poursuivre les expérimentations avec des études de cas réelles. Ainsi, nous pourrions voir si notre démarche est cohérente et peut s'appliquer à d'autres systèmes que ceux déjà étudiés.

Il existe des travaux sur la sélection de tests et la non-régression à partir du code [RH97, JW92]. Nous pensons qu'il serait intéressant de coupler notre démarche avec celles-ci.

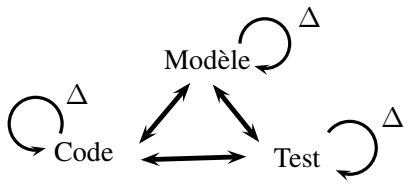

FIGURE 13.1 – Liens entre modèle, test et code

La figure 13.1 présente les deux approches : la nôtre et l'approche basée sur le code. D'une part, celle présentée dans ce mémoire permet d'avoir le lien entre le modèle et les tests. D'autre part, les approches à partir du code permettent de faire le lien entre les tests et le code, les tests étant l'entité partagée entre les deux approches. La valeur ajoutée de cette collaboration serait de pouvoir maintenir un lien entre le code et le modèle. Ainsi, nous pourrions avoir une approche boite grise. Les deux cas à traiter seraient :

1. Modèle $\Longrightarrow$ Code : Les évolutions du modèle sont prises en compte par la génération de tests. Les tests sont joués sur le code. Nous sommes dans l'approche classique présentée dans ce mémoire.
2. Code $\Longrightarrow$ Modèle : Les outils de couverture permettent de connaître les parties du code qui sont couvertes par les tests. En utilisant l'information sur le delta entre les deux versions du code et la couverture, il serait possible de confronter les informations. Alors, les résultats obtenus seraient :
   - Les éléments de code ajoutés sont bien couverts par les tests, et si ce n'est pas le cas, à quoi sert le code ajouté ? Dans ce cas, nous serions face à un ajout de comportement non demandé dans l'itération courante ou alors dans la partie de code non atteignable dans l'ensemble fonctionnel considéré.
   - Les éléments de code modifiés sont bien activés par les tests d'évolution ou des tests qui échouaient précédemment (correction de bugs).
   - Les éléments de code supprimés ne sont plus utilisés par les tests (sauf les tests de non stagnation).

Toutes ces perspectives montrent qu'il existe encore beaucoup de travail dans le domaine du test des systèmes évolutifs et critiques. Nous espérons que les travaux présentés dans ce mémoire aideront à améliorer le processus de sélection des tests et contribueront à l'adoption de méthodes basées sur les modèles, autant dans l'objectif du test fonctionnel que du test de sécurité.

Annexes

Annexe A

Détails sur les exigences fonctionnelles pour les modèles d'eCinema

Dans cette annexe nous donnons les détails pour les exigences fonctionnelles d'eCinema. Les tableaux A.1 et A.3 synthétisent les exigences globales pour les modèles v1 et v2 d'eCinema, dénotées *REQ*. Le tableau A.2 synthétise les raffinements, dénotées *AIM*, qui identifient un comportement issu d'une transition ou d'une opération. De plus, les transitions/comportements modifiés, supprimés et nouveaux sont décrits dans les tableaux A.4 et A.5.

Nb	**REQ**	**Nom**	**Description**
1	ACCOUNT_MNGT/LOG	Log	Capacité du système à garantir ou interdire l'accès au compte des utilisateurs
2	ACCOUNT_MNGT/REGISTRATION	Registration	Capacité du système à gérer un compte utilisateur
3	BASKET_MNGT/BUY_TICKETS	Buy_Tickets	Capacité du système à gérer l'achat de tickets d'un utilisateur identifié
4	BASKET_MNGT/DISPLAY_BASKET and DISPLAY_BASKET_PRICE	Display_Basket and Display_Basket_Price	Capacité du système à afficher le panier d'un utilisateur identifié et le prix total du panier
5	BASKET_MNGT/REMOVE_TICKETS	Remove_Tickets	Capacité du système à supprimer des tickets d'un utilisateur identifié
6	CLOSE_APPLICATION	Close_Application	La capacité du système à fermer à l'application
7	NAVIGATION	Navigation	La capacité du système de naviguer d'une page à une autre dans l'application web

TABLE A.1 – Exigences pour eCinema

Comportement		**Transition**	
REQ_Nb	AIM	Nom	Id
1	LOG_Empty_User_Name	login(in_userName, in_userPassword) - LGN_EmptyUserName	t3
1	LOG_Invalid_Password	login(in_userName, in_userPassword) - LGN_InvalidPassword	t4
1	LOG_Invalid_User_Name	login(in_userName, in_userPassword) - LGN_InvalidUserName	t5
1	LOG_Logout	logout() - LGN_Logout	t9
1	LOG_Success	login(in_userName, in_userPassword) - LGN_SUCCESS	t2
2	REG_Empty_Password	registration(in_userName,in_userPassword) - REG_EmptyPwd	t21
2	REG_Empty_User_Name	registration(in_userName,in_userPassword) - REG_EmptyUser	t19
2	REG_Go_To_Register	goToRegister()	t17
2	REG_Login_Already_Exists	registration(in_userName,in_userPassword) - REG_Existing_UserName	t20
2	REG_Success	registration(in_userName,in_userPassword) - REG_Success	t18
2	REG_Unregister	unregister() - REG_Unregister	t1
3	BUY_Login_Mandatory	buyTicket(in_parameter) - BASKET_MNGT/BUY_TICKETS - LOGIN	t7
3	BUY_Sold_Out	buyTicket(in_parameter) - BASKET_MNGT/BUY_TICKETS-SOLDOUT	t8
3	BUY_Success	buyTicket(in_parameter) - BASKET_MNGT/BUY_TICKETS - SUCCESS	t6
4	DIS_Check_Basket	showBoughtTickets() - Logged in	t11
5	REM_Del_All_Tickets	deleteAllTickets(in_title)	t16
5	REM_Del_Ticket	deleteOneTicket(in_title) - REM_Del_Ticket	t15
6	CLOSE_success	close	t0
7	NAV_Go_To_Home-display	goToHome() - NAV_Go_To_Home	t13
7	NAV_Go_To_Home-register	goToHome() - NAV_Go_To_Home - 2	t14

TABLE A.2 – Comportement/Transitions pour eCinema.

Nb	REQ	Nom	Description
8	BALANCE_MNGT	BalanceManagement	La capacité du système de gérer le solde des comptes des utilisateurs
9	SUBSCRIPTION_MNGT	SubscriptionManagement	La capacité du système de gérer les abonnements des utilisateurs

TABLE A.3 – Nouvelles exigences - eCinema évolution

Comportement	Transition	Évolution
Register_Success	t18	Modifiée
Empty_User_Name	t19	Modifiée
Login_Already_Exists	t20	Modifiée
Empty_Password	t21	Modifiée
Del_Ticket	t15	Modifiée
Del_All_Tickets	t16	Modifiée
Buy_Success	t6	Supprimée

TABLE A.4 – Comportements/Transitions modifiés et supprimés - eCinema

Comportement		**Transition**	
REQ Nb	AIM	Nom	Id
2	REG_Invalid_Young_Subs	registration(in_userSubscription) - REG_YoungIncompSubscription	t35-1
2	REG_Invalid_Senior_Subs	registration(in_userSubscription) - REG_SeniorIncompSubscription	t35-2
3	BUY_No_More_Money	buyTicket(in_parameter) - BASKET_MNGT/BUY_TICKETS-NO-MORE-MONEY	t24
3	BUY_Success, BUY_Young	buyTicket(in_parameter) - BASKET_MNGT/BUY_TICKETS - SUCCESS - YOUNG	t36
3	BUY_Success, BUY_Normal	buyTicket(in_parameter) - BASKET_MNGT/BUY_TICKETS - SUCCESS - NORMAL	t38
3	BUY_Success, BUY_OneFree	buyTicket(in_parameter) - BASKET_MNGT/BUY_TICKETS - SUCCESS - ONEFREE	t40
3	BUY_Success, BUY_OneFree, BUY_Free	buyTicket(in_parameter) - BASKET_MNGT/BUY_TICKETS - SUCCESS - ONEFREE4FREE	t41
3	BUY_Success, BUY_Senior	buyTicket(in_parameter) - BASKET_MNGT/BUY_TICKETS - SUCCESS - SENIOR	t39
3	BUY_Success, BUY_Student	buyTicket(in_parameter) - BASKET_MNGT/BUY_TICKETS - SUCCESS - STUDENT	t37
5	REM_Del_Ticket, REM_One_Free	deleteOneTicket(in_title) - REM_Del_Ticket-ONEFREE	t48
5	REM_Del_Ticket, REM_One_Free, REM_Free	deleteOneTicket(in_title) - REM-Del-Ticket-ONEFREE4FREE	t47
5	REM_NO_MORE_TICKET	deleteOneTicket(in_title) - REM-Del-Ticket-No_tickets	t46-1
5	REM_Not_Enough_Tickets	deleteAllTickets() - REM-DelleteAll-not_enough	t46-2
6	CLOSE_success_FROM_BALANCE	close - BALANCE	t33
6	CLOSE_success_FROM_SUBSCRIPTION	close - SUBSCRIPTION	t30
7	NAV_Go_To_Home-balance	goToHome - BALANCE	t34
7	NAV_Go_To_Home-subscription	goToHome() - SUBSCRIPTION	t29
8	BAL_ADD_NEGATIVE_AMOUNT	addUnits(in_units) - BALANCE_MNGT/Add - NEGATIVE_AMOUNT	t45
8	BAL_ADD_SUCCESS	addUnits(in_units) - BALANCE_MNGT/Add - SUCCESS	t31
8	BAL_Go_To_8	goToBalance_mngt()	t32
8	BAL_RETRIEVE_BalanceInsufficient	retrieveUnits(in_units) - BALANCE_MNGT - Retrieve_BALANCE_INSUFFICIENT	t43
8	BAL_RETRIEVE_NEGATIVE_AMOUNT	retrieveUnits(in_units) - BALANCE_MNGT - Retrieve_NEGATIVE_AMOUNT	t44
8	BAL_RETRIEVE_SUCCESS	retrieveUnits(in_units - BALANCE_MNGT - Retrieve_SUCCESS	t42
9	SUBS_Error_Sub	setSubscription(in_subscription) - SUBSCRIPTION_MNGT - INCOMPATIBLE	t26
9	SUBS_Go_To_SUBSCRIPTION	goToSubscription_mngt()	t28
9	SUBS_Set_Up	setSubscription(in_subscription) - SUBSCRIPTION_MNGT - SUCCESS	t25

TABLE A.5 – Nouveaux comportements/transitions - eCinema évolution

Annexe B

Relations de dépendances pour les modèles d'eCinema

Dans cette annexe nous donnons les relations des dépendances, pour les deux diagrammes d'eCinema, avec et sans diagrammes d'états/transitions. Le tableau B.1 donne les dépendances de données et de contrôle pour la v1 d'eCinema avec diagramme d'états/-transitions. Les tableaux B.4 et B.3 représentent de leur côté les dépendances de données et de contrôle pour le modèle évolué avec diagramme d'états/transitions. Le tableau B.2 donne les dépendances comportementales pour eCinema premiere version, sans diagramme d'états/transitions. Le tableau B.5 donne les dépendances comportementales pour eCinema version évoluée, sans diagramme de classes.

