Génération automatique de scénarios de tests à partir de propriétés temporelles et de modèles comportementaux

Kalou Cabrera Castillos

N° d'ordre : Année 2013

THÈSE

présentée à

L'UFR DES SCIENCES ET TECHNIQUES
DE L'UNIVERSITÉ DE FRANCHE-COMTÉ

pour obtenir le

GRADE DE DOCTEUR
DE L'UNIVERSITÉ DE FRANCHE-COMTÉ
spécialité Informatique

Génération automatique de scénarios de tests à partir de propriétés temporelles et de modèles comportementaux

par

Kalou Cabrera Castillos

Soutenue le 28 novembre 2013 devant la commission d'examen :

Directeurs	Jacques JULLIAND, Professeur à l'Université de Franche-Comté Frédéric DADEAU, Maître de conférences à l'Université de Franche-Comté
Rapporteurs	Stéphane MAAG, Maître de conférences HDR à TELECOM SudParis, Paris Marie-Laure POTET, Professeur à l'Ensimag, Grenoble
Examinateurs	Fabrice BOUQUET, Professeur à l'Université de Franche-Comté Safouan TAHA, Professeur assistant à Supélec, Gif-sur-Yvette

Table des matières

Partie I Problématique et contributions **1**

Chapitre 1 Introduction : contexte et problématique **3**

1.1 Contexte . . . 3
1.1.1 Contexte historique . . . 3
1.1.2 Test à partir de modèles . . . 4
1.2 Problématique . . . 5
1.3 Contributions . . . 7
1.3.1 Processus de tests à partir de propriété . . . 8
1.3.2 Synthèse des contributions . . . 8
1.4 Présentation du plan . . . 10

Chapitre 2 Etat de l'art : Test à partir de modèles **13**

2.1 Introduction au test . . . 14
2.1.1 Classification des techniques de tests . . . 14
2.1.2 Critères de sélection . . . 17
2.2 Modèles comportementaux . . . 18
2.2.1 Paradigmes de modélisation . . . 18
2.2.2 Formalismes de modèles pre/post . . . 19
2.3 Critères statiques . . . 20
2.3.1 Couverture du graphe de flot de contrôle . . . 21
2.3.2 Couverture de flot de données . . . 21
2.3.3 Couverture des états/transitions . . . 23
2.3.4 Couverture de données . . . 25
2.4 Critères dynamiques . . . 26
2.4.1 Approches par scénarios . . . 26

2.4.2 Approches par propriétés sur le système 28
2.4.3 Approches par modèles de fautes 31
2.5 Synthèse 32

Chapitre 3 Modéliser et tester avec UML/OCL 35
3.1 Langage UML4MBT 36
3.1.1 Diagramme de classes 36
3.1.2 Diagramme d'objets 39
3.1.3 Diagramme d'états-transitions 39
3.2 Langage OCL4MBT 41
3.2.1 Interprétation du langage 41
3.2.2 Constructions OCL4MBT 43
3.2.3 Interprétation de la valeur indéfinie 48
3.2.4 Expression des exigences fonctionnelles 48
3.3 Génération de tests dans CertifyIt 49
3.3.1 Calcul des cibles de tests 50
3.3.2 Génération de tests 51
3.4 Exemple fil rouge 53
3.4.1 Exigences fonctionnelles 53
3.4.2 Description du modèle 54
3.4.3 Diagramme de classes 54
3.4.4 Diagramme d'objets 57
3.4.5 Exemples de tests générés par CertifyIt 59
3.5 Synthèse 59

Partie II Contributions 61

Chapitre 4 Génération de tests à partir de scénarios 63
4.1 Langage de scénarios 64
4.1.1 Syntaxe 64
4.1.2 Sémantique des instructions élémentaires 66
4.1.3 Sémantique des instructions de contrôle de flot 69
4.2 Génération de tests 72
4.2.1 Dépliage du scénario 73

4.2.2 Mise en œuvre . . . 74
4.2.3 Exemple de dépliage . . . 76
4.2.4 Comment recourir à l'expertise de l'ingénieur ? . . . 79
4.3 Synthèse . . . 81

Chapitre 5 Langage de propriétés **85**
5.1 Syntaxe . . . 86
5.1.1 Propriétés . . . 87
5.1.2 Les évènements . . . 88
5.2 Automates de substitution . . . 89
5.2.1 Définition . . . 89
5.2.2 Substitution . . . 91
5.2.3 Evénements . . . 95
5.3 Sémantique du langage . . . 96
5.3.1 Sémantique des portées . . . 96
5.3.2 Sémantique des motifs . . . 99
5.3.3 Application de la composition d'automate . . . 102
5.4 Différences sémantiques avec Dwyer *et al.* . . . 103
5.4.1 Types d'évènements . . . 104
5.4.2 Notion de satisfiabilité de propriétés . . . 104
5.5 Quelques exemples de propriétés . . . 104
5.6 Synthèse . . . 106

Chapitre 6 Critères et mesure de couverture de propriétés **109**
6.1 Définitions et notations . . . 110
6.1.1 Exemple de propriété . . . 110
6.1.2 Définitions . . . 110
6.1.3 Animation des tests . . . 111
6.2 Critères et mesure de couverture des automates . . . 113
6.2.1 Critère all-α : toutes les α-transitions . . . 113
6.2.2 Critère *all-α-pairs* : toutes les paires d'α-transitions . . . 114
6.2.3 Critère *k-pattern* : k itérations de `precedes` et `follows` . . . 115
6.2.4 Critère *k-scope* : k itérations de `between/and` et `after/until` . . . 116
6.3 Discussion sur les critères . . . 117
6.3.1 Hiérarchisation des critères . . . 117

6.3.2 Applicabilité des critères 121
6.4 Synthèse 123

Chapitre 7 Critère de robustesse 125
7.1 Définitions préliminaires 126
7.2 Opérateurs de mutation 128
7.2.1 Suppression de précondition 128
7.2.2 Suppression de tags 129
7.2.3 Suppression de proposition atomique 130
7.2.4 Echange de tags 131
7.2.5 Mutations dans l'exemple de propriété 132
7.3 Critère de robustesse 133
7.3.1 Mutation de l'automate 134
7.3.2 Critère de robustesse 136
7.4 Synthèse 137

Chapitre 8 Génération de tests à partir des critères de couverture des automates 139
8.1 Génération de tests guidée par les critères 140
8.1.1 Rappel de l'exemple 140
8.1.2 Définitions 140
8.1.3 Critère *all-α* 141
8.1.4 Critère $\alpha-$*pairs* 142
8.1.5 Critères *k-scope* et *k-pattern* 143
8.1.6 Critère de robustesse 145
8.2 Traduction en scénarios 146
8.2.1 Règles de traduction basiques 147
8.2.2 Traduction des restrictions sur un évènement simple 147
8.2.3 Traduction des restrictions sur un évènement sans opération ni comportement 151
8.2.4 Traduction de l'exemple 154
8.3 Synthèse 155

Partie III Evaluation expérimentale des contributions 157

Chapitre 9 Etude de cas : ECinema 159

9.1 Propriétés considérées . . . 160
9.2 Mesure de couverture . . . 164
9.2.1 Remarques générales . . . 164
9.2.2 Couverture des propriétés . . . 166
9.3 Capacité de détection . . . 167
9.3.1 Protocole expérimental . . . 168
9.3.2 Mutations considérées . . . 170
9.3.3 Analyse . . . 174
9.4 Discussion . . . 175
9.5 Conclusions . . . 177

Chapitre 10 Mise en œuvre et application dans le cadre du projet ANR TASCCC 179

10.1 Présentation . . . 180
10.1.1 Les acteurs du projet . . . 181
10.1.2 Détail des sous-projets . . . 181
10.2 L'étude de cas : GlobalPlatform . . . 183
10.2.1 Gestion du contenu de la carte . . . 184
10.2.2 Chargement d'une application . . . 184
10.3 Prototype logiciel . . . 185
10.3.1 Interface générale . . . 185
10.3.2 Définition des éléments de traçabilité . . . 187
10.3.3 Mesure de couverture d'une suite de tests . . . 187
10.3.4 Génération de tests . . . 188
10.3.5 Rapport de génération de tests . . . 190
10.4 Evaluation de l'approche . . . 191
10.4.1 Point de vue de l'évaluateur Critères Communs . . . 191
10.4.2 Evaluation utilisateur de la méthode . . . 193
10.5 Synthèse . . . 194

Partie IV Conclusions et perspectives **195**

Chapitre 11 Conclusions et perspectives **197**

11.1 Processus 198
11.2 Le langage de propriétés 200
11.3 Les critères de couverture 202
11.4 Le langage de scénarios 203
11.5 Expérimentations 204
11.6 Autres pistes 205

Bibliographie **207**

Table des figures

1.1 Processus classique du *model-based testing* 6
1.2 Schéma du processus de génération de tests proposé 9

2.1 Les différents axes de classification des techniques de tests 15
2.2 Hiérarchie des critères de couverture de flot de données 22
2.3 Exemple d'automate simple 23
2.4 Exemple de système à plusieurs automates 23

3.1 Exemple de classe 37
3.2 Exemple de classe d'énumération 37
3.3 Multiplicité d'association supporté dans UML4MBT 38
3.4 Exemple d'associations entre deux classes 38
3.5 Exemple de diagramme d'objet 39
3.6 Les différents types d'états d'un diagramme d'états-transitions 40
3.7 Représentation graphique d'une transition 41
3.8 Exemple de diagramme d'états-transitions 42
3.9 Opérateurs considérés dans OCL4MBT 45
3.10 Code OCL de l'opération *freiner* 47
3.11 Code OCL de l'opération *freiner* avec les tags des exigences 49
3.12 Comportements de l'opération *freiner* 50
3.13 Opérations et comportements du modèle eCinema 55
3.14 Diagramme de classe de eCinema 56
3.15 Code OCL de l'opération *buyTicket* 57
3.16 Comportements de l'opération *buyTicket* 58
3.17 Diagramme d'objets initial de eCinema 58

4.1 Syntaxe du langage de scénarios 65
4.2 Sémantique d'un appel d'opération 67
4.3 Sémantique de l'appel de scénario 68
4.4 Sémantique de `assert(P)` 69
4.5 Sémantiques des structures conditionnelles `if-then` et `if-then-else` . . . 70
4.6 Sémantique de l'instruction `choice{`A_1`}or{`A_2`}or...or{`A_n`}` 70
4.7 Sémantique de `repeat/unfold between` m `and` n `times` {A} `until` (P) . . 71
4.8 Sémantique de `foreach/any` $i `in` {$i_1$, . . . , i_n}{A} 72
4.9 Graphe de l'exemple 17 74

4.10 Algorithme d'exploration combinatoire du graphe du scénario 75
4.11 Graphe sémantique du scénario exemple 77
4.12 Combinaisons possibles entre utilisateurs et mots de passes 77
4.13 Valuations de l'opération *login* 77
4.14 Evaluation de l'animation des trois premières itérations de la répétition de *buyTicket* 82
4.15 Evaluation de l'animation de la quatrième itération de la répétition de *buyTicket* 82
4.16 Tests extraits de l'animation du scénario 83
4.17 Cas de test sans enregistrement d'utilisateur 83
4.18 Cas de test avec enregistrement d'un utilisateur 84

5.1 Syntaxe du langage de propriétés 86
5.2 Représentation graphique des portées 88
5.3 Exemple d'automate de substitution 90
5.4 Composition et restriction d'évènements 94
5.5 Automate de la portée `globally` 97
5.6 Automate de la portée `before` E 97
5.7 Automate de la portée `after` E 97
5.8 Automate de la portée `between` E `and` F 98
5.9 Automate de la portée `between last` E `and` F 98
5.10 Automate de la portée `after` E `until` F 98
5.11 Automate de la portée `after last` E `until` F 98
5.12 Automate du motif `always` P 100
5.13 Automate du motif `never` E 100
5.14 Automate du motif `eventually` de l'Exemple 29 100
5.15 Automate du motif `eventually` E k `times` 100
5.16 Automate du motif `eventually` E `at least` k `times` 100
5.17 Automate du motif `eventually` E `at most` k `times` 100
5.18 Automate du motif `eventually` E 100
5.19 Automate du motif E `follows` F 101
5.20 Automate du motif E `directly follows` F 101
5.21 Automate du motif E `precedes` F 102
5.22 Automate du motif E `directly precedes` F 102
5.23 Automate réduit du motif E `directly precedes` F 102
5.24 Automate de `always` C `between` A `and` B 105
5.25 Automate de `eventually` B `before` A 105
5.26 Automate de A `follows` B `between` C `and` D 106
5.27 Automate de A `follows` B `before` C 106

6.1 Automate de l'exemple C `precedes` D `after` A `until` 110
6.2 Hiérarchie des nouveaux critères vis-à-vis des critères classiques 117
6.3 Automate de A `follows` B `between` C `and` D 117
6.4 Exemple de cas spécial pour *all-*α $\rightarrow$ *all-*α*-pairs* 117
6.5 Automate du motif E `directly precedes` F 120

6.6 Utilisations possibles des critères par combinaison portée/motif 122
6.7 Automate de la propriété `never` A `globally` 123

7.1 Automate de l'exemple `never` A `globally` 126
7.2 Code OCL de l'opération *buyTicket* 130
7.3 Graphe de tags de l'opération *buyTicket* 131
7.4 Automate de l'exemple C`precedes` D `after` A `until` B 133
7.5 Automate de `never` A `before B` 135
7.6 Automate muté de `never` A `before B` sur la transition $3 \xrightarrow{B} 4$ et $\mathcal{D}_\mu = \{B'\}$ 135

8.1 Rappel de l'automate de C`precedes` D `after` A `until` B 140
8.2 Algorithme de génération - critère *all-α* 141
8.3 Algorithme de génération - critère *α−pairs* 142
8.4 Algorithme de génération - critères *k-scope* et *k-pattern* 144
8.5 Algorithme de génération - critère de robustesse 146
8.6 Automate de la FIGURE 8.1 muté 146
8.7 Enumération des cas pour une restriction $E \backslash E_1$ 149
8.8 Enumération des cas pour $E \backslash \{E_1, E_2\}$ 149
8.9 Traduction de la restriction de l'exemple 51 151
8.10 Traduction de l'évènement $\Sigma \backslash \{B, D\}$ 154
8.11 Traduction de l'évènement $\Sigma \backslash \{A\}$ 154
8.12 Scénario du préfixe de l'exemple 155

9.1 Automate de la propriété P_1 160
9.2 Automate de la propriété P_2 160
9.3 Automate de la propriété P_{31} 160
9.4 Automate des propriétés P_{32}, P_{33} et P_4 161
9.5 Automate de la propriété P_5 162
9.6 Automate de la propriété P_6 162
9.7 Automate de la propriété P_{71} 163
9.8 Automate de la propriété P_{72} 163
9.9 Résultats de la couverture 165
9.10 Processus du protocole expérimental 168
9.11 Règles de mutation des remplacements d'opérateurs 172
9.12 Nombres de mutations par opération et par type de mutations 173
9.13 Verdicts d'élimination des mutants par famille de mutations et par suite de tests 174

10.1 Schéma de l'approche proposée par le projet TASCCC 180
10.2 Interface générale du prototype TASCCC 186
10.3 Grammaire du fichier des éléments de traçabilité 187
10.4 Exemple de fichier contenant les éléments de traçabilité 188
10.5 Rapport de couverture 189
10.6 Rapport de génération de tests - vue TSFI 190
10.7 Rapport de génération de tests - vue SFR↔Propriété 191
10.8 Rapport de génération de tests - vue Tests 191

10.9 Rapport de génération de tests - détail des tests 192

11.1 Processus itératif de génération de tests et de validation de modèle 199
11.2 Exemple de graphe de tags 203

Première partie

Problématique et contributions

Chapitre 1

Introduction : contexte et problématique

Sommaire

1.1 Contexte 3
1.1.1 Contexte historique 3
1.1.2 Test à partir de modèles 4
1.2 Problématique 5
1.3 Contributions 7
1.3.1 Processus de tests à partir de propriété 8
1.3.2 Synthèse des contributions 8
1.4 Présentation du plan 10

Ce manuscrit de thèse rapporte des travaux menés à l'institut FEMTO-ST de Besançon, et plus particulièrement au Département Informatique des Systèmes Complexes (DISC), de l'Université de Franche-Comté. Cette thèse a été effectuée dans le cadre du projet ANR TASCCC [1] (**T**est **A**utomatique basé sur des **SC**énarios et évaluation **C**ritères **C**ommuns) et se focalise sur la génération de tests à partir de modèles comportementaux (*model-based testing*) et de propriétés temporelles (*property-based testing*).

Nous abordons dans la section 1.1 le contexte scientifique dans lequel se situe cette thèse. Puis, nous présentons dans la section 1.2 la problématique à laquelle cette thèse s'intéresse. Nous présentons ensuite dans la section 1.3 les contributions apportées par ce travail de recherche. Enfin, la section 1.4 présente le plan de ce document.

1.1 Contexte

1.1.1 Contexte historique

La qualité d'un logiciel est un enjeu important pour les industriels qui les conçoivent. Celle-ci est en général assurée par une étape de test. Cependant, cette tâche, bien que

1. ANR-09-SEGI-014 - `http://disc.univ-fcomte.fr/TASCCC`

coûteuse en terme de temps, est indispensable pour assurer une certaine qualité du logiciel. En effet, un défaut dans un logiciel peut avoir des conséquences plus ou moins graves pour les utilisateurs et l'entreprise.

Ces conséquences peuvent être bénignes tels que des défauts d'affichage, qui ne constituent pas nécessairement un obstacle à l'utilisation du logiciel. Cependant, leur correction peut s'accompagner d'un coût pour l'entreprise qui n'a pas été prévu initialement.

Ces conséquences peuvent avoir des impacts économiques notables pour les exploitants du logiciel. Ce fut le cas, par exemple, de l'explosion du lanceur Ariane 5 à cause d'un dépassement de capacité de la variable qui représentait l'accélération horizontale dans le composant qui mesure les accélérations de l'appareil. Le coût de cet accident s'élève à environ 370 millions de dollars [LL96]. Plus récemment, le 6 janvier 2010, les possesseurs allemands de cartes bancaires conçues par Gemalto, un des leaders de la carte à puce, ont eu la désagréable surprise de ne plus pouvoir utiliser leur carte dans certains terminaux : l'année 2010 n'était pas reconnue correctement. Le lendemain, une mise à jour de ces terminaux, a permis de corriger le problème. Cependant, même si ce problème n'a duré qu'une journée, Gemalto a estimé le coût associé à ce défaut entre 6 et 10 millions d'euros.

Enfin, dans les pires cas, les conséquences d'un défaut dans le système peuvent être dramatiques comme cela a été le cas pour le logiciel intégré aux Therac-25, des appareils de radiothérapie entre 1985 et 1987. Un dysfonctionnement du logiciel est à l'origine d'une surdose de radiations qui a coûté la vie de plusieurs patients [LT93].

Ces accidents montrent à quel point il est nécessaire de valider qualitativement les logiciels, quel que soit le domaine d'application pour lequel ils sont conçus. Pour s'assurer de la qualité d'un logiciel, il existe deux moyens. Le premier est la vérification formelle de l'ensemble du système par une représentation mathématique du logiciel qui est ensuite prouvée. Cependant, avec l'augmentation de la taille des programmes, la difficulté de la vérification formelle augmente aussi. Il existe bien des applications de la vérification formelle à l'échelle industrielle telle que la ligne 14 Météor du métro de Paris [BBFM99] ou sur des composants critiques notamment dans le domaine de l'aéronautique, mais nous sommes loin d'une généralisation de la vérification formelle des logiciels dans l'industrie.

Pour palier cela, nous pouvons vérifier partiellement le système, par une activité de test, intégrée au cycle de développement du logiciel. Cette activité vise à mettre en lumière des défaillances d'un logiciel en l'utilisant dans différentes conditions. Le test a fait l'objet de nombreux travaux de recherche qui ont permis de développer de nouvelles techniques et de nouveaux outils pour automatiser et systématiser le plus possible la génération des tests. C'est à cette activité que se raccrochent les travaux de cette thèse.

1.1.2 Test à partir de modèles

Le test à partir de modèles est une technique de génération de tests à partir d'une abstraction du système, un modèle, qui est une représentation formelle du système. Dans le cadre du test, ce modèle est en général tiré des spécifications du système, ce qui fait du *model-based testing* une technique de génération de tests *boîte noire*.

La FIGURE 1.1 montre le processus classique du test à partir de modèles qui met en parallèle le processus de **développement** du **système sous test** (SUT), que l'on souhaite tester d'une part, et un processus de *conception* d'un **modèle** pour la génération de tests.

Puisque ces deux activités sont décorrélées, nous gardons l'assurance que le modèle n'est pas *influencé* par des détails provenant du développement ce qui nous donne une plus grande confiance dans les tests produits [PP04]. Les étapes manuelles (ou potentiellement manuelles) faites par un ingénieur de test sont représentées par un personnage au dessus des activités du processus.

La génération de tests est indépendante du processus de développement. En d'autres termes, quel que soit la méthode de développement utilisée, il est possible de recourir au test à partir de modèles. Ainsi, nous ne décrivons pas en détail ce processus de **développement** qui doit produire le **système sous test** à partir de la **spécification**.

Le développement du modèle est une activité nécessairement manuelle qui est une représentation abstraite de la *spécification*. Ce modèle peut être décrit dans une multitude de formalismes dont le choix dépend de l'aspect du système que l'ingénieur de test souhaite représenter et des outils de génération de tests utilisés.

Ce modèle est utilisé par un **générateur de tests** qui, par le biais de **critères de sélection**, permet de générer des **cas de tests abstraits**, c'est-à-dire, qu'ils s'appuient sur les éléments du modèle. Le générateur de tests peut prendre plusieurs formes. La première est l'écriture manuelle des tests qui requiert donc l'intervention de l'ingénieur de test pour leur écriture. La seconde est l'utilisation d'outils de génération de tests, plus ou moins automatiques. Ces outils, généralement dédiés à des formalismes précis, embarquent des critères associés au formalisme utilisé. Certains de ces outils requièrent une intervention de l'utilisateur, par exemple pour le choix de stratégies de génération.

Pour l'utilisation effective sur le système des **cas de tests abstraits** générés, il est nécessaire de les traduire par le biais d'une étape de **concrétisation** qui permet alors de faire le lien entre les éléments du modèle et les éléments concrets du système. La concrétisation est une étape encore largement manuelle et nécessite l'expertise de l'ingénieur tant au niveau de la connaissance du modèle que la connaissance des interfaces réelles du système. Cette étape réunit le processus de développement et celui de génération de tests.

Enfin, les **cas de tests concrets** (qui reposent donc sur les éléments du système réel) peuvent être exécutés sur le système sous test, en général par des scripts pour des logiciels simples ou sur des bancs de test pour des systèmes plus complexes. Le résultat de cette exécution, le **verdict** permet de décider si le système respecte ou non le modèle qui en a été fait. Celui-ci est établi via une relation de conformité entre le système sous test et le modèle qui le représente [Tre96a].

Un aspect intéressant du test à partir de modèles est une grande possibilité d'automatisation du processus de génération mais aussi de pouvoir raisonner de manière abstraite sur le système. Ainsi, le test à partir de modèles permet de se focaliser sur ce qui doit être implémenté dans le système, à savoir, les fonctionnalités, et non pas sur la manière dont ces fonctionnalités sont implémentées.

1.2 Problématique

Dans le cadre du test à partir de modèles, il existe plusieurs problématiques que nous pouvons adresser.

Dans le test en général, il y a plusieurs aspects du système à considérer. Nous en

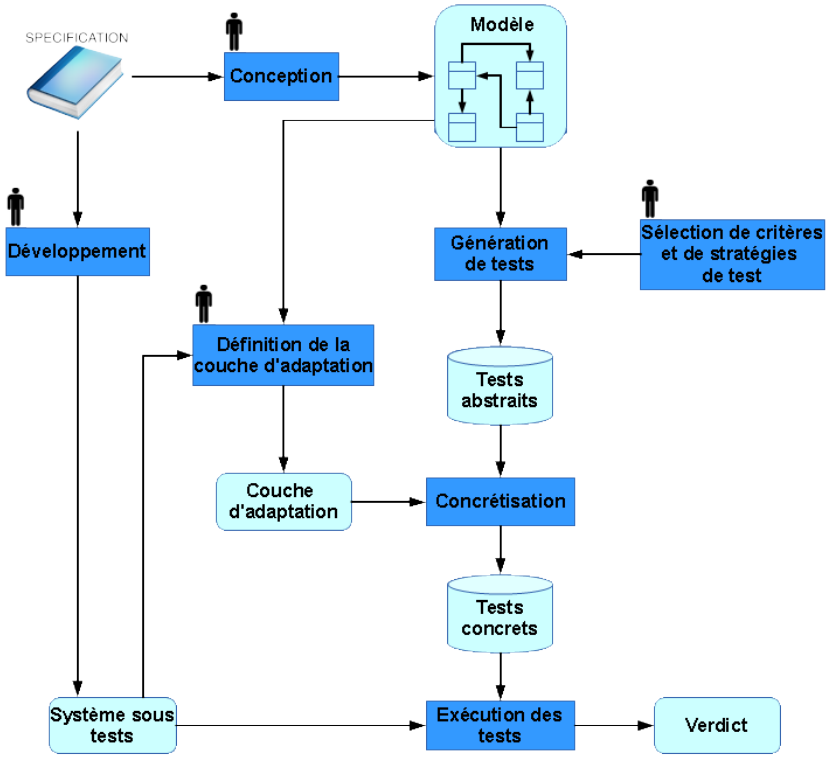


FIGURE 1.1 – Processus classique du *model-based testing*

distinguons deux. Le premier est le test structurel qui s'attache au test de la structure du système, telles que ses structures de données. Le second est un aspect global sur la dynamique du système, généralement représentée par le biais de propriétés temporelles ou d'automates. C'est sur ce deuxième aspect que nous nous focalisons, notamment par l'expression de propriétés temporelles sur le modèle. Ainsi, cette problématique nous permet de nous poser les questions suivantes et nous expliquons, pour chacune d'elles, les différents aspects qui en découlent.

Q I.“Comment exprimer des propriétés sur la dynamique du système ?”

L'ingénieur de test peut écrire des **scénarios de test** qui expriment certains aspects de la dynamique du système. La génération de tests à partir de ces scénarios permet de s'assurer de leur couverture. Cependant, l'écriture de ces scénarios est dépendante du savoir-faire de l'ingénieur de test. Pour remédier à cela, l'idée est de systématiser l'écriture de ces propriétés avec un langage *ad hoc*. De plus, l'expression de ces propriétés doit rester *simple et intuitive*. En effet, malgré la quantité de techniques et d'outils de génération automatique de tests qui existe actuellement, il est surprenant d'apprendre que dans l'industrie, le test logiciel reste une activité encore manuelle. La raison principale est un manque de formation des ingénieurs de test aux techniques et outils issus de la recherche. En effet, certaines techniques de tests demandent de solides connaissances dans des formalismes particuliers, qui ne sont pas nécessairement familiers pour des ingénieurs de tests.

Q II.“Dans quelle mesure peut-on tester le système vis-à-vis de propriétés sur sa dynamique ?”

Les propriétés en elles-mêmes ne suffisent pas pour s'assurer de la couverture de l'aspect dynamique du système. Ainsi, la sélection des tests est aussi un aspect important à prendre en compte dans la génération de tests. Le test exhaustif n'étant pas possible pour des raisons pratiques, il est nécessaire de choisir un sous-ensemble de tests *pertinents* qui soit représentatif des exécutions du système, et plus particulièrement celles permettant d'illustrer la propriété. Cette sélection se fait par le biais de **critères de sélection** qui sont l'expression formelle d'une certaine pertinence des tests vis-à-vis des propriétés. Ainsi, chaque critère exprime le savoir-faire de l'ingénieur de test (dans une certaine mesure).

Q III.“Dans quelle mesure est-il possible d'automatiser un tel processus ? ”

Le test à partir de modèles rend possible l'**automatisation** de la génération de tests. L'automatisation du processus de génération de tests permet un gain de temps dans le cycle de développement du logiciel. La diversité des outils de génération de tests à partir de spécifications formelles du système atteste de cette capacité d'automatisation. Ainsi, celle-ci devrait être possible dans le cadre du test à partir de propriétés fondées sur des modèles.

Ce manuscrit tente d'apporter des réponses à ces questions.

1.3 Contributions

Cette thèse apporte une réponse à ces problématiques en proposant un moyen, d'une part, d'exprimer de manière simple les aspects dynamiques du système et d'autre part,

d'en extraire automatiquement des tests.

1.3.1 Processus de tests à partir de propriété

La FIGURE 1.2 décrit le processus de génération de tests proposé aux utilisateurs. Parallèlement au processus classique de conception du modèle, nous décrivons un processus qui débute par la formalisation manuelle, à partir de la spécification, par un ingénieur validation des propriétés temporelles dans un langage *ad-hoc* (point (1)).

Suite à cela, nous associons à ce langage une sémantique qui nous permet de traduire ces propriétés sous la forme d'automates (point (2)). Les automates ont l'avantage d'avoir une longue histoire dans le domaine et sont au centre de plusieurs techniques de génération de tests. De plus, ils disposent d'une variété de critères de couverture qui peuvent servir d'inspiration pour des critères spécifiques à ces automates.

A partir de cette représentation à base d'automates et à l'aide de critère de couverture spécifiques choisis par l'ingénieur de test, nous générons des scénarios de tests (point (3)). Les scénarios permettent de représenter de manière abstraite des cas de test et permettent alors de ne spécifier, entre autres, que les étapes importantes des tests.

Les critères précédents permettent la couverture d'exécutions nominales du système, mais ne permettent pas d'illustrer nécessairement des cas relevant de la robustesse du système. Ainsi, le point (4) permet d'utiliser les automates de propriétés comme des propriétés de sûreté (*quelque chose de mauvais n'arrive jamais*) par le biais de mutations sur ceux-ci. La couverture de ces automates permet de provoquer ces exécutions potentiellement problématiques.

Ces scénarios sont ensuite dépliés pour générer des cas de tests abstraits (point (5)). Ce processus propose une étape optionnelle (point (6)) qui permet à un ingénieur de test de modifier les scénarios produits. Les scénarios générés automatiquement peuvent être parfois bien trop abstraits, ce qui peut alors entraîner un temps de calcul qui n'est pas acceptable par un ingénieur de test. Cette étape de modification manuelle permet à l'ingénieur de test de modifier le scénario pour accélérer le dépliage ou pour spécifier plus précisément des chemins trop abstraits.

Enfin, le point (7) représente le processus de mesure de couverture d'une suite de tests existante vis-à-vis des critères de sélection pour une propriété considérée. Cette étape permet d'évaluer la prise en compte des propriétés par une suite de tests quelconque. Cela permet de donner une indication à l'ingénieur de test de l'effort de test à produire pour la couverture de la propriété.

1.3.2 Synthèse des contributions

Dans ce document de thèse, nous décrivons nos cinq contributions. La première contribution concerne un **langage de scénarios** (résultat du point (2)) inspiré du langage décrit dans [Tis09] et la technique de dépliage combinatoire associée pour la génération de tests en prenant en compte les directives de dépliage (point (5)). Ce langage permet à l'ingénieur de test de visualiser le produit de la génération de scénarios avant le dépliage effectif, ce qui lui permet éventuellement de les modifier pour les simplifier ou les spécifier plus précisément (point(6)), accelérant ainsi le dépliage [CB11]. Cette contribution fait

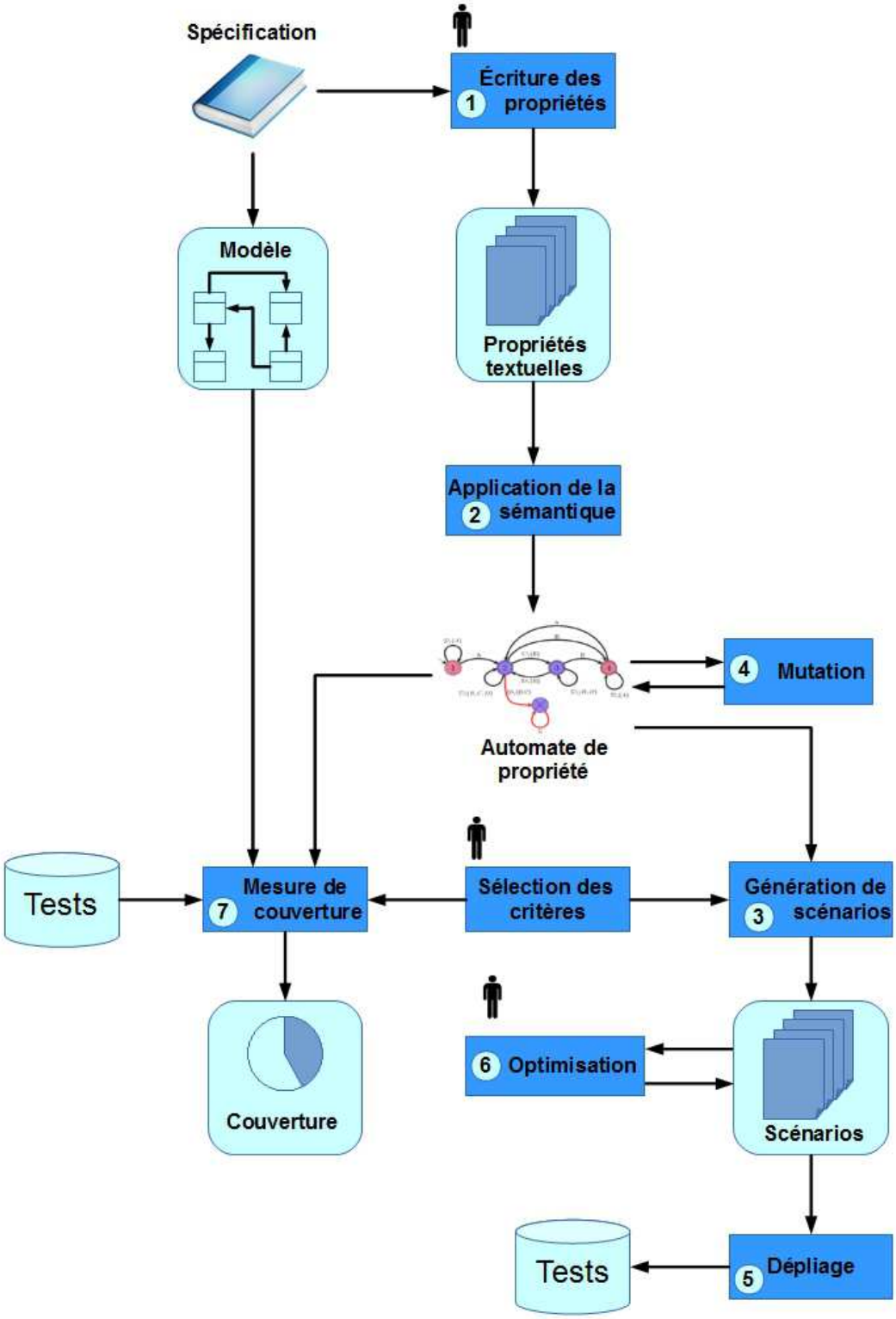


FIGURE 1.2 – Schéma du processus de génération de tests proposé

l'objet du Chapitre 4 et répond directement à la question Q I sur la prise en compte du savoir-faire de l'ingénieur pour la couverture des propriétés du système. Ce document présente une version améliorée du langage de scénarios décrit dans le livrable [CD10].

Notre deuxième contribution concerne le **langage d'expression de propriétés**. Nous nous sommes inspirés du langage décrit par Dwyer *et al.* dans [DAC99] pour une application à des modèles UML/OCL. Nous avons redéfini la sémantique de ces propriétés par des automates : chaque portée et motif est associé à un automate [CDJT11]. Un mécanisme de composition entre ces automates permet d'obtenir l'automate correspondant à la propriété. Cette représentation nous permet de définir des critères de mesure de couverture et de génération de tests pertinents, en fonction des besoins de tests de l'ingénieur de validation. Cette contribution répond à la problématique du test dynamique et de la simplicité d'utilisation par l'utilisation de patrons de propriété. Cette contribution couvre les points (1) et (2) du processus proposé et fait l'objet du Chapitre 5. De plus, elle répond directement à la question Q I sur l'expression des propriétés, mais en systématisant leur expression.

Notre troisième contribution est la **définition de critères de couverture** spécifiques aux automates de propriétés. Ils s'appuient en particulier sur les éléments et la structure de l'automate de propriété. Ces critères sont utilisés pour la génération de tests (point (3)) et pour la mesure de couverture d'une suite de tests existante vis-à-vis de la propriété donnée (point (7)). Ce document présente une version améliorée des travaux décrits dans [DC12]. Cette contribution répond à la problématique du choix des critères de sélection adressée par la question Q II et fait l'objet du Chapitre 6.

Notre quatrième contribution porte sur la **définition d'un critère de robustesse par le biais de mutations** d'automates (point (4)). Les mutations permettent d'exhiber des exécutions potentiellement dangereuses que nous cherchons justement à couvrir pour exercer le système dans des situations particulières. Cette contribution fait l'objet du Chapitre 7 et s'attache à répondre à la question Q II.