Transition utilisatrice/dépendante	**Transitions de définition**	**Transitions contrôleur**
t1	t1,t2,t6,t9,t18	
t2	t1,t2,t9,t18	
t3	t1,t18	
t4	t1,t18	
t6	t1,t2,t6,t9,t15,t16	
t7	t2,t9,t10	
t8	t1,t6,t15,t16	
t9	t1,t2,t9,t18	
t11	t1,t2,t9,t18	
t13		t17
t14		t11
t15	t1,t2,t6,t9,t18	
t16	t1,t2,t6,t9,t18	
t18	t1,t18	t17
t20	t1,t18	

TABLE B.1 – Dépendances de données et de contrôle pour le diagramme d'états/transitions pour eCinema

Comportement de définition / AIM	**Comportement de définition / AIM**	**Variable**
BUY_Success	BUY_Success	available_tickets
BUY_Success	BUY_Success	current_user
BUY_Success	REM_Del_All_Tickets	all_tickets_in_basket
BUY_Success	REM_Del_All_Tickets	available_tickets
BUY_Success	REM_Del_All_Tickets	current_user
BUY_Success	REM_Del_Ticket	all_tickets_in_basket
BUY_Success	REM_Del_Ticket	available_tickets
BUY_Success	REM_Del_Ticket	current_user
LOG_Logout	REM_Del_All_Tickets	current_user
LOG_Logout	REM_Del_Ticket	current_user
LOG_Success	BUY_Success	current_user
LOG_Success	REG_Unregister	current_user
LOG_Success	REM_Del_All_Tickets	current_user
LOG_Success	REM_Del_Ticket	current_user
REG_Success	BUY_Success	current_user
REG_Success	REG_Unregister	all_registered_users
REG_Success	REG_Unregister	current_user
REG_Unregister	LOG_Invalid_Password	all_registered_users
REG_Unregister	LOG_Success	all_registered_users
REG_Unregister	REG_Success	all_registered_users
REG_Unregister	REM_Del_All_Tickets	all_tickets_in_basket
REG_Unregister	REM_Del_All_Tickets	available_tickets
REG_Unregister	REM_Del_All_Tickets	current_user
REG_Unregister	REM_Del_Ticket	all_tickets_in_basket
REG_Unregister	REM_Del_Ticket	available_tickets
REG_Unregister	REM_Del_Ticket	current_user
REM_Del_All_Tickets	BUY_Success	available_tickets
REM_Del_All_Tickets	BUY_Success	owner_ticket
REM_Del_All_Tickets	REG_Unregister	available_tickets
REM_Del_All_Tickets	REM_Del_All_Tickets	available_tickets
REM_Del_All_Tickets	REM_Del_All_Tickets	movie
REM_Del_All_Tickets	REM_Del_Ticket	available_tickets
REM_Del_All_Tickets	REM_Del_Ticket	movie
REM_Del_Ticket	BUY_Success	available_tickets
REM_Del_Ticket	BUY_Success	owner_ticket
REM_Del_Ticket	REG_Unregister	available_tickets
REM_Del_Ticket	REM_Del_All_Tickets	available_tickets
REM_Del_Ticket	REM_Del_All_Tickets	movie
REM_Del_Ticket	REM_Del_Ticket	available_tickets
REM_Del_Ticket	REM_Del_Ticket	movie

TABLE B.2 – Dépendances comportementales pour eCinema

Transition dépendante	Transitions contrôleur
t0	t25,t29,t31,t34
t11	t25,t29,t31,t34
t13	t17
t14	t11
t17	t25,t29,t31,t34
t18	t17
t24	t25,t29,t31,t34
t25	t28
t28	t25,t29,t31,t34
t29	28
t30	28
t31	t24
t33	t24
t34	t24

TABLE B.3 – Dépendances de contrôle pour le diagramme d'états/transitions d'eCinema - évolution

Transition utilisatrice	Transitions de définition
t1	t1,t2,t9,t18,t36-t41
t8	t1,t16,t36-t41,t47-t48
t46-1	t1,t16,t36-t41,t47-t48
t46-2	t1,t16,t36-t41,t47-t48
t16	t1,t2,t9,t18,t36-t41,t47-t48
t24	t16,t36-t41,t47-t48
t36	t1,t2,t9,t16,t18,t31,t36-t41,t42,t47-t48
t38	t1,t2,t9,t16,t18,t31,t36-t41,t42,t47-t48
t39	t1,t2,t9,t16,t18,t31,t36-t41,t42,t47-t48
t40	t1,t2,t9,t16,t18,t31,t36-t41,t42,t47-t48
t41	t1,t2,t9,t16,t18,t31,t36-t41,t42,t47-t48
t42	t16,t31,t36-t41,t42,t47-t48
t43	t16,t31,t36-t41,t42,t47-t48
t47	t1,t2,t9,t16,t18,t36-t41,t47-t48
t48	t1,t2,t9,t16,t18,t36-t41,t47-t48

TABLE B.4 – Dépendances de données pour le diagramme d'états/-transitions eCinema - évolution

Comportement de définition / AIM	Comportement dépendant / AIM	Variable
BAL_ADD_SUCCESS	BAL_ADD_SUCCESS	balance
BAL_ADD_SUCCESS	BAL_ADD_SUCCESS	current_user
BAL_ADD_SUCCESS	BAL_RETRIEVE_SUCCESS	balance
BAL_ADD_SUCCESS	BAL_RETRIEVE_SUCCESS	current_user
BAL_ADD_SUCCESS	BUY_Young, BUY_Success	balance
BAL_ADD_SUCCESS	BUY_Young, BUY_Success	current_user
BAL_ADD_SUCCESS	BUY_Free, BUY_OneFree, BUY_Success	balance
BAL_ADD_SUCCESS	BUY_Free, BUY_OneFree, BUY_Success	current_user
BAL_ADD_SUCCESS	BUY_Normal, BUY_Success	balance
BAL_ADD_SUCCESS	BUY_Normal, BUY_Success	current_user
BAL_ADD_SUCCESS	BUY_OneFree, BUY_Success	balance
BAL_ADD_SUCCESS	BUY_OneFree, BUY_Success	current_user
BAL_ADD_SUCCESS	BUY_Senior, BUY_Success	balance
BAL_ADD_SUCCESS	BUY_Senior, BUY_Success	current_user
BAL_ADD_SUCCESS	BUY_Student, BUY_Success	balance
BAL_ADD_SUCCESS	BUY_Student, BUY_Success	current_user
BAL_ADD_SUCCESS	REG_Unregister	current_user
BAL_ADD_SUCCESS	REM_Del_All_Tickets	balance
BAL_ADD_SUCCESS	REM_Del_All_Tickets	current_user
BAL_ADD_SUCCESS	REM_Del_Ticket, REM_Free, REM_One_Free	balance
BAL_ADD_SUCCESS	REM_Del_Ticket, REM_Free, REM_One_Free	current_user
BAL_ADD_SUCCESS	REM_Del_Ticket, REM_One_Free	balance
BAL_ADD_SUCCESS	REM_Del_Ticket, REM_One_Free	current_user
BAL_ADD_SUCCESS	REM_Del_Ticket	balance
BAL_ADD_SUCCESS	REM_Del_Ticket	current_user
BAL_ADD_SUCCESS	REM_NO_MORE_TICKET	current_user
BAL_RETRIEVE_SUCCESS	BAL_ADD_SUCCESS	balance
BAL_RETRIEVE_SUCCESS	BAL_ADD_SUCCESS	current_user
BAL_RETRIEVE_SUCCESS	BAL_RETRIEVE_SUCCESS	balance
BAL_RETRIEVE_SUCCESS	BAL_RETRIEVE_SUCCESS	current_user
BAL_RETRIEVE_SUCCESS	BUY_Young, BUY_Success	balance

Comportement de définition / AIM	**Comportement dépendant / AIM**	**Variable**
BAL_RETRIEVE_SUCCESS	BUY_Young, BUY_Success	current_user
BAL_RETRIEVE_SUCCESS	BUY_Free, BUY_OneFree, BUY_Success	balance
BAL_RETRIEVE_SUCCESS	BUY_Free, BUY_OneFree, BUY_Success	current_user
BAL_RETRIEVE_SUCCESS	BUY_Normal, BUY_Success	balance
BAL_RETRIEVE_SUCCESS	BUY_Normal, BUY_Success	current_user
BAL_RETRIEVE_SUCCESS	BUY_OneFree, BUY_Success	balance
BAL_RETRIEVE_SUCCESS	BUY_OneFree, BUY_Success	current_user
BAL_RETRIEVE_SUCCESS	BUY_Senior, BUY_Success	balance
BAL_RETRIEVE_SUCCESS	BUY_Senior, BUY_Success	current_user
BAL_RETRIEVE_SUCCESS	BUY_Student, BUY_Success	balance
BAL_RETRIEVE_SUCCESS	BUY_Student, BUY_Success	current_user
BAL_RETRIEVE_SUCCESS	REG_Unregister	current_user
BAL_RETRIEVE_SUCCESS	REM_Del_All_Tickets	balance
BAL_RETRIEVE_SUCCESS	REM_Del_All_Tickets	current_user
BAL_RETRIEVE_SUCCESS	REM_Del_Ticket, REM_Free, REM_One_Free	balance
BAL_RETRIEVE_SUCCESS	REM_Del_Ticket, REM_Free, REM_One_Free	current_user
BAL_RETRIEVE_SUCCESS	REM_Del_Ticket, REM_One_Free	balance
BAL_RETRIEVE_SUCCESS	REM_Del_Ticket, REM_One_Free	current_user
BAL_RETRIEVE_SUCCESS	REM_Del_Ticket	balance
BAL_RETRIEVE_SUCCESS	REM_Del_Ticket	current_user
BAL_RETRIEVE_SUCCESS	REM_NO_MORE_TICKET	current_user
BUY_Young, BUY_Success	BAL_ADD_SUCCESS	balance
BUY_Young, BUY_Success	BAL_ADD_SUCCESS	current_user
BUY_Young, BUY_Success	BAL_RETRIEVE_SUCCESS	balance
BUY_Young, BUY_Success	BAL_RETRIEVE_SUCCESS	current_user
BUY_Young, BUY_Success	BUY_Young, BUY_Success	all_sold_tickets
BUY_Young, BUY_Success	BUY_Young, BUY_Success	all_tickets_in_basket
BUY_Young, BUY_Success	BUY_Young, BUY_Success	available_tickets
BUY_Young, BUY_Success	BUY_Young, BUY_Success	balance
BUY_Young, BUY_Success	BUY_Young, BUY_Success	current_user
BUY_Young, BUY_Success	BUY_Young, BUY_Success	price
BUY_Young, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	all_sold_tickets
BUY_Young, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	all_tickets_in_basket
BUY_Young, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	available_tickets
BUY_Young, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	balance
BUY_Young, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	current_user
BUY_Young, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	price
BUY_Young, BUY_Success	BUY_Normal, BUY_Success	all_sold_tickets
BUY_Young, BUY_Success	BUY_Normal, BUY_Success	all_tickets_in_basket
BUY_Young, BUY_Success	BUY_Normal, BUY_Success	available_tickets
BUY_Young, BUY_Success	BUY_Normal, BUY_Success	balance
BUY_Young, BUY_Success	BUY_Normal, BUY_Success	current_user
BUY_Young, BUY_Success	BUY_Normal, BUY_Success	price
BUY_Young, BUY_Success	BUY_OneFree, BUY_Success	all_sold_tickets
BUY_Young, BUY_Success	BUY_OneFree, BUY_Success	all_tickets_in_basket
BUY_Young, BUY_Success	BUY_OneFree, BUY_Success	available_tickets
BUY_Young, BUY_Success	BUY_OneFree, BUY_Success	balance
BUY_Young, BUY_Success	BUY_OneFree, BUY_Success	current_user
BUY_Young, BUY_Success	BUY_OneFree, BUY_Success	price
BUY_Young, BUY_Success	BUY_Senior, BUY_Success	all_sold_tickets
BUY_Young, BUY_Success	BUY_Senior, BUY_Success	all_tickets_in_basket
BUY_Young, BUY_Success	BUY_Senior, BUY_Success	available_tickets
BUY_Young, BUY_Success	BUY_Senior, BUY_Success	balance
BUY_Young, BUY_Success	BUY_Senior, BUY_Success	current_user
BUY_Young, BUY_Success	BUY_Senior, BUY_Success	price
BUY_Young, BUY_Success	BUY_Student, BUY_Success	all_sold_tickets
BUY_Young, BUY_Success	BUY_Student, BUY_Success	all_tickets_in_basket
BUY_Young, BUY_Success	BUY_Student, BUY_Success	available_tickets

Comportement de définition / AIM	Comportement dépendant / AIM	Variable
BUY_Young, BUY_Success	BUY_Student, BUY_Success	balance
BUY_Young, BUY_Success	BUY_Student, BUY_Success	current_user
BUY_Young, BUY_Success	BUY_Student, BUY_Success	price
BUY_Young, BUY_Success	REG_Unregister	all_tickets_in_basket
BUY_Young, BUY_Success	REG_Unregister	available_tickets
BUY_Young, BUY_Success	REG_Unregister	current_user
BUY_Young, BUY_Success	REM_Del_All_Tickets	all_tickets_in_basket
BUY_Young, BUY_Success	REM_Del_All_Tickets	available_tickets
BUY_Young, BUY_Success	REM_Del_All_Tickets	balance
BUY_Young, BUY_Success	REM_Del_All_Tickets	current_user
BUY_Young, BUY_Success	REM_Del_All_Tickets	price
BUY_Young, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	all_tickets_in_basket
BUY_Young, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	available_tickets
BUY_Young, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	balance
BUY_Young, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	current_user
BUY_Young, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	price
BUY_Young, BUY_Success	REM_Del_Ticket, REM_One_Free	all_tickets_in_basket
BUY_Young, BUY_Success	REM_Del_Ticket, REM_One_Free	available_tickets
BUY_Young, BUY_Success	REM_Del_Ticket, REM_One_Free	balance
BUY_Young, BUY_Success	REM_Del_Ticket, REM_One_Free	current_user
BUY_Young, BUY_Success	REM_Del_Ticket, REM_One_Free	price
BUY_Young, BUY_Success	REM_Del_Ticket	all_tickets_in_basket
BUY_Young, BUY_Success	REM_Del_Ticket	available_tickets
BUY_Young, BUY_Success	REM_Del_Ticket	balance
BUY_Young, BUY_Success	REM_Del_Ticket	current_user
BUY_Young, BUY_Success	REM_Del_Ticket	price
BUY_Young, BUY_Success	REM_NO_MORE_TICKET	all_tickets_in_basket
BUY_Young, BUY_Success	REM_NO_MORE_TICKET	current_user
BUY_Free, BUY_OneFree, BUY_Success	BAL_ADD_SUCCESS	balance
BUY_Free, BUY_OneFree, BUY_Success	BAL_ADD_SUCCESS	current_user
BUY_Free, BUY_OneFree, BUY_Success	BAL_RETRIEVE_SUCCESS	balance
BUY_Free, BUY_OneFree, BUY_Success	BAL_RETRIEVE_SUCCESS	current_user
BUY_Free, BUY_OneFree, BUY_Success	BUY_Young, BUY_Success	all_sold_tickets
BUY_Free, BUY_OneFree, BUY_Success	BUY_Young, BUY_Success	all_tickets_in_basket
BUY_Free, BUY_OneFree, BUY_Success	BUY_Young, BUY_Success	available_tickets
BUY_Free, BUY_OneFree, BUY_Success	BUY_Young, BUY_Success	balance
BUY_Free, BUY_OneFree, BUY_Success	BUY_Young, BUY_Success	current_user
BUY_Free, BUY_OneFree, BUY_Success	BUY_Young, BUY_Success	price
BUY_Free, BUY_OneFree, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	all_sold_tickets
BUY_Free, BUY_OneFree, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	all_tickets_in_basket
BUY_Free, BUY_OneFree, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	available_tickets
BUY_Free, BUY_OneFree, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	balance
BUY_Free, BUY_OneFree, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	cpt
BUY_Free, BUY_OneFree, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	current_user
BUY_Free, BUY_OneFree, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	price
BUY_Free, BUY_OneFree, BUY_Success	BUY_Normal, BUY_Success	all_sold_tickets
BUY_Free, BUY_OneFree, BUY_Success	BUY_Normal, BUY_Success	all_tickets_in_basket
BUY_Free, BUY_OneFree, BUY_Success	BUY_Normal, BUY_Success	available_tickets
BUY_Free, BUY_OneFree, BUY_Success	BUY_Normal, BUY_Success	balance
BUY_Free, BUY_OneFree, BUY_Success	BUY_Normal, BUY_Success	current_user
BUY_Free, BUY_OneFree, BUY_Success	BUY_Normal, BUY_Success	price
BUY_Free, BUY_OneFree, BUY_Success	BUY_OneFree, BUY_Success	all_sold_tickets
BUY_Free, BUY_OneFree, BUY_Success	BUY_OneFree, BUY_Success	all_tickets_in_basket
BUY_Free, BUY_OneFree, BUY_Success	BUY_OneFree, BUY_Success	available_tickets
BUY_Free, BUY_OneFree, BUY_Success	BUY_OneFree, BUY_Success	balance
BUY_Free, BUY_OneFree, BUY_Success	BUY_OneFree, BUY_Success	cpt
BUY_Free, BUY_OneFree, BUY_Success	BUY_OneFree, BUY_Success	current_user
BUY_Free, BUY_OneFree, BUY_Success	BUY_OneFree, BUY_Success	price

Comportement de définition / AIM	**Comportement dépendant / AIM**	**Variable**
BUY_Free, BUY_OneFree, BUY_Success	BUY_Senior, BUY_Success	all_sold_tickets
BUY_Free, BUY_OneFree, BUY_Success	BUY_Senior, BUY_Success	all_tickets_in_basket
BUY_Free, BUY_OneFree, BUY_Success	BUY_Senior, BUY_Success	available_tickets
BUY_Free, BUY_OneFree, BUY_Success	BUY_Senior, BUY_Success	balance
BUY_Free, BUY_OneFree, BUY_Success	BUY_Senior, BUY_Success	current_user
BUY_Free, BUY_OneFree, BUY_Success	BUY_Senior, BUY_Success	price
BUY_Free, BUY_OneFree, BUY_Success	BUY_Student, BUY_Success	all_sold_tickets
BUY_Free, BUY_OneFree, BUY_Success	BUY_Student, BUY_Success	all_tickets_in_basket
BUY_Free, BUY_OneFree, BUY_Success	BUY_Student, BUY_Success	available_tickets
BUY_Free, BUY_OneFree, BUY_Success	BUY_Student, BUY_Success	balance
BUY_Free, BUY_OneFree, BUY_Success	BUY_Student, BUY_Success	current_user
BUY_Free, BUY_OneFree, BUY_Success	BUY_Student, BUY_Success	price
BUY_Free, BUY_OneFree, BUY_Success	REM_Del_All_Tickets	all_tickets_in_basket
BUY_Free, BUY_OneFree, BUY_Success	REM_Del_All_Tickets	available_tickets
BUY_Free, BUY_OneFree, BUY_Success	REM_Del_All_Tickets	balance
BUY_Free, BUY_OneFree, BUY_Success	REM_Del_All_Tickets	current_user
BUY_Free, BUY_OneFree, BUY_Success	REM_Del_All_Tickets	price
BUY_Free, BUY_OneFree, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	all_tickets_in_basket
BUY_Free, BUY_OneFree, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	available_tickets
BUY_Free, BUY_OneFree, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	balance
BUY_Free, BUY_OneFree, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	cpt
BUY_Free, BUY_OneFree, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	current_user
BUY_Free, BUY_OneFree, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	price
BUY_Free, BUY_OneFree, BUY_Success	REM_Del_Ticket, REM_One_Free	all_tickets_in_basket
BUY_Free, BUY_OneFree, BUY_Success	REM_Del_Ticket, REM_One_Free	available_tickets
BUY_Free, BUY_OneFree, BUY_Success	REM_Del_Ticket, REM_One_Free	balance
BUY_Free, BUY_OneFree, BUY_Success	REM_Del_Ticket, REM_One_Free	cpt
BUY_Free, BUY_OneFree, BUY_Success	REM_Del_Ticket, REM_One_Free	current_user
BUY_Free, BUY_OneFree, BUY_Success	REM_Del_Ticket, REM_One_Free	price
BUY_Free, BUY_OneFree, BUY_Success	REM_Del_Ticket	all_tickets_in_basket
BUY_Free, BUY_OneFree, BUY_Success	REM_Del_Ticket	available_tickets
BUY_Free, BUY_OneFree, BUY_Success	REM_Del_Ticket	balance
BUY_Free, BUY_OneFree, BUY_Success	REM_Del_Ticket	current_user
BUY_Free, BUY_OneFree, BUY_Success	REM_Del_Ticket	price
BUY_Free, BUY_OneFree, BUY_Success	REM_NO_MORE_TICKET	all_tickets_in_basket
BUY_Free, BUY_OneFree, BUY_Success	REM_NO_MORE_TICKET	current_user
BUY_Normal, BUY_Success	BAL_ADD_SUCCESS	balance
BUY_Normal, BUY_Success	BAL_ADD_SUCCESS	current_user
BUY_Normal, BUY_Success	BAL_RETRIEVE_SUCCESS	balance
BUY_Normal, BUY_Success	BAL_RETRIEVE_SUCCESS	current_user
BUY_Normal, BUY_Success	BUY_Young, BUY_Success	all_sold_tickets
BUY_Normal, BUY_Success	BUY_Young, BUY_Success	all_tickets_in_basket
BUY_Normal, BUY_Success	BUY_Young, BUY_Success	available_tickets
BUY_Normal, BUY_Success	BUY_Young, BUY_Success	balance
BUY_Normal, BUY_Success	BUY_Young, BUY_Success	current_user
BUY_Normal, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	all_sold_tickets
BUY_Normal, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	all_tickets_in_basket
BUY_Normal, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	available_tickets
BUY_Normal, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	balance
BUY_Normal, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	current_user
BUY_Normal, BUY_Success	BUY_Normal, BUY_Success	all_sold_tickets
BUY_Normal, BUY_Success	BUY_Normal, BUY_Success	all_tickets_in_basket
BUY_Normal, BUY_Success	BUY_Normal, BUY_Success	available_tickets
BUY_Normal, BUY_Success	BUY_Normal, BUY_Success	balance
BUY_Normal, BUY_Success	BUY_Normal, BUY_Success	current_user
BUY_Normal, BUY_Success	BUY_OneFree, BUY_Success	all_sold_tickets
BUY_Normal, BUY_Success	BUY_OneFree, BUY_Success	all_tickets_in_basket
BUY_Normal, BUY_Success	BUY_OneFree, BUY_Success	available_tickets

Comportement de définition / AIM	Comportement dépendant / AIM	Variable
BUY_Normal, BUY_Success	BUY_OneFree, BUY_Success	balance
BUY_Normal, BUY_Success	BUY_OneFree, BUY_Success	current_user
BUY_Normal, BUY_Success	BUY_Senior, BUY_Success	all_sold_tickets
BUY_Normal, BUY_Success	BUY_Senior, BUY_Success	all_tickets_in_basket
BUY_Normal, BUY_Success	BUY_Senior, BUY_Success	available_tickets
BUY_Normal, BUY_Success	BUY_Senior, BUY_Success	balance
BUY_Normal, BUY_Success	BUY_Senior, BUY_Success	current_user
BUY_Normal, BUY_Success	BUY_Student, BUY_Success	all_sold_tickets
BUY_Normal, BUY_Success	BUY_Student, BUY_Success	all_tickets_in_basket
BUY_Normal, BUY_Success	BUY_Student, BUY_Success	available_tickets
BUY_Normal, BUY_Success	BUY_Student, BUY_Success	balance
BUY_Normal, BUY_Success	BUY_Student, BUY_Success	current_user
BUY_Normal, BUY_Success	REG_Unregister	all_tickets_in_basket
BUY_Normal, BUY_Success	REG_Unregister	available_tickets
BUY_Normal, BUY_Success	REG_Unregister	current_user
BUY_Normal, BUY_Success	REM_Del_All_Tickets	all_tickets_in_basket
BUY_Normal, BUY_Success	REM_Del_All_Tickets	available_tickets
BUY_Normal, BUY_Success	REM_Del_All_Tickets	balance
BUY_Normal, BUY_Success	REM_Del_All_Tickets	current_user
BUY_Normal, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	all_tickets_in_basket
BUY_Normal, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	available_tickets
BUY_Normal, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	balance
BUY_Normal, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	current_user
BUY_Normal, BUY_Success	REM_Del_Ticket, REM_One_Free	all_tickets_in_basket
BUY_Normal, BUY_Success	REM_Del_Ticket, REM_One_Free	available_tickets
BUY_Normal, BUY_Success	REM_Del_Ticket, REM_One_Free	balance
BUY_Normal, BUY_Success	REM_Del_Ticket, REM_One_Free	current_user
BUY_Normal, BUY_Success	REM_Del_Ticket	all_tickets_in_basket
BUY_Normal, BUY_Success	REM_Del_Ticket	available_tickets
BUY_Normal, BUY_Success	REM_Del_Ticket	balance
BUY_Normal, BUY_Success	REM_Del_Ticket	current_user
BUY_Normal, BUY_Success	REM_NO_MORE_TICKET	all_tickets_in_basket
BUY_Normal, BUY_Success	REM_NO_MORE_TICKET	current_user
BUY_OneFree, BUY_Success	BAL_ADD_SUCCESS	balance
BUY_OneFree, BUY_Success	BAL_ADD_SUCCESS	current_user
BUY_OneFree, BUY_Success	BAL_RETRIEVE_SUCCESS	balance
BUY_OneFree, BUY_Success	BAL_RETRIEVE_SUCCESS	current_user
BUY_OneFree, BUY_Success	BUY_Young, BUY_Success	all_sold_tickets
BUY_OneFree, BUY_Success	BUY_Young, BUY_Success	all_tickets_in_basket
BUY_OneFree, BUY_Success	BUY_Young, BUY_Success	available_tickets
BUY_OneFree, BUY_Success	BUY_Young, BUY_Success	balance
BUY_OneFree, BUY_Success	BUY_Young, BUY_Success	current_user
BUY_OneFree, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	all_sold_tickets
BUY_OneFree, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	all_tickets_in_basket
BUY_OneFree, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	available_tickets
BUY_OneFree, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	balance
BUY_OneFree, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	cpt
BUY_OneFree, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	current_user
BUY_OneFree, BUY_Success	BUY_Normal, BUY_Success	all_sold_tickets
BUY_OneFree, BUY_Success	BUY_Normal, BUY_Success	all_tickets_in_basket
BUY_OneFree, BUY_Success	BUY_Normal, BUY_Success	available_tickets
BUY_OneFree, BUY_Success	BUY_Normal, BUY_Success	balance
BUY_OneFree, BUY_Success	BUY_Normal, BUY_Success	current_user
BUY_OneFree, BUY_Success	BUY_OneFree, BUY_Success	all_sold_tickets
BUY_OneFree, BUY_Success	BUY_OneFree, BUY_Success	all_tickets_in_basket
BUY_OneFree, BUY_Success	BUY_OneFree, BUY_Success	available_tickets
BUY_OneFree, BUY_Success	BUY_OneFree, BUY_Success	balance
BUY_OneFree, BUY_Success	BUY_OneFree, BUY_Success	cpt