Enfin, notre cinquième contribution concerne la **génération de scénarios à partir des automates** (point (2)). L'application des critères de couverture de notre troisième contribution pour la génération à partir de l'automate nous permet de générer des scénarios qui représentent les séquences respectant ces critères. Cette contribution fait l'objet du Chapitre 8 et se rattache à la question Q II sur la définition de critères de sélection.

Ce processus permet d'apporter une réponse à la question de problématique Q III sur la capacité d'automatisation. En effet, notre approche propose une technique de génération de tests à partir de propriétés temporelles décrites dans le langage de la première contribution, étape qui est manuelle. La suite est automatisée, même s'il reste la possibilité pour l'ingénieur de tests d'intervenir à toutes les étapes du processus.

Ces différentes contributions font l'objet des chapitres de la **Partie II**.

1.4 Présentation du plan

Cette thèse se découpe en onze chapitres, regroupés en quatre parties.

La **Partie I** introduit le contexte et les problématiques adressées par ce travail. Il définit aussi l'état de l'art et les techniques de base sur lesquels ces travaux s'appuient.

Le **Chapitre 1** présente le contexte et les problématiques associés à ces travaux. Il présente succintement le processus mis en place pour adresser ces problèmes.

Le **Chapitre 2** présente l'état de l'art, le contexte scientifique global des différents domaines auxquels cette thèse se rattache. Ce chapitre donne une vue large sur les techniques de génération de tests selon plusieurs points de vues : à partir de modèles, à partir de scénarios et à partir de propriétés.

Le **Chapitre 3** présente la méthode de modélisation pour le test avec les langages UML/OCL. Nous présentons d'abord les sous-ensembles considérés des deux langages puis nous présentons l'exemple fil rouge qui sert de base pour illustrer la démarche de ces travaux.

La **Partie II** qui introduit les contributions de cette thèse est composée des cinq chapitres suivants :

Le **Chapitre 4** décrit le langage de scénarios utilisé pour la représentation de séquences de test, ainsi que la manière dont ces scénarios sont exploités pour la génération de cas de tests abstraits.

Le **Chapitre 5** décrit le langage de propriétés temporelles utilisé pour formaliser de telles propriétés, ainsi que sa sémantique à base d'automates.

Le **Chapitre 6** présente les critères de couverture nominaux spécifiques aux automates de propriété, ainsi que leur utilisation dans la mesure de couverture d'une suite de tests existante.

Le **Chapitre 7** présente les différents opérateurs de mutation et le critère de couverture d'automate associé à ces mutations.

Le **Chapitre 8** présente la génération de scénarios de tests vis-à-vis des critères de couverture définis dans les deux chapitres précédents.

La **Partie III** porte sur l'évaluation expérimentale de cette approche ainsi que du prototype logiciel implémentant les méthodes issues de ces travaux.

Le **Chapitre 10** positionne ce travail dans le cadre du projet ANR TASCCC et de l'étude de cas Global Platform. Ce chapitre présente le prototype logiciel issu du projet et le retour des évaluateurs industriels sur la pertinence de la démarche proposée dans TASCCC.

Enfin, la **Partie IV** présente les conclusions ainsi que les perspectives issues de ces travaux regroupées dans le **Chapitre 11**.

Chapitre 2

Etat de l'art : Test à partir de modèles

Sommaire

2.1 Introduction au test 14
2.1.1 Classification des techniques de tests 14
2.1.2 Critères de sélection 17
2.2 Modèles comportementaux 18
2.2.1 Paradigmes de modélisation 18
2.2.2 Formalismes de modèles pre/post 19
2.3 Critères statiques 20
2.3.1 Couverture du graphe de flot de contrôle 21
2.3.2 Couverture de flot de données 21
2.3.3 Couverture des états/transitions 23
2.3.4 Couverture de données 25
2.4 Critères dynamiques 26
2.4.1 Approches par scénarios 26
2.4.2 Approches par propriétés sur le système 28
2.4.3 Approches par modèles de fautes 31
2.5 Synthèse 32

Nous présentons dans ce chapitre une vue générale des travaux qui se rapprochent de ceux exposés dans cette thèse. Nos travaux se placent dans un contexte de test à partir de modèles comportementaux. Dans nos travaux, nous considérons des modèles de type pre/post et en particulier UML/OCL.

Nous présentons dans la section 2.1 une vue générale sur l'activité de test, les différents types de tests et les critères de sélection, notamment utilisés pour qualifier la pertinence des tests. Dans la section 2.2, nous présentons un panorama rapide des différents paradigmes de modélisation, puis nous approfondissons la présentation des modèles comportementaux dans un paradigme pre/post condition. Ensuite, la section 2.3 présente les critères de sélection statiques, pour l'aspect structurel,et la section 2.4 présente les critères dynamiques, pour l'aspect temporel du système.

2.1 Introduction au test

Dans un cadre général, l'activité de *test* est l'évaluation d'un logiciel qui vise à s'assurer de sa qualité en mettant en œuvre différentes techniques permettant d'identifier d'éventuelles erreurs dans le produit testé. La qualité recherchée dans le test est l'assurance qu'un logiciel, dans ce cas appelé *système sous test*, se comporte de la manière attendue. Ainsi, le but de l'activité de test est la découverte de *fautes* dans un système. Une faute est détectée lorsque le résultat de l'exécution du test sur le système ne correspond pas au résultat attendu par le test.

Dans [UL07], les auteurs attribuent quatre caractéristiques à l'activité de test :

- Le test est une vérification *dynamique* du système dans le sens où le test stimule les interfaces du système avec des valeurs particulières. A l'inverse, la vérification *statique* du système est focalisée sur la structure du code source du système et ne nécessite pas son exécution.
- Le test a un aspect *incomplet* dans le sens où, pour des raisons pratiques, il n'est pas possible de tester exhaustivement toutes les exécutions de la plupart des systèmes. En effet, les paramètres d'entrée d'une opération peuvent prendre leurs valeurs dans des domaines potentiellement grands. A cela s'ajoutent des valeurs invalides ou inattendues pour chaque paramètre, ce qui conduit inévitablement à une combinatoire importante des valuations de paramètres, et donc à autant de tests à générer. Ainsi, il est nécessaire de choisir un sous-ensemble fini, mais représentatif, des séquences du système afin de pouvoir les exécuter dans un temps convenable.
- Le test doit faire l'objet d'une *sélection* qui repose en partie sur l'expertise d'un ingénieur de test. En effet, cette sélection consiste à trouver des données de tests qui permettent d'illustrer un *ensemble représentatif* des exécutions du système. Cela permet de réduire le nombre de tests nécessaires, sans pour autant impacter la qualité de la suite de tests.
- Enfin, le test doit permettre d'observer une défaillance en comparant le résultat de l'exécution du test sur le système et le *verdict* de l'exécution des tests. Une différence entre ces deux résultats indique une faute sur le système.

Il existe plusieurs techniques de génération de tests, dont la plus ancienne est l'écriture manuelle des cas de test qui est encore majoritairement utilisée. Mais cette tâche peut s'avérer délicate car elle dépend grandement du savoir-faire de l'ingénieur de test.

Enfin, les techniques en elles-mêmes ne permettent pas d'identifier les tests qui peuvent être pertinents pour un modèle donné. Pour cela, les approches de génération de tests utilisent des *critères de sélection* qui permettent de filtrer les tests produits. De plus, les suites de tests qui couvrent ces critères permettent assurent une certaine pertinence à la suite de tests vis-à-vis de ces critères.

2.1.1 Classification des techniques de tests

Dans [Tre04], l'auteur identifie trois axes pour catégoriser les différentes techniques de tests. Cette catégorisation par axes est représentée par la Figure 2.1. Un premier axe, *Support de conception*, représente les deux supports sur lesquels nous pouvons nous appuyer pour l'écriture des tests : soit en travaillant à partir du code source (boîte blanche),

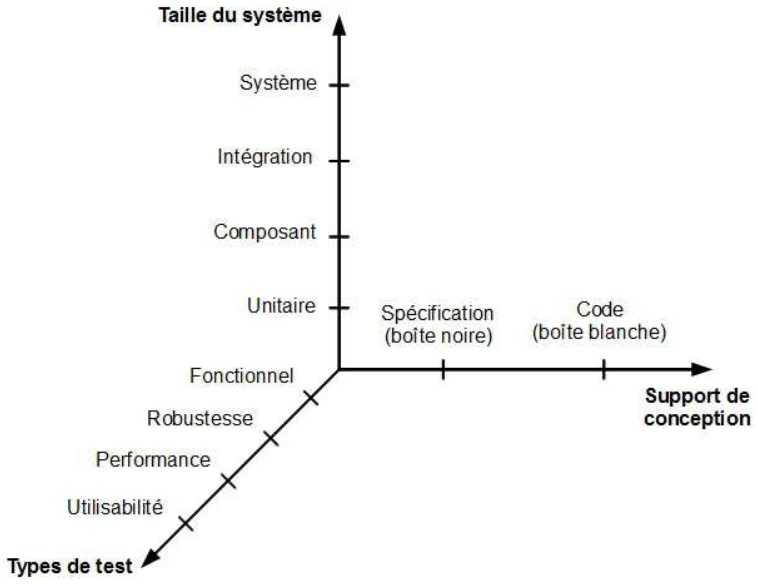


FIGURE 2.1 – Les différents axes de classification des techniques de tests

soit en travaillant à partir de la spécification (boîte noire). Le second axe, *Types de test*, représente les différents types de tests, chacun s'attachant au test d'une caractéristique du système. Enfin, le troisième axe, *Taille du système*, permet de classer les tests en fonction du niveau de détail du système testé. Nous présentons maintenant plus en détail ces trois axes.

Test boîte blanche vs. test boîte noire

Dans [MS04], l'auteur identifie deux stratégies principales, en fonction du support d'où sont dérivées les informations pour identifier les données de test : le test *boîte blanche* et le test *boîte noire*.

Le test *boîte blanche* dérive les données de tests à partir d'une description du fonctionnement interne du programme, tel que son code source. Ces tests permettent de satisfaire des critères de couverture structurelle sur le code du système. Par exemple, le critère *statement coverage* vise à couvrir chaque instruction du code source au moins une fois, le critère *decision coverage* vise à couvrir les différentes branches de nœuds de décision (par exemple, *if-then* ou *switch* en Java) ou encore le critère *path coverage*, qui vise à couvrir tous les chemins du graphe de flot de contrôle du code source.

L'outil Pathcrawler [2] [WMMR05] est un plugin pour la plateforme Frama-C [CKK+12] dédié à l'analyse de code source C et développé par le CEA-LIST [3] et l'INRIA-Saclay [4]. Son objectif est de déterminer des valeurs de paramètres de fonctions C pour maximiser la couverture des chemins du cadre de la fonction (*path coverage*). Le code source est

2. Site Pathcrawler Online : `http://pathcrawler-online.com` (2013)
3. Site CEA-LIST : `http://www-list.cea.fr`
4. Site INRIA-Saclay : `http://www.inria/fr/centre/saclay` (2013)

d'abord instrumenté afin de récupérer les contraintes de chemins lors de l'exécution. Ensuite, elle est exécutée avec des valeurs de paramètres quelconques (mais valides vis-à-vis du typage de la fonction). Son exécution permet de parcourir un chemin et d'identifier, via l'instrumentation, les contraintes associées à ce chemin. Ces contraintes sont ensuite utilisées pour identifier les valeurs de paramètres qui permettent d'emprunter un autre chemin. Le système de contraintes est résolu pour trouver des valeurs de paramètres qui sont ensuite réinjectées pour exécuter la fonction. Ce processus se poursuit jusqu'à ce que tous les chemins faisables ou tous les k-chemins, dans le cadre de programmes itératifs, soient couverts.

L'outil KLEE [5] [CDE08] (successeur d'EXE [CGP+08]) permet de déterminer des valeurs de paramètres d'opérations afin de maximiser la couverture du code source (avec un critère *statement coverage*) et de détecter pour chaque instruction potentiellement dangereuse (déréférencement de variables, assertions, ...) s'il existe des valeurs de paramètre qui risquent de causer des fautes. Cet outil s'appuie sur une exécution symbolique du système. Chaque instruction sur un chemin donne lieu à une contrainte. Ainsi, à la fin de l'opération, le chemin d'exécution de l'opération correspond à un ensemble de contraintes. Pour chacun de ces ensembles, KLEE calcule une solution (une valuation des paramètres de l'opération) et produit alors un test qui, à son exécution, activera le chemin dont il est issu.

L'outil Euclide [6] [Got09] permet la vérification de systèmes critiques en C. Il propose trois applications distinctes : la génération de données de tests, la génération de contre-exemples et la preuve partielle du programme. La génération de données de tests, pour la couverture structurelle du programme C considéré, nécessite le programme et un point à atteindre dans ce programme. En s'appuyant sur une technique mélangeant du symbolique et du concret (approche concolique), Euclide permet de déterminer s'il existe des données de tests permettant d'atteindre le point du programme spécifié, sinon, il est classé comme étant *inatteignable*. La seconde et la troisième application permettent, à partir de propriétés de sûreté décrites en ACSL [BFM+09] (ANSI/ISO C Specification Language) sous la forme de pre/postcondition, d'identifier des contre-exemples vis-à-vis de ces propriétés s'ils existent, ou de prouver que les propriétés sont vérifiées sur le programme.

Le test *boîte noire* dérive les données de test à partir d'une spécification formalisée. C'est cette technique qui est mise en œuvre dans le test à partir de modèles où le modèle est une représentation abstraite, extrait de la spécification. Cette représentation est utilisée de plusieurs manières et notamment pour la génération de tests.

Types de tests

Les tests peuvent aussi être catégorisés en fonction des différents aspects du système qu'ils visent à exercer. Le type de test fréquemment rencontré est le *test fonctionnel* qui vise à identifier des fautes dans les fonctionnalités individuelles du système.

Un autre type de test, le *test de robustesse*, vise à trouver des fautes dans le système en le mettant, ou en tentant de le mettre, dans des situations extrêmes. Ces situations

5. Site KLEE : `http://ccadar.github.io/klee/` (2013)
6. Site Euclide : `http://euclide.gforge.inria.fr/` (2013)

exceptionnelles peuvent prendre la forme, au niveau fonctionnel, de valeurs d'entrée non-prévues, ou plus largement, de problèmes matériels tels qu'une perte de connexion réseau.

Le *test de performance* permet de mettre l'accent sur des caractéristiques telles que la vitesse d'exécution ou sa fiabilité dans des conditions d'utilisation extrêmes. Cet aspect prend en compte différentes vues sur ce qu'est la performance comme la réaction du système face à une forte sollicitation (*test de stress*) ou son comportement après une activité prolongée (*test d'endurance*).

Enfin, une dernière caractéristique est l'utilisabilité du système, par le biais du *test d'utilisabilité*, qui vise en particulier à détecter des problèmes au niveau de l'interface utilisateur et de l'ergonomie du système.

Taille du système et niveau de détail

Un système peut être vu à différents niveaux de détail. Ainsi, au plus bas niveau, nous avons le *test unitaire* qui se focalise sur le test individuel des plus petites parties du système telles que les fonctions ou les procédures. Ces parties sont testées indépendamment les unes des autres. Le niveau supérieur, le *test de composants*, concerne des sous-systèmes d'un système et le test individuel de chaque composant. Le troisième niveau, le *test d'intégration*, se focalise sur les intéractions entre les différents composants d'un système. Enfin, le *test système* considère le système dans sa globalité en se fondant sur les exigences de la spécification du système.

2.1.2 Critères de sélection

L'un des aspects cruciaux de l'activité de test est la sélection des cas de tests, car le test exhaustif n'est en pratique pas possible. Afin d'effectuer cette sélection, le test à partir de modèle s'appuie sur des critères de sélection. Ces critères visent la couverture des modèles qui permettent de détecter certains types de faute.

De plus, ces critères peuvent être utilisés comme guide pour la génération de tests. Ainsi, au lieu de se contenter d'observer ce que l'on attend du système, nous pouvons expliciter au générateur ce que l'on attend des tests produits. Ainsi, les critères nous assurent alors que la suite de tests produite respecte ce critère sur le modèle.

Les critères peuvent être répartis en deux catégories. Les critères *statiques*, présentés dans la section 2.3, qui se focalisent sur la structure du système, et les critères *dynamiques*, présentés à la section 2.4 qui visent la couverture de l'aspect dynamique du système.

Ces critères peuvent être combinés afin de tirer le meilleur parti de chacun d'eux. C'est le cas de notre approche qui utilise des critères *statiques* sur les automates et deux critères *dynamiques*. En effet, nous utilisons des critères dynamiques pour générer des automates de propriété sur lesquels nous appliquons des critères de couverture statiques.

Avant d'étudier plus en profondeur les critères *statiques* et *dynamiques*, nous présentons dans la section 2.2 une description des modèles comportementaux considérés qui sont la base de notre approche.

2.2 Modèles comportementaux

Les modèles comportementaux permettent de représenter le comportement attendu d'un système vis-à-vis de stimulations externes à celui-ci. Cependant, lors de la modélisation de ces systèmes, le choix du formalisme par l'ingénieur est fonction de l'aspect du système qu'il souhaite modéliser. Ainsi, il existe plusieurs formalismes de modélisation que nous pouvons catégoriser en fonction de la représentation du système qu'ils privilégient.

2.2.1 Paradigmes de modélisation

Il existe plusieurs manières de représenter un système, dont le choix dépend de l'aspect à modéliser. Ainsi, certains formalismes favorisent une représentation orientée sur les données du système et leur évolution, alors que d'autres privilégient une représentation orientée sur les évènements qui permettent de faire évoluer le système. Dans [Lam00], van Lamsweerde propose une classification de ces formalismes en différents paradigmes.

Une spécification par **cas d'utilisation** (*history-based specification*) représente le système par un ensemble de comportements possibles admis par le système. Ces comportements sont en général représentés par des propriétés temporelles sur les objets qui composent le système par le biais d'opérateurs temporels sur les états présents, passés et futurs du système. Il existe plusieurs formalismes qui permettent d'exprimer ces propriétés que nous pouvons catégoriser en fonction du type de temps utilisé. Ainsi, le temps peut être linéaire [Pnu77] ou arborescent [EH86]. Il peut être discret [MP92] ou continu [HRR91]. Enfin, d'autres se focalisent sur des instants [MP92] ou sur des intervalles de temps [MRK+97].

Une spécification **algébrique** permet de modéliser le système comme un ensemble de de fonctions mathématiques (premier ordre ou ordre supérieur). Le langage ML [Mar95], un langage de programmation fonctionnelle et utilisé dans l'assistant de preuve interactif HOL, est un représentant de ce paradigme.

Une spécification **opérationnelle** représente un système comme un ensemble de processus. Ces spécifications sont généralement utilisées pour la représentation de protocoles ou de systèmes distribués. Les algèbres de processus telles que CSP [Hoa85] et CCS [Mil82, BHR84] ou encore les langages LOTOS [BB87] et LUSTRE [CPHP87] sont des représentants de ce paradigme.

Une spécification à base de **systèmes de transitions** (*transition-based specification*) permet de représenter un système avec les transitions entre les états du systèmes. Le système, qui est alors représenté par un système d'états-transitions, est alors vu comme un ensemble de transitions. Chaque transition donne, à partir d'un état d'entrée et d'un évènement déclencheur, le nouvel état du système. Ce paradigme regroupe par exemple les machines à états finies (FSM [Gil62]), les systèmes de transitions étiquetés (LTS [Tre96a]), les automates à entrées/sorties (IOLTS [LT89]) et les diagrammes d'états-transitions (Statemate [HLN+88] et les statecharts UML [RJB04]).

Enfin, les spécifications fondées sur les **états du système** (*state-based specification* ou formalismes pre/post) considèrent un système comme un ensemble de variables qui représentent l'état interne du système et un ensemble d'opérations qui permettent de modifier ces variables, entraînant une évolution de l'état du système. Chaque opération

est définie par une *précondition* et une *postcondition*. La précondition exprime la plus faible condition pour l'activation de l'opération, par exemple avec des contraintes sur les valeurs de paramètres d'entrée ou sur les variables du système. La *postcondition* permet de caractériser l'état du système après l'application de l'opération. La notation Z [Spi92], la méthode B [Abr96] et UML/OCL [RJB04] sont des formalismes qui appartiennent à ce paradigme.

Nos travaux sont fondés sur l'utilisation d'UML (Unified Modeling Language) couplé à OCL [WK03] (Object Constraint Language) pour la représentation du modèle comportemental, ce qui implique que nous nous plaçons dans le paradigme de modélisation pre/post. Nous décrivons maintenant plus en détail ce paradigme et certains formalismes qui le représente.

2.2.2 Formalismes de modèles pre/post

Nous présentons ici trois grandes familles de formalismes de modélisation pre/post : les formalismes par contrats, par substitutions et graphiques. Bien qu'elles appartiennent au paradigme pre/post, elles sont cependant très différentes dans la manière d'aborder la représentation de l'état du système. Nous décrivons maintenant ces trois familles de langage pour des modélisations pre/post.

Formalismes par contrat

Les formalismes par contrat permettent de spécifier les invariants d'une opération, ainsi que ses préconditions d'activation d'une opération et de ses différentes sorties (post-condition). Parmi ces sorties, il peut s'agir de la valeur de retour de l'opération, de la modification d'un attribut d'un objet ou de la levée d'un comportement exceptionnel.

Le langage JML [LBR99] (Java Modeling Language) en est un exemple. Il s'agit d'un langage de spécification pour Java, utilisé pour annoter des classes et les différentes méthodes d'une classe. Ces annotations, écrites en tant que commentaires spéciaux dans une classe Java, permet de spécifier le comportement d'une méthode ou l'invariant d'une classe, donnant ainsi une représentation formelle de leur comportement. Ainsi, JML est utilisé pour la programmation par contrat pour Java.

Le langage Spec# [BLS05] est aussi un exemple de langage de spécification par contrat. Il étend la syntaxe du langage C# pour apporter un moyen de spécifier formellement les fonctions et méthodes d'un programme C#. Les constructions utilisées par Spec# sont très semblables à celles utilisées par JML.

Formalismes par substitution

Cette famille de formalismes regroupe les langages qui reposent sur des substitutions de variables, c'est-à-dire, le remplacement de la valeur d'une variable par une autre valeur. Ces substitutions peuvent éventuellement être gardées, ce qui permet d'effectuer ces substitutions uniquement si la condition représentée par la garde est satisfaite.

La méthode B [Abr96] est un exemple de formalisme par substitution. Elle est une méthode formelle de modélisation de systèmes principalement fondée sur la théorie des

ensembles. Une des spécificités de la méthode B est la possibilité de définir un modèle abstrait qui peut ensuite être raffiné par étapes successives vers un modèle plus concret. Le modèle le plus concret, appelé *implémentation*, peut être traduit dans un langage de programmation classique. Cette approche est couplée avec une activité de preuve qui permet d'assurer la cohérence du modèle vis-à-vis du modèle abstrait. Cette approche a été étendue pour modéliser des systèmes à base d'évènements (au lieu d'opérations en B classiques) donnant lieu au B *évènementiel* [Abr10]. Ces deux langages sont implémentés dans l'outil Atelier B [7]. Enfin, l'outil Rodin, issu du projet européen RODIN [8] utilise le B#, une évolution du B évènementiel, qui prend en compte des éléments de la notation Z [Spi92].

Formalisme graphique

Les formalismes graphiques permettent de représenter un système à partir d'une notation graphique, notamment par l'utilisation de diagrammes. Ces diagrammes mettent en relation les différentes entités du système selon plusieurs vues.

Le langage OCL [WK03] (Object Constraint Language) est un exemple de langage de cette famille et il est généralement couplé avec le langage graphique UML [RJB04] (Unified Modeling Language). Dans la plupart des cas, les diagrammes UML (notamment, le diagramme de classes) du modèle sont augmentés par l'ajout de contraintes décrites dans le langage OCL. Ce langage permet l'ajout de contraintes sur des entités du modèle (classes, opérations, transitions, ...). Les contraintes sont formées d'une *précondition* qui établit la condition minimale pour l'activation de la *postcondition* qui spécifie l'effet de la contrainte sur le système.

Ainsi, le couple UML/OCL permet la modélisation de systèmes pre/post, les diagrammes UML représentant les structures de données du système et les contraintes OCL représentant ses aspects évolutifs. Nous détaillons dans le Chapitre 3 une variante de ce formalisme de modélisation [Gra08], utilisé par l'outil CertifyIt de Smartesting [9].

Plus récemment, le langage ALF (Action Language for Foundational UML) est un langage d'action pour UML, standardisé par l'OMG [10] [Obj13], dans le but de concevoir des modèle exécutables. Sa syntaxe est très proche du langage Java mais intègre aussi des constructions OCL telles que les opérations sur les collections (par exemple, *exists*, *forAll* et *collect*).

2.3 Critères statiques

Les critères de sélection *statiques* sont des critères qui reposent sur la structure du modèle. Leur objectif est d'assurer un besoin de couverture de certains éléments du modèle sans s'appuyer sur des besoins de test spécifiques.

7. Site Atelier B : `http://www.atelierb.eu` (2013)
8. Site du projet RODIN : `http://rodin.cs.ncl.ac.uk` (2013)
9. Site Smartesting : `http://www.smartesting.com` (2013)
10. Site OMG pour ALF : `http://www.omg.org/spec/ALF/`

2.3.1 Couverture du graphe de flot de contrôle

Ces critères font référence aux critères de couverture de code source dans une optique test boîte blanche. Les *conditions* sont des propositions booléennes atomiques et les *décisions* sont des expression booléennes composées de une ou plusieurs conditions liées par au moins un opérateur booléen. Nous présentons ces différents critères.

Statement coverage (SC). Chaque instruction doit être couverte au moins une fois.

Decision coverage (DC). Chaque décision doit prendre tous les résultats possibles au moins une fois.

Condition coverage (CC). Chaque condition doit prendre toutes les valeurs possibles au moins une fois.

Condition/decision coverage (C/DC). Ce critère combine les critères *decision coverage* et *condition coverage*. Chaque condition et décision doit prendre toutes les valuations possibles au moins une fois.

Full predicate coverage (FPC). Introduit dans [OXL99], ce critère nécessite que chaque condition soit fixée à *vrai* ou *faux* dans une décision et doit montrer qu'il y existe une dépendance entre la valeur de la condition c considérée et la décision d qui la contient. Ainsi, cela signifie que $d \Leftrightarrow c$ ou $d \Leftrightarrow \neg c$.

Modified condition/decision coverage (MC/DC). Chaque décision prend tous les résultats possibles et l'effet de chaque condition d'une décision peut être montré (en la faisant varier et en fixant toutes les autres) [RTC82].

Multiple condition coverage (MCC). Chaque combinaison possible de conditions dans une décision est considérée au moins une fois.

De tous ces critères, *Multiple condition coverage* est celui qui assure une meilleure couverture, mais au prix d'une explosion combinatoire.

L'outil BZ-TT [LPU02] (BZ-Testing Tools) implémente quatre de ces critères (DC, C/DC, MC/DC et MCC) pour la génération de tests à partir de modèles B ou Z. L'outil, par l'application d'algorithmes spécifiques à chaque critère, permet d'extraire des valuations de paramètres d'opérations qui permettent de générer des cas de test vérifiant le critère dont ils sont issus.

L'approche proposée par l'outil CertifyIt détermine ses cibles de test par l'application du critère *condition/decision coverage* [Gra08] sur les différents comportements du modèle. Ainsi, chaque valuation de condition donne lieu à une cible de test. Ensuite, par l'utilisation d'un moteur symbolique, CertifyIt tente d'atteindre la cible de test à partir d'un état final du modèle. Nous reviendrons plus en détail sur cette technique de génération au cours du chapitre suivant.

2.3.2 Couverture de flot de données

Ces critères se fondent sur les interaction entre les différentes variables et paramètres du système [FW88]. Ainsi, au cours d'une opération, une variable peut être soit *définie* lorsqu'une valeur lui est assignée ou soit *utilisée* lorsqu'elle est utilisée dans un prédicat ou pour calculer la valeur d'une autre variable. De plus, ces critères se fondent sur la couverture de paires *def-use*. Une paire *def-use* est notée $(\mathrm{d}_v, \mathrm{u}_v)$, où d_v est une instruction

FIGURE 2.2 – Hiérarchie des critères de couverture de flot de données

de définition de la variable v et u_v est une instruction d'utilisation de la variable v. La FIGURE 2.2 donne la hiérarchie entre les différents critères de couverture.

All-P-uses coverage. Ce critère nécessite la couverture des instructions qui utilisent une variable donnée dans une condition (p-utilisation). De plus, il ne doit pas y avoir de définition de la variable considérée avant la p-utilisation.

All-C-uses coverage. Ce critère nécessite la couverture des instructions qui utilisent une variable donnée dans le calcul de la valeur (définition) d'une autre variable (c-utilisation). De plus, il ne doit pas y avoir de définition de la variable considérée avant la c-utilisation.

All-defs coverage. Ce critère nécessite au moins un chemin qui permet d'activer chaque *définition* d'une variable donnée. Pour chaque définition de variable ciblée, il ne doit pas y avoir d'autre définition avant la définition considérée.

De plus, il ne doit pas y avoir de définition de la variable considérée avant la définition
.

All-C-uses/Some-P-uses coverage. Ce critère requiert tous les chemins dans le graphe def-use qui permettent une c-utilisation d'une variable. Si un chemin n'admet pas de c-utilisation d'une variable, alors une P-utilisation suffit. Ce critère privilégie la couverture des c-utilisations sur les p-utilisations.

All-P-uses/Some-C-uses coverage. Ce critère requiert tous les chemins dans le graphe def-use qui permettent une c-utilisation d'une variable. Si un chemin n'admet pas de c-utilisation d'une variable, alors une p-utilisation suffit. Ce critère est similaire au précédent mais privilégie les p-utilisations.

All-uses coverage. Ce critère nécessite la couverture de toutes les paires *def-use*. Il a pour but de montrer la faisabilité de chaque *utilisation* après n'importe quelle *définition* d'une même variable donnée.

All-DU-paths coverage. Ce critère nécessite la couverture de toutes les paires *def-use* et de tous les chemins sans boucles séparant la définition et l'utilisation de la variable. Ce critère n'est en pratique pas utilisable.

Dans [USW00], les auteurs utilisent ces critères pour la couverture de modèles décrits en SDL. Ces modèles sont ensuite représentés en tant que graphes de flots de messages étendus (EMFG) qui permettent d'exposer les dépendances de contrôle et de données dans un processus. Les auteurs utilisent ensuite les critères de couverture *all-uses* et *IO-df-chains*. Les séquences permettant de couvrir ces critères deviennent des cas de tests.

L'approche décrite dans [HU00] reprend les travaux précédents mais pour des systèmes multi-processus. A partir d'un modèle SDL [ITU02] (Specification and Description Language), décrit comme une machine à états finie (EFSM), les auteurs appliquent le critère *all-uses* sur le graphe pour en extraire les tests pour chaque configuration du graphe. Ces tests représentent un ensemble de séquences de stimuli à travers tous les processus permettant d'atteindre la configuration attendue.

Dans [KHBC99], les auteurs proposent une technique de génération de tests à partir de modèles UML en utilisant des critères de couverture sur les flots de données. Dans un

FIGURE 2.3 – Exemple d'automate simple

FIGURE 2.4 – Exemple de système à plusieurs automates

premier temps, les auteurs transforment le diagramme d'états-transitions en une machine à nombre d'états finis étendue (EFSM). Dans ce graphe, ils identifient les variables qui sont définies et utilisées à chaque état. l'EFSM est ensuite transformé en graphe de flot de contrôle avec l'information des définitions/utilisations attachées à chaque nœud du graphe. Enfin, les tests sont générés par l'application de critères sur les flots de données tels que *all-defs*, *all-uses* et *all-def-use-paths*.

2.3.3 Couverture des états/transitions

Ces critères sont principalement utilisés pour les modèles à base de transitions telles que les systèmes de transitions étiquetées (LTS), dont la FIGURE 2.3 est un exemple, ou les statecharts UML. De plus, ces notations permettent l'utilisation en parallèle de plusieurs systèmes tels que montrés dans la FIGURE 2.4. Ainsi, à un certain moment, un système peut avoir plusieurs états actifs simultanément, un par automate mis en parallèle.

Nous définissons maintenant les critères de couverture des états/transitions, du plus faible au plus fort.

all-states coverage. Chaque état du système doit être couvert au moins une fois. Dans l'exemple de la FIGURE 2.3, l'ensemble des états à couvrir est {1, 2, 3}.

all-configurations coverage. Chaque configuration du système doit être couverte au moins une fois. Certaines notations autorisent l'utilisation en parallèle de plusieurs systèmes. Ainsi, à un certains moments, il peut y avoir plusieurs états actifs simultanément, un par automate mis en parallèle. Dans ce cas, une *configuration* correspond aux différentes combinaisons d'états actifs dans le système. Par exemple, dans le système présenté en FIGURE 2.4, il existe deux états possibles pour chacun des automates. Ainsi l'ensemble des configurations est défini par le produit cartésien des états {(1,3),(1,4),(2,3),(2,4)}.

all-transitions coverage. Chaque transition du système doit être couverte au moins une fois. Dans l'exemple d'automate FIGURE 2.3, l'ensemble des transitions à couvrir est $\{1\xrightarrow{A}2,\ 1\xrightarrow{B}2,\ 2\xrightarrow{C}3,\ 3\xrightarrow{D}2,\ 3\xrightarrow{E}3\}$.

all-transition-pairs coverage. Chaque paire adjacente de transitions doit être couverte au moins une fois. Il s'agit de l'ensemble des produits cartésiens entre les transitions entrantes et les transitions sortantes à chaque état. Dans l'exemple FIGURE 2.3, les paires de transitions à couvrir sur l'état 2 est l'ensemble de paires $\{(1\xrightarrow{A}2,\ 2\xrightarrow{C}3),\ (1\xrightarrow{B}2,\ 2\xrightarrow{C}3)\}$.

all-loop-free coverage. Tous les chemins qui ne repassent pas par un état (ou une configuration) déjà traversé doivent être couverts. Ainsi, dans l'exemple de la FIGURE 2.3, les chemins $1\xrightarrow{A}2\,;2\xrightarrow{C}3$ et $1\xrightarrow{B}2\,;2\xrightarrow{C}3$ sont les seuls sans boucle. Les chemins ne prennent pas en compte la transition réflexive $3\xrightarrow{E}3$, car elle forme une boucle à elle seule.

all-one-loop coverage. Tous les chemins qui permettent au plus une boucle doivent être couverts. De plus, tous les préfixes de chaque boucle doivent être couverts. Ainsi, les chemins $1\xrightarrow{A}2\,;2\xrightarrow{C}3\,;3\xrightarrow{E}3$, $1\xrightarrow{A}2\,;2\xrightarrow{C}3\,;3\xrightarrow{D}2$, $1\xrightarrow{B}2\,;2\xrightarrow{C}3\,;3\xrightarrow{E}3$ et $1\xrightarrow{B}2\,;2\xrightarrow{C}3\,;3\xrightarrow{D}2$ de la FIGURE 2.3 permettent une boucle (soit avec la transition reflexive, soit avec le retour

sur l'état 2) et satisfont donc le critère. Les chemins $1\xrightarrow{A}2\,;2\xrightarrow{C}3$ et $1\xrightarrow{A}2\,;2\xrightarrow{C}3$ ne forment pas de boucles et satisfont donc le critère. Enfin, le chemin $1\xrightarrow{A}2\,;2\xrightarrow{C}3\,;3\xrightarrow{E}3\,;3\xrightarrow{D}2$ n'est pas considéré car il permet deux boucles.

all-round-trips coverage. Ce critère, proposé dans [Bin99], est similaire au critère précédent car il vise à couvrir au plus une boucle de l'automate, mais est plus faible que le *all-one-loop* car il ne cherche pas à couvrir tous les préfixes de la boucle. Ainsi, dans notre exemple FIGURE 2.3, là où le critère précédent considère quatre chemins, ce critère ne nécessite que deux chemins, un par boucle. Ainsi, les chemins $1\xrightarrow{A}2\,;2\xrightarrow{C}3\,;3\xrightarrow{E}3$ et $1\xrightarrow{A}2\,;2\xrightarrow{C}3\,;3\xrightarrow{D}2$ sont suffisants pour couvrir ce critère.

all-paths coverage. Ce critère vise tous les chemins de l'automate. C'est le critère de couverture exhaustif des chemins, mais puisqu'il existe potentiellement une infinité de chemins dans un automate, ce critère n'est en pratique pas applicable. Ainsi, dans notre exemple d'automate FIGURE 2.3, nous pouvons prendre le chemin $1\xrightarrow{A}2\,;2\xrightarrow{C}3$ qui mène à l'état final. Cependant, à partir de cet état, il est possible d'ajouter la transition $3\xrightarrow{E}3$ à ce chemin, créant un nouveau chemin et ainsi de suite, ce qui permet de créer des chemins infinis et donc non couvrables en pratique. Pour remédier à cela, il existe le critère **all-k-paths** qui permet de limiter le nombre de boucles à k fois et demande la couverture des itérations de 0 à k.