Comportement de définition / AIM	**Comportement dépendant / AIM**	**Variable**
BUY_OneFree, BUY_Success	BUY_OneFree, BUY_Success	current_user
BUY_OneFree, BUY_Success	BUY_Senior, BUY_Success	all_sold_tickets
BUY_OneFree, BUY_Success	BUY_Senior, BUY_Success	all_tickets_in_basket
BUY_OneFree, BUY_Success	BUY_Senior, BUY_Success	available_tickets
BUY_OneFree, BUY_Success	BUY_Senior, BUY_Success	balance
BUY_OneFree, BUY_Success	BUY_Senior, BUY_Success	current_user
BUY_OneFree, BUY_Success	BUY_Student, BUY_Success	all_sold_tickets
BUY_OneFree, BUY_Success	BUY_Student, BUY_Success	all_tickets_in_basket
BUY_OneFree, BUY_Success	BUY_Student, BUY_Success	available_tickets
BUY_OneFree, BUY_Success	BUY_Student, BUY_Success	balance
BUY_OneFree, BUY_Success	BUY_Student, BUY_Success	current_user
BUY_OneFree, BUY_Success	REM_Del_All_Tickets	all_tickets_in_basket
BUY_OneFree, BUY_Success	REM_Del_All_Tickets	available_tickets
BUY_OneFree, BUY_Success	REM_Del_All_Tickets	balance
BUY_OneFree, BUY_Success	REM_Del_All_Tickets	current_user
BUY_OneFree, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	all_tickets_in_basket
BUY_OneFree, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	available_tickets
BUY_OneFree, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	balance
BUY_OneFree, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	cpt
BUY_OneFree, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	current_user
BUY_OneFree, BUY_Success	REM_Del_Ticket, REM_One_Free	all_tickets_in_basket
BUY_OneFree, BUY_Success	REM_Del_Ticket, REM_One_Free	available_tickets
BUY_OneFree, BUY_Success	REM_Del_Ticket, REM_One_Free	balance
BUY_OneFree, BUY_Success	REM_Del_Ticket, REM_One_Free	cpt
BUY_OneFree, BUY_Success	REM_Del_Ticket, REM_One_Free	current_user
BUY_OneFree, BUY_Success	REM_Del_Ticket	all_tickets_in_basket
BUY_OneFree, BUY_Success	REM_Del_Ticket	available_tickets
BUY_OneFree, BUY_Success	REM_Del_Ticket	balance
BUY_OneFree, BUY_Success	REM_Del_Ticket	current_user
BUY_OneFree, BUY_Success	REM_NO_MORE_TICKET	all_tickets_in_basket
BUY_OneFree, BUY_Success	REM_NO_MORE_TICKET	current_user
BUY_Senior, BUY_Success	BAL_ADD_SUCCESS	balance
BUY_Senior, BUY_Success	BAL_ADD_SUCCESS	current_user
BUY_Senior, BUY_Success	BAL_RETRIEVE_SUCCESS	balance
BUY_Senior, BUY_Success	BAL_RETRIEVE_SUCCESS	current_user
BUY_Senior, BUY_Success	BUY_Young, BUY_Success	all_sold_tickets
BUY_Senior, BUY_Success	BUY_Young, BUY_Success	all_tickets_in_basket
BUY_Senior, BUY_Success	BUY_Young, BUY_Success	available_tickets
BUY_Senior, BUY_Success	BUY_Young, BUY_Success	balance
BUY_Senior, BUY_Success	BUY_Young, BUY_Success	current_user
BUY_Senior, BUY_Success	BUY_Young, BUY_Success	price
BUY_Senior, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	all_sold_tickets
BUY_Senior, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	all_tickets_in_basket
BUY_Senior, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	available_tickets
BUY_Senior, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	balance
BUY_Senior, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	current_user
BUY_Senior, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	price
BUY_Senior, BUY_Success	BUY_Normal, BUY_Success	all_sold_tickets
BUY_Senior, BUY_Success	BUY_Normal, BUY_Success	all_tickets_in_basket
BUY_Senior, BUY_Success	BUY_Normal, BUY_Success	available_tickets
BUY_Senior, BUY_Success	BUY_Normal, BUY_Success	balance
BUY_Senior, BUY_Success	BUY_Normal, BUY_Success	current_user
BUY_Senior, BUY_Success	BUY_Normal, BUY_Success	price
BUY_Senior, BUY_Success	BUY_OneFree, BUY_Success	all_sold_tickets
BUY_Senior, BUY_Success	BUY_OneFree, BUY_Success	all_tickets_in_basket
BUY_Senior, BUY_Success	BUY_OneFree, BUY_Success	available_tickets
BUY_Senior, BUY_Success	BUY_OneFree, BUY_Success	balance
BUY_Senior, BUY_Success	BUY_OneFree, BUY_Success	current_user

Comportement de définition / AIM	Comportement dépendant / AIM	Variable
BUY_Senior, BUY_Success	BUY_OneFree, BUY_Success	price
BUY_Senior, BUY_Success	BUY_Senior, BUY_Success	all_sold_tickets
BUY_Senior, BUY_Success	BUY_Senior, BUY_Success	all_tickets_in_basket
BUY_Senior, BUY_Success	BUY_Senior, BUY_Success	available_tickets
BUY_Senior, BUY_Success	BUY_Senior, BUY_Success	balance
BUY_Senior, BUY_Success	BUY_Senior, BUY_Success	current_user
BUY_Senior, BUY_Success	BUY_Senior, BUY_Success	price
BUY_Senior, BUY_Success	BUY_Student, BUY_Success	all_sold_tickets
BUY_Senior, BUY_Success	BUY_Student, BUY_Success	all_tickets_in_basket
BUY_Senior, BUY_Success	BUY_Student, BUY_Success	available_tickets
BUY_Senior, BUY_Success	BUY_Student, BUY_Success	balance
BUY_Senior, BUY_Success	BUY_Student, BUY_Success	current_user
BUY_Senior, BUY_Success	BUY_Student, BUY_Success	price
BUY_Senior, BUY_Success	REM_Del_All_Tickets	all_tickets_in_basket
BUY_Senior, BUY_Success	REM_Del_All_Tickets	available_tickets
BUY_Senior, BUY_Success	REM_Del_All_Tickets	balance
BUY_Senior, BUY_Success	REM_Del_All_Tickets	current_user
BUY_Senior, BUY_Success	REM_Del_All_Tickets	price
BUY_Senior, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	all_tickets_in_basket
BUY_Senior, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	available_tickets
BUY_Senior, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	balance
BUY_Senior, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	current_user
BUY_Senior, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	price
BUY_Senior, BUY_Success	REM_Del_Ticket, REM_One_Free	all_tickets_in_basket
BUY_Senior, BUY_Success	REM_Del_Ticket, REM_One_Free	available_tickets
BUY_Senior, BUY_Success	REM_Del_Ticket, REM_One_Free	balance
BUY_Senior, BUY_Success	REM_Del_Ticket, REM_One_Free	current_user
BUY_Senior, BUY_Success	REM_Del_Ticket, REM_One_Free	price
BUY_Senior, BUY_Success	REM_Del_Ticket	all_tickets_in_basket
BUY_Senior, BUY_Success	REM_Del_Ticket	available_tickets
BUY_Senior, BUY_Success	REM_Del_Ticket	balance
BUY_Senior, BUY_Success	REM_Del_Ticket	current_user
BUY_Senior, BUY_Success	REM_Del_Ticket	price
BUY_Senior, BUY_Success	REM_NO_MORE_TICKET	all_tickets_in_basket
BUY_Senior, BUY_Success	REM_NO_MORE_TICKET	current_user
BUY_Student, BUY_Success	BAL_ADD_SUCCESS	balance
BUY_Student, BUY_Success	BAL_ADD_SUCCESS	current_user
BUY_Student, BUY_Success	BAL_RETRIEVE_SUCCESS	balance
BUY_Student, BUY_Success	BAL_RETRIEVE_SUCCESS	current_user
BUY_Student, BUY_Success	BUY_Young, BUY_Success	all_sold_tickets
BUY_Student, BUY_Success	BUY_Young, BUY_Success	all_tickets_in_basket
BUY_Student, BUY_Success	BUY_Young, BUY_Success	available_tickets
BUY_Student, BUY_Success	BUY_Young, BUY_Success	balance
BUY_Student, BUY_Success	BUY_Young, BUY_Success	current_user
BUY_Student, BUY_Success	BUY_Young, BUY_Success	price
BUY_Student, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	all_sold_tickets
BUY_Student, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	all_tickets_in_basket
BUY_Student, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	available_tickets
BUY_Student, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	balance
BUY_Student, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	current_user
BUY_Student, BUY_Success	BUY_Free, BUY_OneFree, BUY_Success	price
BUY_Student, BUY_Success	BUY_Normal, BUY_Success	all_sold_tickets
BUY_Student, BUY_Success	BUY_Normal, BUY_Success	all_tickets_in_basket
BUY_Student, BUY_Success	BUY_Normal, BUY_Success	available_tickets
BUY_Student, BUY_Success	BUY_Normal, BUY_Success	balance
BUY_Student, BUY_Success	BUY_Normal, BUY_Success	current_user
BUY_Student, BUY_Success	BUY_Normal, BUY_Success	price
BUY_Student, BUY_Success	BUY_OneFree, BUY_Success	all_sold_tickets

Comportement de définition / AIM	**Comportement dépendant / AIM**	**Variable**
BUY_Student, BUY_Success	BUY_OneFree, BUY_Success	all_tickets_in_basket
BUY_Student, BUY_Success	BUY_OneFree, BUY_Success	available_tickets
BUY_Student, BUY_Success	BUY_OneFree, BUY_Success	balance
BUY_Student, BUY_Success	BUY_OneFree, BUY_Success	current_user
BUY_Student, BUY_Success	BUY_OneFree, BUY_Success	price
BUY_Student, BUY_Success	BUY_Senior, BUY_Success	all_sold_tickets
BUY_Student, BUY_Success	BUY_Senior, BUY_Success	all_tickets_in_basket
BUY_Student, BUY_Success	BUY_Senior, BUY_Success	available_tickets
BUY_Student, BUY_Success	BUY_Senior, BUY_Success	balance
BUY_Student, BUY_Success	BUY_Senior, BUY_Success	current_user
BUY_Student, BUY_Success	BUY_Senior, BUY_Success	price
BUY_Student, BUY_Success	BUY_Student, BUY_Success	all_sold_tickets
BUY_Student, BUY_Success	BUY_Student, BUY_Success	all_tickets_in_basket
BUY_Student, BUY_Success	BUY_Student, BUY_Success	available_tickets
BUY_Student, BUY_Success	BUY_Student, BUY_Success	balance
BUY_Student, BUY_Success	BUY_Student, BUY_Success	current_user
BUY_Student, BUY_Success	BUY_Student, BUY_Success	price
BUY_Student, BUY_Success	REM_Del_All_Tickets	all_tickets_in_basket
BUY_Student, BUY_Success	REM_Del_All_Tickets	available_tickets
BUY_Student, BUY_Success	REM_Del_All_Tickets	balance
BUY_Student, BUY_Success	REM_Del_All_Tickets	current_user
BUY_Student, BUY_Success	REM_Del_All_Tickets	price
BUY_Student, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	all_tickets_in_basket
BUY_Student, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	available_tickets
BUY_Student, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	balance
BUY_Student, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	current_user
BUY_Student, BUY_Success	REM_Del_Ticket, REM_Free, REM_One_Free	price
BUY_Student, BUY_Success	REM_Del_Ticket, REM_One_Free	all_tickets_in_basket
BUY_Student, BUY_Success	REM_Del_Ticket, REM_One_Free	available_tickets
BUY_Student, BUY_Success	REM_Del_Ticket, REM_One_Free	balance
BUY_Student, BUY_Success	REM_Del_Ticket, REM_One_Free	current_user
BUY_Student, BUY_Success	REM_Del_Ticket, REM_One_Free	price
BUY_Student, BUY_Success	REM_Del_Ticket	all_tickets_in_basket
BUY_Student, BUY_Success	REM_Del_Ticket	available_tickets
BUY_Student, BUY_Success	REM_Del_Ticket	balance
BUY_Student, BUY_Success	REM_Del_Ticket	current_user
BUY_Student, BUY_Success	REM_Del_Ticket	price
BUY_Student, BUY_Success	REM_NO_MORE_TICKET	all_tickets_in_basket
BUY_Student, BUY_Success	REM_NO_MORE_TICKET	current_user
LOG_Logout	BAL_ADD_SUCCESS	current_user
LOG_Logout	BAL_RETRIEVE_SUCCESS	current_user
LOG_Logout	REM_Del_All_Tickets	current_user
LOG_Logout	REM_Del_Ticket, REM_Free, REM_One_Free	current_user
LOG_Logout	REM_Del_Ticket, REM_One_Free	current_user
LOG_Logout	REM_Del_Ticket	current_user
LOG_Logout	REM_NO_MORE_TICKET	current_user
LOG_Success	BAL_ADD_SUCCESS	current_user
LOG_Success	BAL_RETRIEVE_SUCCESS	current_user
LOG_Success	BUY_Young, BUY_Success	current_user
LOG_Success	BUY_Free, BUY_OneFree, BUY_Success	current_user
LOG_Success	BUY_Normal, BUY_Success	current_user
LOG_Success	BUY_OneFree, BUY_Success	current_user
LOG_Success	BUY_Senior, BUY_Success	current_user
LOG_Success	BUY_Student, BUY_Success	current_user
LOG_Success	REG_Unregister	current_user
LOG_Success	REM_Del_All_Tickets	current_user
LOG_Success	REM_Del_Ticket, REM_Free, REM_One_Free	current_user
LOG_Success	REM_Del_Ticket, REM_One_Free	current_user

Comportement de définition / AIM	Comportement dépendant / AIM	Variable
LOG_Success	REM_Del_Ticket	current_user
LOG_Success	REM_NO_MORE_TICKET	current_user
REG_Success	BUY_Young, BUY_Success	current_user
REG_Success	BUY_Free, BUY_OneFree, BUY_Success	current_user
REG_Success	BUY_Normal, BUY_Success	current_user
REG_Success	BUY_OneFree, BUY_Success	current_user
REG_Success	BUY_Senior, BUY_Success	current_user
REG_Success	BUY_Student, BUY_Success	current_user
REG_Success	REG_Unregister	all_registered_users
REG_Success	REG_Unregister	current_user
REG_Unregister	BAL_ADD_SUCCESS	current_user
REG_Unregister	BAL_RETRIEVE_SUCCESS	current_user
REG_Unregister	BUY_Young, BUY_Success	all_tickets_in_basket
REG_Unregister	BUY_Young, BUY_Success	available_tickets
REG_Unregister	BUY_Young, BUY_Success	current_user
REG_Unregister	BUY_Normal, BUY_Success	all_tickets_in_basket
REG_Unregister	BUY_Normal, BUY_Success	available_tickets
REG_Unregister	BUY_Normal, BUY_Success	current_user
REG_Unregister	LOG_Invalid_Password	all_registered_users
REG_Unregister	LOG_Success	all_registered_users
REG_Unregister	REG_Success	all_registered_users
REG_Unregister	REM_Del_All_Tickets	all_tickets_in_basket
REG_Unregister	REM_Del_All_Tickets	available_tickets
REG_Unregister	REM_Del_All_Tickets	current_user
REG_Unregister	REM_Del_Ticket, REM_Free, REM_One_Free	all_tickets_in_basket
REG_Unregister	REM_Del_Ticket, REM_Free, REM_One_Free	available_tickets
REG_Unregister	REM_Del_Ticket, REM_Free, REM_One_Free	current_user
REG_Unregister	REM_Del_Ticket, REM_One_Free	all_tickets_in_basket
REG_Unregister	REM_Del_Ticket, REM_One_Free	available_tickets
REG_Unregister	REM_Del_Ticket, REM_One_Free	current_user
REG_Unregister	REM_Del_Ticket	all_tickets_in_basket
REG_Unregister	REM_Del_Ticket	available_tickets
REG_Unregister	REM_Del_Ticket	current_user
REG_Unregister	REM_NO_MORE_TICKET	all_tickets_in_basket
REG_Unregister	REM_NO_MORE_TICKET	current_user
REM_Del_All_Tickets	BAL_ADD_SUCCESS	balance
REM_Del_All_Tickets	BAL_RETRIEVE_SUCCESS	balance
REM_Del_All_Tickets	BUY_Young, BUY_Success	available_tickets
REM_Del_All_Tickets	BUY_Young, BUY_Success	balance
REM_Del_All_Tickets	BUY_Young, BUY_Success	owner_ticket
REM_Del_All_Tickets	BUY_Young, BUY_Success	price
REM_Del_All_Tickets	BUY_Free, BUY_OneFree, BUY_Success	available_tickets
REM_Del_All_Tickets	BUY_Free, BUY_OneFree, BUY_Success	balance
REM_Del_All_Tickets	BUY_Free, BUY_OneFree, BUY_Success	cpt
REM_Del_All_Tickets	BUY_Free, BUY_OneFree, BUY_Success	owner_ticket
REM_Del_All_Tickets	BUY_Free, BUY_OneFree, BUY_Success	price
REM_Del_All_Tickets	BUY_No_More_Money	owner_ticket
REM_Del_All_Tickets	BUY_Normal, BUY_Success	available_tickets
REM_Del_All_Tickets	BUY_Normal, BUY_Success	balance
REM_Del_All_Tickets	BUY_Normal, BUY_Success	owner_ticket
REM_Del_All_Tickets	BUY_Normal, BUY_Success	price
REM_Del_All_Tickets	BUY_OneFree, BUY_Success	available_tickets
REM_Del_All_Tickets	BUY_OneFree, BUY_Success	balance
REM_Del_All_Tickets	BUY_OneFree, BUY_Success	cpt
REM_Del_All_Tickets	BUY_OneFree, BUY_Success	owner_ticket
REM_Del_All_Tickets	BUY_OneFree, BUY_Success	price
REM_Del_All_Tickets	BUY_Senior, BUY_Success	available_tickets
REM_Del_All_Tickets	BUY_Senior, BUY_Success	balance

Comportement de définition / AIM	**Comportement dépendant / AIM**	**Variable**
REM_Del_All_Tickets	BUY_Senior, BUY_Success	owner_ticket
REM_Del_All_Tickets	BUY_Senior, BUY_Success	price
REM_Del_All_Tickets	BUY_Student, BUY_Success	available_tickets
REM_Del_All_Tickets	BUY_Student, BUY_Success	balance
REM_Del_All_Tickets	BUY_Student, BUY_Success	owner_ticket
REM_Del_All_Tickets	BUY_Student, BUY_Success	price
REM_Del_All_Tickets	REG_Unregister	available_tickets
REM_Del_All_Tickets	REM_Del_All_Tickets	available_tickets
REM_Del_All_Tickets	REM_Del_All_Tickets	balance
REM_Del_All_Tickets	REM_Del_All_Tickets	movie
REM_Del_All_Tickets	REM_Del_All_Tickets	owner_ticket
REM_Del_All_Tickets	REM_Del_All_Tickets	price
REM_Del_All_Tickets	REM_Del_Ticket, REM_Free, REM_One_Free	available_tickets
REM_Del_All_Tickets	REM_Del_Ticket, REM_Free, REM_One_Free	balance
REM_Del_All_Tickets	REM_Del_Ticket, REM_Free, REM_One_Free	cpt
REM_Del_All_Tickets	REM_Del_Ticket, REM_Free, REM_One_Free	movie
REM_Del_All_Tickets	REM_Del_Ticket, REM_Free, REM_One_Free	owner_ticket
REM_Del_All_Tickets	REM_Del_Ticket, REM_Free, REM_One_Free	price
REM_Del_All_Tickets	REM_Del_Ticket, REM_One_Free	available_tickets
REM_Del_All_Tickets	REM_Del_Ticket, REM_One_Free	balance
REM_Del_All_Tickets	REM_Del_Ticket, REM_One_Free	cpt
REM_Del_All_Tickets	REM_Del_Ticket, REM_One_Free	movie
REM_Del_All_Tickets	REM_Del_Ticket, REM_One_Free	owner_ticket
REM_Del_All_Tickets	REM_Del_Ticket, REM_One_Free	price
REM_Del_All_Tickets	REM_Del_Ticket	available_tickets
REM_Del_All_Tickets	REM_Del_Ticket	balance
REM_Del_All_Tickets	REM_Del_Ticket	movie
REM_Del_All_Tickets	REM_Del_Ticket	owner_ticket
REM_Del_All_Tickets	REM_Del_Ticket	price
REM_Del_All_Tickets	REM_NO_MORE_TICKET	movie
REM_Del_Ticket, REM_Free, REM_One_Free	BAL_ADD_SUCCESS	balance
REM_Del_Ticket, REM_Free, REM_One_Free	BAL_RETRIEVE_SUCCESS	balance
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_Young, BUY_Success	available_tickets
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_Young, BUY_Success	balance
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_Young, BUY_Success	owner_ticket
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_Young, BUY_Success	price
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_Free, BUY_OneFree, BUY_Success	available_tickets
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_Free, BUY_OneFree, BUY_Success	balance
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_Free, BUY_OneFree, BUY_Success	cpt
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_Free, BUY_OneFree, BUY_Success	owner_ticket
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_Free, BUY_OneFree, BUY_Success	price
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_No_More_Money	owner_ticket
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_Normal, BUY_Success	available_tickets
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_Normal, BUY_Success	balance
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_Normal, BUY_Success	owner_ticket
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_Normal, BUY_Success	price
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_OneFree, BUY_Success	available_tickets
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_OneFree, BUY_Success	balance
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_OneFree, BUY_Success	cpt
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_OneFree, BUY_Success	owner_ticket
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_OneFree, BUY_Success	price
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_Senior, BUY_Success	available_tickets
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_Senior, BUY_Success	balance
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_Senior, BUY_Success	owner_ticket
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_Senior, BUY_Success	price
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_Student, BUY_Success	available_tickets
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_Student, BUY_Success	balance
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_Student, BUY_Success	owner_ticket

Comportement de définition / AIM	Comportement dépendant / AIM	Variable
REM_Del_Ticket, REM_Free, REM_One_Free	BUY_Student, BUY_Success	price
REM_Del_Ticket, REM_Free, REM_One_Free	REG_Unregister	available_tickets
REM_Del_Ticket, REM_Free, REM_One_Free	REM_Del_All_Tickets	available_tickets
REM_Del_Ticket, REM_Free, REM_One_Free	REM_Del_All_Tickets	balance
REM_Del_Ticket, REM_Free, REM_One_Free	REM_Del_All_Tickets	movie
REM_Del_Ticket, REM_Free, REM_One_Free	REM_Del_All_Tickets	owner_ticket
REM_Del_Ticket, REM_Free, REM_One_Free	REM_Del_All_Tickets	price
REM_Del_Ticket, REM_Free, REM_One_Free	REM_Del_Ticket, REM_Free, REM_One_Free	available_tickets
REM_Del_Ticket, REM_Free, REM_One_Free	REM_Del_Ticket, REM_Free, REM_One_Free	balance
REM_Del_Ticket, REM_Free, REM_One_Free	REM_Del_Ticket, REM_Free, REM_One_Free	cpt
REM_Del_Ticket, REM_Free, REM_One_Free	REM_Del_Ticket, REM_Free, REM_One_Free	movie
REM_Del_Ticket, REM_Free, REM_One_Free	REM_Del_Ticket, REM_Free, REM_One_Free	owner_ticket
REM_Del_Ticket, REM_Free, REM_One_Free	REM_Del_Ticket, REM_Free, REM_One_Free	price
REM_Del_Ticket, REM_Free, REM_One_Free	REM_Del_Ticket, REM_One_Free	available_tickets
REM_Del_Ticket, REM_Free, REM_One_Free	REM_Del_Ticket, REM_One_Free	balance
REM_Del_Ticket, REM_Free, REM_One_Free	REM_Del_Ticket, REM_One_Free	cpt
REM_Del_Ticket, REM_Free, REM_One_Free	REM_Del_Ticket, REM_One_Free	movie
REM_Del_Ticket, REM_Free, REM_One_Free	REM_Del_Ticket, REM_One_Free	owner_ticket
REM_Del_Ticket, REM_Free, REM_One_Free	REM_Del_Ticket, REM_One_Free	price
REM_Del_Ticket, REM_Free, REM_One_Free	REM_Del_Ticket	available_tickets
REM_Del_Ticket, REM_Free, REM_One_Free	REM_Del_Ticket	balance
REM_Del_Ticket, REM_Free, REM_One_Free	REM_Del_Ticket	movie
REM_Del_Ticket, REM_Free, REM_One_Free	REM_Del_Ticket	owner_ticket
REM_Del_Ticket, REM_Free, REM_One_Free	REM_Del_Ticket	price
REM_Del_Ticket, REM_Free, REM_One_Free	REM_NO_MORE_TICKET	movie
REM_Del_Ticket, REM_One_Free	BAL_ADD_SUCCESS	balance
REM_Del_Ticket, REM_One_Free	BAL_RETRIEVE_SUCCESS	balance
REM_Del_Ticket, REM_One_Free	BUY_Young, BUY_Success	available_tickets
REM_Del_Ticket, REM_One_Free	BUY_Young, BUY_Success	balance
REM_Del_Ticket, REM_One_Free	BUY_Young, BUY_Success	owner_ticket
REM_Del_Ticket, REM_One_Free	BUY_Young, BUY_Success	price
REM_Del_Ticket, REM_One_Free	BUY_Free, BUY_OneFree, BUY_Success	available_tickets
REM_Del_Ticket, REM_One_Free	BUY_Free, BUY_OneFree, BUY_Success	balance
REM_Del_Ticket, REM_One_Free	BUY_Free, BUY_OneFree, BUY_Success	cpt
REM_Del_Ticket, REM_One_Free	BUY_Free, BUY_OneFree, BUY_Success	owner_ticket
REM_Del_Ticket, REM_One_Free	BUY_Free, BUY_OneFree, BUY_Success	price
REM_Del_Ticket, REM_One_Free	BUY_No_More_Money	owner_ticket
REM_Del_Ticket, REM_One_Free	BUY_Normal, BUY_Success	available_tickets
REM_Del_Ticket, REM_One_Free	BUY_Normal, BUY_Success	balance
REM_Del_Ticket, REM_One_Free	BUY_Normal, BUY_Success	owner_ticket
REM_Del_Ticket, REM_One_Free	BUY_Normal, BUY_Success	price
REM_Del_Ticket, REM_One_Free	BUY_OneFree, BUY_Success	available_tickets
REM_Del_Ticket, REM_One_Free	BUY_OneFree, BUY_Success	balance
REM_Del_Ticket, REM_One_Free	BUY_OneFree, BUY_Success	cpt
REM_Del_Ticket, REM_One_Free	BUY_OneFree, BUY_Success	owner_ticket
REM_Del_Ticket, REM_One_Free	BUY_OneFree, BUY_Success	price
REM_Del_Ticket, REM_One_Free	BUY_Senior, BUY_Success	available_tickets
REM_Del_Ticket, REM_One_Free	BUY_Senior, BUY_Success	balance
REM_Del_Ticket, REM_One_Free	BUY_Senior, BUY_Success	owner_ticket
REM_Del_Ticket, REM_One_Free	BUY_Senior, BUY_Success	price
REM_Del_Ticket, REM_One_Free	BUY_Student, BUY_Success	available_tickets
REM_Del_Ticket, REM_One_Free	BUY_Student, BUY_Success	balance
REM_Del_Ticket, REM_One_Free	BUY_Student, BUY_Success	owner_ticket
REM_Del_Ticket, REM_One_Free	BUY_Student, BUY_Success	price
REM_Del_Ticket, REM_One_Free	REG_Unregister	available_tickets
REM_Del_Ticket, REM_One_Free	REM_Del_All_Tickets	available_tickets
REM_Del_Ticket, REM_One_Free	REM_Del_All_Tickets	balance
REM_Del_Ticket, REM_One_Free	REM_Del_All_Tickets	movie