Dans [MRS+97], les auteurs proposent une technique de génération de tests sur des machines de Mealy (les transitions portent à la fois un évènement d'entrée et éventuellement un évènement de sortie), mise en œuvre dans l'outil Conformance Kit. Cet outil propose deux critères de couverture de l'automate :

- *transition tour* qui correpond au critère *all-transitions* et permet donc de couvrir chaque transition de l'automate au moins une fois. Si une séquence couvre plusieurs transitions à la fois, alors elle suffit à elle seule pour assurer leur couverture.
- *partition tour* qui est aussi un critère *all-transitions* mais plus fort que le précédent car il vise à créer une séquence par transition à couvrir. Ainsi, la séquence est composée d'une *séquence de synchronisation* qui permet de ramener le système dans son état initial, d'une *séquence de transfert* qui permet d'atteindre l'état source de la transition à couvrir, de la paire d'entrée/sortie de la transition à couvrir, puis d'une *séquence unique d'entrée/sortie* qui permet de vérifier que l'état atteint est correct.

Dans [OA99], les auteurs décrivent une technique de génération de données de test à partir de statecharts UML. Ils proposent d'appliquer les critères *all-transitions* et *all-transition pairs* pour la couverture des transitions des statecharts, et le critère *full - predicate* sur les gardes des transitions pour plus de finesse. Cette approche est implémentée dans l'outil UMLTest.

Dans [ABM97], les auteurs décrivent l'approche utilisée dans l'outil CASTING. L'outil est architecturé en deux couches : la couche administrateur et la couche utilisateur. La couche administrateur permet de définir les formalismes d'entrée pour l'outil et de définir des règles d'extraction d'information sur ces formalismes. De plus, cette couche permet la définition de stratégies de génération de tests. La couche d'utilisateur permet d'interagir avec l'outil de plusieurs manières. Ainsi, il peut :

- directement utiliser les stratégies définies dans la couche administrateur pour le

formalisme qu'il utilise,
— créer ses propres stratégies,
— assister le solveur de contraintes s'il ne parvient pas à trouver des données de tests pour la stratégie.

L'application des stratégies permet de calculer les spécifications de cas de tests (*tcs*, *test case specification*) qui sont des ensembles de contraintes à satisfaire lors de l'appel d'une opération du modèle. Ensuite, ces spécifications de cas de test sont utilisées pour générer un graphe de suite de tests : à partir de l'état initial, CASTING applique l'opération d'initialisation qui crée alors le premier nœud du graphe. Ensuite, l'algorithme de recherche par animation symbolique identifie toutes les *tcs* applicables à partir de cet état en résolvant ses contraintes vis-à-vis de l'état à partir duquel elle est utilisée. S'il existe une solution, alors la *tcs* est applicable et génère une nouvelle transition dans le graphe pointant vers un nouvel état du graphe ou un état existant si la postcondition de la *tcs* est satisfaite sur cet état. Dans le cas contraire, la *tcs* n'est pas applicable et ne génère pas de transition. Le graphe obtenu est alors traversé pour couvrir toutes les transitions, en cherchant le plus court chemin à partir du nœud initial, tout en couvrant, si cela est possible, les *tcs* non couvertes (afin de limiter le nombre de cas de tests). L'outil propose des stratégies pour les notations B, Z, SDL et UML.

L'outil TorX [TB03], développé au cours du projet *Côte de Resyste*[11], permet la génération de tests à partir d'un modèle comportemental en s'appuyant sur des modèles sous forme de systèmes de transitions étiquetées (LTS). TorX se fonde sur le parcours des systèmes de transitions étiquetées à entrées/sorties (IOLTS, Input/Output Labeled Transition System) comme critères de sélection et propose trois heuristiques de parcours :
— une heuristique sur la longueur des tests qui permet de ne conserver que les traces qui ont une taille inférieure à une constante fixée par l'ingénieur de tests,
— une heuristique sur le nombre d'itération des cycles, qui permet de conserver les traces qui itèrent un certain nombre de fois les cycles,
— une heuristique visant à ne sélectionner qu'un nombre restreint de transitions qui sortent d'un état.

2.3.4 Couverture de données

Les critères de couverture de données permettent de choisir des données pertinentes pour des variables (ce qui inclut les paramètres d'entrée des opérations) pour la génération de tests.

Ainsi, nous disposons de deux critères basiques. Le premier nécessite la couverture d'au moins une valeur du domaine de la variable considérée. A l'inverse, le second critère extrême nécessite la couverture de toutes les valeurs du domaine. Pour une variable avec un très grand domaine, ce critère n'est en pratique pas applicable. Il peut cependant être intéressant lorsque le domaine considéré est petit.

Entre ces deux critères extrêmes qui ne se focalisent sur rien d'autre que le nombre de valeurs à couvrir, il est nécessaire d'avoir des critères qui correspondent à une intuition pour le test.

11. Site du projet Côte de Resyste : `http://fmt.cs.utwente.nl/CdR` (2013)

Dans l'outil BZ-TT [ABC+02], en plus de l'utilisation des critères de couverture du graphe de flot de contrôle [LPU02], la génération de tests utilise des critères de couverture des variables d'état et des paramètres d'opération aux bornes de leur domaine pour limiter l'explosion combinatoire. Dans un premier temps, à partir des préconditions et postconditions, BZ-TT calcule les valeurs aux limites des variables qui constituent les *buts* à atteindre. Le moteur symbolique CLPS, fondé sur la théorie des ensembles, tente de trouver une séquence pour atteindre un état permettant à la variable de prendre la valeur limite considérée dans le but.

Dans [BDL06], les auteurs utilisent les annotations JML comme modèle pour déterminer les données de test pertinentes, notamment pour la couverture d'un critère *boundary values*. Ces travaux font écho à ceux de [ABC+02] mais pour les modèles JML au lieu de B. Le modèle JML est partitionné entre les comportements normaux et les comportements exceptionnels et donne ainsi une représentation sous la forme d'un graphe représentant ces comportements. Ensuite, les comportements sont réécrits afin de permettre la couverture du graphe par différents critères (SC, DC, C/DC, FPC et MCC). Enfin, l'approche s'appuie sur le solveur de contraintes CLPS-BZ [BLP02] afin de déterminer les valeurs aux limites (*boundary values*) des variables en fonction des contraintes apparaissant dans chaque comportement réécrit. Lorsqu'une variable est contrainte, alors son domaine est partionné entre les valeurs satisfaisant la contrainte et les autres que ne la satisfont pas. Les valeurs limites sont définie comme étant les valeur aux bornes de ces sous-domaines. Cette approche est implémentée dans l'outil JML-TT (JML Testing Tools) [BDLU05].

Dans [BJE94], les auteurs proposent une stratégie de combinaison par paire de chaque valeur de deux paramètres significatifs (*pair-wise testing* et en s'assurant de la couverture de toutes les valeurs des autres paramètres au moins une fois. Cette technique a été étendue dans [CDPP96] en combinant plus de deux paramètres *t-wise testing*). L'outil AETG [CDFP97] (Automatic Efficient Test Generator) implémente ces deux critères de couverture.

2.4 Critères dynamiques

Les critères *dynamiques*, par opposition aux critères *statiques*, sont des critères sur l'aspect temporel du système et représentent des enchaînements particuliers d'opérations ou d'états du système. Nous distinguons deux catégories de critères dynamiques : les scénarios et les propriétés du système. Les scénarios sont l'expression de séquences particulières, d'opération ou d'états, sur le système définis par l'ingénieur de test. Les propriétés sont des exigences au niveau de la spécification que le système doit respecter.

Nous décrivons maintenant ces deux familles de critères dynamiques.

2.4.1 Approches par scénarios

Un scénario représente une abstraction de cas de tests. Il permet de spécifier des points clef que le test doit considérer, et ce qu'il se passe entre deux de ces points est laissé libre. Un scénario permet à un ingénieur d'exprimer un objectif de test et lui permet alors de décrire les tests qui peuvent répondre à cet objectif sans avoir à spécifier toute la séquence.

Nous avons deux catégories de scénarios : les scénarios décrits textuellement avec un langage *ad hoc* et ceux écrits à l'aide de diagrammes de séquence qui représentent alors les séquences avec les points clefs

De manière textuelle

Les approches textuelles sont celles fondées sur une description textuelle de l'objectif de test. L'intérêt de ces approches est de faciliter l'écriture des objectifs de test en camouflant des formalismes plus difficiles à appréhender.

L'approche PUT (*Parameterized Unit Test*) proposée dans [TS05a] permet la définition de cas de test unitaires avec des paramètres, qui sont des cas de tests symboliques. Ensuite, par un mécanisme d'animation symbolique de la méthode de test, chaque chemin est associé à un système de contraintes. La résolution de ces systèmes de contraintes permet alors de déduire des valuations pour chaque chemin d'exécution de la méthode de test et ainsi de produire des tests complètement valués. Cette approche est implémentée dans l'outil *Unit Meister* [TS05b] pour le test d'applications en *.Net*.

Dans [FOB+11], les auteurs proposent de décrire des objectifs de test, appelés *test purposes*, sous une forme textuelle dans un langage de schémas développé dans le cadre du projet SecureChange [12] pour des modèles UML/OCL. Ce langage permet de spécifier un squelette de cas de test qui est ensuite injecté dans le générateur de test de l'outil CertifyIt de Smartesting. En d'autres termes, cette approche a l'avantage de guider le moteur de génération de l'outil en lui spécifiant des appels d'opération spécifiques (éventuellement avec des valeurs de paramètres spécifiques) afin d'obtenir des tests couvrant l'objectif de test décrit par le schéma. Le plugin de CertifyIt pour Rational Software Architect de IBM [13] propose un éditeur dédié à l'écriture de ces schémas

Dans [BMdB+01], les auteurs présentent une technique de génération de tests combinatoire qui a donné lieu à l'outil TOBIAS, développé dans le cadre du projet RNTL COTE [14]. Cet outil permet, à partir d'un schéma (une spécification de cas de test) écrit sous la forme d'expressions régulières sur les appels d'opérations, de générer des tests par dépliage combinatoire sur les valeurs de paramètres des opérations dans les domaines définis par le schéma. L'approche combinatoire permet la génération d'un grand nombre de cas de test. Cependant, puisque le dépliage ne repose pas sur les données du modèle, alors certains tests générés peuvent ne pas être faisables.

Pour remédier à cela, dans [MLdB03], les auteurs décrivent plusieurs scénarios pour un couplage entre TOBIAS et UCASTING, la version UML de CASTING [ABM97]. Cette intégration permet, en premier lieu, de vérifier la conformité des séquences générées par TOBIAS vis-à-vis du modèle UML. Ensuite, elle permet de calculer les valeurs adéquates des paramètres des opérations. Enfin, elle permet, à partir des séquences partiellement valuées, d'éliminer les séquences impossibles en fonction des valeurs déjà présentes.

L'outil Conformiq Designer de Conformiq [15] (anciennement Conformiq Qtronic) utilise

12. Site SecureChange : `http://www.securechange.eu` (2013)
13. Site IBM : `http://www.ibm.com(2013)`
14. COTE a rassemblé des équipes de Softeam, de France Télécom R&D, de Gemplus, de l'IRISA et du LSR, autours du test de conformité de composants logiciels (2000-2002)
15. Site Conformiq : `http://www.conformiq.com` (2013)

un langage de modélisation qui est un sur-ensemble de Java et peut aussi utiliser les diagrammes d'états-transitions d'UML pour représenter la dynamique du système. Cet outil génère des tests en utilisant des critères de couverture connus (couverture des transitions, couverture des données, ...). Dans cet outil, les scénarios peuvent être écrits dans le langage de modélisation en tant que méthode *main*. Ensuite, par exploration symbolique de l'espace d'états, Conformiq permet d'extraire des séquences de test qui correspondent au scénario.

Par des diagrammes de séquences

Pour UML, plusieurs approches utilisent les différents diagrammes pour en extraire des objectifs de test. C'est l'approche choisie dans SCENTOR [WM01] qui génère des scénarios sous forme de diagrammes de séquences. Cette approche utilise un modèle UML exporté dans un format XMI qui est utilisé par SCENTOR. Après la génération des scénarios (sous forme de diagrammes de séquences), l'ingénieur validation doit ajouter les valeurs de paramètres adéquates et spécifier les résultats attendus dans le langage cible. Les scénarios sont ensuite exportés dans le langage cible tel que JUnit pour un programme Java.

L'approche SCENT [RG99] (SCENario-Based Validation and Test of software) est similaire à la précédente mais exprime les scénarios sous la forme de diagrammes d'états-transitions (statecharts). Ces diagrammes sont ensuite complétés avec des préconditions et des postconditions, des informations sur les données manipulées afin d'en dériver des cas de test concrets.

Une autre approche fondée sur la représentation des scénarios en diagrammes UML est l'approche *Telling Test Stories* [FBCO+09, FABA10] qui permet de décrire manuellement des *stories*, utilisées pour exprimer des exigences de sécurité. Cette approche se fonde sur quatre éléments :

- un *modèle du système* qui est une représentation abstraite exécutable du système sous test,
- une *implémentation du système* qui est le système sous test à considérer,
- un *modèle de test* qui contient toutes les *stories*,
- une *implémentation des tests* qui fait office de couche d'adaptation des tests.

Le modèle du système est utilisé pour assurer la cohérence des tests d'une part, et pour quantifier la couverture du modèle par le biais de critères adéquats. Les tests générés du modèle de tests sont traduits dans du code exécutable vis-à-vis de l'implémentation des tests pour faire le lien entre le modèle et l'implémentation.

2.4.2 Approches par propriétés sur le système

Les propriétés temporelles d'un système sont d'autres éléments de la spécification à partir desquels il est possible de tirer des objectifs de tests. Ici, nous considérons une propriété temporelle au sens large, c'est-à-dire qu'il s'agit de la représentation d'un sous-ensemble d'exécutions du système qui respectent certains enchaînements d'opérations. Nous distinguons deux catégories : les propriétés temporelles décrites avec des opérateurs temporels tels que la LTL ou la CTL, et les propriétés temporelles décrites par des

automates.

Par des logiques temporelles

Dans [HCL+03], les auteurs décrivent une approche de couverture de flot de données sur des systèmes à base de transitions à partir de propriétés temporelles décrites en CTL [EH86] et en utilisant le model-checker symbolique SMV [McM92]. En reprenant des critères de couverture de flot de données déjà existants [LK83, Nta84, RW85], les auteurs de cette approche proposent de caractériser chacun de ces critères en tant que propriété temporelle décrite en WCTL, un sous-ensemble de CTL. Ensuite, le model-checker est utilisé pour trouver un témoin de cette propriété sur le modèle. Ce témoin est alors un test issu du critère par la propriété CTL.

Dans [DAC99], les auteurs proposent un langage de patrons de propriétés temporelles. Cette étude part du principe que la majorité des propriétés temporelles considérées dans le test logiciel découlent d'un ensemble restreint de patrons paramétrables, comme cela est constaté dans [DAC11]. Ainsi, pour les auteurs, les propriétés temporelles sont définies comme étant la composition d'une portée (*scope*), qui représente la portion d'exécution du système considérée pour la propriété, et d'un motif (*pattern*) qui est une propriété devant être continuellement satisfaite dans une portée donnée. L'outil Bandera [HD01] implémente ces patrons de propriété pour la vérification de systèmes concurrents par model-checking. Il se fonde sur le langage du même nom [CDHR02] utilisé en tant qu'annotation de programmes Java.

Une autre approche est celle décrite dans [BDGJ06] où les auteurs s'intéressent à la génération de tests à partir de propriétés temporelles et plus particulièrement, à partir de propriétés de sûreté décrites à l'aide de JTPL [TH02] (Java Temporal Property Language), une extension de JML. A partir de critères de couverture de la spécification, des données et des décisions, les auteurs décrivent des stratégies pour générer des tests qui permettent de couvrir la propriété temporelle. Dans [GG06], les auteurs présentent l'outil JAG (JML Annotation Generator) qui permet, à partir de propriétés temporelles sur le système, de générer des annotation en JML pour un programme Java. Ainsi, ces annotations assurent le respect de la propriété au niveau du modèle représenté par ces annotations. Couplé à l'outil JML-TT, il estcela est fait dans [BDL06].

Dans [FW06], les auteurs présentent une technique permettant de caractériser la pertinence d'un cas de test vis-à-vis d'une propriété temporelle par le biais d'un critère de pertinence (*property relevance criterion*). Les auteurs utilisent un modèle sous la forme d'une structure de Kripke et des propriétés temporelles en LTL, et partitionnent les cas de test selon deux catégories : les cas de test *positifs*, qui détectent une faute si leur exécution échoue, et les cas de tests *négatifs*, qui détectent une faute si leur exécution passe avec succès. La pertinence d'un cas de test *négatif* est établie si le cas de test est un contre-exemple de la propriété sur le modèle (tel que l'on pourrait obtenir avec un model-checker). La pertinence d'un cas de test *positif* est établie si le cas de tests *peut* échouer et produit une violation de la propriété. Ainsi, si l'exécution d'un test échoue mais ne provoque pas la violation de la propriété considérée, alors il n'est pas considéré comme pertinent. En s'appuyant sur ces définitions, les auteurs proposent un critère de pertinence pour un test vis-à-vis d'une propriété, qui permettra ensuite d'évaluer une

suite entière vis-à-vis de la propriété.

Dans [FMFR08], les auteurs présentent l'outil jPOST (Property-Oriented Software Testing) qui permet la génération de tests à partir de propriétés temporelles. Cet outil utilise un modèle sous la forme d'un système de transitions étiquetées (LTS), dont les étiquettes sont des actions du système (partitionnées entre les actions *visibles*, qui représentent les interfaces du système, et les actions *internes* au système). Il s'appuie aussi sur des propriétés temporelles génériques. Les auteurs utilisent la LTL-X (la LTL sans l'opérateur *next*) dans leur exemple, mais il peut s'agir de n'importe quelle logique temporelle. Les tests sont représentés comme des ensembles de processus qui communiquent entre eux au moyen de variables partagées et de canaux de communication. Les auteurs distinguent les *test modules*, qui sont des processus de test qui représentent les prédicats apparaissant dans la propriété temporelle et les *test controllers* qui représentent les opérateurs spécifiques à la logique temporelle utilisée. Ainsi, dans la propriété LTL $\neg A \mathcal{U} B$, A et B donnent lieu à des *test modules* et les opérateurs $\neg$ et $\mathcal{U}$ donnent lieu à des *test controllers*. Un cas de test est représenté comme l'arbre de syntaxe abstraite de la propriété temporelle avec les *test modules* et les *test controllers* correspondants comme nœuds. Ensuite, lors de l'exécution des tests, l'outil opère une composition parallèle entre le cas de test et le modèle en se synchronisant sur les actions visibles du modèle. Cette composition représente toutes les exécutions du système qui satisfont la propriété. L'ingénieur de test définit ensuite des obectifs de test sur cette composition pour sélectionner certaines exécutions. Ces objectifs de tests sont représentés par des systèmes de transitions étiquetés qui contiennent deux états puits, *Accept* et *Reject*. Enfin, une composition synchrone entre un objectif de test et un cas de tests permet de sélectionner les exécutions ciblées par l'objectif de test dans le cas de test considéré.

Par des automates

Nous considérons les représentations par automate comme des propriétés temporelles. En effet, dans les approches qui suivent, les automates représentent des ensembles d'exécutions du système bien précises. Ainsi, nous pouvons les considérer comme des propriétés temporelles sur le système.

Certaines approches utilisent les automates pour représenter leurs objectifs de tests, i.e le sous-ensemble des traces possibles du système qui illustre l'objectif de test. La plupart de ces approches utilisent une représentation du modèle et des objectifs de tests sous forme d'automate pour en faire un produit représentant les exécutions du modèle qui respectent l'objectif de test.

L'outil TGV [JJ05] (Test Generation with Verification technology), développé par l'IRISA [16] conjointement avec l'équipe Verimag [17] de Grenoble, est axé sur la génération de tests de conformité pour des systèmes réactifs. Similairement à TorX, cet outil utilise des modèles et des objectifs de tests sous la forme d'IOLTS. La première étape synchronise l'IOLTS du modèle avec celui de l'objectif de test pour identifier les comportements autorisés par celui-ci. TGV détermine ensuite les comportements visibles de cet automate (trace et quiescence). A partir de cet automate, la sélection de tests reconstruit un IOLTS

16. Site IRISA : `http://www.irisa.fr` (2013)
17. Site Verimag : `http://www-verimag.imag.fr` (2013)

en extrayant les comportements acceptés par le produit, en inversant les entrées et les sorties et en complétant éventuellement les entrées manquantes de l'automate (redirigées vers un état FAIL). Enfin, les traces qui permettent d'atteindre un état PASS sont conservées en tant que tests. Cet outil prend aussi en compte des spécifications SDL (Specification Description Language) et LOTOS.

Dans [RBJ00], les auteurs proposent une approche fondée sur des systèmes de transitions symboliques à entrées/sorties (IOSTS) étendus avec des variables et des paramètres. Le modèle et les objectifs de test sont représentés par ces IOSTS et leur produit permet d'obtenir des traces dans le modèle qui représentent les objectifs de test. Les traces de l'automate produit sont alors considérées comme des tests symboliques qu'il est nécessaire d'instancier pour obtenir des cas de tests abstraits. C'est cette technique qui est mise en œuvre dans l'outil STG [CJRZ01] (Symbolic Test Generation tool) développéà l'IRISA.

2.4.3 Approches par modèles de fautes

Ces approches reposent sur l'introduction volontaire d'erreurs dans le modèle afin de déterminer les objectifs de test qui permettent la génération de tests dédiés à leur détection.

Dans [AD06], les auteurs génèrent des mutants pour des modèles LOTOS. Ces modèles sont ensuite traduits en tant qu'IOLTS puis simplifiés par une relation d'équivalence appelée *Safety Equivalence*. Ensuite, les deux modèles sont utilisés pour déterminer s'il existe une relation de bisimulation forte entre les deux. Si les IOLTS sont équivalents, alors il s'agit peut-être d'un mutant équivalent. Dans le cas contraire, un contre-exemple est trouvé. Il est ensuite complété par des transitions supplémentaires permettant d'observer la faute. Enfin, l'objectif de test est formé de la trace du contre-exemple et de la séquence supplémentaire pour observer la faute.

Dans [AWW08], similairement à la technique précédente, les auteurs utilisent des règles de mutation sur le modèle pour produire des mutants. Ces mutations permettent de modifier une des transitions du système qui est alors marquée comme telle dans le modèle original. Ensuite, les auteurs font appel à un model-checker pour générer une trace permettant d'atteindre la transition mutante. Un graphe de test complet est créé pour le modèle original et pour le mutant, puis ces automates sont ensuite comparés selon la relation de conformité *ioco* (Input/Output Conformance [Tre96b]). Enfin, si un contre-exemple est trouvé, celui-ci est considéré comme un test permettant de détecter la mutation.

Dans [TSL04], les auteurs définissent leurs objectifs de test par le biais de propriétés temporelles décrites en LTL. Leur approche s'appuie sur la mutation de la propriété considérée. Implicitement, la propriété temporelle doit être valide pour au moins une exécution. Techniquement, une propriété est de la forme Aϕ et sa mutation est de la forme Eϕ, où ϕ est la formule de la propriété. Les auteurs définissent ensuite un critère de couverture vis-à-vis de la propriété qui permet d'évaluer le pouvoir de détection des tests vis-à-vis des propriétés mutées $\exists$LTL. La génération de tests s'appuie alors sur ce critère pour la génération de séquences se terminant par une boucle (séquences *lasso*), pertinentes pour la propriété considérée. Ces séquence sont ensuite tronquées, car potentiellement infinies, pour donner lieu à des séquences finies, qui sont les tests permettant de cibler la

mutation dont ils sont issus.

Dans les travaux décrits dans [SHJ11], issus du projet européen MOGENTES [18], les auteurs écrivent une approche de génération de tests à partir de modèles de fautes sur des modèles UML/OCL. La dynamique du système est décrite avec un diagramme d'états-transitions. L'objectif de ces travaux est d'effectuer des mutations sur ce diagramme et de le comparer à l'original. Pour cela, les auteurs proposent l'utilisation de quatre mutations sur les diagrammes d'états-transitions : le remplacement de l'évènement déclencheur (*trigger*) par un autre, le remplacement d'une garde de transition par *vrai* ou *faux* et le changement de destination d'une transition. Ensuite, les auteurs transforment les diagrammes d'états-transitions du modèle original et de chaque mutant en systèmes de transitions étiquetées (LTS). L'automate du modèle original est synchronisé avec chaque automate mutant en utilisant la relation de conformité *ioco*. S'il y a non-conformité alors la génération de tests produit des tests qui ciblent les états d'erreur de l'automate synchronisé.

2.5 Synthèse

Cet état de l'art nous a permis de positionner nos travaux dans le vaste domaine du test à partir de modèles. Nous fondons notre approche sur des modèles comportementaux en UML/OCL et plus particulièrement le formalisme utilisé par l'outil CertifyIt de Smartesting que nous introduisons dans le Chapitre 3.

Ce formalisme a la particularité de bien séparer la partie structurelle avec la définition des structures de données (diagramme de classe) et les données (diagramme d'objets) d'une part, et la dynamique du modèle par le biais du diagramme d'états-transitions et de spécifications OCL des opérations d'autre part. De plus, c'est un formalisme qui est utilisé (ou du moins, connu) par les industriels, ce qui permet une meilleure perspective de transfert industriel pour ces travaux.

Nous avons décrit des critères de sélection dynamiques et avons pointé le manque de travaux sur l'utilisation de propriétés temporelles pour des modèles UML/OCL qui ne s'attachent pas au diagramme d'états-transitions. Dans le Chapitre 4, nous décrivons un langage de scénarios textuel proche d'un langage de programmation, similairement à l'approche Conformiq. L'autre moyen pour systématiser l'écriture de ces séquences est l'utilisation de propriétés temporelles. Mais malgré tous les travaux pour le test à partir de propriétés temporelles, il en existe très peu sur la génération de tests à partir de propriétés temporelles pour des modèles UML/OCL. Ainsi, dans le Chapitre 5, nous nous inspirons des patrons de Dwyer *et al.* [DAC99] pour la définition d'un langage de propriétés temporelles. Cette approche a le grand avantage de proposer un langage textuel, contrairement aux logiques temporelles telles que la LTL ou la CTL qui utilisent des symboles. De plus, il utilise un nombre restreint de constructions, tout en étant très expressif.

Nous avons décrit différentes classes de critères de couverture statiques. En particulier, les critères classiques sur les transitions attirent notre attention. Dans le Chapitre 6, nous nous inspirons de ces critères classiques en les adaptant à nos automates de propriété. Ces

18. Site MOGENTES : `http://www.mogentes.eu/` (2013)

critères nous permettent alors d'illustrer différents aspects de la dynamique du système. Pour cibler des exécutions permettant la violation de propriétés, nous nous inspirons dans le Chapitre 7 des approches par modèles de fautes en mutant l'automate de propriété original. Cette technique, proche des travaux de [FW06], permet de faire ressortir des évènements proches d'évènements permettant la violation de la propriété.

Chapitre 3

Modéliser et tester avec UML/OCL

Sommaire

3.1 Langage UML4MBT 36
3.1.1 Diagramme de classes 36
3.1.2 Diagramme d'objets 39
3.1.3 Diagramme d'états-transitions 39
3.2 Langage OCL4MBT 41
3.2.1 Interprétation du langage 41
3.2.2 Constructions OCL4MBT 43
3.2.3 Interprétation de la valeur indéfinie 48
3.2.4 Expression des exigences fonctionnelles 48
3.3 Génération de tests dans CertifyIt 49
3.3.1 Calcul des cibles de tests 50
3.3.2 Génération de tests 51
3.4 Exemple fil rouge 53
3.4.1 Exigences fonctionnelles 53
3.4.2 Description du modèle 54
3.4.3 Diagramme de classes 54
3.4.4 Diagramme d'objets 57
3.4.5 Exemples de tests générés par CertifyIt 59
3.5 Synthèse 59

Ce chapitre présente la notation UML/OCL utilisée pour la conception de modèles pour le test. La notation UML permet la représentation de structures de données dans un paradigme orienté-objet. La notation OCL, initialement conçue pour ajouter des contraintes sur des éléments des diagrammes UML, voit sa sémantique adaptée en vue de l'utiliser comme un langage de description des opérations du modèle sous une forme pre/-post. Cette modélisation est celle utilisée en tant que langage d'entrée de l'outil CertifyIt de Smartesting [Gra08].

La section 3.1 présente la notation UML4MBT, un sous-ensemble d'UML, utilisée dans le cadre de la modélisation de spécifications pour le test suivie de la section 3.2 qui

présente la notation OCL4MBT, un sous-ensemble d'OCL. Ensuite, la section 3.3 décrit le processus de génération de tests de l'outil CertifyIt de Smartesting. Enfin, la section 3.4 introduit l'exemple de modèle qui va nous servir d'exemple fil rouge tout au long de ce document.

3.1 Langage UML4MBT

Nous introduisons ici la notation UML utilisée pour la modélisation de la structure de données des modèles et d'une partie de leur aspect dynamique à l'aide de différents diagrammes issus du standard UML. La modélisation de systèmes comportementaux utilise trois des diagrammes d'UML : les diagrammes de classes, les diagrammes d'objets et les diagrammes d'états-transitions. Ce sous-ensemble est nommé UML4MBT pour *UML for Model-Based Testing*.

3.1.1 Diagramme de classes

Ce diagramme exprime l'aspect structurel des données du modèle à l'aide de *classes* et d'*associations* entre ces classes. Il est une abstraction des objets et des liens qui composent le système sous test qui sont, quant à eux, représentés dans le diagramme d'objets. Il existe donc une relation d'instanciation entre les deux diagrammes : une *classe* décrit un ensemble d'*objets* et une *association*, un ensemble de *liens*.

Classe UML

Une classe UML décrit de manière abstraite un ensemble d'objets du système qui possèdent une sémantique commune. Cette sémantique est définie à partir des attributs et des opérations de la classe.

Les **attributs** permettent de décrire les données de la classe. Un attribut possède un type, une visibilité et peut éventuellement être caractérisé par une valeur par défaut.

Les **opérations** décrivent l'aspect comportemental de la classe. Il s'agit de fonctions qui peuvent prendre des valeurs d'entrées (les *paramètres d'entrée*) et modifier les attributs et/ou produire des résultats.

La Figure 3.1 donne la représentation graphique d'une classe UML. Dans cet exemple, un véhicule est caractérisé par sa couleur et sa vitesse. De plus, il lui est possible de démarrer, de couper le moteur, d'accélérer, de freiner et de s'autodétruire par le biais des opérations éponymes. Enfin, il est toujours possible d'observer la vitesse du véhicule.

Attributs de classe. Les attributs de classe représentent des données typées qui varient tout au long de la vie du système. UML4MBT dispose d'un ensemble de types présents dans le standard UML : les types **primitifs** *booléens* et *entiers*, et les types **énumérés** par les *classes d'énumération*. Il y a deux exceptions, parmi les types natifs d'UML : le type *String* et les rationnels (et par conséquent, les réels). Pour le type *String*, cette restriction vient du fait que les combinaisons possibles de caractères qui la compose est de l'ordre de a^n avec a étant le nombre d'éléments de l'alphabet considéré dans la chaîne de

FIGURE 3.1 – Exemple de classe

FIGURE 3.2 – Exemple de classe d'énumération

caractère de taille n. Ainsi, pour une chaîne de caractères de taille 5 n'utilisant que les 26 caractères de l'alphabet, il existe environ $1,4 \times 10^{14}$ chaînes possibles ! Pour contourner le problème, il convient d'abstraire les chaînes de caractères par des littéraux définis par un type d'énumération. Ainsi, dans notre exemple de véhicule, les couleurs "rouge", "noir", "bleu" sont représentées par une classe d'énumération *COULEURS* qui contient les littéraux *ROUGE*, *NOIR* et *BLEU* comme le montre la FIGURE 3.2.

Dans la FIGURE 3.1, la classe *Vehicule* possède deux attributs :

— *couleur* qui représente la couleur du véhicule. Son type correspond à l'énumération *Couleurs* dont il prend une des valeurs.

— *vitesse* qui représente la vitesse du véhicule.

Les différentes valuations de cet ensemble d'attributs définissent les différents états des objets.

Opération de classe. Les opérations représentent l'aspect dynamique d'un objet en particulier et du système en général. Une opération permet à l'objet qui l'invoque de modifier son état (en modifiant la valeur de ses attributs ou en créant de nouveaux liens) et/ou celui d'autres objets liés. Les profils de ces opérations représentent les interfaces du système à tester. Dans le cadre d'UML4MBT, le comportement des opérations est décrit par une précondition (la condition d'activation de l'opération) et une postcondition (l'effet de l'opération) en OCL4MBT. Enfin, contrairement à UML où les paramètres des opérations peuvent prendre les directions IN (paramètre d'entrée), OUT (paramètre de sortie), INOUT (paramètre d'entrée/sortie) et RETURN (paramètre de retour), UML4MBT n'autorise que les directions IN et RETURN.

Enfin, les opérations peuvent être marquées avec le stéréotype *observation* qui la définit alors comme une opération d'observation sur le système. Typiquement, ces opérations permettent de retourner la valeur d'un attribut d'une classe du système. C'est le cas de l'opération *compteurVitesse* de notre exemple.

Associations UML

Une association représente une relation structurelle et sémantique entre plusieurs classes du diagramme. Dans UML4MBT, une association est toujours binaire (entre deux classes), elle peut être réflexive (la classe source et la classe destination peuvent être identiques) et n'est pas orientée (elle est navigable dans les deux sens).

Multiplicité	Signification
1	Un seul
0..1	Au plus un
* 0..*	Zéro ou plus
1..*	Au moins un
$n..m$	De n à m fois, (avec $m > n$)

FIGURE 3.3 – Multiplicité d'association supporté dans UML4MBT

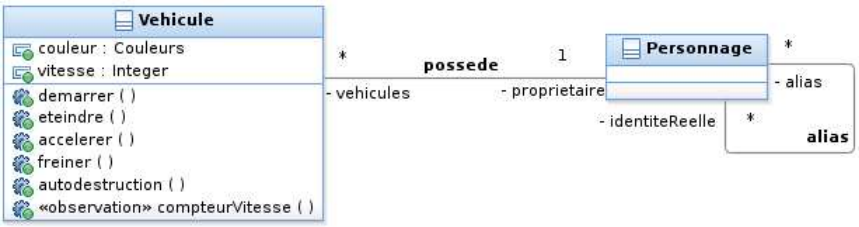


FIGURE 3.4 – Exemple d'associations entre deux classes

Outre les classes à chacune de ses extrémités, une association possède trois caractéristiques supplémentaires :

— le *nom* de l'association, qualifiant la relation entre deux classes.
— les *rôles*, apparaissant à chaque extrémité de l'association, et décrivant comment une classe voit l'autre classe. Ces rôles permettent de naviguer de classe en classe par ces associations.
— les *multiplicités*, associées à chaque *rôle*, indiquent la quantité d'objets de la classe pouvant être liés à un objet de l'autre classe. Le tableau de la FIGURE 3.3 récapitule les différentes multiplicités possibles.

La navigation d'une association permet d'accéder à un objet, pour des multiplicités 1 ou 0..1, ou à une collection d'objets pour les autres multiplicités. Dans UML4MBT, les collections sont toujours considérées comme étant des ensembles, dans le sens mathématique du terme (non-ordonnés et absence de doublons).

La FIGURE 3.4 donne un exemple d'associations entre classes. Dans cet exemple, l'association *possede* représente la relation de propriété entre un véhicule et son propriétaire. Le rôle *vehicules* du coté de la classe *Vehicule* indique que le propriétaire possède les véhicules associées dans son garage. La multiplicité * indique qu'un personnage peut avoir plusieurs véhicules dans son garage ou éventuellement aucun. Le rôle *propriétaire*, de l'autre coté de l'association, indique qui possède le véhicule en question. La multiplicité 1 indique qu'un véhicule ne peut avoir qu'un seul et unique propriétaire. L'association réflexive *alias* sur la classe *Personnage* permet de représenter les différentes identités d'un même personnage : les multiplicités permettent de représenter le fait qu'un alias peut être endossé par plusieurs personnages (rôle *identité*) et qu'un même personnage peut avoir plusieurs alias (rôle *alias*).