Comportement de définition / AIM	**Comportement dépendant / AIM**	**Variable**
REM_Del_Ticket, REM_One_Free	REM_Del_All_Tickets	owner_ticket
REM_Del_Ticket, REM_One_Free	REM_Del_All_Tickets	price
REM_Del_Ticket, REM_One_Free	REM_Del_Ticket, REM_Free, REM_One_Free	available_tickets
REM_Del_Ticket, REM_One_Free	REM_Del_Ticket, REM_Free, REM_One_Free	balance
REM_Del_Ticket, REM_One_Free	REM_Del_Ticket, REM_Free, REM_One_Free	cpt
REM_Del_Ticket, REM_One_Free	REM_Del_Ticket, REM_Free, REM_One_Free	movie
REM_Del_Ticket, REM_One_Free	REM_Del_Ticket, REM_Free, REM_One_Free	owner_ticket
REM_Del_Ticket, REM_One_Free	REM_Del_Ticket, REM_Free, REM_One_Free	price
REM_Del_Ticket, REM_One_Free	REM_Del_Ticket, REM_One_Free	available_tickets
REM_Del_Ticket, REM_One_Free	REM_Del_Ticket, REM_One_Free	balance
REM_Del_Ticket, REM_One_Free	REM_Del_Ticket, REM_One_Free	cpt
REM_Del_Ticket, REM_One_Free	REM_Del_Ticket, REM_One_Free	movie
REM_Del_Ticket, REM_One_Free	REM_Del_Ticket, REM_One_Free	owner_ticket
REM_Del_Ticket, REM_One_Free	REM_Del_Ticket, REM_One_Free	price
REM_Del_Ticket, REM_One_Free	REM_Del_Ticket	available_tickets
REM_Del_Ticket, REM_One_Free	REM_Del_Ticket	balance
REM_Del_Ticket, REM_One_Free	REM_Del_Ticket	movie
REM_Del_Ticket, REM_One_Free	REM_Del_Ticket	owner_ticket
REM_Del_Ticket, REM_One_Free	REM_Del_Ticket	price
REM_Del_Ticket, REM_One_Free	REM_NO_MORE_TICKET	movie
REM_Del_Ticket	BAL_ADD_SUCCESS	balance
REM_Del_Ticket	BAL_RETRIEVE_SUCCESS	balance
REM_Del_Ticket	BUY_Young, BUY_Success	available_tickets
REM_Del_Ticket	BUY_Young, BUY_Success	balance
REM_Del_Ticket	BUY_Young, BUY_Success	owner_ticket
REM_Del_Ticket	BUY_Young, BUY_Success	price
REM_Del_Ticket	BUY_Free, BUY_OneFree, BUY_Success	available_tickets
REM_Del_Ticket	BUY_Free, BUY_OneFree, BUY_Success	balance
REM_Del_Ticket	BUY_Free, BUY_OneFree, BUY_Success	owner_ticket
REM_Del_Ticket	BUY_Free, BUY_OneFree, BUY_Success	price
REM_Del_Ticket	BUY_No_More_Money	owner_ticket
REM_Del_Ticket	BUY_Normal, BUY_Success	available_tickets
REM_Del_Ticket	BUY_Normal, BUY_Success	balance
REM_Del_Ticket	BUY_Normal, BUY_Success	owner_ticket
REM_Del_Ticket	BUY_Normal, BUY_Success	price
REM_Del_Ticket	BUY_OneFree, BUY_Success	available_tickets
REM_Del_Ticket	BUY_OneFree, BUY_Success	balance
REM_Del_Ticket	BUY_OneFree, BUY_Success	owner_ticket
REM_Del_Ticket	BUY_OneFree, BUY_Success	price
REM_Del_Ticket	BUY_Senior, BUY_Success	available_tickets
REM_Del_Ticket	BUY_Senior, BUY_Success	balance
REM_Del_Ticket	BUY_Senior, BUY_Success	owner_ticket
REM_Del_Ticket	BUY_Senior, BUY_Success	price
REM_Del_Ticket	BUY_Student, BUY_Success	available_tickets
REM_Del_Ticket	BUY_Student, BUY_Success	balance
REM_Del_Ticket	BUY_Student, BUY_Success	owner_ticket
REM_Del_Ticket	BUY_Student, BUY_Success	price
REM_Del_Ticket	REG_Unregister	available_tickets
REM_Del_Ticket	REM_Del_All_Tickets	available_tickets
REM_Del_Ticket	REM_Del_All_Tickets	balance
REM_Del_Ticket	REM_Del_All_Tickets	movie
REM_Del_Ticket	REM_Del_All_Tickets	owner_ticket
REM_Del_Ticket	REM_Del_All_Tickets	price
REM_Del_Ticket	REM_Del_Ticket, REM_Free, REM_One_Free	available_tickets
REM_Del_Ticket	REM_Del_Ticket, REM_Free, REM_One_Free	balance
REM_Del_Ticket	REM_Del_Ticket, REM_Free, REM_One_Free	movie
REM_Del_Ticket	REM_Del_Ticket, REM_Free, REM_One_Free	owner_ticket
REM_Del_Ticket	REM_Del_Ticket, REM_Free, REM_One_Free	price

Comportement de définition / AIM	**Comportement dépendant / AIM**	**Variable**
REM_Del_Ticket	REM_Del_Ticket, REM_One_Free	available_tickets
REM_Del_Ticket	REM_Del_Ticket, REM_One_Free	balance
REM_Del_Ticket	REM_Del_Ticket, REM_One_Free	movie
REM_Del_Ticket	REM_Del_Ticket, REM_One_Free	owner_ticket
REM_Del_Ticket	REM_Del_Ticket, REM_One_Free	price
REM_Del_Ticket	REM_Del_Ticket	available_tickets
REM_Del_Ticket	REM_Del_Ticket	balance
REM_Del_Ticket	REM_Del_Ticket	movie
REM_Del_Ticket	REM_Del_Ticket	owner_ticket
REM_Del_Ticket	REM_Del_Ticket	price
REM_Del_Ticket	REM_NO_MORE_TICKET	movie
SUBS_Set_Up	BAL_ADD_SUCCESS	current_user
SUBS_Set_Up	BAL_RETRIEVE_SUCCESS	current_user
SUBS_Set_Up	BUY_Young, BUY_Success	current_user
SUBS_Set_Up	BUY_Free, BUY_OneFree, BUY_Success	current_user
SUBS_Set_Up	BUY_Normal, BUY_Success	current_user
SUBS_Set_Up	BUY_OneFree, BUY_Success	current_user
SUBS_Set_Up	BUY_Senior, BUY_Success	current_user
SUBS_Set_Up	BUY_Student, BUY_Success	current_user
SUBS_Set_Up	REG_Unregister	current_user
SUBS_Set_Up	REM_Del_All_Tickets	current_user
SUBS_Set_Up	REM_Del_Ticket, REM_Free, REM_One_Free	current_user
SUBS_Set_Up	REM_Del_Ticket, REM_One_Free	current_user
SUBS_Set_Up	REM_Del_Ticket	current_user
SUBS_Set_Up	REM_NO_MORE_TICKET	current_user

Table B.5: Dépendances comportementales pour eCinema -évolution

Annexe C

Catégorisation des tests pour les modèles d'eCinema

Nous donnons dans cette annexe les tests générés pour la première version d'eCinema, leur catégorisation et les tests *new* pour la version évoluée. Ceci est détaillé pour les types de diagrammes avec et sans diagrammes d'états/transitions, notés respectivement *Modèle 1* et *Modèle 2*. Le tableau C.1 donne les tests de sécurité et le tableau C.2 les tests fonctionnels.

Modèle 1		**Modèle 2**	
Test	Statut	Test	Statut
requirement11(bb-25-c2)	outdated	requirement11(bb-7c-93)	outdated
requirement11(bb-d3-b1)	outdated	requirement12(bb-fd-f9)	failed
requirement15(bb-91-ae)	failed	requirement15(bb-8f-fe)	failed
requirement28(bb-0c-3d)	outdated	requirement28(bb-f1-e3)	outdated
requirement210(bb-ca-6c)	outdated	requirement210(bb-72-5c)	outdated
requirement29(bb-c0-2c)	outdated	requirement29(bb-5c-3f)	outdated
requirement211(bb-24-b0)	outdated	requirement211(bb-be-43)	outdated
requirement12(bb-24-b0)	new	requirement12(bb-1e-8u)	new
requirement13(bb-14-b1)	new	requirement13(bb-5e-po)	new
requirement14(bb-04-e0)	new	requirement14(bb-b1-5p)	new
requirement16(bb-34-b0)	new	requirement16(bb-87-tt)	new
requirement17(bb-21-c0)	new	requirement17(bb-25-4n)	new
requirement11(bb-25-6a)	adapted	requirement11(bb-7c-b3)	adapted
requirement11(bb-d3-b1)	adapted	requirement12(bb-fd-f9)	adapted
requirement15(bb-91-ae)	adapted	requirement15(bb-8f-fe)	adapted
requirement28(bb-0c-3d)	adapted	requirement28(bb-f1-e3)	adapted
requirement210(bb-ca-6c)	adapted	requirement210(bb-72-5c)	adapted
requirement29(bb-c0-2c)	adapted	requirement29(bb-5c-3f)	adapted
requirement211(bb-24-b0)	adapted	requirement211(bb-be-43)	adapted