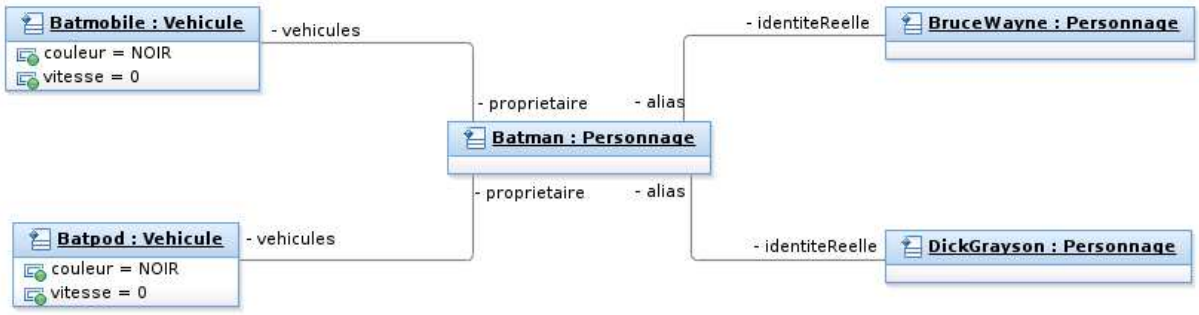


FIGURE 3.5 – Exemple de diagramme d'objet

3.1.2 Diagramme d'objets

Le diagramme d'objet représente une instanciation du diagramme de classe. Ainsi, il contient des objets, qui instancient les classes, et des liens qui instancient les associations. La FIGURE 3.5 donne un exemple de diagramme d'objet. Dans cet exemple, les objets *Batmobile* et *Batpod* sont des instances de la classe *Vehicule*, tandis que les objets *Batman*, *DickGrayson* et *BruceWayne* instancient la classe *Personnage*.

Les liens reliant *Batman* aux deux véhicules *Batmobile* et *Batpod* sont les concrétisations de l'association *possede* du diagramme de classe. De plus, avec les rôles, nous voyons bien que ces véhicules appartiennent au garage de *Batman* qui en est alors leur propriétaire. La multiplicité * du rôle *vehicules* est visible par le fait que Batman possède deux véhicules.

Les deux autres liens concrétisent l'association réflexive *alias*. La multiplicité 1..* du rôle *identité* représente le fait que plusieurs personnages ont endossé le costume de Batman : Bruce Wayne, l'original, et, parmi d'autres, Dick Grayson [Win09], un des Robins de Batman.

Dans le cadre d'UML4MBT, le diagramme a deux spécificités. D'une part, il représente l'état initial du système sous test. D'autre part, il contient tous les objets présents lors de l'exécution du modèle, car UML4MBT n'autorise pas la création ou la suppression d'objets du diagramme. Ainsi, tous les objets potentiellement nécessaires au système pour le test doivent être définis dans ce diagramme : les instances qui ne sont pas censées exister à l'état initial sont isolées des autres instances (aucun lien ne les relient).

Enfin, il est à noter que dans UML4MBT, le diagramme de classes possède une classe spécifique représentant le système sous test et dont les opérations servent à décrire les interfaces du système disponibles à l'extérieur.

3.1.3 Diagramme d'états-transitions

Les diagrammes d'états-transitions sont des automates d'états finis hiérarchiques. Ils sont rattachés à une classe dont ils décrivent, de manière formelle, le comportement. Ils permettent donc de représenter, d'une manière abstraite, un aspect dynamique du système. Il est composé d'un ensemble d'états, reliés entre eux par des transitions.

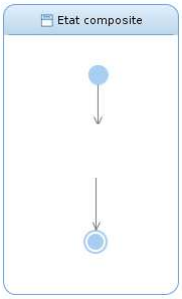
Nom	Représentation	Description
Etat initial		Représente le point de départ du diagramme. Il ne peut y en avoir qu'un.
Etat final		Représente la fin de l'exécution du diagramme.
Etat simple		Représente un état simple du diagramme.
Etat composite		Représente un état composé d'un sous-diagramme d'états-transitions. En entrant dans ce genre d'état, c'est le sous-diagramme qui est considéré pour l'évolution du système, jusqu'à atteindre son état final.

FIGURE 3.6 – Les différents types d'états d'un diagramme d'états-transitions

Etats

Un état représente une certaine configuration du diagramme d'objet en terme de valeurs d'attributs et des liens qui le définissent : chacun d'eux représente donc des combinaisons particulières de valeurs d'attributs et de liens. Un état est aussi caractérisé par un ensemble d'actions d'*entrée* et de *sortie*. Le tableau de la FIGURE 3.6 présente les différents types d'états rencontrés dans les diagrammes d'états-transitions UML4MBT.

Transitions

Une **transition** exprime un changement d'état du diagramme, déclenché par un évènement du système. La FIGURE 3.7 donne la représentation graphique d'une transition. L'étiquette de la transition est caractérisée par trois composants :

— un **évènement déclencheur** qui permet le déclenchement de la transition. En UML4MBT, cet évènement est l'appel d'une opération du système sous test donc ce diagramme est de fait associé à la classe du système sous test.
— une **garde** qui est une expression booléenne. C'est la condition à satisfaire pour activer la transition associée.
— une **action** qui est éxécutée lorsque la transition est activée. L'action peut agir directement sur l'objet associé à l'automate et indirectement sur ses objets liés.

La FIGURE 3.8 donne un exemple de diagramme d'états-transitions attaché à la classe *Vehicule*. Dans cet exemple, un véhicule peut être dans l'un des états suivants :

— *Eteint* : dans ce cas, il peut passer à l'état *Démarré* par le déclenchement de l'opération *démarrer()*.
— *Démarré* : il peut repasser à l'état *Eteint* par l'opération *éteindre()*, ou passer dans l'état *EnMouvement* en accélérant par l'opération *accelerer()* pour atteindre une

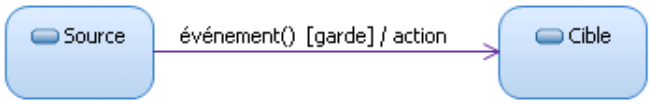


FIGURE 3.7 – Représentation graphique d'une transition

vitesse positive.
— *EnMouvement* : il peut continuer à l'être jusqu'à ce qu'un appel à l'opération *freiner()* amène la vitesse du véhicule à 0, auquel cas le véhicule repasse dans un état *Démarré*.
— *final* : représenté par l'état final qui est atteint par l'appel de l'opération *autodestruction()*, quel que soit l'état. Le véhicule étant détruit il est alors impossible de faire une action quelconque.

3.2 Langage OCL4MBT

Le langage OCL4MBT (Object Constraint Language) est utilisé pour décrire l'aspect comportemental du système. Cependant, à l'instar d'UML4MBT, il ne s'appuie que sur un sous-ensemble des constructions offertes par OCL. Initialement, OCL permet de définir des contraintes sur les différents éléments UML (attributs, classes, opérations, ...).

3.2.1 Interprétation du langage

Une des grandes différences d'OCL4MBT par rapport à OCL est l'introduction d'une interprétation opérationnelle du langage. Alors qu'OCL est défini comme un langage d'expressions logiques de type précondition/postcondition, OCL4MBT interprète la postcondition comme une action interprétée de manière séquentielle.

Langage d'action

La différence majeure entre OCL et OCL4MBT est une contextualisation de l'évaluation des expressions OCL, suivant leur place dans le code OCL. Ainsi, une expression peut être évaluée comme une **contrainte** (Définition 1) ou comme une **action** (Définition 2).

Définition 1 (Contrainte OCL)**.** Une expression OCL est une contrainte si elle est dans un des cas suivants :
— en précondition d'une opération, pour exprimer la condition d'activation de l'opération,
— en tant que garde d'une transition, pour exprimer la condition d'activation d'une transition,
— en condition d'un bloc *if-then-else* dans la postcondition d'une opération.

Définition 2 (Action OCL)**.** Une expression OCL est une action si elle est dans un des cas suivants :
— en postcondition d'opération pour en exprimer ses comportements,

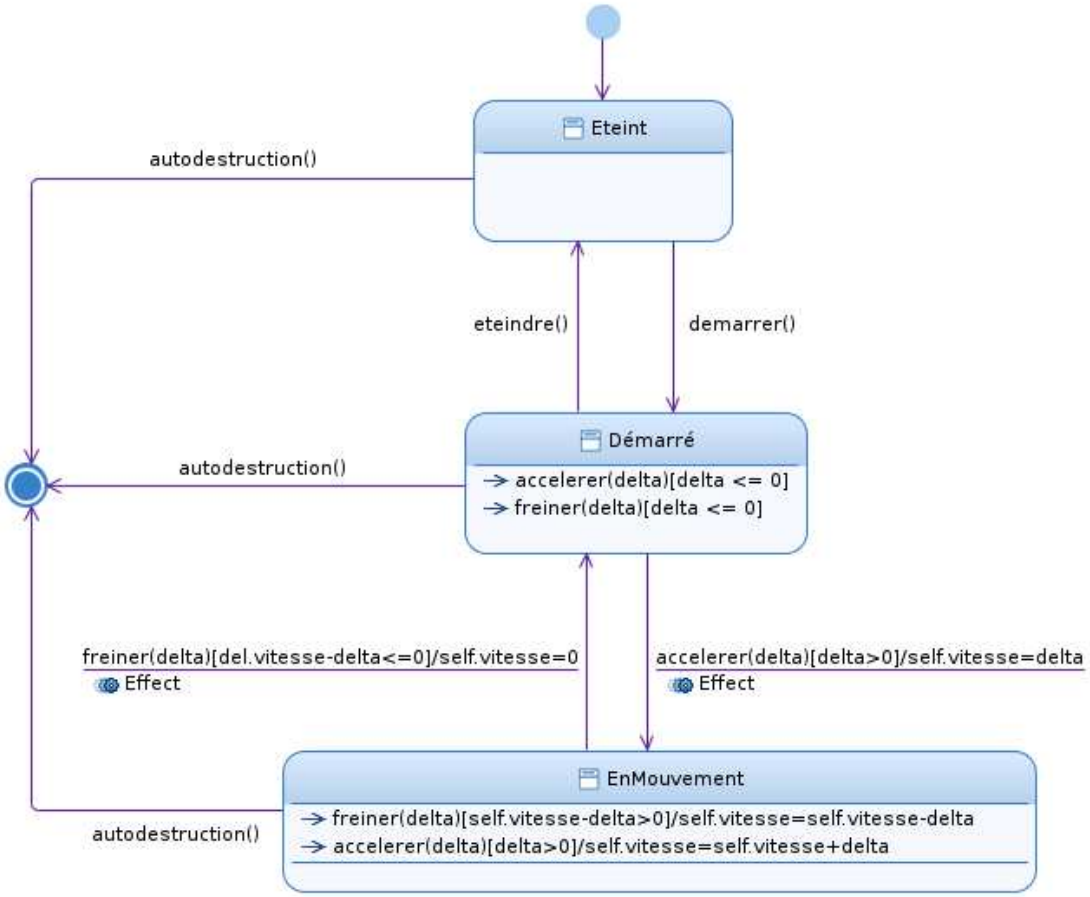


FIGURE 3.8 – Exemple de diagramme d'états-transitions

— dans les blocs d'actions d'une structure conditionnelle, (`if-then-else`),
— en action d'une transition du diagramme d'états-transitions.

Avec la définition d'action, on s'aperçoit que l'interprétation d'OCL4MBT dans une postcondition diffère de celle de l'OCL classique : la postcondition n'indique plus *ce à quoi on s'attend* après l'opération (condition à respecter), mais *ce que fait l'opération* (l'action exécutée par l'appel).

Interprétation séquentielle

A l'origine, le langage OCL est un langage de contraintes sans effet de bord : une expression OCL ne peut pas modifier l'état du système. Ainsi, une variable utilisée dans une opération existe à l'état avant et à l'état après et n'a pas de valeur intermédiaire lors de l'exécution de l'opération. Pour les variables dans une postcondition, OCL permet de faire référence à la fois à sa valeur au début de l'opération et à la fin de celle-ci. Le mot-clef `@pre` est utilisé pour faire référence à sa valeur initiale. Cependant, dans OCL4MBT, cette construction pour les variables n'a pas lieu d'être : l'interprétation étant séquentielle, la valeur peut changer plusieurs fois au cours de l'opération et la valeur de la variable à la fin de l'opération est celle qui lui aura été affectée en dernier.

3.2.2 Constructions OCL4MBT

Comme OCL, OCL4MBT est un langage fortement typé et permet de manipuler des types de donnés connus dans le standard UML. Ainsi, nous retrouvons les types *booléens* et *entiers* ainsi que les types des classes du diagramme de classe et les énumérations. Pour chacun de ces types, OCL4MBT définit un ensemble d'opérateurs qui leur sont applicables.

Opérateurs booléens

OCL4MBT supporte l'ensemble des opérateurs booléens utilisés dans OCL qui sont récapitulés dans la catégorie *Booléens* dans la FIGURE 3.9, avec b_1 et b_2 des expressions booléennes. Nous retrouvons les opérateurs classiques de la logique propositionnelle, l'implication en moins.

Il est cependant à noter que les opérateurs = et *and* peuvent être interprétés de deux manières différentes, en fonction du contexte dans lequel ils sont utilisés.

Définition 3. [Double interprétation du =] Dans un contexte d'action l'opérateur = est interprété comme une affectation, alors que dans un contexte d'évaluation, il est interprété comme l'opérateur d'égalité de valeurs entre deux expressions.

Définition 4. [Double interprétation du *and*] Dans un contexte d'action l'opérateur *and* est utilisé comme un séparateur entre les actions. L'interprétation étant séquentielle, il perd sa propriété de commutativité. Dans un contexte d'évaluation, il est interprété comme l'opérateur de conjonction logique entre deux expressions booléennes.

Opérateurs sur les entiers

OCL4MBT supporte l'ensemble des opérateurs sur les entiers utilisés par OCL. La catégorie Entier dans la FIGURE 3.9 récapitule ces opérateurs avec leur type, avec e_1 et e_2 des expressions entières. Nous retrouvons ici les opérateurs classiques de l'arithmétique élémentaire (l'addition, la soustraction, la multiplication et la division) sur les entiers. Nous disposons aussi du support des fonctions minimum, maximum, valeur absolue et modulo. Enfin, tous les opérateurs de comparaison classiques entre les entiers sont supportés ($\leq$, $<$, $\geq$, $>$, $=$ et $<>$).

La double interprétation de l'opérateur = décrite précédemment s'applique aussi dans ce cas (voir Définition 3).

Opérateurs sur les classes d'objets

OCL4MBT supporte l'ensemble des opérateurs sur les classes d'objets présentés dans la catégorie Objet de la FIGURE 3.9, avec *obj*, obj_1 et obj_2 des objets, *class* une classe du diagramme de classes.

Les expressions *obj* peuvent faire référence :

— aux attributs de classes de types énumérés, car définis par des classes d'énumération,
— aux navigations par associations entre classes,
— à l'expression d'un objet dans une collection (*any* par exemple),
— aux paramètres d'opérations de type classe.

La double interprétation de l'opérateur = décrite précédemment s'applique aussi dans ce cas (voir Définition 3), à la différence qu'il s'applique aux objets ou aux classes. A cela, nous ajoutons la double interprétation de *oclIsUndefined()*.

Définition 5. [Double interprétation du $oclIsUndefined()$] Dans un contexte d'évaluation, *oclIsUndefined()* permet de tester l'existence d'un objet par navigation via une association (i.e. si le lien existe). Dans un contexte d'action, il permet de supprimer ce lien.

Il est à noter que, quel que soit le contexte, il ne peut être utilisé que sur une association dont le rôle a une multiplicité 0..1.

Opérateurs sur les collections

OCL4MBT supporte les opérateurs sur les ensembles présentés dans la catégorie Collection de la FIGURE 3.9, avec col_1 et col_2 des ensembles d'éléments. Ces collections peuvent être obtenues :

— par navigation d'une association dont l'extrémité a une multiplicité 0..* ou 1..*,
— par *class.allInstances()*,
— ou par une expression de type *ensemble d'objets* issue de ces opérateurs.

Dans les opérateurs supportés nous retrouvons les opérateurs permettant :

— de tester le vide d'une collection avec *isEmpty()* et *notEmpty()*,
— le test d'inclusion et d'exclusion d'un élément, avec *includes(obj)* et *excludes(obj)*,

Type	**Opérateur**	**Nom**	**Type d'expression**
Booléens	$b_1 = b_2$	égalité	booléen
	$b_1 <> b_2$	différence	booléen
	b_1 or b_2	disjonction	booléen
	b_1 xor b_2	disjonction exclusive	booléen
	b_1 and b_2	conjonction	booléen
	not b_2	négation	booléen
Entiers	$e_1 = e_2$	égalité	booléen
	$e_1 <> e_2$	différence	booléen
	$e_1 < e_2$	inférieur strict	booléen
	$e_1 <= e_2$	inférieur	booléen
	$e_1 > e_2$	supérieur strict	booléen
	$e_1 >= e_2$	supérieur	booléen
	$e_1 + e_2$	addition	entier
	e_1 - e_2	soustraction	entier
	- e_1	opposé	entier
	$e_1 * e_2$	multiplication	entier
	e_1.div(e_2)	division entière	entier
	e_1.abs()	valeur absolue	entier
	e_1.mod(e_2)	modulo	entier
	e_1.max(e_2)	maximum	entier
	e_1.min(e_2)	minimum	entier
Classes	$obj_1 = obj_2$	égalité entre objets	booléen
	$obj_1 <> obj_2$	différence entre objets	booléen
	obj.oclIsUndefined()	existence d'objet	booléen
	$class$.allInstances()	instances d'une classe	ensemble d'objets
Collections	$e_1 = e_2$	égalité	booléen
	$col_1 <> col_2$	différence	booléen
	col_1→size()	cardinalité	entier
	col_1→includes(obj)	inclusion d'objet	booléen
	col_1→excludes(obj)	exclusion d'objet	booléen
	col_1→includesAll(col_2)	inclusion multiple d'objets	booléen
	col_1→excludesAll(col_2)	exclusion multiple d'objets	booléen
	col_1→isEmpty()	test de l'ensemble vide	booléen
	col_1→notEmpty()	test de l'ensemble non-vide	booléen
Itération sur les ensembles	set→collect(obj)	collection d'objets	ensemble d'objets
	set→select($bool$)	filtre sur un ensemble d'objets	ensemble d'objets
	set→exists($bool$)	test d'existence	booléen
	set→forAll($bool$)	test d'universalité	booléen
	set→any($bool$)	extraction d'objet	objet

FIGURE 3.9 – Opérateurs considérés dans OCL4MBT

— le test d'inclusion et d'exclusion d'une collection d'éléments avec *includesAll(coll)* et *excludesAll(coll)*,
— la comparaison entre collections (égalité et différence),
— le calcul du nombre d'éléments avec *size()*.

La double interprétation de l'opérateur = est valable lors de la comparaison de collections (voir Définition 3). A cela, nous ajoutons la double interprétation de l'inclusion ensembliste (pour *includes(obj)* et *includesAll(coll)*), de l'exclusion ensembliste (pour *excludes(obj)* et *excludesAll(coll)*) et de l'opérateur *isEmpty()*.

Définition 6. [Double interprétation de l'inclusion] Dans un contexte d'évaluation, *includes(obj)* permet de tester l'existence de l'objet *obj* dans l'ensemble considéré. Dans un contexte d'action, il permet d'ajouter l'objet *obj* à l'ensemble considéré.

Dans un contexte d'évaluation, *includesAll(coll)* permet de tester l'existence de tous les éléments de *col* dans l'ensemble considéré. Dans un contexte d'action, il permet d'ajouter tous les objets de *col* à l'ensemble considéré.

Définition 7. [Double interprétation de l'exclusion] Dans un contexte d'évaluation, *excludes(obj)* permet de tester l'absence de l'objet *obj* dans l'ensemble considéré. Dans un contexte d'action, il permet de supprimer l'objet *obj* de l'ensemble considéré.

Dans un contexte d'évaluation, *excludesAll(coll)* permet de tester l'absence de tous les éléments de *col* dans l'ensemble considéré. Dans un contexte d'action, il permet de supprimer tous les objets de *col* de l'ensemble considéré.

Définition 8. [Double interprétation de *isEmpty()*] Dans un contexte d'évaluation, *isEmpty()* permet de tester le vide de l'ensemble considéré. Dans un contexte d'action, il permet de vider l'ensemble considéré.

Structure conditionnelle

Il est possible d'exprimer la structure conditionnelle `if-then-else` dans OCL4MBT avec la structure suivante :

```
1 if condition then
2    action1
3 else
4    action2
5 endif
```

où *condition* est une expression booléenne, et *action*1 et *action*2 sont des expressions interprétables dans un contexte d'action. Une structure conditionnelle est toujours exprimée dans un contexte d'action.

Exemple 1 (Code OCL de l'opération *freiner*)**.** L'opération *freiner* prend un paramètre `delta` représentant la perte de vitesse dûe au freinage et ajuste la nouvelle vitesse en conséquence. Cette opération distingue plusieurs cas, appelés *comportements*. Le premier est un freinage qui réduit la vitesse du véhicule, mais ne l'arrête pas (vitesse-delta>0). Ensuite, les deux autres cas distinguent un freinage brusque (vitesse-delta<0) ou un freinage pour un arrêt en douceur (vitesse-delta=0). Bien que ces deux derniers cas sont identiques

```
1  pre  : delta >= 0
2  post :
3      if vitesse - delta <= 0 then
4          if vitesse - delta = 0 then
5              vitesse = 0
6          else
7              vitesse = 0
8          endif
9      else
10         vitesse = vitesse - delta
11     endif
```

FIGURE 3.10 – Code OCL de l'opération *freiner*

à ce niveau d'abstraction du système (vitesse = 0), la distinction est effectuée du fait que dans la spécification, ces cas sont différents. Puisqu'il s'agit d'un modèle, les détails qui permettent d'identifier la différence n'apparaissent pas à ce niveau d'abstraction, mais nous conservons tout de même cette distinction pour la génération de tests.

Définition de variables locales

La structure OCL `let` permet de définir une variable locale à l'opération substituant une expression. Cette construction est utilisée lorsque l'on a besoin d'utiliser une même expression à plusieurs reprises. Elle n'est cependant exprimable que dans un contexte d'action et s'écrit de la manière suivante :

```
1  let var1 : Type1 = exp1,
2      var2 : Type2 = exp2,
3      ...
4      varn : Typen = expn in
5      action
```

Dans cette expression :

- $\texttt{var}_i$ est le nom de la variable de substitution,
- $\texttt{Type}_i$ est un type du modèle et représente le type de la variable, qui doit correspondre au type de l'expression $\texttt{exp}_i$,
- $\texttt{exp}_i$ est l'expression substituée par la variable,
- `action` est une expression interprétable dans un contexte d'action. La variable peut être utilisée dans cette expression.

Une telle variable, définie dans un contexte d'action, ne peut cependant pas être modifiée dans l'action : elle permet uniquement de remplacer la valeur d'une expression par un nom. S'il s'agit d'un objet, il est cependant toujours possible de modifier les attributs et les liens de cet objet.

Exemple 2 (Structure *let*)**.** Soit une classe comportant un attribut entier `att` et `in_param` le nom d'un paramètre d'entrée de l'opération considérée. Considérons l'exemple suivant :

```
1  let temp : Integer = 5*in_param  in
2          att = 3*temp
```

Dans cet exemple la variable `temp` reçoit la valeur de l'expression qui la suit, ici, 5*`in_param`. En réalité, ce code OCL est équivalent à :

```
1  att = 15*in_param
```

3.2.3 Interprétation de la valeur indéfinie

L'évaluation d'une expression OCL peut conduire à une valeur non définie. Cette notion de valeur non définie est à rapprocher du concept de l'objet *null* dans une programmation orientée objet. La valeur indéfinie d'un objet peut être testée par l'opérateur *oclIsUndefined()*, dans un contexte d'évaluation. De plus, si une valeur indéfinie apparaît dans une expression, cette valeur se propage et rend alors l'expression indéfinie.

Cette valeur indéfinie peut apparaître à plusieurs occasions :

— par une expression de sélection d'objet par l'opérateur *any*. Si aucun élément ne permet de répondre au critère de sélection du *any*, alors l'expression est indéfinie.

— par navigation, si un lien entre deux objets n'existe pas ou s'il a été supprimé (par l'opérateur *oclIsUndefined()* interprété comme une action).

3.2.4 Expression des exigences fonctionnelles

OCL4MBT permet d'exprimer des exigences fonctionnelles en instrumentant le modèle UML. Pour ce faire, OCL4MBT permet cette instrumentation au travers de commentaires OCL spéciaux et de deux marqueurs spécifiques @REQ et @AIM :

— le marqueur @REQ permet d'identifier des exigences fonctionnelles de haut niveau, comme la possibilité de freiner pour un véhicule (@REQ :FREIN),

— le marqueur @AIM permet d'identifier des exigences au niveau d'un comportement spécifique d'une opération, comme le comportement particulier du freinage brutal d'un véhicule (@AIM :FREIN_Stop_Urgent)

En d'autres termes, une exigence de type @REQ peut être composée de plusieurs exigences de type @AIM, chacune marquant un aspect spécifique du comportement global de l'opération considérée. Ils peuvent ainsi être considérés comme des raffinements d'une exigence plus abstraite. Ces exigences sont conservées tout au long du processus de génération de test pour établir une traçabilité entre les exigences et les tests.

Exemple 3 (Marquage des exigences)**.** Considérons l'opération *freiner()* introduite dans l'exemple 1. La FIGURE 3.11 montre les annotations ajoutées à la postcondition de *freiner()*. Nous identifions l'opération par @REQ :FREIN afin de l'associer à l'exigence fonctionnelle relative au freinage du véhicule. Ensuite, nous associons un nom à chacun des comportements de l'opération :

— @AIM :FREIN_Stop : cas où le véhicule freine pour s'arrêter,

— @AIM :FREIN_Stop_Urgent : cas où le véhicule freine brusquement,

— @AIM :FREIN_Frein_Normal : cas où le véhicule ralentit.

Cela nous permet d'identifier les comportements d'une opération de manière unique. Un comportement est une séquence d'*actions gardées* et caractérise un ensemble de modifications sur l'état du système (action) si la condition (garde) est satisfaite.

Afin de définir la notion d'*action gardée*, nous devons d'abord définir la notion d'action sur une variable.

```
1   ---@REQ:FREIN
2   if vitesse - delta <= 0 then
3     if vitesse - delta = 0 then
4       ---@AIM:FREIN_Stop
5       vitesse = 0
6     else
7       ---@AIM:FREIN_Stop_Urgent
8       vitesse = 0
9     endif
10  else
11    ---@AIM:FREIN_Frein_Normal
12    vitesse = vitesse - delta
13  endif
```

FIGURE 3.11 – Code OCL de l'opération *freiner* avec les tags des exigences

Définition 9 (Action)**.** Une action est une affectation, ou le changement de valeur, de la variable var par l'expression val. Elle est notée $var := val$.

Définition 10 (Action gardée)**.** Une action gardée est une affectation d'une variable var par une valeur val, gardée par la condition $guard$. Le changement de valeur de var a lieu, uniquement si la condition $guard$ est vraie. Une telle substitution gardée est notée $[guard]var := val$, où $guard$ est une expression booléenne, var est une variable d'état du système, une variable locale à l'opération ou un paramètre, et val est une expression dans le domaine de var.

Cette définition d'*action gardée* permet de définir la notion de comportement comme étant une séquence d'actions gardées.

Définition 11 (Comportement)**.** Un comportement C est une séquence d'actions, éventuellement gardées, sous forme SSA (Single Static Assignement), représenté de la manière suivante avec $n \geq 0$:

$$C = \quad act_0; act_1; \ldots; act_n = \quad [guard_0]var_0 := val_0; \ldots; [guard_n]var_n := val_n$$

Il est à noter que puisque l'exécution est séquentielle, alors la garde $guard_i$ s'exprime sur l'état du système après la réalisation de l'action act_{i-1} avec $i > 0$. La garde $guard_0$ est la condition initiale d'activation du comportement.

Exemple 4 (Exemple de comportements)**.** Reprenons l'exemple de l'opération *freiner* décrite dans la FIGURE 3.11. Cette opération dispose de trois comportements. Le tableau de la FIGURE 3.12 suivant identifie les gardes et les effets de chaque comportement de cette opération.

3.3 Génération de tests dans CertifyIt

CertifyIt, qui prend en entrée des modèles écrits dans le formalisme décrit dans ce chapitre, est un générateur de tests fonctionnels qui visent à exercer tous les comportements

FREIN_Stop = [vitesse - delta ≤ 0 and vitesse - delta = 0](vitesse := 0)
FREIN_Stop_Urgent = [vitesse - delta ≤ 0 and vitesse - delta ≠ 0](vitesse := 0)
FREIN_Frein_Normal = [vitesse - delta ≤ 0](vitesse := vitesse - delta)

FIGURE 3.12 – Comportements de l'opération *freiner*

atomiques des opérations du modèle. La stratégie de génération de CertifyIt est en deux étapes. Dans un premier temps, il identifie les cibles à atteindre à partir de la description en OCL4MBT des comportements des opérations. Ensuite, le moteur de preuve embarqué dans CertifyIt cherche un chemin faisable dans le modèle permettant d'atteindre chaque cible de test. Nous décrivons ces deux étapes.

3.3.1 Calcul des cibles de tests

La première étape est l'identification des cibles de test à atteindre. Le moteur de CertifyIt détermine tous les comportements du système à couvrir à partir des expressions OCL utilisée pour la représentation des comportements du modèle UML. En UML4MBT et OCL4MBT, un comportement désigne :

— une branche d'exécution dans une opération,
— une transition (interne ou non) dans un diagramme d'états-transitions,
— une action d'entrée ou de sortie d'un état du diagramme états-transitions.

Les cibles de tests sont, similairement aux comportements, des séquences de substitution gardées. Elles sont calculées par l'application de règles de production γ sur un comportement C_{pt} du modèle.

Définition 12 (Cible de test)**.** Une cible de test C_t est une séquence d'actions éventuellement gardées. Elle est définie par :

$$\begin{aligned} C_t &= Seq_{i=0}^{n} act_i \\ &= [guard_0]var_0 := val_0; \ldots [guard_n]var_n := val_n \end{aligned}$$

Dans [Gra08], l'auteur définit de manière générique les règles de production de cibles. Cependant, la règle de production de cibles de tests dans CertifyIt utilise le critère *Condition/Decision Coverage* (C/DC).

Définition 13 (Règle de production $\gamma_{C/DC}$)**.** La règle de production $\gamma_{C/DC}$ permettant la production de cibles de test en appliquant le critère C/DC est défini par :

$$\gamma_{C/DC}(Seq_{i=0}^{n}[\bigvee_{k=0}^{m_i} cond_k]a_i) = \bigcup_{j\in 0..n} \bigcup_{k\in 0..m_j} \{Seq_{i=0}^{j-1}[g_i]a_i; [cond_k]a_j; Seq_{i=j+1}^{n}[g_i]a_i\}$$

avec m_i le nombre de clauses *cond* dans la disjonction de la garde i, les a_i sont des actions et les g_i sont des gardes d'actions.

Cette règle considère toutes les gardes comme étant des disjonctions de clauses $\bigvee_{k=0}^{m_i} cond_k$. Pour chacune de ces clauses c, cette règle génère une cible de test avec toutes les gardes et les actions du comportement d'origine, mais en remplaçant la garde de c par c.

Exemple 5 (Production $\gamma_{C/DC}$)**.** Soit l'exemple d'opération OCL suivant :

```
1   if A or B then
2     x = 2
3   else
4     y = 3 and
5     if C or D then
6       x = 5
7     else
8       x = 4
9     endif
10  endif and
11  z = 6
```

Cette opération contient les trois comportements suivants :

C_{pt1} = [A or B](x = 2) ; (z = 6)
C_{pt2} = [not(A or B)](y = 3) ; [C or D](x = 5) ; (z = 6)
C_{pt2} = [not(A or B)](y = 3) ; [not(C or D)](x = 4) ; (z = 6)

A partir de ces comportements, nous utilisons la règle de production $\gamma_{C/DC}$ pour produire les cinq cibles de tests associées :

$\gamma_{C/DC}(C_{pt1})$ = {[A](x = 2) ; (z = 6),
[B](x = 2) ; (z = 6)}
$\gamma_{C/DC}(C_{pt2})$ = {[not(A or B)](y = 3) ; [C](x = 5) ; (z = 6),
[not(A or B)](y = 3) ; [D](x = 5) ; (z = 6)}
$\gamma_{C/DC}(C_{pt2})$ = {[not(A or B)](y = 3) ; [not(C or D)](x = 4) ; (z = 6)}

La génération de tests se fera donc vis-à-vis de ces cinq cibles de tests.

3.3.2 Génération de tests

Les cibles des tests et le modèle sont ensuite transformés puis transmis à l'outil *Prover Plugin*[19] qui va alors tenter de déterminer s'il existe un chemin dans le modèle (une séquence d'appels d'opérations) permettant d'atteindre et d'exécuter le comportement associé à la cible de test.

L'outil peut ensuite conclure sur l'atteignabilité du comportement ciblé de trois manières :

— *atteignable* signifie qu'il existe au moins un chemin dans le modèle qui permet l'activation de la cible,
— *inatteignable* signifie qu'il n'existe aucun chemin permettant d'activer le comportement de la cible ou que l'évaluation d'une expression dans le code OCL est indéfinie,
— *indéterminé* si la cible n'est ni atteignable, ni inatteignable.

Le statut *indéterminé* est un statut temporaire permettant de qualifier toutes les cibles jusqu'à ce que la preuve de leur (in)atteignabilité soit établie. Cependant, dans la pratique une preuve d'atteignabilité de cible de test peut ne pas être concluante dans un temps

19. Site Prover Plugin : `http://www.prover.com/product/prover_plugin` (2013)

imparti : il est donc possible d'avoir des cibles étiquetées comme étant *indéterminé* à l'issue de la génération de tests.

S'il existe un chemin permettant de réaliser la cible de test, alors ce chemin est un test qui permet de couvrir la cible de test. Le test est découpé en plusieurs parties :

- le *préambule* représente la séquence d'appels d'opérations depuis l'état initial jusqu'à l'état qui permet d'activer le comportement associé à la cible de test considérée,
- le *corps* du test représente l'activation effective du comportement dont la cible considérée est issue,
- le *postambule* représente une séquence d'opérations qui permet de ramener le système à l'état initial. Le postambule peut être vu comme une cible de test et peut ainsi être qualifié d'*atteignable* (en donnant une séquence ramenant le système dans l'état initial), *inatteignable* (l'état initial ne peut pas être atteint), ou *indéterminé*.

Pour chaque préambule qui atteint avec succès l'état visé, le cas de test est alors la concaténation de ce préambule et de l'invocation de l'opération comportant le comportement visé avec les valeurs de paramètres adéquates.

Enfin, CertifyIt donne la possibilité d'attribuer deux modes aux opérations d'observation. Une opération d'observation peut être appelée systématiquement après chaque pas du test, ou elle peut être appelée plus finement seulement si l'un des attributs du système qu'elle utilise a pu subir une modification au cours du pas de test considéré. Ce dernier mode permet de limiter le nombre d'opérations d'observation considérées à chaque pas. La relation de conformité se fonde uniquement sur les valeurs de retour des opérations.

Exemple 6 (Génération de test)**.** Le code OCL de l'opération *freiner* décrit à la Figure 3.11, fait apparaître les trois comportements suivants @AIM:FREIN_Stop_Urgent, @AIM:FREIN_Stop, et @AIM:FREIN_Frein_Normal. Puisqu'il n'y a aucune disjonction dans les gardes de ces comportement, alors les cibles produites par la règle $\gamma_{C/DC}$ sont équivalentes aux comportements.

Dans un premier temps, afin d'activer ces comportements, il est nécessaire d'atteindre l'état *EnMouvement* de la machine à états-transitions. En effet, ces comportements ne peuvent être activés que lorsque la vitesse est positive. Par conséquent, le préambule de test est commun pour ces 3 comportements et est défini par la séquence *demarrer() ;accelerer(2)*. Pour cet exemple, 2 est arbitraire et nous permet de tout activer.

Nous considérons dans notre exemple que l'opération *compteurVitesse* est systématiquement ajoutée à chaque pas.

Les tests correspondant aux trois comportements sont alors décrits par :

Cible de test	Opération		Observation
@AIM:FREIN_Frein_Normal	demarrer()		
	compteurVitesse()	→	0
	accelerer(2)		
	compteurVitesse()	→	2
	freiner(1)		
	compteurVitesse()	→	1
@AIM:FREIN_Stop	demarrer()		
	compteurVitesse()	→	0
	accelerer(2)		
	compteurVitesse()	→	2
	freiner(2)		
	compteurVitesse()	→	0
@AIM:FREIN_Stop_Urgent	demarrer()		
	compteurVitesse()	→	0
	accelerer(2)		
	compteurVitesse()	→	2
	freiner(3)		
	compteurVitesse()	→	0

3.4 Exemple fil rouge

Afin d'illustrer les différents concepts introduits dans ce document, nous présentons l'application *eCinema*. Cette application est un site web permettant la réservation de tickets pour des séances de cinéma. Elle propose pour les utilisateurs enregistrés, en navigant sur les pages adéquates, de réserver/supprimer les tickets placés dans leur panier d'achat. Nous détaillons dans un premier temps les exigences fonctionnelles associées à *eCinema*, puis la modélisation de l'application proprement dite.

3.4.1 Exigences fonctionnelles

La spécification *eCinema* identifie plusieurs exigences fonctionnelles :

1. un utilisateur authentifié sur le site doit pouvoir accéder aux services proposés (acheter/supprimer des tickets et voir ses achats),
2. le système doit permettre l'enregistrement d'un nouvel utilisateur seulement si le nom d'utilisateur n'est pas déjà utilisé par un utilisateur enregistré et que son mot de passe est valide,
3. un utilisateur peut acheter des tickets de cinéma s'il reste encore des tickets disponibles,
4. le système doit pouvoir afficher les tickets achetés par un utilisateur,
5. un utilisateur doit pouvoir supprimer un, plusieurs, voire même tous les tickets qu'il a achetés (avant le début de la séance),

6. un utilisateur authentifié doit pouvoir naviguer entre les pages du site web et accéder aux services proposés.

Dans notre modèle, ces exigences fonctionnelles sont réalisées par des comportements d'opérations du système. Ainsi, les exigences en rapport avec l'achat de tickets sont réalisées dans l'opération *buyTicket*. En prenant en compte les différents aspects des exigences (assurer un mécanisme d'enregistrement et d'authentification d'utilisateurs, achat et suppression de tickets, navigation à l'intérieur du site, ...), nous pouvons extraire différents comportements en accord avec ces exigences.

3.4.2 Description du modèle

Le modèle *eCinema* est formalisé en utilisant le formalisme UML/OCL décrit précédemment. Nous utilisons donc un diagramme de classe pour les structures de données de l'application et un diagramme d'objet pour représenter l'état initial.

3.4.3 Diagramme de classes

L'application fait intervenir différentes entités telles que l'utilisateur, qui doit s'enregistrer/s'authentifier, les films à l'affiche et les tickets à vendre. Le diagramme de classes UML, représenté en FIGURE 3.14, contient donc les classes suivantes :

— *eCinema*, le système sous test, qui contient les interfaces de l'application,
— *User* qui représente un utilisateur dans le système,
— *Movie* qui représente un film dans le système,
— *Ticket* qui représente un ticket pour un film.

Les utilisateurs connus (*all_registered_users*) sont rattachés au système par l'association *knows* et l'utilisateur courant *current_user* est identifié par l'association *has*. Les films actuellement à l'affiche sont identifiés par l'association *offers*. Enfin, chaque film propose un ensemble de tickets (*is_reserved_through*) qui peuvent être réservés par un utilisateur (*has_reserved*).

Les opérations de la classe *eCinema* peuvent être classées en trois catégories : celles relatives au mécanisme d'enregistrement et d'authentification, celles relatives au système de réservation de tickets et celles relatives à la navigation entre les différentes pages.

Les opérations suivantes sont en rapport avec le mécanisme d'enregistrement et d'authentification des utilisateurs dans le système :

— *register(USER_NAMES, PASSWORDS)* permet d'enregistrer un nouvel utilisateur dans le système,
— *unregister(USER_NAMES)* permet de supprimer un utilisateur du système,
— *login(USER_NAMES, PASSWORDS)* permet à un utilisateur enregistré de s'authentifier dans le système,
— *logout()* permet à un utilisateur de se déconnecter du système.

Les opérations suivantes interagissent directement avec le système d'achat et de suppression de tickets de l'application :

— *buyTicket(TITLES)* permet d'acheter des tickets, s'il en reste,
— *deleteTicket(TITLES)* permet de supprimer un ticket acheté pour un film particulier,

Opérations	**Exigences fonctionnelles**	**Comportements**
buyTicket	Aucun utilisateur authentifié, échec de l'achat	BUY_Login_Mandatory
	Achat du dernier ticket pour un film	BUY_Last_Ticket_Sold
	Achat normal de ticket	BUY_Nominal_Success
	Tous les tickets ont déjà été vendus, échec de l'achat	BUY_Sold_Out
	Achat uniquement sur la page d'accueil, échec de l'achat	BUY_Wrong_Page
deleteAllTickets	Suppression de tous les tickets d'un utilisateur	REMALL_Del_All_Tickets
	Pas de ticket à supprimer, échec de la suppression	REMALL_No_Tickets
	Suppression des tickets uniquement sur la page de visualisation du panier, échec de la suppression	REMALL_Wrong_Page
deleteTicket	Suppression d'un ticket pour un certain film	REM_Del_Ticket
	Pas de ticket à supprimer pour le film sélectionné, échec de la supression	REM_No_Ticket_To_Delete
	Suppression de tickets uniquement sur la page de visualisation du panier, échec de la suppression	REM_Wrong_Page
goToHome	Navigation à la page d'accueil par la page de visualisation	NAV_HOME_From_Display
	Navigation à la page d'accueil par la page d'enregistrement	NAV_HOME_From_Register
	Navigation réflexive sur la page d'accueil, échec de la navigation	NAV_HOME_Wrong_Page
goToRegister	Navigation à la page d'enregistrement	NAV_REG_To_Register
	Navigation uniquement à partir de la page d'accueil, échec de la navigation	NAV_REG_Wrong_Page
login	Authentification réussie	LOGIN_Success
	Nom d'utilisateur vide, échec de l'enregistrement	LOGIN_Empty_User_Name
	Nom d'utilisateur invalide, échec de l'enregistrement	LOGIN_Invalid_User_Name
	Mot de passe invalide, échec de l'enregistrement	LOGIN_Invalid_Password
	Utilisateur déjà authentifié, échec de l'enregistrement	LOGIN_Already_Logged_In
	Authentification uniquement à partir de la page d'accueil, échec de l'enregistrement	LOGIN_Wrong_Page
logout	Désauthentification réussie	LOGOUT_Logout
	Aucun utilisateur authentifié, échec de la désauthentification	LOGOUT_Not_Logged_In
	Désauthentification uniquement à partir de la page d'accueil	LOGOUT_Wrong_Page
registration	Enregistrement réussi	REG_Success
	Nom d'utilisateur vide, échec de l'enregistrement	REG_Empty_User_Name
	Mot de passe vide, échec de l'enregistrement	REG_Empty_Password
	Utilisateur déjà enregistré, échec de l'enregistrement	REG_Login_Already_Exists
	Désauthentification uniquement à partir de la page d'enregistrement	REG_Wrong_Page
showBoughtTickets	Navigation à la page de visualisation du panier	DIS_Check_Basket
	Navigation uniquement pour un utilisateur authentifié, échec de la navigation	DIS_Not_Logged_In
	Navigation uniquement à partir de la page d'accueil, échec de la navigation	DIS_Wrong_Page
unregister	Suppression de l'utilisateur courant de la base	UNREG_Unregister
	Aucun utilisateur authentifié, échec de la suppression	UNREG_Not_Logged_In
	Suppression uniquement à partir de la page d'enregistrement, échec de la suppression	UNREG_Wrong_Page

FIGURE 3.13 – Opérations et comportements du modèle eCinema

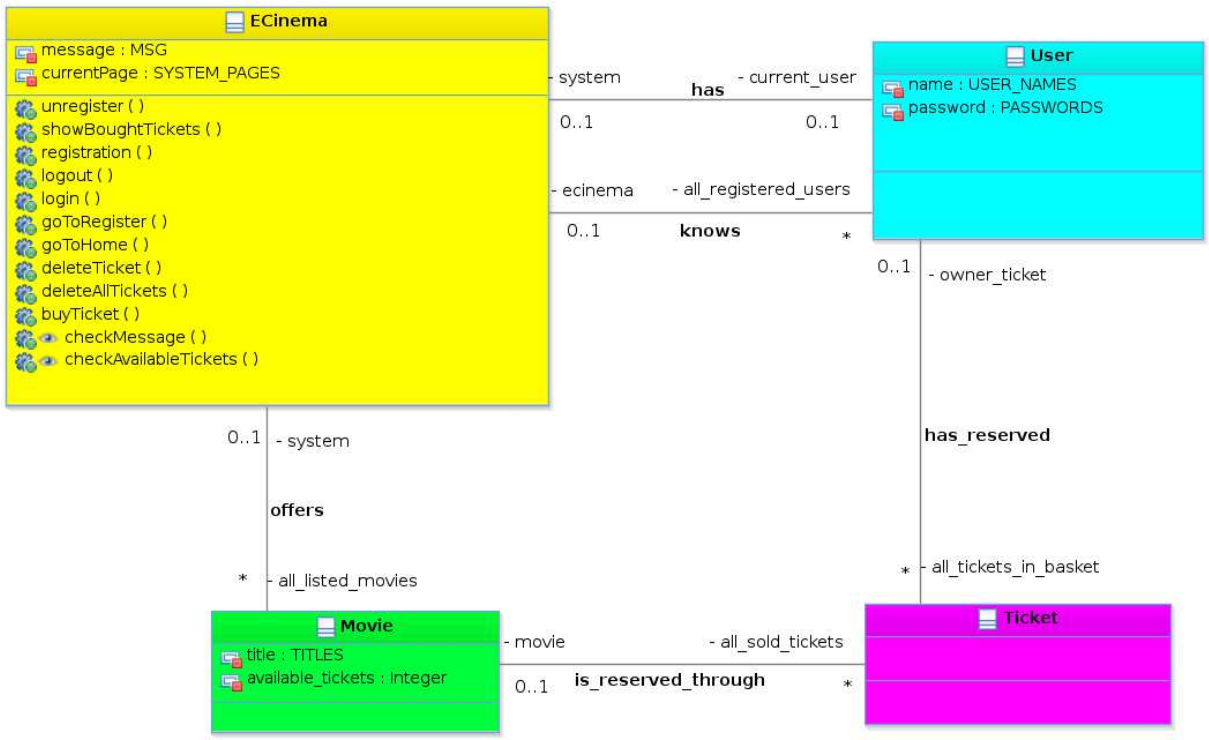


FIGURE 3.14 – Diagramme de classe de eCinema

— *deleteAllTickets()* permet de supprimer tous les tickets du panier d'un utilisateur.

Le changement de page est simulé par l'attribut *currentPage* qui peut prendre trois valeurs définies dans la classe d'énumération *SystemState* :

— REGISTER si l'utilisateur est à la page d'enregistrement,
— WELCOME si l'utilisateur est à la page d'accueil,
— DISPLAY si l'utilisateur est sur la page de visualisation de tickets.

Il est possible de passer d'une page à une autre par le biais d'opérations adéquates :

— *goToRegister()* pour passer de la page d'accueil à la page d'enregistrement,
— *goToHome()* pour retourner à la page d'accueil, quelle que soit la page actuelle,
— *showBoughtTickets()* pour passer de la page d'accueil à la page de visualisation des tickets déjà achetés.

Il est à noter que pour passer de la page d'enregistrement à la page de visualisation (et inversement) il est nécessaire de repasser par la page d'accueil. La FIGURE 3.13 présente tous les comportements associés à chaque opération ainsi que le tag qui leur est associé à l'intérieur du modèle.

Enfin, le modèle dispose aussi de deux opérations d'observation :

— *checkMessage()* permet de récupérer la valeur de l'attribut *message* de la classe *ECinema*. En pratique elle nous est utile pour observer le dernier message renvoyé par la dernière opération.
— *checkAvailableTickets(TITLES)* permet de récupérer le nombre de tickets disponibles pour le film passé en paramètre.

```
1  --@REQ: BASKET_MNGT/BUY_TICKETS
2  if self.currentState = SYSTEM_STATES::WELCOME then
3   if self.all_listed_movies→any(t : Movie | t.title = in_title)
4                .available_tickets >= 1 then
5    if self.current_user.oclIsUndefined() then
6     --@AIM: BUY_Login_Mandatory
7     message = MSG::LOGIN_FIRST
8    else
9     let targeted_movie : Movie = self.all_listed_movies
10                 →any(t : Movie | t.title = in_title) in
11    let unallocated_ticket : Ticket = (Ticket.allInstances())
12                 →any(owner_ticket.oclIsUndefined()) in
13     self.current_user.all_tickets_in_basket→includes(unallocated_ticket) and
14     targeted_movie.all_sold_tickets→includes(unallocated_ticket) and
15     if targeted_movie.available_tickets > 1 then
16         --@AIM: BUY_Nominal_Success
17         targeted_movie.available_tickets = targeted_movie.available_tickets - 1
18     else
19         --@AIM: BUY_Last_Ticket_Sold
20         targeted_movie.available_tickets = 0
21     endif and
22     message= MSG::NONE
23    endif
24   else
25    --@AIM: BUY_Sold_Out
26    message = MSG::ALL_MOVIES_SOLD_OUT
27   endif
28  else
29   --@AIM: BUY_Wrong_Page
30   message = MSG::WRONG_PAGE
31  endif
```

FIGURE 3.15 – Code OCL de l'opération *buyTicket*

Exemple 7 (Exemple de code OCL)**.** La FIGURE 3.15 présente le code OCL de la postcondition de l'opération *buyTicket*. Nous retrouvons l'exigence fonctionnelle de haut niveau @REQ :BASKET_MNGT/BUY_TICKETS ainsi que tous les comportements différents qui composent cette opération, identifiés dans la FIGURE 3.13.

La FIGURE 3.16 présente les séquences d'actions gardées des différents comportements de l'opération *buyTicket*.

3.4.4 Diagramme d'objets

L'état initial du système est représenté par le diagramme d'objets présenté à la FIGURE 3.17. Il contient plusieurs instances :

- *sut*, l'unique instance de la classe du système sous test *eCinema*. Les appels d'opération utiliseront cette instance comme référentiel.
- deux instances d'utilisateurs : une instance *registeredUser* pour un utilisateur déjà enregistré, déjà liée à l'instance *sut*, et un utilisateur *unregisteredUser* qui ne l'est pas.
- deux instances de films *film1* et *film2*, le premier proposant sept places à vendre et le deuxième seulement trois.
- dix instances de la classe *Ticket*

@AIM :BUY_Wrong_Page =[self.currentUser¬SYSTEM_STATES : :WELCOME](message :=WRONG_PAGE)

@AIM :BUY_Sold_Out =[self.currentUser = SYSTEM_STATES : :WELCOME and
self.all_listed_movies→any(t :Movie | t.title = in_title).available_tickets<1)]
(message := ALL_TICKETS_SOLD_OUT)

@AIM :BUY_Login_Mandatory =[self.currentUser = SYSTEM_STATES : :WELCOME and
self.all_listed_movies→any(t :Movie | t.title = in_title).available_tickets≥1) and
self.currentUser.oclIsUndefined()](message := LOGIN_FIRST)

@AIM :BUY_Nominal_Success =[self.currentUser = SYSTEM_STATES : :WELCOME and
self.all_listed_movies→any(t :Movie | t.title = in_title).available_tickets ≥ 1) and
not(self.currentUser.oclIsUndefined())]
(targeted_movie := self.all_listed_movies→any(t :Movie | t.title = in_title)) ;
(unallocated_movie := Tickets.allInstances()→any(owner_ticket.oclIsUndefined())) ;
[targeted_movie.availableTickets > 1]
(targeted_movie.availableTickets = targeted_movie.availableTickets-1)

@AIM :BUY_Last_Ticket_Sold =[self.currentUser = SYSTEM_STATES : :WELCOME and
self.all_listed_movies→any(t :Movie | t.title = in_title).available_tickets ≥ 1) and
not(self.currentUser.oclIsUndefined())]
(targeted_movie := self.all_listed_movies→any(t :Movie | t.title = in_title)) ;
(unallocated_movie := Tickets.allInstances()→any(owner_ticket.oclIsUndefined())) ;
[targeted_movie.availableTickets = 1]
(targeted_movie.availableTickets = targeted_movie.availableTickets-1)

FIGURE 3.16 – Comportements de l'opération *buyTicket*

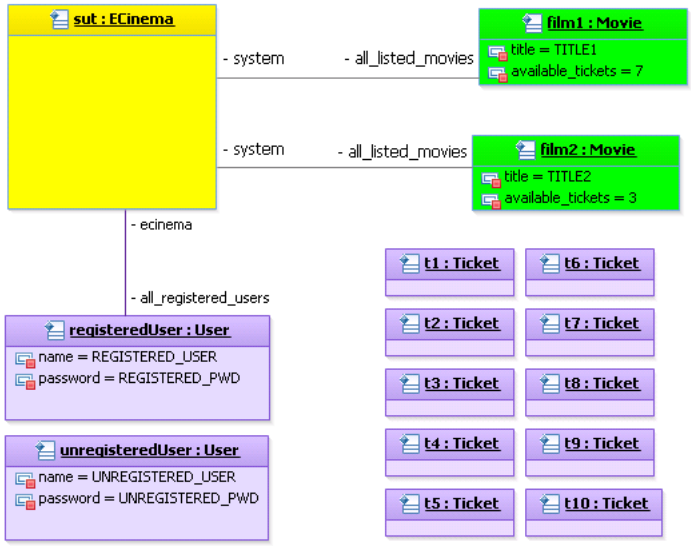


FIGURE 3.17 – Diagramme d'objets initial de eCinema

A l'état initial, les tickets sont déjà présents, même s'ils n'ont pas encore été vendus et ne correspondent alors à rien de concret.

3.4.5 Exemples de tests générés par CertifyIt

Nous avons décrit dans la section 3.3 la manière dont CertifyIt génére les tests à partir d'un modèle donné. Nous donnons ici quelques exemples de tests générés sur notre exemple fil rouge.

Déconnexion de l'utilisateur courant. La couverture du comportement de déconnexion de l'utilisateur courant @AIM :LOGOUT_Logout est assuré par le test suivant :

login(REGISTERED_USER, REGISTERED_PWD)	→	AIM :LOGIN_Success
logout()	→	AIM :LOGOUT_Logout

Echec d'achat de ticket. L'achat de ticket peut échouer de plusieurs manières mais nous nous intéressons ici au cas où il n'existe plus de ticket pour le film sélectionné. Ainsi, afin d'activer le comportement @AIM :BUY_Sold_Out, il est nécessaire d'acheter tous les tickets d'un même film avant de l'activer. Ce comportement est couvert par le test suivant :

login(REGISTERED_USER, REGISTERED_PWD)	→	AIM :LOGIN_Success
buyTicket(TITLE2)	→	AIM :BUY_Success
buyTicket(TITLE2)	→	AIM :BUY_Success
buyTicket(TITLE2)	→	AIM :BUY_Success
buyTicket(TITLE2)	→	AIM :BUY_Sol_Out

3.5 Synthèse

UML et OCL sont des langages de modélisation qui, par leur expressivité, permettent de réaliser des modèles dédiés au test (model-based testing).

Dans UML, un modèle est décrit à l'aide de diagrammes et de contraintes associées à différents éléments de ces diagrammes. Cependant, dans le cadre de notre approche, nous ne considérons qu'un sous-ensemble des diagrammes décrits par le standard UML : ce sous-ensemble est appellé UML4MBT et utilise les diagrammes de classes pour les structures de données, d'objets pour représenter l'état initial, et d'états-transitions pour l'aspect dynamique du système.

L'aspect comportemental est décrit en utilisant OCL4MBT, un dérivé du langage OCL. Ce langage est un sous-ensemble d'OCL, qui s'en démarque par les constructions disponibles, des sémantiques différentes pour certains opérateurs selon le contexte et une interprétation opérationnelle en tant que langage d'action. De plus, il permet d'annoter le code avec des commentaires spéciaux, les *tags*, qui associent un nom à un comportement ou un ensemble de comportements.

Dans le cadre de notre approche, nous faisons aussi l'hypothèse que nos modèles sont complètement défensifs : une opération est **toujours** activable quel que soit l'état du système. Cela implique de gérer des cas supplémentaires, tels que les valeurs non-définies pour les paramètres et les attributs. Ainsi, la précondition ne porte que sur les valeurs des paramètres de l'opération pour ne considérer que certaines valeurs, afin d'éviter l'utilisation de valeurs trop grandes pour la génération de test, mais pas sur les variables

d'état du système. Cette pratique, courante dans le MBT, permet de limiter l'explosion combinatoire sur les transitions du graphe d'états.

Nous avons aussi décrit le processus de génération de tests implémenté dans l'outil CertifyIt de Smartesting. Cet outil fonde sa génération de test sur l'activation de cibles de tests, issues des comportements du modèle par l'application d'une règle de production. Un moteur de preuve tente de déterminer si ces cibles sont atteignables dans le modèle avant de rechercher un chemin pour les atteindre. Ces chemins sont ensuite considérés comme des tests permettant de couvrir ces cibles.

Cependant, cette technique permet d'activer les comportements en ne considérant systématiquement que les chemins les plus courts pour y parvenir. Ainsi, il peut être intéressant pour un ingénieur de test de considérer d'autres chemins qui sont le reflet de sa connaissance du modèle et de son savoir-faire. A cette fin, nous proposons dans le chapitre suivant, un langage de scénarios qui permet à l'ingénieur de définir ses propres séquences d'opérations.

Deuxième partie

Contributions

Chapitre 4

Génération de tests à partir de scénarios

Sommaire

4.1 Langage de scénarios 64
4.1.1 Syntaxe 64
4.1.2 Sémantique des instructions élémentaires 66
4.1.3 Sémantique des instructions de contrôle de flot 69
4.2 Génération de tests 72
4.2.1 Dépliage du scénario 73
4.2.2 Mise en œuvre 74
4.2.3 Exemple de dépliage 76
4.2.4 Comment recourir à l'expertise de l'ingénieur ? 79
4.3 Synthèse 81

Dans ce chapitre, nous proposons un langage d'expression de spécifications de cas de tests. En effet, les propriétés dynamiques d'un système peuvent représenter des enchaînements d'évènements du système, ces évènements pouvant prendre la forme d'appels d'opérations ou de changements d'état du système. Le langage que nous proposons permet de formaliser nos spécifications de cas de test sous une forme proche d'un langage de programmation impératif, en tant qu'enchaînement d'instructions. Ce langage tire son origine du projet POSE [Tis09] où le langage de scénarios utilisé est fondé sur les expressions régulières dont il reste très proche. Il propose des directives de pilotage destinées à guider le dépliage du scénario lors de la génération de tests. Ainsi, pour chaque point de choix du scénario, il est possible de spécifier si l'on désire considérer une seule ou toutes les séquences concrètes dérivées du dépliage de ce point de choix particulier. Il est cependant à noter que cette formalisation était très lourde à écrire, c'est pourquoi nous proposons une amélioration de la syntaxe afin de la rendre plus proche d'un langage de programmation et faciliter sa prise en main.

Dans un premier temps, nous donnons dans la section 4.1 la définition de notre langage de scénarios, sa syntaxe et sa sémantique. Ensuite, dans la section 4.2, nous décrivons la technique de génération de tests à partir de ces scénarios, accompagnée d'un exemple

illustrant ce processus, avant de conclure sur les forces et les faiblesses d'une approche à partir de scénarios.

4.1 Langage de scénarios

Nous décrivons maintenant le langage de scénarios et la sémantique à base de graphes de flot de contrôle acycliques qui lui est associée.

4.1.1 Syntaxe

La syntaxe de notre langage de scénarios ressemble à la syntaxe des langages de programmation courant tels que *Java* ou *C*. Cette ressemblance vise une prise en main rapide. La FIGURE 4.1 donne la syntaxe du langage. Les mot-clefs sont représentés en **gras** et les symboles terminaux sont soulignés. Ils ont la signification suivante :

- scenarioName est le nom d'un scénario,
- $varName est le nom d'une variable locale du scénario ou d'une boucle,
- instance est le nom d'une instance du modèle,
- paramName est le nom d'un paramètre d'une opération,
- integer est un entier,
- modelEnumeration est un litéral d'une énumération du modèle,
- predicate est un prédicat OCL,
- tagName est le nom d'un tag.

Enfin, pour représenter les listes d'éléments, nous utilisons la forme e^r_s où e est l'élément à répéter, r est le type de répétition (qui peut être un entier, un intervalle, $*$, $+$ ou ?) et s est le séparateur entre les éléments de la liste.

Les éléments de base du langage sont des instructions dont il existe plusieurs types. L'élément le plus simple correspond à un appel d'opération nommée ou quelconque du système (règle *OPERATION_CALL*) dont il est possible de spécifier les comportements (identifiés par des tags) à utiliser (resp. à interdire) par l'utilisation du mot-clef `including` (resp. `excluding`). Si un appel d'opération peut donner lieu à plusieurs instances de cet appel, par exemple avec des paramètres laissés libres, les mot-clefs `one` et `all` nous permettent de spécifier si l'on souhaite conserver une seule ou toutes les instances de cet appel respectivement. Enfin, il est aussi possible de récupérer la valeur de retour dans une variable locale afin de la réutiliser plus tard.

Nous avons aussi la possibilité de faire référence à un scénario qui existe déjà (règle *SCENARIO_CALL*) notamment dans le but de factoriser certaines parties du scénario.

Nous disposons d'une structure de contrôle conditionnelle *if-then-else* (règle *IFTHENELSE*) permettant de choisir l'exécution d'instructions en fonction de la valeur de vérité de *CONDITION*.

Enfin, nous disposons d'instructions plus complexes (des points de choix et des itérateurs) qui permettent de choisir entre plusieurs chemins (règle *CHOICE*), de répéter un certain nombre de fois un ensemble d'instructions (règle *REPEAT*) ou de définir différents chemins en fonction de la valeur d'une variable prise dans une liste de valeurs (règle *FOR*).

```
SCENARIO ::= scenario scenarioName ( PARAMETERS_LIST ){ (INSTRUCTIONS ;)*}
PARAMETERS_LIST ::= $paramName*,
INSTRUCTIONS ::= OPERATION_CALL
               | SCENARIO_CALL
               | IFTHENELSE
               | CHOICE
               | ASSERT
               | REPEAT
               | FOR
OPERATION_CALL ::= ($varName =)? INSTANCE . OP_CALL ((including | excluding){TAG_LIST})?
                   (one | all)?
INSTANCE ::= instance | $I
OP_CALL ::= $OP(\{OP_LIST})?
          | opName (VALUES_LIST)
OP_LIST ::= opName+,
TAG_LIST ::= TAG+,
VALUES_LIST ::= VALUE*,
VALUE ::= _
        | $varName
        | integer
        | modelEnumeration
SCENARIO_CALL ::= scenarioName (VALUES_LIST)
IFTHENELSE ::= if(CONDITION){INSTRUCTIONS} (else{INSTRUCTIONS})?
CHOICE ::= (exclusive)? choice{INSTRUCTIONS} (or{INSTRUCTIONS})+
ASSERT ::= assert(CONDITION)
CONDITION ::= predicate on instance
REPEAT ::= (repeat | unfold)((at most integer)
                           | (between integer and integer)
                           | integer) times{ INSTRUCTIONS } (until (CONDITION))?
FOR ::= (any | foreach) $varName ((between integer and integer)
                                | (in{VALUES_LIST}){INSTRUCTIONS}
TAG ::= TAG_TYPE : tagName
TAG_TYPE ::= AIM | REQ
```

FIGURE 4.1 – Syntaxe du langage de scénarios

Il est à noter que chacune des instructions complexes possède deux mots-clef qui permettent de distinguer la directive de pilotage du dépliage associée à cette instruction. Nous avons, d'une part, les instructions *choice*, *foreach* et *unfold* qui correspondent à des instructions où l'on considère toutes les séquences générées par le point de choix. D'autre part, nous avons *exclusive choice*, *any* et *repeat* qui correspondent à des instructions où l'on ne considère qu'une seule séquence. Cet aspect d'assistance au dépliage des scénarios est important car il permet de limiter une explosion combinatoire tout en restant suffisamment fin pour pouvoir changer ces directives au cas par cas. Il est à noter que chaque paire d'instructions partage la même sémantique de graphe (à la différence de la représentation du noeud de choix) mais l'interprétation du graphe lors du dépliage n'est pas la même.

Nous associons à ce langage une sémantique de graphes de flot de contrôle auquel nous ajoutons la possibilité de distinguer les options de pilotage associées à chacun des

points de choix du graphe. Nous définissons la sémantique de manière schématique en représentant graphiquement les graphes de flot de contrôle associés aux instructions du langage.

Définition 14 (Graphe de flot de contrôle du langage de scénarios)**.** Un graphe de flot de contrôle est un graphe orienté acyclique $(In, Cp, i, f, Calls, T, L)$ avec :

- In l'ensemble des nœuds représentant des appels d'opérations complètement valués, c'est-à-dire où les paramètres ne possèdent qu'une seule valeur,
- Cp l'ensemble des nœuds représentant les points de choix introduits par les instructions de choix. Il est divisé en deux sous-ensembles distincts : Cp_i représente les points de choix inclusifs (instructions `choice`, `unfold`, `foreach` et les appels d'opérations partiellement valués avec le mot-clef `all`) et Cp_e représente les points de choix *exclusifs* (instructions `exclusive choice`, `repeat`, `any` et les appels d'opérations partiellement valués avec le mot-clef `one`),
- i le nœud initial du graphe ($i \in In$),
- f le nœud final du graphe ($f \in In$),
- $Calls$ est une fonction qui permet la correspondance pour chaque noeud d'appel d'opération valué, l'opération considérée *op*, sa liste de valeurs de paramètres *params* et l'ensemble des comportements *tags* autorisé qui doivent être activés $Calls \in (In \rightarrow \langle op \times params \times tags \rangle)$,
- L l'ensemble des étiquettes du graphe, ici des prédicats, décrits en OCL,
- T l'ensemble des arcs du graphe $T \subseteq (In \backslash \{f\} \cup Cp) \times L \cup \{\top\} \times (In \backslash \{i\} \cup Cp)$.

Nous définissons ici la sémantique avec des graphes de flot de contrôles acycliques. En effet, les constructions permettant les boucles telles que *REPEAT* et *FOR* sont bornées, ce qui nous permet de leur associer une sémantique où les boucles sont déjà dépliées, en générant autant de chemin que d'itérations. Cette acyclicité nous permet de nous assurer que l'exploration du graphe finit par atteindre le nœud final du graphe.

Chaque type de nœud a une représentation particulière : les nœuds de In sont représentés par des cercles, les nœuds de Cp_e sont représentés par des losanges, les nœuds de Cp_i sont représentés par des losanges au contour épais, le nœud initial i est représenté par un disque plein et le nœud final est représenté par disque plein au contour doublé. Les transitions portent les étiquettes qui leur sont associées. Dans un souci de lisibilité, les étiquettes qui portent le prédicat $\top$ (prédicat *vrai*) ne sont pas représentées. Dans les représentations graphiques des sémantiques, nous représentons par un nœud rectangulaire le graphe de flot de contrôle (GFC) du bloc *INSTRUCTIONS* (dénoté par A, B et A_i) que l'on peut trouver dans l'expression des instructions *if-then-else* ou *choice* par exemple.

Enfin, il est à noter que l'état final et l'état initial ne possèdent pas de sémantique particulière et n'indiquent que le point de départ (resp. la fin) du graphe. Ainsi, ils ne sont pas considérés lors de l'exploration du graphe en tant que nœuds d'appel d'opération.

Nous présentons maintenant les différentes instructions du langage et leur sémantique sous forme de graphe de flot de contrôle acyclique associée.

4.1.2 Sémantique des instructions élémentaires

Nous présentons dans un premier temps les instructions élémentaires du langage.

Appel d'opération

Il s'agit de l'instruction de plus bas niveau du langage qui fait appel à une opération du modèle. Il est possible d'appeler une opération particulière par son nom (*login*, *logout*, *buyTicket*, ...) ou une opération quelconque du modèle en utilisant l'alias *$OP*. Pour chaque invocation d'opération, nommée ou quelconque, nous pouvons aussi spécifier quels comportements sont considérés (ou exclus) afin de restreindre les possibilités.

Dans le cas d'opérations nommées, il est possible de spécifier des valeurs aux paramètres pour contraindre l'appel. Si tous les paramètres sont spécifiés alors on parle d'un appel d'opération *complètement valué*. Au contraire, nous parlons d'un appel d'opération *partiellement valué* si au moins un des paramètres peut prendre plusieurs valeurs.

Dans le cas des opérations quelconques, il est possible d'interdire certaines opérations par la construction $\$OP \backslash \{op_1, \ldots, op_n\}$. Lors du dépliage de cette construction, les opérations de cette liste ne seront alors pas considérées.

Enfin, il est possible de capturer la valeur de retour de l'opération dans une variable afin de la réutiliser plus tard, par exemple en tant que paramètre d'une autre opération ou dans le prédicat d'une assertion. La FIGURE 4.2 donne la sémantique sous forme de graphe de flot de contrôle d'un appel d'opération où A est une instruction d'appel d'opération. Il existe deux cas distincts. La FIGURE 4.2a donne la sémantique d'une opération complètement valuée (tous les paramètres ont une valeur associée). La FIGURE 4.2b représente la sémantique d'un appel d'opération dont au moins un paramètre est laissé libre (avec le caractère _) où chaque A_k représente un appel d'opération A pour une des valuations possibles des paramètres de l'opération laissés libres. Ce dépliage fait la combinatoire des valeurs de paramètres laissés libres.

Dans le cas d'une opération quelconque $OP, chaque branche correspond à toutes les valuations des opérations du modèle, à l'exception de celles qui apparaissent dans la liste d'exclusion. Il est à noter que, dans le cas de cette construction, la combinatoire des opérations et des paramètres associés peut conduire à une explosion du nombre de branches du point de choix et la situation s'aggrave avec l'utilisation de répétitions. Il est donc préférable d'éviter l'utilisation de cette construction et de spécifier au maximum les sous-séquences qu'elle peut représenter.

Dans les deux types d'appels d'opération il existe un nouveau point de choix, le type *inclusif* ou *exclusif* (par défaut) du point de choix dépend de la directive de pilotage associée à l'instruction (**all** ou **one** respectivement).

(a) Appel d'opération entièrement valuée

(b) Appel d'opération avec paramètres libres pouvant prendre n valeurs

FIGURE 4.2 – Sémantique d'un appel d'opération

Exemple 8 (Appel d'opération nommée)**.** Un exemple d'appel nommé est l'achat avec succès d'un ticket pour le film TITLE1. Ici, nous faisons explicitement l'appel à l'opération *buyTicket* avec TITLE1 en paramètre. De plus, nous forçons le fait que l'appel doit être

un succès. Les comportements avec succès de *buyTicket* sont ceux identifiés par les tags @AIM:BUY_Success et @AIM:BUY_Last_Ticket_Sold. Ainsi, dans notre langage, cet appel s'écrit :

```
sut.buyTicket(TITLE1) including {@AIM:BUY_Nominal_Success,
                                 @AIM:BUY_Last_Ticket_Sold};
```

Exemple 9 (Appel de $OP)**.** Un exemple d'appel d'opération quelconque est celui représentant les appels d'opération qui ne permettent pas de modifier le contenu du panier d'un utilisateur. Les opérations concernées sont *buyTicket*, qui permet d'en ajouter, *deleteTicket* et *deleteAllTickets* qui permettent d'en supprimer. Au lieu de faire un choix entre toutes les autres opérations, nous pouvons nous contenter d'écrire l'appel à une opération quelconque en restreignant les opérations interdites :

```
sut::$OP \ {buyTicket, deleteTicket, deleteAllTickets}
```

Appel de scénario

Cette règle permet d'invoquer, à partir d'un scénario de base (l'appellant) un autre scénario. Ceci a pour effet de substituer le graphe du scénario appellé à l'instruction représentant l'appel du scénario.

En pratique, la transition entrante sur le nœud de l'instruction d'invocation de scénario est redirigé sur le nœud pointé par l'état initial du sous-scénario et la transition sortante a comme source les nœuds qui pointent sur l'état final du sous-scénario. La Figure 4.3 montre ce processus de substitution. En bas à gauche, le scénario qui fait l'appel au scénario avec le nœud SC qui représente l'appel du scénario. En haut à gauche, le scénario à substituer. A droite, le résultat de la substitution. Les états initiaux et finaux sont retirés du scénario à substituer. Enfin, les prédicats portés sur les transitions entrantes (resp. sortantes) du nœud SC et les prédicats portés sur les transitions entrantes (resp. sortantes) du premier (resp. dernier) nœud sont combinés, par conjonction, sur la transition entrante (resp. sortante) résultante.

Figure 4.3 – Sémantique de l'appel de scénario

Assertion

L'instruction spéciale `assert` se distingue des autres dans le sens où ce n'est pas une instruction permettant de faire avancer le scénario (pas d'appel d'opération). En effet, elle permet de spécifier une condition sur l'état du système à satisfaire après la précédente instruction et avant l'exécution de l'instruction suivante.

La Figure 4.4 présente la sémantique de cette instruction. Nous considérons que chaque arc du graphe qui n'est pas étiqueté par un prédicat donné, est considéré comme portant le prédicat *vrai*.

Exemple 10 (Assertion)**.** Supposons qu'un utilisateur tente une authentification dans le système dans le but d'acheter un ticket. Après l'authentification, nous devons nous assurer qu'il reste au moins un ticket disponible pour au moins un film. C'est le rôle de l'instruction `assert` et dans notre langage, l'exemple s'écrit ainsi :

```
sut.login(_,_);
assert("all_listed_movies->any(available_tickets > 0)" on "sut");
sut.buyTicket(_) including {@AIM:BUY_Nominal_Success,
                            @AIM:BUY_Last_Ticket_Sold};
```

FIGURE 4.4 – Sémantique de `assert(P)`

Variables

Un autre aspect du langage est l'utilisation de variables. Ces variables peuvent être définies soit pour recevoir la valeur d'un retour d'une opération, soit en tant que variable d'itération dans les instructions `foreach` et `any`.