TABLE C.1 – Résultats de SeTGaM pour le test de sécurité - eCinema

Modèle 1		**Modèle 2**	
Test	Statut	Test	Statut
buyTicket (f2-e8-37)	outdated	buyTicket (f2-13-11)	outdated
buyTicket (f2-f7-37)	unimpacted	buyTicket (f2-12-41)	unimpacted
closeApplication (f2-18-1c)	unimpacted	closeApplication (f2-96-40)	reexecuted
deleteAllTickets (f2-e0-93)	outdated	deleteAllTickets (f2-9c-e5)	updated
deleteOneTicket (f2-e5-ca)	outdated	deleteTicket (f2-17-a6)	outdated
goToHome (f2-73-71)	reexecuted	goToHome (f2-4c-3a)	unimpacted
goToHome (f2-e7-b9)	reexecuted	goToHome (f2-0c-87)	reexecuted
login (f2-2e-11)	unimpacted	login (f2-38-9b)	unimpacted
login (f2-7f-d7)	unimpacted	login (f2-9b-c1)	reexecuted
login (f2-c8-d9)	unimpacted	login (f2-bf-10)	unimpacted
logout (f2-9b-b5)	unimpacted	logout (f2-41-3d) (f2-41-94)	reexecuted
registration (f2-1e-fd)	failed	registration (f2-2c-ef)	failed
registration (f2-74-d8)	failed	registration (f2-d2-5d)	failed
registration (f2-a6-5a)	failed	registration (f2-f0-bf)	failed
registration (f2-e2-b1)	failed	registration (f2-f2-a7)	failed
unregister (f2-06-46)	unimpacted	unregister (f2-87-d6)	reexecuted
registration-a (f2-1e-fd)	adapted	registration-a(f2-2c-ef)	adapted
registration-a (f2-a6-5a)	adapted	registration-a (f2-f0-bf)	adapted
registration-a (f2-e2-b1)	adapted	registration-a (f2-f2-a7)	adapted
deleteAllTickets (f2-6d-23)	adapted		
deleteOneTicket (f2-a1-ec)	adapted	deleteTicket (f2-83-15)	adapted
deleteOneTicket (f2-b3-13)	adapted		
addUnits (f2-2b-e1)	new	addUnits (f2-56-66)	new
addUnits (f2-cc-30)	new	addUnits (f2-92-ac)	new
buyTicket (f2-01-9a)	new	buyTicket (f2-23-e5)	new
buyTicket (f2-d1-4a)	new	buyTicket (f2-f5-8e)	new
buyTicket (f2-d7-da)	new	buyTicket (f2-6c-df)	new
buyTicket (f2-da-58)	new	buyTicket (f2-ad-da)	new
buyTicket (f2-bb-bc)	new	buyTicket (f2-ed-82)	new
closeApplication (f2-7c-7e)	new	closeApplication (f2-84-9c)	new
closeApplication (f2-d4-0f)	new	closeApplication (f2-b0-e1)	new
deleteAllTickets (f2-11-ac)	new	deleteAllTickets (f2-14-4e)	new
deleteOneTicket (f2-9c-01)	new	deleteTicket (f2-b5-43)	new
		deleteTicket (f2-fe-6a)	new
		deleteTicket (f2-de-n0)	new
goToHome (f2-b8-2b)	new	goToHome (f2-70-a1)	new
goToHome (f2-d6-17)	new	goToHome (f2-e5-ed)	new
registration (f2-2f-01)	new	registration (f2-6d-47)	new
registration (f2-4a-a3)	new	registration (f2-a7-c7)	new
retrieveUnits (f2-0a-49)	new	retrieveUnits (f2-3d-ec)	new
retrieveUnits (f2-3d-97)	new	retrieveUnits (f2-5d-2a)	new
		retrieveUnits (f2-e9-69)	new
setSubscription (f2-51-a2)	new	setSubscription (f2-c1-45)	new
setSubscription (f2-b2-99)	new		

TABLE C.2 – Résultats de SeTGaM pour le test fonctionnel - eCinema

Annexe D

Le langage de schémas de Smartesting

Nous présentons dans cette annexe la grammaire du langage de schémas de Smartesting. Le tableau D.1 donne les règles de la syntaxe de la grammaire. Le tableau D.2 définit précisément les symboles terminaux, pour une complète compréhension du langage.

```
test_purpose      : := (quantifier_list COMA) ? seq EOF ;
quantifier_list   : := quantifier (COMA quantifier)* ;
quantifier        : := FOR_EACH BEHAVIOUR var FROM behaviour_choice
                     | FOR_EACH OPERATION var FROM op_choice
                     | FOR_EACH LITERAL var FROM literal_choice
                     | FOR_EACH INSTANCE var FROM instance_choice
                     | FOR_EACH INTEGER var FROM integer_choice
                     | FOR_EACH CALL var FROM call_choice ;
op_choice         : := ANY_OPERATION
                     | ANY_OPERATION_BUT op_list
                     | op_list ;
call_choice       : := call_list ;
behaviour_choice  : := ANY_BEHAVIOUR_TO_COVER
                     | ANY_BEHAVIOUR_TO_COVER_BUT behaviour_list
                     | behaviour_list ;
literal_choice    : := IDENTIFIER (OR IDENTIFIER)* ;
instance_choice   : := instance (OR instance)* ;
integer_choice    : := CURLY_OPEN INT (COMA INT)+ CURLY_CLOSE ;
var               : := DOLLAR IDENTIFIER ;
state             : := ocl_constraint ON_INSTANCE instance ;
instance          : := IDENTIFIER ;
ocl_constraint    : := STRING_LITERAL ;
seq               : := bloc (THEN bloc)* ;
bloc              : := USE control restriction ? target ? ;
restriction       : := AT_LEAST_ONCE
                     | ANY_NUMBER_OF_TIMES
                     | INT TIMES
                     | var TIMES ;
target            : := TO_REACH state
                     | TO_ACTIVATE behaviour
                     | TO_ACTIVATE var ;
control           : := op_choice
                     | behaviour_choice
                     | var
                     | call_choice ;
call_list         : := call (OR call)* ;
op_list           : := operation (OR operation)* ;
operation         : := IDENTIFIER ;
call              : := instance '.' operation parameters ;
parameters        : := PARENTHESIS_OPEN (parameter (COMA parameter)*) ? PARENTHESIS_CLOSE ;
parameter         : := FREE_VALUE
                     | IDENTIFIER
                     | var
                     | INT ;
behaviour_list    : := behaviour (OR behaviour)* ;
behaviour         : := BEHAVIOUR_WITH_TAGS tag_list
                     | BEHAVIOUR_WITHOUT_TAGS tag_list ;
tag_list          : := CURLY_OPEN tag (COMA tag)* CURLY_CLOSE ;
tag               : := REQ COLON IDENTIFIER
                     | AIM COLON IDENTIFIER ;
```

TABLE D.1 – Syntaxe des schémas : règles de grammaire

TIMES	: :=	'times' ;
FOR_EACH	: :=	'for_each' ;
BEHAVIOUR	: :=	'behaviour' ;
OPERATION	: :=	'operation' ;
INTEGER	: :=	'integer' ;
CALL	: :=	'call' ;
INSTANCE	: :=	'instance' ;
LITERAL	: :=	'literal' ;
FROM	: :=	'from' ;
THEN	: :=	'then' ;
USE	: :=	'use' ;
TO_REACH	: :=	'to_reach' ;
TO_ACTIVATE	: :=	'to_activate' ;
ON_INSTANCE	: :=	'on_instance' ;
ANY_OPERATION	: :=	'any_operation' ;
ANY_OPERATION_BUT	: :=	'any_operation_but' ;
OR	: :=	'or' ;
ANY_BEHAVIOUR_TO_COVER	: :=	'any_behaviour_to_cover' ;
ANY_BEHAVIOUR_TO_COVER_BUT	: :=	'any_behaviour_to_cover_but' ;
BEHAVIOUR_WITH_TAGS	: :=	'behaviour_with_tags' ;
BEHAVIOUR_WITHOUT_TAGS	: :=	'behaviour_without_tags' ;
AT_LEAST_ONCE	: :=	'at_least_once' ;
ANY_NUMBER_OF_TIMES	: :=	'any_number_of_times' ;
COMA	: :=	',' ;
CURLY_OPEN	: :=	'{' ;
CURLY_CLOSE	: :=	'}' ;
PARENTHESIS_OPEN	: :=	'(' ;
PARENTHESIS_CLOSE	: :=	')' ;
COLON	: :=	' :' ;
DOLLAR	: :=	'$' ;
REQ	: :=	'REQ' ;
AIM	: :=	'AIM' ;
FREE_VALUE	: :=	'_' ;
DOT	: :=	'.' ;
IDENTIFIER	: :=	'identifier' ;
EOF	: :=	<EOF> ;

TABLE D.2 – Syntaxe des schémas : symboles terminaux

Bibliographie

[AGA10] Ahmed Alnatheer, Andrew M. Gravell, and David Argles. Agile security issues : an empirical study. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 58 :1–58 :1, New York, NY, USA, 2010. ACM.

[Aml00] Stale Amland. Risk-based testing : : Risk analysis fundamentals and metrics for software testing including a financial application case study. *Journal of Systems and Software*, 53(3) :287–295, 2000.

[AO08] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, Cambridge, UK, 2008.

[BD04] Pierre Bourque and Robert Dupuis, editors. *Software Engineering Body of Knowledge (SWEBOK)*. IEEE Computer Society Press, EUA, 2004.

[BFG+03] Céline Bigot, Alain Faivre, Jean-Pierre Gallois, Arnault Lapitre, David Lugato, Jean-Yves Pierron, and Nicolas Rapin. Automatic test generation with AGATHA. In *TACAS*, pages 591–596, 2003.

[BGL+07] Fabrice Bouquet, Christophe Grandpierre, Bruno Legeard, Fabien Peureux, Nicolas Vacelet, and M. Utting. A subset of precise UML for model-based testing. In *Proceedings of the 3rd International Workshop on Advances in Model Based Testing*, pages 95–104, 2007.

[BGLP08] Fabrice Bouquet, Christophe Grandpierre, Bruno Legeard, and Fabien Peureux. A test generation solution to automate software testing. In *Proceedings of the 3rd International Workshop on Automation of Software Test*, AST '08, pages 45–48, New York, NY, USA, 2008. ACM.

[BLH09] Lionel Briand, Yvan Labiche, and Siyuan He. Automating regression test selection based on UML designs. *Information and Software Technology (Elsevier)*, 51(1), 2009.

[BMSS11] S. Biswas, R. Mall, M. Satpathy, and S. Sukurman. Regression test selection techniques : A survey. *Informatica*, 35(3) :289–321, October 2011.

[CAB12] J. Cantenot, F. Ambert, and F Bouquet. Transformation rules from uml4mbt meta-model to smt meta-model for model animation. In *Proceedings of the 12th International Workshop OCL, co-located with MODELS 2012*, 2012.

[CCB11] Kalou Cabrera Castillos and Julien Botella. Scenario based test generation using test designer. In *Proceedings of the 1st International Workshop on Scenario Based Testing – co-located with ICST'2011*, Berlin, Germany, March 2011. IEEE Computer Society Press.

[CDKP94] D. M. Cohen, S. R. Dalal, A. Kajla, and G. C. Patton. The automatic efficient test generator (AETG) system. In *In International Conference on Testing Computer Software*, 1994.

[Con08] Global Platform Consortium. Global platform uicc configuration version 1.0, October 2008. GPC_GUI_010.

[DBr08] L. De Moura and N. Bjø rner. Z3 : An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[DHBT07] Werner Damm, Holger Hermanns, Clark Barrett, and Cesare Tinelli. *Computer Aided Verification*, volume 4590 of *LNCS*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[DHK11] Frédéric Dadeau, Pierre-Cyrille Héam, and Rafik Kheddam. Mutation-based test generation from security protocols in HLPSL. In M. Harman and B. Korel, editors, *roceedings of the 4th International Conference on Software Testing, Verification and Validation (ICST)*, pages 240–248, Berlin, Germany, March 2011. IEEE Computer Society Press.

[DMTR10] Hyunsook Do, Siavash Mirarab, Ladan Tahvildari, and Gregg Rothermel. The effects of time constraints on test case prioritization : A series of controlled experiments. *IEEE Transactions on Software Engineering*, 36(5) :593–617, 2010.

[ESR08] Emelie Engström, Mats Skoglund, and Per Runeson. Empirical evaluations of regression test selection techniques : a systematic review. In *Proceedings of the 2nd International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 22–31, New York, NY, USA, 2008. ACM.

[FABA10] Michael Felderer, Berthold Agreiter, Ruth Breu, and Alvaro Armenteros. Security testing by telling teststories. In Gregor Engels, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung 2010*, volume 161 of *LNI*, pages 195–202. GI, 2010.

[FAZB11] M. Felderer, B. Agreiter, P. Zech, and R. Breu. A classification for model-based security testing. In *Proceedings of the 3rd International Conference on Advances in System Testing and Validation Lifecycle(VALID 2011)*, pages 109–114, 2011.

[FB97] George Fink and Matt Bishop. Property-based testing : a new approach to testing for assurance. *SIGSOFT Software Engineering Notes*, 22 :74–80, 1997.

[FBO+12] Elizabeta Fourneret, Fabrice Bouquet, Martin Ochoa, Jan Jürjens, and Sven Wenzel. Vérification et test pour des systèmes évolutifs. In *AFADL'12, Congrès Approches Formelles dans l'Assistance au Développement de Logiciels*, pages 150–164, Grenoble, France, January 2012.

[FIMN07] Qurat-ul-ann Farooq, Muhammad Zohaib Z. Iqbal, Zafar I Malik, and Aamer Nadeem. An approach for selective state machine based regression testing. In *Proceedings of the 3rd international workshop on Advances in model-based testing*, A-MOST '07, pages 44–52, New York, NY, USA, 2007. ACM.

[FIMR10] Qurat-ul-ann Farooq, Muhammad Zohaib Z. Iqbal, Zafar I. Malik, and Matthias Riebisch. A model-based regression testing approach for evolving software systems with flexible tool support. In *Proceedings of the 17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, ECBS '10, pages 41–49, Washington, DC, USA, 2010. IEEE Computer Society Press.

[FOB+11] E. Fourneret, M. Ochoa, F. Bouquet, J. Botella, J. Jurjens, and P. Yousefi. Model-based security verification and testing for smart-cards. In *Proceedings of the 6th International Conference on Availability, Reliability and Security (ARES)*, pages 272–279. IEEE, 2011.