Il est possible de faire référence à une variable dans les contextes suivants :

— en tant que paramètre de l'invocation d'un scénario,
— en tant que paramètre d'une opération,
— dans n'importe quel prédicat OCL du scénario (pour une assertion ou le prédicat de fin des instructions `repeat` et `unfold`).

Les variables ne sont pas typées à leur déclaration (de la même manière que dans le langage Python). Elles ne sont typées que par le type de la valeur qu'elles contiennent.

4.1.3 Sémantique des instructions de contrôle de flot

Nous présentons maintenant la sémantique des instructions de contrôle de flot. Dans le cas de ces instructions, nous présentons de manière graphique les versions *exclusives*. La représentation graphique des versions *inclusives* est identique, mais le losange possède un contour gras.

Structure conditionnelle if-then-else

Cette instruction représente une exécution conditionnelle. Si un certain prédicat P est évalué à vrai, alors une séquence d'instructions est exécutée (branche *then*), dans le cas contraire, une autre séquence est exécutée (branche *else*). Dans le cas où la branche *else* est omise, alors cette branche équivaut à une séquence vide.

La FIGURE 4.5 donne la sémantique de la structure conditionnelle if-then en 4.5a et celle du if-then-else en 4.5b. Il est à noter que cette construction est nécessairement inclusive. En effet, elle permet le choix entre deux alternatives, qui peuvent être très différentes, contrairement à un *repeat* où les opérations sont les mêmes mais répétées plusieurs fois.

(a) Sémantique du `if(P)then{A}` (b) Sémantique du `if(P)then{A}else{B}`

FIGURE 4.5 – Sémantiques des structures conditionnelles `if-then` et `if-then-else`

Ainsi, pour la construction *if-then-else*, les deux chemins sont systématiquement considérés.

Exemple 11 (Structure if-then-else)**.** Un exemple de l'utilisation de la structure complète est le cas où l'utilisateur courant achète un ticket pour le film TITLE1 s'il en reste, sinon il se désauthentifie du système, déçu de ne pas pouvoir aller voir un film. Dans notre langage, ce scénario s'écrit :

```
if("self.all_listed_movies->any()->size()>0" on "sut")then{
  sut.buyTicket(TITLE1);
}else{
  sut.logout();
}
```

Exemple 12 (Structure if-then)**.** Supposons que l'utilisateur courant puisse acheter un ticket uniquement s'il en reste au moins un qui n'a pas été vendu. Dans ce cas, l'utilisateur peut acheter un ticket pour n'importe quel film. Dans notre langage, cela s'écrit ainsi :

```
if("Ticket.allInstances()->exists(Ticket t|
        t.ownerTicket.oclIsUndefined())" on "sut")then{
  sut.buyTicket(_);
}
```

La condition décrit le fait que parmi tous les tickets proposés à la vente, il en existe au moins un qui n'a pas de possesseur (`t.ownerTicket.oclIsUndefined()`).

Choix

FIGURE 4.6 – Sémantique de l'instruction `choice{`A_1`}or{`A_2`}or...or{`A_n`}`

Cette instruction permet de choisir, de manière non-déterministe, entre plusieurs séquences. Elle propose l'utilisation d'un mot-clef optionnel *exclusive* qui permet de modifier l'interprétation du choix lors du dépliage. En effet, sans mot-clef, le dépliage considère chaque bloc de l'instruction pour la génération de tests, alors qu'il n'en considère qu'un seul avec le mot-clef *exclusive*.

La FIGURE 4.6 donne la sémantique de l'instruction du choix avec le mot-clef *exclusive*.

Exemple 13 (Choix d'instruction)**.** Supposons que l'utilisateur courant puisse choisir d'acheter un ticket pour le film TITLE1 ou pour le film TITLE2. Dans notre langage, cela s'écrit :

```
choice{
  sut.buyTicket(TITLE1);
} or {
  sut.buyTicket(TITLE2);
}
```

Répétition

FIGURE 4.7 – Sémantique de `repeat/unfold between` m `and` n `times` {A} `until` (P)

Les instructions `repeat` et `unfold` permettent de répéter, un certain nombre de fois (fixe ou borné), une séquence d'instructions. Il est possible de spécifier une condition qui doit être respectée après la dernière itération de la boucle par l'intermédiaire d'un prédicat sur l'état du système avec le mot-clef `until`. Si cette condition est omise (`until` n'est pas utilisé), alors on considère que cette condition équivaut à *vrai*.

Enfin, ces instructions possèdent des variantes portant sur le nombre de répétitions avec l'utilisation de mot-clefs différents :

- `between` m `and` n `times` : la séquence d'instructions doit être répétée au moins m fois et au plus n fois,
- `at most` n `times` : la séquence d'instructions doit être répétée au plus n fois. Cette variante est équivalente à `between` 0 `and` n.
- `n times` : exactement n fois. Cette variante est équivalente à `between` n `and` n.

La FIGURE 4.7 donne la sémantique des instructions de répétition `repeat` et `unfold` qui est la même, indépendamment du choix de pilotage. Comme toutes les variantes de dépliage peuvent être réécrites à l'aide d'un intervalle, nous définissons uniquement la sémantique avec un intervalle spécifié par `between` m `and` n `times`.

Exemple 14 (Répétition d'instructions (borne fixe))**.** L'achat successif de trois tickets pour le film TITLE1, s'écrit simplement :

```
repeat 3 times {
  sut.buyTicket(TITLE1) including {@AIM:BUY_Nominal_Success,
                                   @AIM:BUY_Last_Ticket_Sold};
}
```

Exemple 15 (Répétition d'instructions (intervalle))**.** L'achat à répétition d'au moins 3 et au plus 5 tickets pour le film TITLE1 s'écrit de la manière suivante :

```
repeat between 3 and 5 times {
  sut.buyTicket(TITLE1) including {@AIM:BUY_Nominal_Success,
                                   @AIM:BUY_Last_Ticket_Sold};
}
```

Itération

Les instructions `foreach` et `any` permettent d'itérer une séquence d'instructions sur une variable dans un certain domaine.

FIGURE 4.8 – Sémantique de `foreach/any` $i `in` {$i_1$, ..., i_n}{A}

La FIGURE 4.8 donne la représentation graphique du graphe de flot de contrôle d'une instruction d'itération. Ici, les blocs $GFC(A(i_k))$ représentent le graphe de flot de contrôle des instructions à itérer en remplaçant chaque occurrence de la variable $i par la valeur i_k.

Exemple 16 (Itérations)**.** Supposons que l'on souhaite faire représenter les appels de *login* en fonction de toutes les combinaisons possibles de noms d'utilisateurs (enregistré et non enregitré) et de mots de passe (le mot de passe valide de l'utilisateur enregistré et un mot de passe invalide). Nous itérons avec une variable d'énumération des utilisateurs et une variable d'énumération des mots de passe. Dans notre langage, cela s'écrit ainsi :

```
foreach $user in {REGISTERED_USER, UNREGISTERED_USER}{
  foreach $password in {REGISTERED_PWD, UNREGISTERED_PWD}{
    login($user,$password);
  }
}
```

4.2 Génération de tests

Le langage de scénarios est un langage qui intreprète un scénario comme un ensemble d'exécutions de chaque pas sur le modèle. Les scénarios sont dépliés de manière combinatoire sur plusieurs éléments du langage :

— les paramètres des opérations laissés libres (avec _),
— les points de choix introduits par les itérations/choix/répétitions,
— les opérations dans le cas d'appels d'opérations quelconques (avec $OP).

Il est à noter qu'avec la construction $OP, la combinatoire des paramètres s'ajoute à la combinatoire des opérations. Cet effet est encore multiplié avec des opérateurs de répétition. Il devient alors évident que les répétitions de constructions $OP sont à éviter le plus possible et qu'il convient alors de spécifier plus précisément ces parties de scénario avec des opérations nommées.

Bien que ces règles de dépliage puissent amener à une explosion combinatoire, nous avons cependant la possibilité de la contrôler. En effet, chacun des éléments introduisant de la combinatoire peut être contraint par les directives de pilotage à ne considérer qu'une seule valeur qui permet de trouver un chemin dans le graphe, de l'état initial à l'état final, limitant ainsi l'exploration du graphe.

Dans la suite, nous utilisons une représentation *dépliée* des graphes de sémantique. Cela signifie que tous les nœud d'opération du graphe font référence à un appel d'opération

complètement valué. Ainsi, chaque nœud d'appel d'opération qui n'est pas entièrement valué doit être déplié en plusieurs nœuds qui représentent alors toutes ses valuations.

4.2.1 Dépliage du scénario

Le dépliage du scénario permet de générer des cas de tests en animant chaque pas sur le modèle en utilisant un parcours en profondeur du graphe de flot de contrôle du scénario. Avant de définir les cas de tests extraits d'un scénario nous devons d'abord déterminer quand un pas du scénario est faisable vis-à-vis du modèle. Pour cela, nous nous appuyons sur une animation du modèle afin de déterminer l'état du système et les comportements activés par l'appel d'opération.

Définition 15 (Faisabilité d'un pas de scénario)**.** Soit le graphe d'appel d'opération suivant :

Soient s l'état courant, P et Q des prédicats sur les instances de classes i et j respectivement, et $A = \langle op, \rho_{op}, tags \rangle$. L'arc d'appel d'opération précédent est dit *faisable* si les conditions suivantes sont satisfaites :

- s satisfait P sur l'instance i,
- l'exécution d'au moins un des comportements de $tags$ de l'opération op avec les valeurs de paramètres ρ_{op} produit un état s',
- s' satisfait Q sur l'instance j'.

Exemple 17 (Faisabilité d'une instruction)**.** Soit l'instruction suivante qui permet de tester le fait qu'un utilisateur enregistré ne peut être authentifié que s'il utilise son mot de passe :

```
  ...
sut.login(REGISTERED_USER,_) including {AIM:LOG_Success};
assert("not(current_user.oclIsUndefined())" on "sut");
...
```

Le premier appel d'opération n'est pas complètement valué (le mot de passe est laissé libre). D'après la sémantique, il y a dépliage de cette instruction. Chaque nouveau nœud est l'appel à l'opération *login* mais avec une valuation du mot de passe différente : UNREGISTERED_PWD, REGISTERED_PWD, EMPTY_PWD et INVALID_PWD. La FIGURE 4.9 montre le graphe complètement déplié correspondant au scénario avec :

- A_1=⟨sut.login,(REGISTERED_USER,UNREGISTERED_PWD), {@AIM:LOG_Success}⟩
- A_2=⟨sut.login,(REGISTERED_USER,REGISTERED_PWD), {@AIM:LOG_Success}⟩
- A_3=⟨sut.login,(REGISTERED_USER,EMPTY_PWD), {@AIM:LOG_Success}⟩
- A_4=⟨sut.login,(REGISTERED_USER,INVALID_PWD), {@AIM:LOG_Success}⟩
- P="not(current_user.oclIsUndefined())" on "sut"

Dans un premier temps, nous animons la première valuation A_1 avec le mot de passe UNREGISTERED_PWD. L'animation sur le modèle de cette opération est un échec car le tag activé est @AIM:LOG_Invalid_Password et le prédicat du `assert` n'est pas satisfait (l'utilisateur est resté non-défini). La deuxième valuation est valide vis-à-vis du modèle car il s'agit du bon mot de passe et cette fois, l'utilisateur est bien défini. Les deux autres valuations échouent comme la première.

FIGURE 4.9 – Graphe de l'exemple 17

La faisabilité d'un pas du scénario étant définie, nous pouvons maintenant définir ce qu'est un test extrait d'un scénario.

Définition 16 (Extraction de tests d'un scénario)**.** Un cas de test extrait d'un scénario est un chemin de l'état initial à l'état final du graphe de sémantique dont chaque pas est faisable. Cela signifie que la séquence d'instructions sur le chemin a été animée avec succès sur le modèle.

4.2.2 Mise en œuvre

Pour parcourir le graphe, nous utilisons un parcours en profondeur avec un mécanisme de retour en arrière (backtrack). Si à un certain point du parcours, un pas n'est pas faisable, alors on remonte jusqu'au dernier point de choix pour emprunter la branche suivante non explorée. Les directives de pilotage, représentées par le type des nœuds de choix, permettent d'élaguer des branches du graphe pour limiter l'explosion combinatoire. Dans le cas de directives *inclusives*, par exemple avec les instructions `unfold`, `foreach`, `choice` et la directive `all` des opérations, toutes les branches sont considérées. Dans le cas des directives exclusives, si une branche appartient à un chemin faisable du graphe ayant permis de produire un test, alors les autres branches ne seront plus considérées pour cette instance du point de choix.

La FIGURE 4.10 donne l'algorithme récursif d'exploration combinatoire d'un graphe déplié du scénario (chaque nœud d'appel d'opération est nécessairement entièrement valué).

Lors de l'exploration d'un nœud, il y a quatre cas à distinguer.

Le premier est le cas où le nœud courant est le nœud initial. Dans ce cas, l'exploration continue directement à son fils.

Le second est le cas où le nœud courant est l'état final. Si l'exploration arrive à ce nœud, alors il existe une séquence de nœuds du graphe qui a été animée avec succès et est stockée dans *sequenceCourante*. Cette séquence courante est alors conservée en tant que nouveau cas de test car elle correspond à la définition d'un *chemin faisable*. De plus, le retour de l'opération signale qu'un cas de test a été trouvé.

Le troisième est le cas où le nœud courant est un point de choix. Dans ce cas, l'algorithme explore chaque fils du nœud successivement. L'algorithme garde une trace de la séquence déjà effectuée et, en cas d'exploration infructueuse ou de dépliage complet du

```
1  Résultats Globaux tests : ensemble des tests générés
2                    sequenceCourante : séquence de noeuds animés avec succès

4  Algorithme explore
5  Données : G : graphe sémantique du scénario
6            noeudCourant ∈ In ∪ Cp
7  Résultat : resultat : booléen      /* vrai si l'animation est un succès */
8  Variables : testTrouve, animResult : booléen
9  Début
10   testTrouve ← false
11   si noeudCourant = i_G alors      /* cas noeud initial */
12     sequenceCourante ← sequence vide
13     resultat ← explore(G, successeur de i_G)
14   sinon
15     si noeudCourant = f_G alors      /* cas noeud final */
16       tests ← tests ∪ {sequenceCourante}
17       resultat ← true
18     sinon
19       si noeudCourant ∈ Cp_G alors      /* cas  noeud de choix */
20         prefixeCourant ← sequenceCourante
21         pour chaque successeur n du noeudCourant faire
22           sequenceCourante ← prefixeCourant
23           testTrouvé ← explore(G,n) ∨ testTrouvé
24           si testTrouvé = true ∧ noeudCourant est un choix exclusif alors
25             break
26           fsi
27         finpour
28         resultat ← testTrouvé
29       sinon      /* cas d'un noeud d'appel d'opération */
30         animResult, s' ← animationSurModèle(noeudCourant)
31         si animResult = true ∧
32           s' satisfait le prédicat de l'arc sortant alors
33           sequenceCourante ← sequenceCourante.noeudCourant
34           resultat ← explore(G, successeur du noeudCourant)
35         sinon
36           resultat ← false
37         fsi
38       fsi
39     fsi
40   fsi
41 Fin
```

FIGURE 4.10 – Algorithme d'exploration combinatoire du graphe du scénario

nœud, remplace l'état courant et continue l'exploration des autres fils. Lors de l'exploration d'un de ses fils, si un test a été trouvé et si le nœud est identifié comme un choix exclusif, alors on arrête l'itération sur les fils, même s'il reste encore des fils inexplorés. Le retour signale si un cas de test a été trouvé lors de l'exploration d'un de ses fils.

Enfin, le quatrième est le cas d'un nœud d'opération entièrement valuée. Dans ce cas, l'animation de l'opération associée au nœud courant sur le modèle est réalisée par la fonction *animationSurModèle* qui exécute l'opération et produit l'état résultant s' et vrai si l'opération est faisable. Si l'animation est un succès et si le nouvel état du système satisfait le prédicat sortant du nœud courant, alors le nœud courant est ajouté à la séquence et l'exploration continue sur le fils du nœud courant. Si l'animation échoue ou si le prédicat n'est pas satisfait, alors le retour signale qu'aucun test n'a été trouvé.

A la fin de l'algorithme, la variable *tests* contient alors tous les chemins qui ont été animés avec succès sur le modèle en tenant compte des directives de pilotage. Ceux-ci

sont ainsi les cas de tests extraits à partir du scénario.

4.2.3 Exemple de dépliage

Afin d'illustrer la génération de tests à partir d'un scénario nous utilisons l'exemple suivant.

Exemple 18 (Exemple de scénario)**.** Soit le scénario de test où l'on souhaite tester les différents comportements de l'opération *buyTicket* lorsqu'un utilisateur est authentifié. Pour ce faire, un utilisateur peut directement s'authentifier ou s'enregistrer avant, le but étant de montrer que l'opération *buyTicket* se comporte de manière égale avec un utilisateur déjà ou nouvellement enregistré. Ensuite, il achète au moins un ticket (avec un maximum de 4) jusqu'à ce qu'il ne reste plus qu'un seul ticket disponible pour ce film. Il tente ensuite d'acheter encore deux tickets avant de se désauthentifier. Ce scénario s'écrit de la manière suivante :

```
scenario buyTicketTest(){
  unfold between 0 and 1 times {
    sut.registration(_,_) including {@AIM:REG_Success};
  }
  sut.login(_,_) including {@AIM:LOG_Success};
  unfold between 1 and 4 times until ("self.all_listed_movies->
                                      any().available_tickets=1" on "sut"){
    sut.buyTicket(_);
  }
  sut.buyTicket(_);
  sut.buyTicket(_);
  sut.logout();
}
```

Donner la possibilité d'enregistrer un utilisateur avant l'authentification permet d'avoir les deux cas de figure où le test authentifie un utilisateur déjà enregistré et celui où l'utilisateur est enregistré juste avant l'authentification. Il y a ensuite trois comportements distincts de l'opération *buyTicket* que l'on souhaiterait observer :

- les invocations de *buyTicket* dans la boucle permettent de répéter cette opération jusqu'à ce qu'il ne reste plus qu' un seul ticket pour le film. D'après notre connaissance du modèle, à chaque itération, le comportement activé doit toujours être @AIM:BUY_Success.
- l'invocation suivante de *buyTicket* après la boucle, permet d'illustrer l'achat du dernier ticket pour le film, car à ce stade, il ne doit en rester plus qu'un à cause de la condition de sortie de la boucle. De ce que l'on sait du modèle, cette invocation doit activer le comportement @AIM:BUY_Last_Ticket_Sold.
- à ce stade, il ne reste plus de ticket pour le film. Par conséquent, l'invocation suivante doit signaler que l'achat est un échec. Cela se traduit par l'activation du comportement @AIM:BUY_Sold_Out.

Pour cet exemple, nous considérons que le nombre d'itérations maximales pour les répétitions est fixé à 4. La FIGURE 4.11 donne le graphe complet du scénario de l'exemple 18,

FIGURE 4.11 – Graphe sémantique du scénario exemple

	INVALID_PWD	REGISTERED_PWD	UNREGISTERED_PWD
EMPTY_USER	1	2	3
INVALID_USER	4	5	6
REGISTERED_USER	7	8	9
UNREGISTERED_USER	10	11	12

FIGURE 4.12 – Combinaisons possibles entre utilisateurs et mots de passes

annoté pour relier les différents éléments du scénario. La FIGURE 4.12 numérote toutes les combinaisons possibles d'utilisateurs et de mots de passe. Nous utilisons cette table pour numéroter les différents appels des opérations *login* et *registration*. Par exemple, la combinaison d'un utilisateur enregistré REGISTERED_USER et d'un mot de passe correct REGISTERED_PWD porte le numéro 8.

Ainsi, dans le graphe de la FIGURE 4.11, les l_i (resp. r_i) avec $i \in 1..12$ font référence à l'opération *login* (resp. *registration*) et à la combinaison de paramètres numérotée par i dans la FIGURE 4.12. Par exemple, l_7 fait référence à l'opération *login* avec un utilisateur enregistré et un mot de passe invalide. Enfin, les nœuds étiquetés par bt_1 et bt_2 font référence à l'appel de l'opération *buyTicket* avec le film TITLE1 et le film TITLE2 respectivement. Enfin, le prédicat P correspond au prédicat de fin de la boucle `unfold` :

"self.all_listed_movies→any().available_tickets=1" **on** "sut".

Lors du parcours en profondeur du graphe, nous croisons un premier point de choix qui représente la répétition optionnelle de l'enregistrement. Dans un premier temps nous considérons que nous traversons l'arc qui amène directement au point de choix de l'opération *login* (pas d'enregistrement d'utilisateur). Il est à noter que ce choix est inclusif et par conséquent, le processus de génération considérera les autres branches, qu'un test soit trouvé ou non avec la branche courante.

Les paramètres de *login* ont été laissés libres donc nous avons le choix entre toutes les valuations possibles (voir FIGURE 4.11). Nous animons la première valuation (utilisateur invalide et mot de passe vide) sur le modèle. Or, avec ces paramètres, l'animation active le comportement @AIM:LOGIN_Empty_User_Name alors que nous attendions l'activation du comportement @AIM:LOGIN_Success. De fait, ce pas échoue et il en va de même pour les valuations suivantes jusqu'à l'appel de l'opération avec un utilisateur enregistré et son mot de passe (cas 8 de la FIGURE 4.11) tel que montré dans la FIGURE 4.13.

FIGURE 4.13 – Valuations de l'opération *login*

Après l'authentification avec succès, le parcours continue sur la répétition des appels à *buyTicket*. Les seuls dépliages possibles sont ceux qui contiennent exactement deux appels à *buyTicket* avec le titre TITLE2, les autres cas ne permettant pas de respecter le prédicat

de fin d'itération. En effet, l'unique possibilité pour satisfaire le prédicat consiste à acheter deux tickets pour le film TITLE2, qui possède au maximum 3 tickets. Ainsi, après deux achats, il ne reste plus qu'un seul ticket disponible pour ce film. Tous les chemins possibles des différentes itérations sont représentés dans la FIGURE 4.14 pour les trois premières itérations et dans la FIGURE 4.15 pour la dernière.

Puisque le dépliage des appels de chaque *buyTicket* est exclusif pour chaque itération, nous choisissons le premier chemin valide pour chacune d'elles, représenté en gras dans les FIGURE 4.14 et FIGURE 4.15. Ainsi, au dépliage, nous ne considérons que les trois chemins suivants :

$buyTicket(TITLE2); buyTicket(TITLE2);$
$buyTicket(TITLE1); buyTicket(TITLE2); buyTicket(TITLE2);$
$buyTicket(TITLE1); buyTicket(TITLE1); buyTicket(TITLE2); buyTicket(TITLE2);$

A la première itération, il n'y a qu'un seul appel à *buyTicket*. La première valuation avec TITLE1 ne permet pas de satisfaire la condition de la répétition, et il en va de même pour la valuation avec TITLE2. Ainsi, aucune valuation de l'unique appel de *buyTicket* ne permet de satisfaire la postcondition : cette branche de la répétition est donc abandonnée. Le processus retourne sur le point de choix et continue le parcours sur la branche suivante (deuxième itération). La seule possibilité de satisfaire la condition de fin répétition est d'enchaîner deux appels à *buyTicket* avec TITLE2 en paramètre, les autres valuations utilisant TITLE1 ne permettent pas de la satisfaire.

La condition de répétition étant satisfaite, il reste donc les deux derniers achats. Pour la première invocation, la première valuation avec TITLE1 est animable sur le modèle (en activant le comportement @AIM:BUY_Success). Le nœud de choix étant exclusif, la valuation avec TITLE2 n'est pas considérée. Il en va de même pour la seconde invocation.

Enfin, l'opération *logout* est aussi animable sur le modèle (avec le comportement @AIM:LOGOUT_Logout) et permet de former un chemin de l'état initial à l'état final. Cette séquence est donc un cas de test extrait du scénario. Un cas de test a été trouvé, mais il reste encore des points de choix inclusifs à considérer (les itérations suivantes de la répétition de *buyTicket* et l'enregistrement d'un utilisateur). Les autres choix des points de choix exclusifs ne sont plus considérés : c'est le cas du choix de la valuation de l'opération *login*, la valuation courante étant valide pour extraire un cas de test, les autres valuations ne seront pas considérées.

Le dernier point de choix inclusif rencontré est la répétition de l'opération *buyTicket*. La branche de l'itération suivante est donc empruntée.

Dans ce graphe, il existe 9720 chemins possibles. Cependant, après animation, seuls 6 tests sont produits. La FIGURE 4.16 récapitule les cas de tests produits avec les comportements associés à chaque invocation d'opération.

Nous remarquons dans un premier temps qu'aucun des tests ne correspond complètement à l'intention de test initiale : le comportement nominal de *buyTicket* est bien activé, mais il n'y a aucune activation des comportements d'achat de dernier ticket et d'échec d'achat, @AIM:BUY_Last_Ticket_Sold et @AIM:Sold_Out respectivement. Cela s'explique par le fait que la boucle a acheté des tickets pour TITLE2, mais les deux invocations

suivantes ont considéré le film TITLE1. Par défaut, la directive de pilotage des opérations est `one` : les invocations avec TITLE1 ont pu être animées avec succès, donc les valuations avec TITLE2 ont été ignorées. Les comportements attendus auraient été activés si les valuations avec TITLE2 avaient été choisies.

Dans un second temps, les tests 4, 5 et 6 enregistrent bien un nouvel utilisateur, mais ils choisissent l'utilisateur déjà enregistré lors de l'authentification, alors que l'intention de test précise que l'on souhaite observer le comportement de *buyTicket* avec un utilisateur nouvellement créé.

En conclusion, ce scénario ne permet donc pas de générer des cas de tests qui correspondent précisément à l'intention de test qu'il était censé représenter. Toutefois, ce scénario a été volontairement grossièrement écrit afin d'illustrer l'impact d'une faible expertise de l'ingénieur validation lors de son écriture. Nous allons montrer comment pallier ce problème dans la section suivante.

4.2.4 Comment recourir à l'expertise de l'ingénieur ?

L'exemple de dépliage précédent montre que les tests générés ne sont pas tous ou pas totalement pertinents, ou du moins, ne correspondent pas à l'intention de test initiale du scénario. Cela vient du fait que le scénario n'est pas assez finement spécifié. Il montre ainsi le fort impact de l'expertise de l'ingénieur de test lors de la spécification de scénarios à la main. Un scénario peut être amélioré de plusieurs manières que nous décrivons ci-dessous et que nous illustrons sur l'exemple.

Restreindre les valeurs de certains paramètres. Au lieu de laisser le titre du film à acheter libre dans l'opération *buyTicket*, nous pouvons le spécifier en tant que paramètre du scénario. Cette restriction permet d'éviter d'utiliser des titres différents pour les différents appels de *buyTickets*. En effet, en laissant le paramètre libre, le scénario produit des cas de test tels que ceux de la FIGURE 4.16 qui ne permettent pas d'illustrer les comportements d'achat du dernier ticket ni celui d'échec parce qu'il ne reste plus de ticket. Cette restriction du titre permet d'éviter des cas de tests qui ne correspondent pas à l'intention de test en changeant le titre du film sur les deux dernières invocations de *buyTicket*.

Affiner le préambule. L'enregistrement facultatif d'un utilisateur avant l'authentification ne nous intéresse que si celui-ci réussi. En effet, l'intérêt de cet appel est de montrer que *buyTicket* se comporte de la même manière avec un utilisateur déjà enregistré et un nouvel utilisateur. Ainsi, nous pouvons spécifier les paramètres de l'opération *registration* pour enregistrer un utilisateur qui ne l'est pas déjà et éviter les cas d'erreur d'enregistrement avec un utilisateur déjà inscrit ou avec un mot de passe invalide. De plus, dans le cas de la création d'un nouvel utilisateur, l'intention de test est de tester l'opération *buyTicket* avec ce nouvel utilisateur. Nous pouvons donc préciser plus finement les paramètres de l'opération *login* pour qu'elle utilise les identifiants du nouvel utilisateur plutôt que ceux d'un utilisateur déjà enregistré.

Limiter l'explosion des répétitions. Les répétitions sont aussi responsables des explosions combinatoires, en particulier l'instruction `repeat between 1 and 4 times` en choisissant un borne maximale trop grande. Dans le dépliage précédent, la borne a été choisie à 4, trop petite pour illustrer l'intention de test avec TITLE1 (il faudrait la fixer

à 6, car TITLE1 propose 7 tickets) et trop grande pour TITLE2 (où une borne maximale fixée à 2 est suffisante). De plus, en connaissant le modèle, nous pouvons aussi épargner au dépliage les itérations qui ne sont pas suffisantes et répéter l'instruction un nombre de fois fixe avec `repeat 2 times` par exemple.

En prenant en compte les points précédents, le nouveau scénario devient :

```
scenario buyTicketTest2($title, $rep){
  choice {
    sut.login(REGISTERED_USER, REGISTERED_PWD)
                                including {@AIM:LOG_Success};
  } or {
    sut.register(UNREGISTERED_USER, UNREGISTERED_PWD)
                                    including {@AIM:REGISTRATION_Success};
    sut.login(UNREGISTERED_USER, UNREGISTERED_PWD)
                                    including {@AIM:LOG_Success};
  }
  repeat $rep times until
                  ("self.all_listed_movies->any().available_tickets=1"){
    sut.buyTicket($title);
  }
  sut.buyTicket($title);
  sut.buyTicket($title);
  sut.logout();
}
```

Dans un premier temps, le scénario prend deux nouveaux paramètres : le titre du film à considérer et la répétition maximale pour la boucle d'achat de tickets. Ce dernier permet de fixer une borne *optimale* en fonction du nombre de tickets disponible pour le film en paramètre. La première répétition est devenue un choix entre l'authentification d'un utilisateur déjà enregistré, et l'enregistrement d'un nouvel utilisateur et son authentification. Ensuite, la répétition d'achat de tickets a sa borne maximale limitée par le paramètre *$rep* du scénario. Enfin, les deux derniers achats de tickets ont leur paramètre valué avec le titre passé en paramètre du scénario.

Un cas d'utilisation de ce scénario utilise le film TITLE2 et une borne maximale 2. En effet, avec notre connaissance du modèle, nous savons que le film TITLE2 propose 3 tickets à la vente. La boucle d'achat effectue alors 2 achats de tickets avec le comportement @AIM:BUY_Success. L'achat de ticket suivant achète le dernier ticket (puisqu'il n'en reste qu'un d'après la condition de sortie de boucle) exhibant l'activation du comportement @AIM:BUY_Last_Ticket_Sold. Enfin, la dernière tentative doit être un échec car il ne reste plus de ticket. Les figures 4.17 et 4.18 donnent les deux cas de tests issus de ce scénario. Le test 4.17 utilise un utilisateur déjà inscrit pour l'authentification, alors que le test 4.18 enregistre un nouvel utilisateur et c'est celui-ci qui s'authentifie. Le reste du scénario reste identique.

Ces deux tests correspondent bien à l'intention de test du départ (tester les comportements de *buyTicket* avec un utilisateur authentifié) sans les cas de tests qui ne répondent pas à l'objectif de la première version du scénario.

Nous aurions pu montrer la même chose avec le film TITLE1 et une borne maximale à 6, mais le test aurait alors été plus long, pour un même résultat.

4.3 Synthèse

Dans ce chapitre, nous avons introduit un langage de scénarios qui permet à un ingénieur de test de spécifier des cas de tests abstraits. Ce langage est inspiré du langage de scénarios de [Tis09] qui est un formalisme fondé sur les expressions régulières, qui rendent un scénario de grande taille difficilement lisible. Notre langage permet de remédier à ce problème en proposant une syntaxe proche de celle d'un langage de programmation en y intégrant les directives de dépliage [Tis09]. De plus, la compositionnalité des scénarios permet de les fragmenter pour gagner en lisibilité.

Pour la génération de tests à partir de ce langage, nous utilisons une technique de dépliage combinatoire en considérant les directives de dépliage associées aux instructions qui permettent de limiter l'explosion combinatoire. Cependant, un scénario trop grossièrement spécifié conduira à une explosion combinatoire ou à la génération de tests qui n'ont pas de rapport avec l'intention de test du scénario. La pertinence et l'efficacité (en terme de temps de dépliage) de la génération de test dépend grandement de l'expertise que l'ingénieur de test a du modèle et de sa capacité à spécifier finement des scénarios.

Selon le processus décrit dans le Chapitre 1, ces tests sont ensuite traduits, par le biais d'une couche d'adaptation, vers un formalisme qui permet de stimuler le système sous test réel (JUnit par exemple). Cependant, puisque les tests produits par les scénarios sont au même niveau d'abstraction que le modèle, nous pouvons aussi utiliser ces tests pour une première validation du modèle. Il y a deux intérêts à cette démarche. Le premier est que ce langage permet l'expression des cas de test exerçant la dynamique du système au niveau du modèle. Le second intérêt est l'obtention d'un oracle pour les tests. En effet, l'animation de ces tests directement sur le modèle permet d'obtenir les valeurs des observations attendues à chaque étape du test, que nous pouvons associer au test en tant qu'oracle. Ainsi, cette démarche de *test du modèle* nous assure une plus grande confiance dans le modèle et dans les tests générés à partir de ce modèle.

Pour aider les ingénieurs de tests, nous proposons dans le chapitre suivant un langage de plus haut niveau, inspiré des patrons de propriétés de Dwyer *et al.*, qui permet de définir des propriétés temporelles sur le système puis d'engendrer automatiquement des scénarios.

FIGURE 4.14 – Evaluation de l'animation des trois premières itérations de la répétition de *buyTicket*

FIGURE 4.15 – Evaluation de l'animation de la quatrième itération de la répétition de *buyTicket*

Tests	Opérations		Comportements
1	login(REGISTERED_USER, REGISTERED_PWD)	→	@AIM:LOG_Success
	buyTicket(TITLE2)	→	@AIM:BUY_Success
	buyTicket(TITLE2)	→	@AIM:BUY_Success
	buyTicket(TITLE1)	→	@AIM:BUY_Success
	buyTicket(TITLE1)	→	@AIM:BUY_Success
	logout()	→	@AIM:LOGOUT_Logout
3	login(REGISTERED_USER, REGISTERED_PWD)	→	@AIM:LOG_Success
	buyTicket(TITLE1)	→	@AIM:BUY_Success
	buyTicket(TITLE2)	→	@AIM:BUY_Success
	buyTicket(TITLE2)	→	@AIM:BUY_Success
	buyTicket(TITLE1)	→	@AIM:BUY_Success
	buyTicket(TITLE1)	→	@AIM:BUY_Success
	logout()	→	@AIM:LOGOUT_Logout
3	login(REGISTERED_USER, REGISTERED_PWD)	→	@AIM:LOG_Success
	buyTicket(TITLE1)	→	@AIM:BUY_Success
	buyTicket(TITLE1)	→	@AIM:BUY_Success
	buyTicket(TITLE2)	→	@AIM:BUY_Success
	buyTicket(TITLE2)	→	@AIM:BUY_Success
	buyTicket(TITLE1)	→	@AIM:BUY_Success
	buyTicket(TITLE1)	→	@AIM:BUY_Success
	logout()	→	@AIM:LOGOUT_Logout
4	registration(UNREGISTERED_USER, UNREGISTERED_PWD)	→	@AIM:REG_Success
	login(REGISTERED_USER, REGISTERED_PWD)	→	@AIM:LOG_Success
	buyTicket(TITLE2)	→	@AIM:BUY_Success
	buyTicket(TITLE2)	→	@AIM:BUY_Success
	buyTicket(TITLE1)	→	@AIM:BUY_Success
	buyTicket(TITLE1)	→	@AIM:BUY_Success
	logout()	→	@AIM:LOGOUT_Logout
5	registration(UNREGISTERED_USER, UNREGISTERED_PWD)	→	@AIM:REG_Success
	login(REGISTERED_USER, REGISTERED_PWD)	→	@AIM:LOG_Success
	buyTicket(TITLE1)	→	@AIM:BUY_Success
	buyTicket(TITLE2)	→	@AIM:BUY_Success
	buyTicket(TITLE2)	→	@AIM:BUY_Success
	buyTicket(TITLE1)	→	@AIM:BUY_Success
	buyTicket(TITLE1)	→	@AIM:BUY_Success
	logout()	→	@AIM:LOGOUT_Logout
6	registration(UNREGISTERED_USER, UNREGISTERED_PWD)	→	@AIM:REG_Success
	login(REGISTERED_USER, REGISTERED_PWD)	→	@AIM:LOG_Success
	buyTicket(TITLE1)	→	@AIM:BUY_Success
	buyTicket(TITLE1)	→	@AIM:BUY_Success
	buyTicket(TITLE2)	→	@AIM:BUY_Success
	buyTicket(TITLE2)	→	@AIM:BUY_Success
	buyTicket(TITLE1)	→	@AIM:BUY_Success
	buyTicket(TITLE1)	→	@AIM:BUY_Success
	logout()	→	@AIM:LOGOUT_Logout

FIGURE 4.16 – Tests extraits de l'animation du scénario

sut::login(REGISTERED_USER,REGISTERED_PWD)	→	@AIM:LOG_Success
sut::buyTicket(TITLE2)	→	@AIM:BUY_Success
sut::buyTicket(TITLE2)	→	@AIM:BUY_Success
sut::buyTicket(TITLE2)	→	@AIM:BUY_Last_Ticket_Sold
sut::buyTicket(TITLE2)	→	@AIM:BUY_Sold_Out
sut::logout()	→	@AIM:LOGOUT_Logout

FIGURE 4.17 – Cas de test sans enregistrement d'utilisateur

sut::registration(UNREGISTERED_USER,UNREGISTERED_PWD)	→	@AIM:REG_Success
sut::login(UNREGISTERED_USER,UNREGISTERED_PWD)	→	@AIM:LOG_Login
sut::buyTicket(TITLE2)	→	@AIM:BUY_Success
sut::buyTicket(TITLE2)	→	@AIM:BUY_Success
sut::buyTicket(TITLE2)	→	@AIM:BUY_Last_Ticket_Sold
sut::buyTicket(TITLE2)	→	@AIM:BUY_Sold_Out
sut::logout()	→	@AIM:LOGOUT_Logout

FIGURE 4.18 – Cas de test avec enregistrement d'un utilisateur

Chapitre 5

Langage de propriétés

Sommaire

5.1 Syntaxe 86
5.1.1 Propriétés 87
5.1.2 Les évènements 88
5.2 Automates de substitution 89
5.2.1 Définition 89
5.2.2 Substitution 91
5.2.3 Evénements 95
5.3 Sémantique du langage 96
5.3.1 Sémantique des portées 96
5.3.2 Sémantique des motifs 99
5.3.3 Application de la composition d'automate 102
5.4 Différences sémantiques avec Dwyer *et al.* 103
5.4.1 Types d'évènements 104
5.4.2 Notion de satisfiabilité de propriétés 104
5.5 Quelques exemples de propriétés 104
5.6 Synthèse 106

Le langage de scénarios introduit dans le chapitre précédent permet l'expression de séquences d'opérations, mais la faisabilité et la pertinence de cette tâche restent étroitement liées au savoir-faire et à l'expérience de l'ingénieur de test. Pour palier cela, nous proposons un langage plus simple d'utilisation, pour exprimer des propriétés à partir desquelles sont ensuite dérivées les scénarios.