[GPCL08] Ravi Prakash Gorthi, Anjaneyulu Pasala, Kailash KP Chanduka, and Benny Leong. Specification-based approach to select regression test suite to validate changed software. In *Proceedings of the 15th Asia-Pacific Software Engineering Conference*, APSEC '08, pages 153–160, Washington, DC, USA, 2008. IEEE Computer Society Press.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[HXEK+12] JeeHyun Hwang, Tao Xie, Donia El Kateb, Tejeddine Mouelhi, and Yves Le Traon. Selection of regression system tests for security policy evolution. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 266–269. ACM, 2012.

[ISO94] ISO/IEC. *Information technology – open systems interconnection – conformance testing methodology and framework*, 1994. International ISO/IEC multi–part standard No. 9646.

[JÖ5] Jan Jürjens. *Secure Systems Development with UML*. Springer-Verlag, 2005.

[JÖ8] Jan Jürjens. Model-based security testing using UMLsec. *Electronic Notes Theory Computer Science*, 220(1) :93–104, December 2008.

[JJ05] Claude Jard and Thierry Jéron. TGV : theory, principles and algorithms. *STTT*, 7(4) :297–315, 2005.

[JTG+10] Bo Jiang, T. H. Tse, Wolfgang Grieskamp, Nicolas Kicillof, Yiming Cao, and Xiang Li. Regression testing process improvement for specification evolution of real-world protocol software. In *Proceedings of the 10th International Conference on Quality Software*, pages 62–71, 2010.

[JW92] Laski J. and Szermer W. Identification of program modifications and its applications in software maintentance. In *Conference on Software Maintenance*, pages 282–290, 1992.

[Kap04] Gregory M. Kapfhammer. *The Computer Science and Engineering Handbook*, chapter Chapter 105 : Software Testing. Allen Tucker, CRC Press, Boca Raton, FL, second edition, 2004.

[KFA81] Fischer K., Raji F., and Chrusckicki A. A methodology for retesting modified software. In *National Tele. Conference B-6-3*, pages 1–6, 1981.

[KHV02] Bogdan Korel, Luay H.Tahat, and Boris Vaysburg. Model based regression test reduction using dependence analysis. In *IEEE ICSM'02*, page 10, 2002.

[Kon06] Vidar Kongsli. Towards agile security in web applications. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 805–808, New York, NY, USA, 2006. ACM.

[Lan96] Kevin Lano. *The B Language and Method : A Guide to Practical Formal Development*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1996.

[LBP09] Bruno Legeard, Fabrice Bouquet, and Natacha Pickaert. *Industrialiser le test fonctionnel*. Dunod, Paris, France, 2009.

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML : a behavioral interface specification language for java. *SIGSOFT Software Engineering Notes*, 31(3) :1–38, May 2006.

[LdBMB04] Yves Ledru, Lydie du Bousquet, Olivier Maury, and Pierre Bontron. Filtering tobias combinatorial test suites. In *FASE*, pages 281–294, 2004.

[LSS11] Mass Lund, Bjørnar Solhaug, and Ketil Stølen. *Model-Driven Risk Analysis : The CORAS Approach*. Springer, 1st edition, 2011.

[LW89] H. K. N. Leung and L White. Insights into regression testing (software testing). In *Proceedings Conference on Software Maintenance 1989*, pages 60–69. IEEE Computer Society Press, 1989.

[Meh] Dharmesh M Mehta. Effective software security management.

[MLdB03] O. Maury, Y. Ledru, and L. du Bousquet. Intégration de TOBIAS et UCASTING pour la génération de tests. In *In 16th International Conference Software & Systems Engineering and their Applications-ICSSEA'03*, 2003.

[Mou10] Tejeddine Mouelhi. *Testing and Modeling Security Mechanisms in Web Applications*. PhD thesis, Université de Rennes, 2010.

[MP04] Gary McGraw and Bruce Potter. Software security testing. *IEEE Security and Privacy*, 2(5) :81–85, September 2004.

[MPJ+10] Pierre-Alain Masson, Marie-Laure Potet, Jacques Julliand, Régis Tissot, Georges Debois, Bruno Legeard, Boutheina Chetali, Fabrice Bouquet, Eddie Jaffuel, Lionel Van Aertrick, June Andronick, and Amal Haddad. An access control model based testing approach for smart card applications : Results of the POSÉ project. *JIAS, Journal of Information Assurance and Security*, 5(1) :335–351, 2010.

[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, Centrum voor Wiskunde en Informatica (CWI). Amsterdam (NL), Amsterdam, The Netherlands, The Netherlands, 1997.

[NMS+11] Agastya Nanda, Senthil Mani, Saurabh Sinha, Mary Jean Harrold, and Alessandro Orso. Regression testing in the presence of non-code changes. In *Proceedings of the 4th International Conference on Software Testing, Verification and Validation (ICST)*, ICST '11, pages 21–30, Washington, DC, USA, 2011. IEEE Computer Society Press.

[NR07] Leila Naslavsky and Debra J. Richardson. Using traceability to support model-based regression testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 567–570, New York, NY, USA, 2007. ACM.

[PMBD12] F. Paci, F. Massacci, F. Bouquet, and S. Debricon. Managing evolution by orchestrating requirements and testing engineering processes. In *Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST)*, pages 834–841, april 2012.

[Pro03] S. J. Prowell. JUMBL : A tool for model-based statistical testing. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences*. Society Press, 2003.

[RBGW10] Thomas Rossner, Christian Brandes, Helmut Goetz, and Mario Winter. *Basiswissen Modellbasierter Test*. dpunkt Verlag, 2010. in German.

[RH96] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, pages 529–551, 1996.

[RH97] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, pages 173–210, 1997.

[RHG+01] Gregg Rothermel, Mary Jean Harrold, Todd L. Graves, Jung-Min Kim, and Adam Porter. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodoligy*, 10 : 184–208, april 2001.

[Sch12] Ina Schieferdecker. Model-based testing. *IEEE Software*, 29(1) :14–18, 2012.

[Sec10] Work Package 1 SecureChange. Deliverable 1.1 : Description of scenarios and their requirements, January 2010. SecureChange (EU ICT-FET-231101) [accessed : September 29, 2012].

[Sec12a] Work Package 1 SecureChange. Deliverable 1.3 : Report on the industrial validation of securechange solutions, January 2012. SecureChange (EU ICT-FET-231101) [accessed : October 12, 2012].

[Sec12b] Work Package 7 SecureChange. Deliverable 7.4 : Results of test campaign on case studies, 2012. SecureChange (EU ICT-FET-231101) [accessed : September 1, 2012].

[SGR] Ina Schieferdecker, Jürgen Großmann, and Axel Rennoch. Model based security testing *Selected Considerations*. Keynote at SECTEST at ICST 2011 [accessed : Septmber 25, 2012].

[SGS12] Ina Schieferdecker, Juergen Grossmann, and Martin Schneider. Model-based security testing. In *MBT*, pages 1–12, 2012.

[Smi00] Graeme Smith. *The Object-Z specification language*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[SMP08] Heiko Stallbaum, Andreas Metzger, and Klaus Pohl. An automated technique for risk-based test case generation and prioritization. In *Proceedings of the 3rd International Zorkshop on Automation of Software Test*, AST '08, pages 67–70, New York, NY, USA, 2008. ACM.

[TB03] G. J. Tretmans and H. Brinksma. TorX : Automated model-based testing. In *Proceedings of the 1st Europeqn Conference on Model-Driven Software Engineering*, pages 31–43, Nuremberg, Germany, December 2003.

[TBVK01] Luay Ho Tahat, Atef Bader, Boris Vaysburg, and Bogdan Korel. Requirement-based automated black-box test generation. In *Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development*, COMPSAC '01, pages 489–495, Washington, DC, USA, 2001. IEEE Computer Society Press.

[Tre04] Jan Tretmans. Model–based testing : Property checking for real. Keynote addressed at the International Workshop for Construction and Analysis of Safe Secure, and Interoperable Smart Devices, 2004. Available at `http://www-sop.inria.fr/everest/events/cassis04` [accessed : August 17, 2012].

[TyYsYy10] Gu Tian-yang, Shi Yin-sheng, and Fang You-yuan. Research on software security testing. *World Academy of Science, Engineering and Technology*, 70, 2010.

[UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing : A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[UPC07] Hasan Ural, Robert L. Probert, and Yanping Chen. Model based regression test suite generation using dependence analysis. In *Proceedings of the 3rd International Workshop on Advances in model-based testing*, pages 54–62, 2007.

[UPL12] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability (STVR)*, 22(5) :297–312, August 2012.

[VAJ03] L. Van Aertryck and T. Jensen. UML-CASTING : Test synthesis from UML models using constraint resolution. In *AFADL'03*, 2003.

[VCG+08] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. In *Formal Methods and Testing*, pages 39–76, 2008.

[VF98] F. I. Vokolos and P. G. Frankl. Empirical evaluation of the textual differencing regression testing technique. *International Conference on Software Maintenance*, pages 44–53, 1998.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language : Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.

[XP06] D. Xu and J. J. Pauli. Threat-Driven Design and Analysis of Secure Software Architectures. *Journal of Information Assurance and Security*, 1(3) :171–180, 2006.

[XTK+12] Dianxiang Xu, Lijo Thomas, Michael Kent, Tejeddine Mouelhi, and Yves Le Traon. A model-based approach to automated testing of access control policies. In *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, SACMAT '12, pages 209–218, New York, NY, USA, 2012. ACM.

[YH10] Shin Yoo and Mark Harman. Regression testing minimisation, selection and prioritisation : A survey. *Software Testing, Verification, and Reliability*, 2010.

[Zec11] Philippe Zech. Risk-based security testing in cloud computing environments. In *PhD Symposium at the 4th International Conference on Software Testing, Verification and Validation (ICST)*, ICST '11, pages 411 – 414. IEEE, 2011.

Résumé

Cette thèse porte sur l'étude d'une démarche et de techniques pour prendre en compte les spécificités des systèmes sécurisés évolutifs lors de la génération des tests à partir de modèles UML/OCL. Dans ce travail, trois axes sont étudiés : (i) le cycle de vie des tests, (ii) les exigences fonctionnelles et (iii) les exigences de sécurité. Dans un premier temps, nous avons défini la clé de voûte de notre approche qui est la caractérisation des statuts du cycle de vie des tests. À l'issu de ces travaux, nous avons pu définir la démarche de classification des tests pour les systèmes évolutifs, appelée *SeTGaM*. La notation UML, accompagnée du langage de spécification OCL, permet de formaliser les comportements du système. Le langage OCL spécifie ainsi les gardes/actions des transitions et les pré/post conditions des opérations. La méthode propose ainsi deux classifications des tests : la première s'appuie sur les comportements issus des transitions du diagramme d'états/transitions, tandis que l'autre repose sur l'étude des comportements issus des opérations du diagramme de classes. Dans le domaine du test de logiciels critiques, une des questions clés concerne la sécurité. Pour cette raison, nous avons enrichi l'approche SeTGaM en prenant en compte l'aspect sécurité. Ainsi, SeTGaM permet de générer sélectivement des tests qui ciblent les différentes exigences de sécurité lors des évolutions du système. Finalement, le prototype de SeTGaM a été intégré, avec l'outil Smartesting CertifyIt, à l'environnement IBM Rational Software Architect. Ceci nous a permis de valider expérimentalement le passage à l'échelle de la méthode sur des études de cas industriels, notamment proposées par Gemalto/Trusted Labs dans le cadre du projet européen SecureChange.

Mots-clés: génération de tests à partir de modèles, UML/OCL, test fonctionnel, test de sécurité, systèmes critiques, systèmes évolutifs

Abstract

This thesis is focused on methods and approaches taking into account the evolution in case of UML/OCL model-based test generation. In this framework I am studying three major axes : (i) the test's life cycle for evolving critical systems, (ii) functional requirements and (iii) security requirements. In order to respond these challenges we have defined the core of our approach, which is the test life cycle. Then, we have created *SeTGaM*, a test classification approach for validation of critical evolving systems. The Object Constraint Language (OCL) is used with UML to formalize different behaviors of the system under test. It specifies the transition's guards and actions and operation's pre and post conditions. Thus, on one hand, we have created two "panels" of SeTGaM : the first dedicated for selecting tests basing on behaviors issued from transitions and the second basing on behaviors issued from operations. On the other hand, in the industry testing the security is unavoidable issue. That is why, we have worked on the security aspect in the scope of model-based testing. We have extended the SeTGaM approach by adding the security "panel". SeTGaM thus allows to generate selectively tests and manage their life cycle through different evolutions of the system under test. Finally, a prototype was smoothly integrated with the Smartesting CertifyIt tool and the IBM Software Rational Architect. The prototype permitted to define the benefits of such a methodology through the case studies of our project's industrial partners, particularly Gemalto/Trusted Labs.

Keywords: Model-Based Testing, UML/OCL, functional testing, security testing, critical systems, evolving systems