Dans ce chapitre, nous introduisons un langage d'expression de propriétés temporelles inspiré des travaux de Dwyer *et al.* [DAC99] sur des patrons de propriétés. Il existe déjà des formalismes de propriétés temporelles tels que la LTL [MP92] (Linear Temporal Logic) ou la CTL [EH86] (Computation Tree Logic), mais ces formalismes requièrent une expertise qui n'est pas toujours acquise par les ingénieurs. Les patrons de propriétés sont un moyen plus simple d'écrire des propriétés temporelles sur un système.

Dans un premier temps, dans la section 5.1, nous donnons la syntaxe de notre langage de spécification de propriétés temporelles. Ensuite, dans la section 5.2, nous donnons une

```
TEMP_EXPR      ::=  TEMP_PATTERN TEMP_SCOPE
TEMP_PATTERN   ::=  always ( PREDICATE )
                |   never EVENT
                |   eventually EVENT (TIMES)?
                |   EVENT (directly)? precedes EVENT
                |   EVENT (directly)? follows EVENT
TEMP_SCOPE     ::=  globally
                |   before EVENT
                |   after EVENT
                |   between (last)? EVENT and EVENT
                |   after (last)? EVENT until EVENT
EVENT          ::=  CHANGE_EVENT (|| EVENT )?
                |   CALL_EVENT ( || EVENT )?
CHANGE_EVENT   ::=  becomesTrue(PREDICATE )
CALL_EVENT     ::=  isCalled((name .)? name
                                (, pre:  PREDICATE)?
                                (, post:  PREDICATE)?
                                (, TAG_LIST)? )
TAG_LIST       ::=  including: { TAGS}
                |   excluding: { TAGS}
TIMES          ::=  integer times
                |   at least integer times
                |   at most integer times
TAGS           ::=  @REQ:  name (, TAGS)?
                |   @AIM:  name (, TAGS)?
PREDICATE      ::=  OclExpression on name
```

FIGURE 5.1 – Syntaxe du langage de propriétés

définition formelle des automates de substitution qui nous servent à exprimer la sémantique de notre langage, décrite dans la section 5.3. Dans la section 5.4, nous comparons notre sémantique à celle définie par Dwyer *et al.* Avant de conclure, la section 5.5 donne des exemples de propriétés sur eCinema et leur traduction en automates.

5.1 Syntaxe

Dans la FIGURE 5.1, nous présentons la syntaxe du langage d'expression de propriétés. Il permet de définir des propriétés temporelles comme la composition d'une portée et d'un motif en se basant sur des évènements de deux types différents : les appels d'opération (éventuellement conditionnés) et les changements d'états du système. Ce langage fait intervenir quatre concepts que nous allons détailler dans la suite de ce chapitre : les propriétés (règle TEMP_EXPR), les portées (règle TEMP_SCOPE), les motifs (règle TEMP_PATTERN) et les évènements (règles CHANGE_EVENT et CALL_EVENT).

Plusieurs terminaux apparaissent dans cette grammaire. Les terminaux OclExpression représentent des propositions dans le langage OCL. Les termes name représentent de manière générique des noms et désignent plus précisément des opérations, des objets ou des tags. Enfin, les terminaux integer représentent des valeurs entières.

Dans la suite de ce chapitre, nous notons les événements *EVENT* par une lettre majuscule de A à F, les prédicats logiques *PREDICATE* par P et Q et les ensembles de tags par T. Pour finir les notations k et n référence à des entiers et une opération quelconque est notée *op*.

5.1.1 Propriétés

Cette syntaxe reprend le principe des patrons de propriétés de Dwyer *et al.* en définissant les propriétés comme la composition d'une portée et d'un motif.

Les motifs

Les motifs (ou *pattern* dans le vocabulaire de Dwyer) sont des enchaînements d'évènements particuliers et correspondent à certaines exigences dynamiques sur le système. Il existe cinq motifs :

— `always` P : la propriété d'état P doit toujours être satisfaite,
— `never` E : l'évènement E ne doit jamais arriver,
— `eventually` E : l'évènement E doit nécessairement arriver,
— E `precedes` F : l'évènement E doit précéder l'évènement F,
— F `follows` E : l'évènement F doit suivre l'évènement E.

Il est à noter que les motifs E `precedes` F et F `follows` E se différencient par une subtile nuance. Le motif E `follows` F signifie que s'il y a un F, alors nécessairement par la suite il doit y avoir un E. Dans le cas du motif F `precedes` E, s'il y a un E alors nécessairement il y a eu un F avant. Par exemple, le motif E `follows` F permet la séquence $E.F.E$ qui n'est pas acceptée par F `precedes` E car il n'y a pas de F précédent le premier E. Inversement, la séquence $F.E.F$ est acceptée par F `precedes` E (il y a bien eu un F précédant E) mais pas par E `follows` F car il n'y a pas de E suivant le dernier F.

En plus de ces motifs de base, certains motifs présentent des variantes. Le premier cas est celui du motif `eventually` qui dispose de trois variantes, avec k un entier naturel :

— `eventually` E k `times` : E doit nécessairement arriver k fois,
— `eventually` E `at least` k `times` : E doit nécessairement arriver au moins k fois,
— `eventually` E `at most` k `times` : E doit nécessairement arriver au plus k fois.

A ces variantes s'ajoutent celles des motifs E `precedes` F et E `follows` F :

— E `directly precedes` F : E doit directement précéder F,
— E `directly follows` F : E doit immédiatement suivre F.

Ces variantes n'autorisent qu'aucun autre évènement n'arrive entre E et F, alors que la version basique autorise l'occurrence d'évènements entre E et F.

Les portées

Les portées, appelées *scope* dans les travaux de Dwyer, spécifient une partie de chaque exécution du système dans laquelle un certain motif doit être satisfait. En d'autres termes, une portée définit un contexte d'exécution dans lequel le motif qui lui est associé doit être satisfait. Les portées sont au nombre de cinq, définies par la règle TEMP_SCOPE :

— `globally` représente l'intégralité de la séquence d'exécution,

FIGURE 5.2 – Représentation graphique des portées

— `before` E : partie d'exécution avant l'arrivée de l'évènement E,
— `after` E : partie d'exécution après l'arrivée de l'évènement E,
— `between` E `and` F : partie d'exécution entre les évènements E et F, l'arrivée de F est nécessaire pour que la satisfaction du motif soit exigée,
— `after` E `until` F : partie d'exécution entre les évènements E et F. Si F n'arrive pas après E, le motif doit tout de même être satisfait.

La FIGURE 5.2 donne une représentation graphique des portées avec E et F des évènements du système. Les rectangles gris indiquent les intervalles considérés pour la portée.

Il est nécessaire de donner des précisions lorsque l'on est en dehors de la portée. La portée spécifie uniquement les parties où le motif doit être satisfait et ne donne par conséquent aucune contrainte sur la satisfaction du motif lorsque l'on est au dehors de la portée. Hors de la portée, la propriété spécifiée par le motif n'a pas à être satisfaite.

Les portées `between` E `and` F et `after` E `until` F peuvent être complétées avec le mot-clef `last`. Il a pour effet de ne considérer que la dernière occurrence de l'évènement E avant l'arrivée de l'évènement F. Par défaut, c'est la première occurrence de l'évènement E qui est retenue comme déclencheur de l'ouverture de la portée.

5.1.2 Les évènements

Les évènements sont les atomes du langage dans le sens où ils représentent les déclencheurs entre les différents états de la propriété. Un évènement d'une portée ou d'un motif peut aussi être une disjonction d'évènements, qui peuvent représenter alors plusieurs déclencheurs différents pour l'ouverture d'une portée. Par exemple, la portée `after` E | F représente les parties d'exécutions du système après l'arrivée de l'évènement E ou de l'évènement F.

Il existe deux types d'évènements dans le langage. Le premier est un appel d'opération dénoté `isCalled`(op, `pre:` P, `post:` Q, `including:` T) défini par la règle CALL_EVENT et le second est un évènement de changement d'état dénoté par `becomesTrue`(P) défini par la règle CHANGE_EVENT.

Evènement isCalled

Ce type d'évènement représente l'appel à une opération du modèle sous certaines conditions. En effet, il est possible de spécifier l'état du système à partir duquel l'opération doit être appelée par le biais d'un prédicat sur l'état du système dans la clause `pre` de la construction `isCalled`. Il en va de même pour l'état du système après l'exécution de l'opération, le prédicat étant précisé dans la clause `post`.

Il est aussi possible de restreindre l'ensemble des comportements à utiliser en spécifiant les comportements autorisés pour cet évènement par le biais d'une liste de tags en dernière clause. L'utilisation de la clause `including` indique que l'évènement considère les comportements identifiés par les tags de sa liste et nécessite l'activation d'au moins un

des comportements de la liste. Inversement, avec la clause `excluding`, les comportements listés ne doivent pas être activés par l'évènement.

Exemple 19 (Evènements `isCalled`)**.** Par exemple, l'achat de ticket avec succès qui amène le total de ticket de l'acheteur à 5 peut s'écrire de la manière suivante :

`isCalled`(sut.buyTicket, `post:` "self.current_user.all_tickets_in_basket=5" `on` "sut", `including:` {@AIM :BUY_Success})

La tentative d'authentification alors qu'un utilisateur est déjà authentifié peut s'écrire ainsi :

`isCalled`(sut.login, `pre` :"not(self.current_user.oclIsUndefined())" `on` "sut")

Evènements becomesTrue

Les évènements de type `becomesTrue`(P) représentent des changements d'état du système. Intuitivement, ces évènements représentent l'appel d'une opération quelconque du système avec une restriction sur l'état de départ où la proposition P doit être évaluée à *faux* et qui aboutira à un état où P doit être évaluée à *vrai*.

Exemple 20 (Evènement `becomesTrue`)**.** Pour représenter une authentification réussie, il est possible d'écrire l'évènement suivant :

`becomesTrue`("not(self.current_user.oclIsUndefined())" `on` "sut")

signifiant qu'avant l'évènement l'utilisateur était non défini et qu'il devient défini suite à une opération quelconque permettant de l'authentifier.

5.2 Automates de substitution

Nous introduisons, dans cette section, la notion d'automate de substitution. Les automates de substitution sont des systèmes de transitions étiquetées par des évènements du système. Ces automates sont des automates hiérarchiques où des états particuliers, appelés *états de substitution*, sont des emplacements destinés à être remplacés par d'autres automates. Celui-ci peut être substitué par un automate qui définit une instance de cette état particulier. Ces automates sont utilisés pour la définition de la sémantique des portées et des motifs et un mécanisme de composition permet d'obtenir l'automate de la sémantique de la propriété.

5.2.1 Définition

L'intuition derrière cette formalisation est l'expression de cette sémantique de manière compositionnelle comme cela est fait pour la définition d'une propriété temporelle, avec une portée et un motif. Ces derniers sont, dans un premier temps, représentés séparément avec ces automates.

Définition 17 (Automates de substitution)**.** Un automate de substitution a est un 8-uplet tel que a = $\langle$Q, F, I, S, R, E, Σ, T$\rangle$ dans lequel :

FIGURE 5.3 – Exemple d'automate de substitution

— Q est un ensemble fini d'états,
— F est l'ensemble des états finaux ($F \subseteq Q$),
— I est l'ensemble des états initiaux ($I \subseteq Q$),
— S est l'ensemble des états de substitution ($S \subseteq Q$),
— R est une fonction qui associe à chaque état de substitution un ensemble d'étiquettes ($R \in S \rightarrow \mathcal{P}(\Sigma)$), où $\mathcal{P}(\Sigma)$ est l'ensemble des parties de l'ensemble des évènements Σ,
— E est l'ensemble des états d'erreur ($E \subset Q$),
— Σ est l'ensemble des étiquettes des transitions,
— et T est l'ensemble des transitions du système, étiquetées par un ensemble de labels ($T \subseteq Q \times \mathcal{P}(\Sigma) \times Q$).

Cette définition fait apparaître plusieurs éléments la différenciant de celle des automates classiques. Tout d'abord, nous ajoutons l'ensemble S qui représente l'ensemble des états de substitution. Ces états de substitution peuvent être vus comme des "emplacements" dans l'automate que nous pourrons remplacer par un autre automate via un mécanisme de composition (Section 5.2.2). Ce concept est à rapprocher de celui des automates hiérarchiques, tels que les diagrammes d'états-transitions dans le langage UML [Obj12, RJB04]. Une des différences avec ces derniers est l'introduction du concept de *restriction* sur un état de substitution (par la fonction R) afin de restreindre l'alphabet de l'automate qui remplacera l'état de substitution au moment de la composition.

Graphiquement, nous représentons les états de substitution par des carrés, à la place des cercles qui représentent des états dits "normaux". Si un état de substitution porte une restriction d'alphabet, alors les évènements sur lesquels portent la restriction étiquettent l'état en question. Les états d'erreur sont représentés par un cercle barré d'une croix. Les états initiaux sont représentés par une petite flèche entrante et n'ayant pas de source. Les états finaux possèdent un contour doublé : un état final normal sera classiquement représenté par un double cercle et un état de substitution final sera représenté par un double carré. Les transitions sont représentées par des flèches étiquetées par un ensemble d'évènements {E,F, ...} acceptés par la transition. Par abus de notation, et pour alléger les schémas, un ensemble d'évènements {E} ne contenant qu'un seul élément est noté E. Un ensemble d'évènements sur une transition signifie une disjonction entre chacun de ces évènements : la transition est déclenchée si au moins un des évènements survient. Les évènements peuvent être définis par soustraction de l'ensemble d'évènements. Nous notons la soustraction d'un évènement E par un évènement F par E\F. Cette définition peut être généralisée à une disjonction d'évènements {E,F,...}\G. Cette restriction est alors partagée par tous les évènements et peut être réécrite de la manière suivante :

$$\{E,F,\dots\}\backslash G \rightsquigarrow \{E\backslash G,\ F\backslash G,\ \dots\}$$

Enfin, les transitions étiquetées par $\Sigma\backslash\{A, B, \dots\}$ représentent tous les évènements sauf ceux représentés par l'ensemble soustrait $\{A, B, \dots\}$. Le label Σ, l'alphabet de l'automate, est un abus de notation qui représente l'ensemble de tous les évènements du

système et est une facilité d'écriture de l'ensemble de tous les évènements : il s'agit de l'évènement qui active n'importe quelle opération, à partir d'un état quelconque, amenant dans un état tout aussi quelconque et utilisant n'importe quel comportement du système.

La FIGURE 5.3 donne un exemple d'un tel automate. Il est composé de quatre états : q_0 est l'état initial, q_1 est un état de substitution final, q_2 est un état de substitution qui porte une restriction de l'évènement B, et q_3 est un état final.

Formellement, cet automate de substitution se décrit de la manière suivante :

— $Q = \{q_0, q_1, q_2, q_3\}$,
— $F = \{q_1, q_3\}$,
— $I = \{q_0\}$,
— $S = \{q_1, q_2\}$,
— $R = \{(q_1 \mapsto \emptyset), (q_2 \mapsto \{B\})\}$,
— $\Sigma = \{A, B, C\}$,
— $T = \{(q_0, \Sigma \backslash \{A, B\}, q_0), (q_0, B, q_1), (q_0, A, q_2),$
$(q_2, B, q_1), (q_1, C, q_3), (q_2, B, q_3), (q_3, \Sigma, q_3)\}$,
— $E = \emptyset$.

Enfin, dans le cadre des portées et des motifs définis précédemment, nous pouvons déjà leur donner une certaine forme. Les portées du langage de la FIGURE 5.1 sont formalisées par un automate contenant un seul état de substitution, celui destiné à accueillir le motif de la propriété. Les motifs sont représentés par un automate de substitution sans état de substitution mais avec potentiellement un état d'erreur. Autrement dit, les automates de portée sont des automates de substitution dont l'état de substitution représente un motif générique, c'est-à-dire non défini.

5.2.2 Substitution

Afin d'avoir une représentation des propriétés en tant que combinaison d'une portée et d'un motif, nous définissons l'opération de substitution qui permet de remplacer les états de substitution d'un automate de base par d'autres automates. Ce remplacement se fait par une fonction σ qui permet d'associer à chaque état de substitution l'automate qui le remplace.

La définition 18 donne cette opération de substitution. Dans un souci de lisibilité, nous notons X_a le composant X de l'automate a.

Définition 18 (Substitution d'automate)**.** Soient l'automate a muni de l'ensemble d'états de substitution $S_a = \{s_0, \ldots, s_n\}$ et la fonction de substitution σ qui associe à chaque s_k un automate a_k avec $k \in 0..n$. Soit e_c un nouvel état distinct de tous les états de a et a_k. Nous considérons que cette fonction totale associe un automate à chaque état de substitution de l'automate de base a. Sous l'hypothèse que tous les états des différents automates sont distincts, l'automate résultat c est défini comme suit :

I. $Q_c = (Q_a - S_a) \cup \bigcup_{s \in S_a \backslash F_a} Q_{\sigma(s)} \cup (\bigcup_{s \in S_a \cap F_a} Q_{\sigma(s)} \backslash E_{\sigma(s)}) \cup \{e_c\}$

II. $I_c = (I_a - S_a) \cup \bigcup_{s \in S_a \cap I_a} I_{\sigma(s)}$

III. $F_c = (F_a - S_a) \cup \bigcup_{s \in S_a \cap F_a} F_{\sigma(s)}$

IV. $S_c = \bigcup_{s \in S_a} S_{\sigma(s)}$

V. $R_c = \bigcup_{s \in S_a} R_{\sigma(s)}$

VI. $E_c = \{e_c\}$ est un singleton contenant un nouvel état d'erreur e_c distinct de tous les états

VII. $\Sigma_c = \Sigma_a \cup \bigcup_{s \in S_a} \Sigma_{\sigma(s)}$

VIII. $q \xrightarrow{L} q' \in T_c$ si et seulement si l'une des conditions suivantes est satisfaite :

a) $q, q' \in Q_a - S_a \wedge q \xrightarrow{L} q' \in T_a$

b) $q \in Q_a - S_a, s \in S_a \wedge q \xrightarrow{L} s \in T_a \wedge q' \in I_{\sigma(s)}$

c) $q' \in Q_a - S_a, s \in S_a \wedge s \xrightarrow{L} q' \in T_a \wedge q \in F_{\sigma(s)}$

d) $s, s' \in S_a \wedge s \xrightarrow{L} s' \in T_a \wedge q' \in I_{\sigma(s')} \wedge (q \in F_{\sigma(s)} \vee (s = s' \wedge s \in S_a \backslash F_a \wedge q \in E_{\sigma(s)}))$

e) $s \in S_a \cap F_a \wedge q, q' \in Q_{\sigma(s)} \backslash E_{\sigma(s)} \wedge q \xrightarrow{L'} q' \in T_{\sigma(s)} \wedge L = L' - R_{\sigma(s)} \wedge L \neq \emptyset$

f) $s \in S_a \backslash F_a \wedge q, q' \in Q_{\sigma(s)} \wedge q \xrightarrow{L'} q' \in T_{\sigma(s)} \wedge L = L' - R_{\sigma(s)} \wedge L \neq \emptyset$

g) $s \in S_a \cap F_a \wedge q \in Q_{\sigma(s)} \wedge q \xrightarrow{L} e_{\sigma(s)} \in T_{\sigma(s)} \wedge q' = e_c$

h) $s \in S_a \backslash F_a \wedge \exists q'' \cdot (e_{\sigma(s)} \in E_{\sigma(s)} \wedge q'' \xrightarrow{L''} e_{\sigma(s)} \in T_{\sigma(s)} \wedge \exists q''' \in Q_a \backslash \{s\} \cdot$
$(s \xrightarrow{L} q''' \in T_a \wedge q = e_{\sigma(s)} \wedge q' = e_c))$

L'ensemble des états Q_c de c, défini par la règle I., est composé de l'ensemble des états de a privé des états de substitution de a, remplacés par les états des automates de substitution selon la fonction σ, et l'ensemble des états des automates substitués. Si certains des automates substitués contiennent des états de substitution, alors ceux-ci sont conservés dans c tout comme les restrictions d'alphabet dont ils font l'objet.

La règle II. définit les nouveaux états initiaux. Si un état initial i_{k_a} est un état de substitution alors les états initiaux de l'automate substitué à sa place seront des états initiaux de c.

Le même principe est appliqué aux états finaux avec la règle III. : si un état final f de l'automate a est un état de substitution, alors les états finaux de l'automate substitué à f deviennent des états finaux de c. Si l'état de substitution n'est pas final, alors ses états finaux redeviennent normaux.

Le remplacement des états de substitution est défini par la règle IV.. La fonction σ étant totale, il ne reste aucun des états de substitution de l'automate dans lequel la substitution a eu lieu (a dans la définition). Par contre, si les automates substitués en contenaient, alors nous les retrouverions non modifiés dans l'automate résultat.

Il en va de même pour les restrictions définies par la règle V.. Celles associées à des états de substitution de l'automate a disparaissent (car prises en compte lors de l'intégration des transitions des automates substitués) et celles provenant des automates substitués

affectent toujours les états de substitution qu'elles ciblent. Synthétiquement, les états de substitution de l'automate de base et leurs restrictions disparaissent et ceux des automates substitués sont conservés.

Afin de prendre en compte tous les évènements de chaque automate, la règle VII. définit le nouvel alphabet comme étant l'union de l'alphabet de la portée et des alphabets de chacun des automates substitués.

Enfin, la règle VIII. définit les différentes règles pour les transitions. Une transition $q \xrightarrow{L} q'$ de l'automate c est définie en fonction des transitions de l'automate de base a et des automates substitués. Elle est définie par cas de la manière suivante :

a) S'il s'agit d'une transition de l'automate de base a entre deux états qui ne sont pas de substitution (q *et* $q' \in Q_a - S_a$) alors elle est intégrée telle quelle dans l'automate c.

b) Si l'on considère une transition de l'état q de l'automate de base a entrant vers un état de substitution s de a, alors il existe dans c les transitions qui relient l'état q à chaque état initial de l'automate remplaçant s, marquant ainsi l'entrée dans l'automate substitué.

c) Si l'on considère une transition de l'automate de base qui relie un état de substitution s à un état normal q', alors, dans c, il existe autant de transitions qu'il y a d'états finaux dans l'automate remplaçant s, chacune reliant un de ces états finaux q à q'.

d) Si, dans l'automate de base, il existe une transition entre deux états de substitution s et s', alors il existe autant de transitions qu'il y a de combinaisons possibles entre les états finaux de s et les états initiaux de s', chacune reliant un état final de s à un état initial de s'. Cette règle définit le cas particulier où la transition est réflexive sur un état de substitution non-final. Dans ce cas, une transition est créée de l'ancien état d'erreur du motif à son état initial. Ce cas est spécifique à la construction *last*.

e) Cette règle considère les transitions internes d'un automate substitué à la place d'un état de substitution final. Dans ce cas, d'après la règle I., l'état d'erreur de l'automate substitué disparaît, ainsi toutes les transitions qui y pointaient ne sont pas ajoutées. En revanche, toutes les autres transitions apparaissent bien dans le nouvel automate.

f) Cette règle est similaire à la précédente dans le sens où elle considère les transitions internes de l'automate substitué mais dans le cas où il prend la place d'un état de substitution final. Dans ce cas, toutes les transitions internes sont considérées, y compris celles menant à l'état d'erreur s'il existe. D'après la règle I., ce dernier apparaît bien dans l'ensemble des états mais a perdu son statut d'état d'erreur.

g) Concernant les états d'erreur, si un état de substitution est final et si l'automate le remplaçant contient un état d'erreur, alors les transitions de l'automate substitué qui pointent vers l'état d'erreur sont redirigées pour pointer vers le nouvel état d'erreur e_c de l'automate c.

h) Si l'état de substitution n'est pas final, alors d'après la règle I., l'état d'erreur de l'automate substitué est redevenu normal. Par conséquent, cette règle précise qu'il existe une transition supplémentaire reliant l'ancien état d'erreur au nouvel état d'erreur de la propriété par une transition étiquetée par toutes les étiquettes des transitions de sortie de l'état de substitution. Cette règle a pour but d'exprimer le fait que l'on a

FIGURE 5.4 – Composition et restriction d'évènements

atteint un état d'erreur dans le motif mais que celui-ci n'est visible que lorsque l'on sort de l'état de substitution, d'où le décalage de l'état d'erreur.

La sémantique de la restriction d'évènements sur un label Σ est la suivante : à l'ensemble de tous les évènements possibles du système, nous en interdisons certains par le biais de cette restriction. Ainsi, un évènement qui n'appartient pas à l'ensemble des évènements de la restriction, est accepté.

La FIGURE 5.4 montre un exemple de restriction sur un évènement. Dans cet exemple, l'évènement A est restreint par l'évènement B, ce qui signifie qu'un évènement E du système accepté par A et par B ne l'est pas par A\{B}. En revanche, E est accepté par A\B s'il est accepté uniquement par A et pas par B.

Nous appliquons cette définition aux portées et motifs du langage de la FIGURE 5.1, ces automates ne contenant qu'un état initial, un état final et un seul état de substitution pour l'automate de portée, aucun pour celui du motif. L'unique état de substitution de l'automate de portée est remplacé par l'automate du motif. Le résultat de la substitution de ces automates donne un automate classique, *i.e.* sans état de substitution. La définition précédente, plus générale que ce dont nous avons besoin, permet d'envisager la définition de propriétés plus générales avec plusieurs motifs selon les fragments de portées.

Enfin, l'automate tel quel ne présente pas toutes les exécutions erronées du système. Afin de mettre en évidence tous les cas d'erreur de la propriété, nous ajoutons une étape supplémentaire : la *complétion de l'automate*. Cette étape va permettre d'identifier les transitions dans l'automate qui permettent la violation de la propriété. Intuitivement, la composition identifie les transitions valides de la propriété et la complétion permet d'identifier les transitions d'erreur. La définition 19 formalise la complétion d'automate.

Définition 19 (Complétion d'automate)**.** Soit a un automate. Si l'ensemble des états d'erreur de l'automate a est vide, alors nous en ajoutons un nommé e. Sinon, c'est l'état d'erreur existant qui est considéré ? L'automate complété $\{Q, F, I, S, R, E, T\}$ est défini ainsi :

- $Q = Q_a \cup \{e\}$ si $E_a = \emptyset$, $Q = Q_a$ sinon,
- $F = F_a$,
- $I = I_a$,
- $R = R_a$ et $S = S_a$,
- $E = E_a$ ou $E = \{e\}$ si $E_a = \emptyset$,
- $q \xrightarrow{L} q' \in T$ si et seulement si une des conditions suivante est satisfaite :

1. $q \xrightarrow{L} q' \in T_a$
2. $q \in Q_a \wedge q' = e \wedge L = \Sigma_a - \{L' | \exists q'' \cdot q'' \in Q_a \backslash E_a \wedge q \xrightarrow{L'} q'' \in T_a\} \wedge L \neq \emptyset$

Nous considérons les transitions sortantes de chaque état et, plus particulièrement, l'ensemble des étiquettes des transitions sortantes d'un état. Deux cas sont possibles :

— soit l'ensemble des étiquettes des transitions sortantes représente tous les évènements possibles du système (nous avons donc une complétude sur cet état). Dans ce cas, il n'y a pas de complétion,
— soit l'ensemble est incomplet par rapport à Σ. Dans ce cas, tous les évènements qui ne sont pas pris en compte étiquetteront une transition ciblant l'état d'erreur.

Cette étape de complétion permet d'obtenir un automate complet qui met en évidence les transitions violant la propriété.

5.2.3 Evénements

Dans la sémantique introduite par Dwyer *et al.*, le type des évènements n'est pas défini. Cependant, dans notre langage, il existe deux types d'évènements. Nous devons donc uniformiser la notation pour les utiliser en tant qu'étiquettes des automates. Nous utilisons la structure suivante pour décrire un évènement étiquetant une transition :

[opération, précondition, postcondition, tags]

Le composant *opération* est le nom de l'opération déclenchée par l'évènement. Le composant *précondition* est la précondition de l'évènement. C'est un prédicat qui doit être satisfait sur l'état du système *avant* l'activation de l'évènement. La *postcondition* représente un prédicat d'état qui doit être satisfait *après* activation de l'évènement. Enfin, *tags* représente l'ensemble des comportements autorisés par cet évènement. Une transition est activable si et seulement si ces quatre composantes sont satisfaites, en particulier le comportement activé doit être un de ceux de l'ensemble des tags.

Cette structure générique permet de capturer les deux types d'évènements. Un évènement de type `isCalled` du langage trouve une correspondance directe dans la forme d'évènement que nous proposons. En effet, les deux contiennent une référence à un nom d'opération, aux prédicats de précondition et de postcondition sur l'état du système avant et après l'appel d'opération et une liste de tags représentant les comportements autorisés par l'évènement. Certains composants de notre notation peuvent rester non-définis, dans ce cas, ils seront dénotés par "_".

Exemple 21 (Réécriture des exemples d'évènements `isCalled`)**.** Pour montrer cette traduction, nous reprenons les évènements décrits dans l'Exemple 19 :

`isCalled`(sut.buyTicket, `post` : "self.current_user.all_tickets_in_basket=5" **on** "sut",
`including` :{@AIM :BUY_Success})
⇓
[sut.buyTicket, _, "self.current_user.all_tickets_in_basket=5" **on** "sut", {@AIM :BUY_Success}]

Et le deuxième exemple se réécrit de la manière suivante :

`isCalled`(sut.login, `pre` :"not(self.current_user.oclIsUndefined())" **on** "sut")
⇓
[sut.login, "not(self.current_user.oclIsUndefined())" **on** "sut", _, _]

En ce qui concerne les évènements de type `becomesTrue`, cela signifie que, avant l'appel d'une opération quelconque, le prédicat sur l'état est évalué à *faux* et après cet évènement il est évalué à *vrai*. Un évènement *becomeTrue(P)* se traduira dans notre notation par

[_, not(P), P, _]

Exemple 22 (Réécriture de l'exemple d'évènement `becomesTrue`)**.** Pour illustrer, reprenons l'exemple de l'authentification réussie d'un utilisateur de l'Exemple 20 :

`becomesTrue`("not(self.current_user.oclIsUndefined())" **on** "sut")
⇓
[_, "self.current_user.oclIsUndefined()" **on** "sut",
"not(self.current_user.oclIsUndefined())" **on** "sut", _]

Cette notation permet d'étiqueter de manière uniforme les transitions de nos automates de substitution. De plus, elle nous permet de définir la soustraction d'évènements plus facilement. En effet, un évènement $[op, pre, post, tags]$ est activable si et seulement tous ses composants sont satisfaits, c'est-à-dire qu'il y a un appel de l'opération op, à partir d'un état satisfaisant pre, amenant dans un état satisfaisant $post$ et utilisant au moins un des tags de $tags$.

L'évènement $[op, pre, post, tags] \backslash [op', pre', post', tags']$, noté E\F, n'autorise pas l'activation de l'évènement $[op', pre', post', tags']$. Toutefois, puisque l'activation d'un évènement nécessite la satisfaction des quatre composantes, la négation d'au moins une de ces composantes rend l'évènement possible. Ainsi, l'évènement $[op, pre \wedge pre', post \wedge not(post'), tags]$ est capturé par la soustraction car il active l'évènement E mais n'active pas F car il ne satisfaitpas sa postcondition $post'$, même si l'état de départ satisfait pre. Ainsi, dans le cas où $op = op'$, cette restriction est équivalente à l'ensemble des évènements suivants :

- $[op, pre \wedge not(pre'), post \wedge not(post'), tags \backslash tags']$
- $[op, pre \wedge not(pre'), post \wedge post', tags \backslash tags']$
- $[op, pre \wedge pre', post \wedge not(post'), tags \backslash tags']$
- $[op, pre \wedge pre', post \wedge post', tags \backslash tags']$
- $[op, pre \wedge not(pre'), post \wedge not(post'), tags]$
- $[op, pre \wedge not(pre'), post \wedge post', tags]$
- $[op, pre \wedge pre', post \wedge not(post'), tags]$

Dans le cas où $op \neq op'$, cette restriction n'a pas d'incidence sur l'évènement de base car il s'agit du cas où la composante d'opération est niée. Dans ce cas, la restriction $[op, pre, post, tags] \backslash [op', pre', post', tags']$ est équivalente à l'évènement $[op, pre, post, tags]$.

5.3 Sémantique du langage

Les automates de substitution étant définis, nous pouvons maintenant définir la sémantique des portées et des motifs de notre langage par des automates. La sémantique d'une propriété est alors obtenue par un mécanisme de substitution de l'automate de motif dans l'automate de la portée.

5.3.1 Sémantique des portées

Nous allons dans un premier temps définir la sémantique des portées. Les portées permettent de délimiter des portions d'exécutions qui doivent vérifier un certain motif.

FIGURE 5.5 – Automate de la portée `globally`

FIGURE 5.6 – Automate de la portée `before` E

FIGURE 5.7 – Automate de la portée `after` E

Une portée ne donne aucune indication sur le comportement du système vis-à-vis de la propriété hors de la portion d'exécution définie. Ainsi, pour une propriété donnée, si le système n'est pas dans l'intervalle d'exécution défini par sa portée, alors il peut arriver n'importe quoi, ce qui explique l'apparition de boucles autorisant n'importe quel évènement hors de la portée. Chaque automate de portée contient un et un seul état de substitution qui sera substitué par un automate de motif. Les énoncés textuels des exemples comportent des "...", qui sont à remplacer par le motif dans le cadre d'une propriété complète. Sur l'automate, ils font référence à l'état de substitution.

Sémantique de `globally`

Il s'agit de la portée la plus large. En effet, elle spécifie qu'à tout moment de l'exécution du système le motif qu'elle encadre doit être validé. Par conséquent, il n'y a aucun évènement délimiteur, juste le motif. La FIGURE 5.5 montre l'automate représentant cette portée.

Sémantique de `before` E

Cette portée spécifie que le motif doit être satisfait avant un certain évènement E. Après cet évènement, le satisfiabilité du motif n'a plus d'importance. La FIGURE 5.6 montre l'automate associé à cette portée et l'Exemple 23 illustre un cas d'utilisation de cette portée.

Exemple 23 (Utilisation de la portée `before` E)**.** Considérons l'énoncé suivant : "Avant une authentification réussie ...". Il s'écrit de la manière suivante :

```
...
before isCalled(login, including : {@AIM:LOG_Success})
```

et est représenté par l'automate de la FIGURE 5.6 où

E=[login, _, _, {@AIM :LOG_Success}]

Sémantique de `after` E

Cette portée est le pendant dans le futur de la portée `before`. Le motif doit être satisfait après la survenue de l'évènement E. La FIGURE 5.7 donne l'automate associé à cette portée et l'Exemple 24 illustre un cas d'utilisation de cette portée.

Exemple 24 (Utilisation de la portée `after` E)**.** Considérons l'énoncé suivant : "Après l'achat du dernier ticket ...". Il se traduit de la manière suivante :

```
...
after isCalled(buyTicket, including: {@AIM:BUY_Last_Ticket_Sold})
```

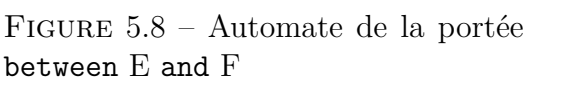


FIGURE 5.8 – Automate de la portée `between` E `and` F

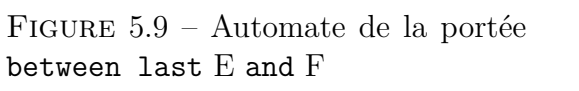


FIGURE 5.9 – Automate de la portée `between last` E `and` F

FIGURE 5.10 – Automate de la portée `after` E `until` F

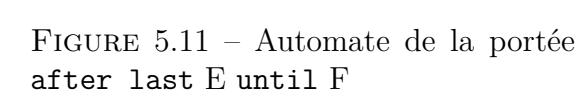


FIGURE 5.11 – Automate de la portée `after last` E `until` F

et est représenté par l'automate de la FIGURE 5.7 où

E=[buyTicket, _, _, {@AIM :BUY_Last_Ticket_Sold}]

Sémantique de `between` E `and` F

Cette portée spécifie un intervalle, entre l'arrivée de l'évènement E et l'arrivée de l'évènement F, durant lequel le motif qu'elle encadre doit être satisfait. Pour vérifier le motif, F doit être nécessairement arrivé pour fermer la portée. En attendant, nous ne pouvons pas conclure sur la satisfaction de la propriété, même si le motif a déjà été invalidé, jusqu'à ce que l'évènement F arrive. Enfin, il est à noter que l'évènement F est interdit dans l'état de substitution : cela nous permet de contrôler l'arrivée de la fin de la portée. La FIGURE 5.8 donne la sémantique associée à cette portée. Cette portée dispose d'une variante, avec le mot-clef `last` qui, au lieu de considérer la première occurrence de l'évènement E, considère la dernière occurrence de E. Par conséquent, cet évènement ramène au début du motif. La FIGURE 5.9 donne la sémantique de cette variante. L'automate de base et sa variante ne se distinguent au final que par la transition réflexive sur l'état de substitution, étiquetée par E qui apparaît dans la variante. L'Exemple 25 illustre un cas d'utilisation de cette portée.

Exemple 25 (Utilisation de la portée `between` E `and` F)**.** Considérons l'énoncé suivant : "Entre une connexion réussie et une déconnexion réussie . . . ". Il est traduit de la manière suivante :

```
...
between isCalled(login, including: {@AIM:LOG_Success})
and isCalled(logout, including: {@AIM:LOGOUT_Success})
```

et est représenté par l'automate de la FIGURE 5.8 où

E = [login, _, _, {@AIM :LOG_Success}]
F = [logout, _, _, {@AIM :LOGOUT_Success}]

Sémantique de `after` E `until` F

Cette portée est très similaire à la portée `between` E `and` F. Cependant, là où la portée précédente était inconclusive tant qu'elle n'était pas fermée, cette portée exige que le motif soit satisfait même sans la fermeture de la portée. Comme pour la portée précédente, nous restreignons l'état de substitution en interdisant l'évènement F, qui marque la fin de la portée. La FIGURE 5.10 donne la sémantique associée à cette portée. L'utilisation du

mot-clef `last`, illustrée en FIGURE 5.11 a exactement le même effet que pour la portée `between` : une transition réflexive est ajoutée sur l'état de substitution portant l'étiquette E. L'Exemple 26 illustre un cas d'utilisation de cette portée.

Exemple 26 (Utilisation de la portée `after` E `until` F)**.** Considérons l'énoncé suivant : "Après une authentification réussie et jusqu'à ce qu'il n'y ait plus d'utilisateur courant, ...". Il est traduit de la manière suivante :

```
...
after isCalled(login, including: {@AIM:LOG_Success})
until becomesTrue("current_user.oclIsUndefined()" on "sut")
```

et est représenté par l'automate de la FIGURE 5.10 où :

$$\begin{array}{lcl} \text{E} & = & [\text{login}, \_, \_, \{@\text{AIM} :\text{LOG\_Success}\}] \\ \text{F} & = & [\_, \text{"not(current\_user.oclIsUndefined())" on "sut"}, \\ & & \text{"current\_user.oclIsUndefined()" on "sut"}, \_] \end{array}$$

5.3.2 Sémantique des motifs

Nous allons maintenant définir la sémantique des motifs du langage. Il est à noter que les automates des motifs permettent de répéter le motif, à l'exception du motif `eventually`. Autrement dit, la répétition des enchaînements décrits par le motif doit se faire uniquement dans l'automate du motif.

Dans cette section, les exemples donnent des motifs généraux mais sont à remettre dans un contexte (la portée) pour représenter une propriété de notre étude de cas.

Sémantique de `always` P

Ce motif est particulier du fait qu'il ne se base pas sur un évènement, mais sur un prédicat sur l'état du système. Il permet de spécifier qu'à tout moment, dans la portée considérée, l'état doit satisfaire le prédicat P. Il peut être traduit par la répétition d'un évènement activant une opération quelconque dont la précondition et la postcondition respectent toutes les deux le prédicat P. La FIGURE 5.12 montre l'automate associé à ce motif.

Exemple 27 (Exemple de motif `always`)**.** Considérons le motif suivant : "Il y a toujours un utilisateur courant". Cela peut se traduire par le motif suivant :

```
always ("not(current_user.oclIsUndefined())" on "sut")
```

qui signifie qu'il y a toujours un utilisateur authentifié. La FIGURE 5.12 donne la représentation en automate de ce motif où :

$$\text{P} \quad = \quad \text{"not(current\_user.oclIsUndefined())" } \textbf{on} \text{ "sut"}$$

FIGURE 5.12 – Automate du motif `always` P

FIGURE 5.13 – Automate du motif `never` E

FIGURE 5.14 – Automate du motif `eventually` de l'Exemple 29

Sémantique de `never E`

Ce motif spécifie l'absence de l'évènement E dans la portée considérée. En d'autres termes, E ne doit jamais arriver. La FIGURE 5.13 donne l'automate représentant ce motif.

Exemple 28 (Exemple de motif `never`)**.** Considérons le motif suivant : "Il est impossible d'acheter un ticket si l'utilisateur n'est pas défini". Il se traduit dans le langage de propriétés par :

```
never isCalled(buyTicket,
               pre : "not(current_user.oclIsUndefined())" on "sut"),
               including : {@AIM:BUY_Success})
```

qui s'interpète comme le fait qu'il n'y a jamais un appel de *buyTicket* avec succès si l'utilisateur n'est pas défini. La FIGURE 5.13 donne la représentation en automate de ce motif où :

E = [buyTicket, "not(current_user.oclIsUndefined())" **on** "sut", {@AIM :BUY_Success}]

Sémantique de `eventually E`

Ce motif spécifie la necessité de l'occurrence d'un évènement E dans la portée considérée. Il dispose de plusieurs variantes basées sur le nombre d'occurrences attendues de l'évènement :

— exactement k fois, représenté par l'automate de la FIGURE 5.15
— au moins k fois avec le mot-clef `at least`, représenté en FIGURE 5.16
— au plus k fois avec le mot-clef `at most`, représenté en FIGURE 5.17

FIGURE 5.15 – Automate du motif `eventually` E k `times`

FIGURE 5.16 – Automate du motif `eventually` E `at least` k `times`

FIGURE 5.17 – Automate du motif `eventually` E `at most` k `times`

FIGURE 5.18 – Automate du motif `eventually` E

FIGURE 5.19 – Automate du motif E `follows` F

FIGURE 5.20 – Automate du motif E `directly follows` F

— aucun mot-clef indique au moins un occurrence de l'évènement. Il s'agit d'un raccourci d'écriture pour le motif `eventually at least` E 1 `times`. Il est représenté en FIGURE 5.18.

Exemple 29 (Exemple de motif `eventually`)**.** Considérons le motif suivant : "Un utilisateur authentifié doit acheter au moins 2 tickets". Cela se traduit dans le langage de propriétés :

```
eventually isCalled(buyTicket,
          pre : "not(current_user.oclIsUndefined())" on "sut",
          including : {@AIM:BUY_Success}) at least 2 times
```

La FIGURE 5.14 donne la représentation en automate de ce motif qui est une instanciation de l'automate de la FIGURE 5.16 avec $k = 2$ où

E = [buyTicket, "current_user.oclIsUndefined())" **on** "sut", _, {@AIM :BUY_Success}]

Sémantique de E `follows` F

Ce motif spécifie l'enchaînement de deux évènements : si l'évènement E survient, alors il doit y avoir au moins une occurrence de l'évènement F avant la fermeture de la portée. Dans le cas où la portée ne se ferme pas, typiquement la portée `after`, le verdict de la propriété est inconclusif tant que l'évènement F n'arrive pas. Dans ce cas, la propriété est une propriété de vivacité.

Ce motif propose une variante qui spécifie l'enchaînement direct des deux évènements E et F : un autre évènement que F après un E entraîne une erreur dans le motif et par conséquent, dans la propriété. La FIGURE 5.19 donne la sémantique du motif et la FIGURE 5.20 celle de sa variante.

Exemple 30 (Exemple de motif `follows`)**.** Considérons le motif suivant : "L'utilisateur s'authentifie puis se désauthentifie". Ce motif se traduit dans le langage de propriétés par :

```
isCalled(logout, including : {@AIM:LOG_Success})
follows isCalled(login, including : {@AIM:LOGOUT_Success})
```

La FIGURE 5.19 donne l'automate de motif associé à cet exemple où

$$\begin{aligned} \text{E} &= [\text{logout}, \_, \_, \{\text{@AIM :LOGOUT\_Success}\}] \\ \text{F} &= [\text{login}, \_, \_, \{\text{@AIM :LOG\_Success}\}] \end{aligned}$$

FIGURE 5.21 – Automate du motif E `precedes` F

FIGURE 5.22 – Automate du motif E `directly precedes` F

FIGURE 5.23 – Automate réduit du motif E `directly precedes` F

Sémantique de E `precedes` F

Ce motif dispose d'une variante, similaire à celle du motif précédent, qui impose un enchaînement direct des deux évènements. La FIGURE 5.21 donne la sémantique du motif `precedes` et la FIGURE 5.22 donne sa variante (à gauche). La différence entre les deux automates est bien plus importante que pour le motif `follows` et sa variante. En effet, la FIGURE 5.23 montre une version réduite de la variante et possède une taille comparable au motif de base. Cependant, cet automate n'exhibe pas clairement l'enchaînement E-F, l'évènement F étant caché dans l'évènement $\Sigma\backslash\{\text{E}\}$. De fait, nous privilégions l'automate étendu (FIGURE 5.22). Dans le cas contraire, nous ne pouvons pas différencier l'évènement F d'un autre évènement de la transition.

Enfin, il est à noter que tous les états de ces deux motifs sont finaux, à l'exception de l'état d'erreur. Cela s'explique par le fait qu'il n'y a jamais d'incertitude concernant sa satisfaction : ces motifs fondent leur satisfiabilité sur un évènement du passé (propriété de sûreté), contrairement à d'autres motifs où la satisfiabilité est fondée sur des évènements futurs (propriété de vivacité), le motif `follows` en est un exemple. Ainsi, le fait que tous les états de ces motifs soient finaux représente le fait qu'à tout moment du motif qu'il n'y a pas d'incertitude quant à sa satisfaction.

5.3.3 Application de la composition d'automate

Les automates de portée et de motif étant définis, nous allons illustrer l'application de la substitution d'automates en utilisant une portée comme automate de base et un motif pour l'automate substitué. Prenons l'exemple de la propriété générique A `directly follows` B `between` C `and` D. Cette propriété permet de montrer la plupart des règles de substitution. Initialement, nous avons respectivement les automates de portée (automate de base a, à gauche) et de motif (automate à substituer, à droite) suivants :

Dans un premier temps, nous appliquons les définitions de Q_c, I_c, F_c et E_c (règles I., II., III. et VI. respectivement) :

L'état de substitution n'est pas final, donc l'état d'erreur du motif e est inclus dans les états de l'automate résultat mais perd son statut d'état d'erreur par définition de E_c et redevient normal (partie $\bigcup_{s \in S_a \backslash F_a} Q_{\sigma(s)}$ de la règle I. de la définition 18). Notons que sur l'exemple $S_c = \emptyset$ et $R_c = \emptyset$, car il n'y a pas d'état de substitution dans l'automate de motif.

Ensuite, nous pouvons appliquer les points a et f de la règle VIII. pour les transitions, ce qui a pour effet de créer les transitions entre nœuds simples (qui ne sont ni d'erreur, ni de substitution), en restreignant par l'évènement D toutes les transitions de l'automate du motif :

Nous appliquons ensuite le point b de VIII. afin de relier l'état initial de l'automate de substitution à toutes les transitions qui entraient sur l'état de substitution :

Ensuite, nous appliquons le point c de VIII. pour fermer la portée :

A ce stade, il ne reste qu'à relier les transtions aux états d'erreurs. L'état de substitution n'étant pas final, c'est la règle h de VIII. qui s'applique. Ainsi, une nouvelle transition apparaît entre l'ancien état d'erreur et le nouvel état d'erreur, étiqueté par la transition de sortie de la portée (D en l'occurrence) :

Nous terminons par la complétion de l'automate afin d'ajouter les dernières transitions d'erreur de la propriété :

L'automate final présente bien tous les cas possibles et interdits de la propriété.

5.4 Différences sémantiques avec Dwyer *et al.*

Les patrons de propriétés décrits dans ce chapitre, issus de ceux introduits par Dwyer *et al.*, ne font cependant que s'en inspirer. Nous récapitulons ici les différences entre les sémantiques originales et notre sémantique à base d'automates de substitution. Ces différences sont dues à une adaptation pour leur application dans le cadre de la génération de tests. Par exemple, du point de vue vérification, une séquence d'exécution où la portée ne s'exerce pas satisfait la propriété. En revanche, du point de vue du test, l'objectif est de générer des séquences d'exécutions où la portée s'exerce afin de valider le système sur de telles exécutions. Deux différences les séparent : le choix du type d'évènements considérés et la notion de satisfaction d'une propriété donnée par une séquence d'exécution.

5.4.1 Types d'évènements

Le premier point sur lequel notre sémantique s'écarte des travaux de Dwyer *et al.* est le type des évènements. Dans les travaux originaux, les évènements sont indéfinis, ce qui permet aux auteurs de définir plusieurs sémantiques basées sur un type d'évènement. Ainsi, parmi les sémantiques que les auteurs donnent, la sémantique à base d'expressions régulières quantifiées (QRE) utilise des évènements du système, alors que les sémantiques en LTL et CTL utilisent des propriétés d'états comme évènements. De fait, en restreignant le type des évènements à des appels d'opérations du système, nous définissons une logique d'action alors que les logiques telles que LTL et CTL sont des logiques d'états.

5.4.2 Notion de satisfiabilité de propriétés

Chaque automate est une définition de ce que signifie chaque portée et motif. Si nous comparons le langage décrit par l'expression régulière du motif `follows` telle que donnée par Dwyer *et al.* (qui est en fait mise dans la portée `globally` dans la sémantique originale), nous nous apercevons que l'automate que nous avons proposé définit un sous-langage de ce qu'autorise l'expression régulière décrite par Dwyer *et al.*

La différence entre ces sémantiques est due à la différence de la notion de *satisfaction* d'une propriété. Dwyer *et al.* considèrent qu'une séquence d'opérations qui n'active pas la portée satisfait la propriété. Nous avons, de notre coté, fait le choix qu'une séquence d'opérations qui n'atteint pas au moins un état final de l'automate, n'est pas considérée comme exerçant la propriété. L'exemple 31 illustre cette différence sémantique.

Exemple 31 (Différence sémantique)**.** Considérons la séquence suivante :

$$\Sigma\backslash\{E\}; \Sigma\backslash\{E\}; \Sigma\backslash\{E\}; \Sigma\backslash\{E\}$$

et une propriété qui utilise la portée `between` E `and` F. Dans le cas de la sémantique définie par des expressions régulières, Dwyer *et al.* considérent cette séquence comme satisfaisant la propriété, bien qu'elle n'entre jamais dans sa portée. Dans notre cas, l'automate représentant la portée `between` (voir FIGURE 5.8) reconnaît cette séquence en restant sur l'état initial qui n'est pas final. La propriété n'est donc pas considérée comme satisfaite.

5.5 Quelques exemples de propriétés

Nous donnons ici trois exemples de propriétés qui illustrent la sémantique que nous avons donnée. Dans les chapitres suivants, nous ferons référence à ces propriétés et aux automates associés, pour illustrer les exploitations possibles de ces automates. L'exemple 32 est une propriété que l'on pourrait qualifier d'*invariant* dans une portée donnée.

Exemple 32 (Propriété 1)**.** Considérons la propriété suivante : "Entre une authentification réussie et une désauthentification, il y a toujours un utilisateur courant". Elle peut être écrite dans le langage de propriétés de la matière suivante :

```
always ("not(current_user.oclIsUndefined())" on "sut")
between isCalled(login,including : {@AIM:LOG_Login_Success})
and isCalled(logout)
```

FIGURE 5.24 – Automate de `always` C `between` A `and` B

FIGURE 5.25 – Automate de `eventually` B `before` A

Sa sémantique est représentée par l'automate de la FIGURE 5.24 où :

A = [login, _, _, {@AIM :LOG_Login_Success}]
B = [logout, _, _, _]
C = [_,"not(current_user.oclIsUndefined())" **on** "sut",
"not(current_user.oclIsUndefined())" **on** "sut",_]

L'exemple 33 illustre la transformation de l'état d'erreur.

Exemple 33 (Propriété 2)**.** Considérons la propriété suivante : "Pour qu'un utilisateur puisse supprimer avec succès un ticket, il a dû acheter au moins un ticket avec succès". Cette propriété s'écrit par :

```
eventually isCalled(buyTicket, including:{@AIM:BUY_Success})
before isCalled(deleteTicket, including:{@AIM:REM_Success})
```

Sa sémantique est représentée par l'automate de la FIGURE 5.25 où

A = [deleteTicket,_,_,{@AIM :REM_Success}]
B = [buyTicket,_,_,{@AIM :BUY_Success}]

Enfin, l'exemple 34 montre ce qu'entraînent des interprétations d'une même propriété textuelle. La première est l'écriture la plus intuitive. La seconde, au contraire, l'est moins. Cependant, cette différence d'interprétation dans l'écriture d'une propriété permet d'illustrer différents aspects complémentaires de la propriétés.

Exemple 34 (Propriétés 3a et 3b)**.** Considérons la propriété suivante : "En étant sur la page d'enregistrement, un utilisateur authentifié qui souhaite supprimer un de ses tickets acheté doit d'abord se rendre sur la page de bienvenue puis sur la page de visualisation".
Deux interprétations sont possibles. La première se formule de la manière suivante :

```
isCalled(showBoughtTicket,
        including : {@AIM:NAV_Show_Bought_Tickets})
follows isCalled(goToHome, including : {@AIM:NAV_Go_To_Home})
between becomesTrue("currentPage = REGISTER" on "sut")
    and isCalled(deleteTicket, including : {@AIM:REM_Success})
```

Sa sémantique est représentée par l'automate de la FIGURE 5.26 où

A = [showBoughtTicket,_,_,{@AIM :NAV_Show_Bought_Tickets}]
B = [goToHome,_,_,{@AIM :NAV_Go_To_Home}]
C = [_,"currentPage=REGISTER" **on** "sut",
"currentPage<>REGISTER" **on** "sut",_]
D = [deleteTicket,_,_,{@AIM :REM_Success}]

FIGURE 5.26 – Automate de A `follows` B `between` C `and` D

FIGURE 5.27 – Automate de A `follows` B `before` C

La seconde interprétation se formule de la manière suivante :

```
        isCalled(goToHome,
                 pre : "currentPage = REGISTER" on "sut",
                 including : {@AIM:NAV_Go_To_Home})
follows isCalled(showBoughtTicket,
                 including : {@AIM:NAV_Show_Bought_Tickets})
 before isCalled(deleteTicket, including : {@AIM:REM_Success})
```

Sa sémantique est représentée par l'automate de la FIGURE 5.27 où

A = [goToHome,"currentPage = REGISTER" **on** "sut",_,{@AIM :NAV_Go_To_Home}]
B = [showBoughtTickets, _, _, {@AIM :NAV_Show_Bought_Tickets}]
C = [deleteTicket,_,_,{@AIM :REM_Success}]

Cette formalisation correspond à la propriété textuelle suivante : "Avant de pouvoir supprimer un ticket avec succès, à partir de la page d'enregistrement, il est nécessaire de passer par la page de bienvenue puis par la page de visualisation". Il est à noter que cet énoncé, bien que très proche de l'original, a changé l'automate associé.

5.6 Synthèse

Nous avons défini dans ce chapitre un langage d'expression de propriétés temporelles et sa sémantique fondée sur les automates de substitution.

Ce langage s'inspire des patrons de propriétés définis par Dwyer *et al.* dans [DAC99]. Ils permettent d'exprimer un large sous-ensemble des propriétés que l'on rencontre fréquemment dans les spécifications de systèmes, tout en utilisant un ensemble restreint de constructions. En effet, les auteurs montrent que ces constructions sont suffisantes pour couvrir 92% des propriétés temporelles sur les 555 exemples étudiés. Nous augmentons encore cette proportion avec l'ajout du mot clef `last` pour certaines portées.

La sémantique, basée sur les automates, est définie de manière compositionnelle entre les portées et les motifs. Cela permet de conserver, après composition, la distinction entre la portée et le motif d'une propriété (que nous avons représentée en colorant les états de nos automates). Cette distinction, qui était perdue avec les sémantiques proposées par Dwyer *et al.* est centrale pour la définition de certains critères spécifiques sur nos automates que nous décrivons dans le Chapitre 6.

Enfin, nous avons comparé la sémantique à base d'automates avec celles introduites par Dwyer *et al.*. Bien que ce langage s'inspire en grande partie des patrons de Dwyer *et al.*, la sémantique que nous proposons ne correspond pas complètement aux sémantiques

originales, notamment en ce qui concerne la notion de satisfiabilité d'une propriété. Cette adaptation de la sémantique est nécessaire dans le cadre de l'utilisation des automates pour le test, d'une part pour la définition de critères de couverture spécifiques et la mesure de leur couverture dans les Chapitre 6 et 7, et d'autre part, pour la génération de tests à partir de ces critères dans le Chapitre 8.

Chapitre 6

Critères et mesure de couverture de propriétés

Sommaire

6.1 Définitions et notations 110
6.1.1 Exemple de propriété 110
6.1.2 Définitions 110
6.1.3 Animation des tests 111
6.2 Critères et mesure de couverture des automates 113
6.2.1 Critère all-α : toutes les α-transitions 113
6.2.2 Critère *all-α-pairs* : toutes les paires d'α-transitions 114
6.2.3 Critère *k-pattern* : k itérations de `precedes` et `follows` . . 115
6.2.4 Critère *k-scope* : k itérations de `between/and` et `after/until` 116
6.3 Discussion sur les critères 117
6.3.1 Hiérarchisation des critères 117
6.3.2 Applicabilité des critères 121
6.4 Synthèse 123

Nous avons introduit dans le chapitre précédent un langage d'expression de propriétés inspiré des travaux de Dwyer *et al.* sur les patrons de propriétés. Nous avons donné à ce langage une sémantique à base d'automates spécifiques appelés *automates de substitution*. Ces automates représentent les exécutions permises par les propriétés qu'ils représentent et donnent une visibilité aux exécutions erronées par le biais d'états

Dans ce chapitre, nous nous intéressons à la définition de critères de couverture spécifiques aux automates représentant les propriétés et à leur couverture. Dans un premier temps, la section 6.1 définit les concepts utilisés dans la suite du chapitre. Ensuite, la section 6.2 présente les critères associés aux automates qui permettent de définir les exécutions intéressantes parmi celles autorisées par la propriété. Enfin, la section 6.3 propose une discussion sur ces nouveaux critères, notamment leur positionnement par rapport aux critères classiques et leur applicabilité.

6.1 Définitions et notations

Dans un premier temps, afin d'illustrer les définitions et les concepts introduits dans ce chapitre, nous utilisons la propriété suivante comme exemple.

6.1.1 Exemple de propriété

Cet exemple illustre des cas d'utilisation des opérations *deleteAllTickets* et de *buyTicket* de notre exemple fil rouge eCinema. Cette propriété vise à tester le fait qu'il est possible pour un utilisateur qui se connecte avec un panier vide, de supprimer tous les tickets s'il y a eu au moins un achat de ticket auparavant. Ce motif doit être maintenu tant qu'un utilisateur est authentifié. Cette propriété est exprimée dans le langage de propriétés de la manière suivante :

```
            isCalled(buyTicket, pre :"current_user.all_tickets_in_basket→size()=0" on "sut",
            including :{@AIM :BUY_Success}])
precedes    isCalled(deleteAllTickets, pre : "not(current_user.oclIsUndefined()) and
            current_user.all_tickets_in_basket→size()>0" on "sut",
            including :{@AIM :REMALL_Dell_All_Tickets})
after        isCalled(login, post :"current_user.all_tickets_in_basket→size()=0" on "sut",
            including :{@AIM :LOG_Success})
until       isCalled(logout, including :{@AIM :LOG_Logout})
```

Il est à noter qu'il peut y avoir des suppressions de tickets entre l'achat et la suppression de tous les tickets. Cependant, la précondition du second événement du *precedes* avec l'opération *deleteAllTickets* impose l'existence d'au moins un ticket dans le panier, ce qui présuppose au moins un achat entre la dernière suppression et la prochaine suppression avec succès.

La FIGURE 6.1 donne l'automate correspondant à cette propriété. Par souci de lisibilité, nous étiquetons les transitions par A, B, C et D qui correspondent aux évènements suivants de la propriété :

```
A = [login,_,"current_user.all_tickets_in_basket→size()=0" on "sut",
    {@AIM :LOG_Success}]
B = [logout,_,_,{@AIM :LOG_Logout}]
C = [buyTicket, "current_user.all_tickets_in_basket→size()=0" on "sut", _,
    {@AIM :BUY_Success}]
D = [deleteAllTickets, "not(current_user.oclIsUndefined()) and
    current_user.all_tickets_in_basket→size()>0" on "sut", _,
    {@AIM :REMALL_Dell_All_Tickets}]
```

FIGURE 6.1 – Automate de l'exemple C `precedes` D `after` A `until`

6.1.2 Définitions

Dans la suite du chapitre, nous faisons la distinction entre deux types de transitions dans l'automate : les α-transitions et les Σ-transitions.

Définition 20 (α-transition)**.** Une α-transition est une transition qui est étiquetée par un seul évènement de la propriété, auquel peut éventuellement être soustrait un ensemble d'évènements.

Définition 21 (Σ-transition)**.** Une Σ-transition est une transition étiquetée par un évènement Σ, auquel peut éventuellement être soustrait un ensemble d'évènements.

Dans l'exemple en FIGURE 6.1, les Σ-transitions sont toutes les transitions réflexives de l'automate car elles sont étiquetées par un évènement de la forme $\Sigma\backslash\{\ldots\}$. Les autres transitions, comme $1\xrightarrow{A}2$ et $2\xrightarrow{C\backslash\{B\}}3$ sont des α-transitions. Une transition est toujours soit une α-transition, soit une Σ-transition.

L'automate présente toutes les exécutions possibles, et interdites dans le cadre de propriété de sûreté, vis-à-vis de la propriété. Puisque nous considérons que le modèle est correct vis-à-vis de cette propriété, tous les sous-chemins qui mènent vers un état d'erreur ne peuvent jamais être activés et sont considérés comme fautifs. Pour les identifier, nous distinguons les transitions d'erreur et les transitions activables du modèle.

Définition 22 (Transition d'erreur/activable)**.** Une *transition d'erreur* est une transition qui amène à l'état d'erreur de l'automate. A l'inverse, une transition *activable* ne mène pas à un état d'erreur.

Les transitions *activables*, sont toutes les transitions qui sont *potentiellement* activables sur le modèle. En effet, bien que ces transitions soient valides vis-à-vis de la propriété, nous n'avons pas la garantie que le modèle puisse effectivement les activer. Les transitions d'erreur, quant à elles, ne peuvent jamais être activées car elle permettent la violation de la propriété, ce que le modèle ne permet pas, par hypothèse.

Définition 23 (Chemin fautif)**.** Un chemin fautif est un chemin dans l'automate qui contient au moins une transition d'erreur. Autrement dit, tous les chemins fautifs se terminent par l'état d'erreur de l'automate.

Exemple 35 (Exemple de chemin fautif)**.** Dans l'exemple de propriété FIGURE 6.1, il existe des chemins fautifs qui passent par la transition menant à l'état d'erreur $2\xrightarrow{D\backslash\{B,C\}}e$, e étant l'état d'erreur de l'automate (représenté par l'état barré d'une croix). La transition $e\xrightarrow{\Sigma}e$ appartient aussi aux chemins fautifs de l'automate.

La FIGURE 6.1 illustre en gras les transitions d'erreur de l'automate.

6.1.3 Animation des tests

Afin de mesurer la couverture, nous avons besoin d'une suite de tests existante sur laquelle nous devons évaluer les activations des éléments à couvrir pour chaque critère. La provenance de cette suite n'a pas d'importance : il peut s'agir de tests générés automatiquement ou des tests écrits manuellement. Nous définissons dans un premier temps ce qu'est un *test* et les éléments qui le composent, les *pas de test*.

Définition 24 (Test et pas de test)**.** Un pas de test $c_i = \vec{o_i}, tags_i \leftarrow op_i(\vec{in_i})$ est l'appel de l'opération op_i avec des paramètres d'entrée $\vec{in_i}$, de sortie $\vec{o_i}$ et qui a activé les tags de $tags_i$ de l'opération op_i.

Un test t est une séquence de pas de tests $t = [c_1, \ldots, c_n]$ représentant une trace d'exécution du système.

Nous pouvons maintenant définir l'animation de tests sur un modèle.

Définition 25 (Animation d'un test)**.** Soit un test $t = [c_1, \ldots, c_n]$ et un état initial d'animation q_0. L'animation $\chi(t)$ du test t est définie par :

$$\chi(t) = [(q_0, c_1, q_1), (q_1, c_2, q_2), \ldots, (q_{n-1}, c_n, q_n)]$$

Chaque triplet (q_i, c, q_{i+1}) de la séquence correspond à un pas d'animation, où q_i est l'état du système avant l'exécution du pas, c est le pas de test animé et q_{i+1} est l'état après l'exécution du pas de test. Il est à noter que l'état initial de chacun des pas est l'état final du pas précédent, à l'exception du premier pas où l'état initial correspond à l'état initial du système q_0.

Ces tests sont ensuite animés sur l'automate dont on mesure la couverture. Pour ce faire, chaque pas du test doit correspondre une transition de l'automate. Ainsi, les correspondances successives des pas de tests sur des transitions forment un chemin dans l'automate. C'est à partir de ces chemins que nous pouvons raisonner pour la couverture de l'automate.

Définition 26 (Couverture pas d'animation-transition)**.** Un pas d'animation $(q_i, tags_i \leftarrow op_i(\vec{in_i}), q_{i+1})$ couvre une transition étiquetée par l'évènement $[op, pre, post, tags]$ si et seulement si :

- $op = op_i$ ou op est non défini (par le symbole "_")
- pre est satisfait sur l'état q_i
- $post$ est satisfait sur l'état q_{i+1}
- $tags \cap tags_i \neq \emptyset$

Si un pas d'animation ne correspond à aucun évènement de la propriété, alors il correspond à la transition étiquetée par Σ.

Pour chaque cas de test, chacun de ces *pas* est comparé aux transitions qui peuvent être activées à partir de l'état courant, en utilisant q_0 lorsqu'il s'agit du premier état. Quand une correspondance est établie, l'exploration continue avec le pas d'animation suivant et en utilisant l'état cible de la transition correspondante comme nouvel état courant. L'automate étant déterministe et complet par construction grâce aux restrictions d'évènements, il y a exactement une transition correspondante pour chaque pas du cas de test considéré.

Les chemins considérés commencent nécessairement à l'état q_0 et passent par au moins un des états finaux de l'automate.

Définition 27 (Animation conforme, non conforme et inconclusive)**.** Soit une animation $\chi = [(q_0, c_0, q_1), \ldots, (q_{n-1}, c_{n-1}, q_n)]$ Une animation peut être qualifiée de trois manières différentes :

- *conforme* si, parmi les états traversés lors de l'animation du test, il y a au moins un état final $(\exists k \cdot 0 \leq k \leq n \wedge q_k \in F)$ et aucun état d'erreur n'a été atteint $(\forall k \cdot 0 \leq k \leq n \wedge q_k \notin E)$,

— *non conforme* si, lors de l'animation du test, l'état d'erreur a été atteint ($\exists k \cdot 0 \leq k \leq n \wedge q_k \in E$),
— *inconclusive* si, parmi les états traversés lors de l'animation du test, aucun état final ni d'erreur n'a été atteint ($\forall k \cdot 0 \leq k \leq n \Rightarrow q_k \notin E \wedge q_k \notin F$).

Si l'exécution est *conforme* alors le test satisfait la propriété et il est de plus considéré comme pertinent pour cette propriété, dans le sens où il l'illustre. Si l'exécution est *inconclusive*, nous ne pouvons rien dire quant à la pertinence ou la satisfaction de la propriété par le test.

Le cas d'une animation *non conforme* est intéressant. En effet, nous avons un vrai problème qui peut avoir deux causes : soit la propriété est mal décrite, soit le modèle est mal écrit. Quelle que soit la cause, il existe un problème de cohérence entre ces deux éléments. Dans le cas où la propriété est incorrectement décrite, il est nécessaire de la reprendre pour la faire correspondre à une propriété de la spécification. Par contre, si l'on considère que la propriété est correcte, alors l'erreur se trouve dans le modèle. Cette méthode nous permet de vérifier la cohérence entre le modèle et les propriétés et d'évaluer la pertinence, par un taux de couverture, des tests vis-à-vis de la propriété.

6.2 Critères et mesure de couverture des automates

L'automate de propriété décrit toutes les exécutions autorisées par la propriété dont il est issu. Or, dans le cadre du test qui est le thème central de cette thèse, seules certaines séquences particulières nous intéressent. A cette fin, il existe les critères de sélection classiques sur les automates tels que *all-transitions* et *all-k-paths*, mais ceux-ci, bien qu'applicables sur nos automates, ne tiennent pas compte des différentes caractéristiques de nos automates de propriété. Ainsi, ils traitent toutes les transitions de l'automate de manière indifférenciée, quelle que soit leur origine. Or, dans notre cas, nous voyons bien que les $\Sigma-$transitions ne sont pas aussi importantes que les $\alpha-$transitions.

Pour remédier à cela, nous proposons de nouveaux critères de couverture, spécifiques à nos automates de propriété. Ces critères sont définis comme des ensembles d'éléments à couvrir, différents selon le critère. De plus, pour chaque critère, nous définissons un taux de couverture qui permet d'évaluer la couverture de l'automate donc de la propriété.

6.2.1 Critère all-α : toutes les α-transitions

Ce critère est proche du critère *classique* de couverture *all-transitions* des automates. Cependant, il ne considère pas toutes les transitions, uniquement celles étiquetées par les évènements de la propriété. En d'autres termes, ce critère vise à n'activer que les α-transitions.

Nous restreignons ce critère en ne couvrant que les α-transitions activables : la couverture des transitions d'erreur amènerait à considérer des exécutions erronées, qui ne pourraient alors pas être animées sur le modèle supposé satisfaire la propriété.

Définition 28 (Critère *all-α*)**.** Soit $\mathcal{A}$ un automate de propriété, soit C_α l'ensemble des α-transitions activables de $\mathcal{A}$. Soit Ts une suite de tests. Le critère *all-α* est satisfait par

la suite Ts si chaque transition activable de C_α est couverte par au moins un test de la suite Ts.

Le taux de couverture associé T_α est défini ainsi :

$$\mathrm{T}_\alpha = \frac{\text{nombre d'}\alpha\text{-transitions couvertes par la suite } Ts}{\text{nombre total d'}\alpha\text{-transitions activables}}$$

Dans notre exemple de propriété de la FIGURE 6.1, C_α est défini par l'ensemble suivant :

$$\mathrm{C}_\alpha \quad = \quad \{1\xrightarrow{A}2,\ 2\xrightarrow{C\setminus\{B\}}3,\ 3\xrightarrow{D\setminus\{B\}}2,\ 2\xrightarrow{B}4,\ 3\xrightarrow{B}4,\ 4\xrightarrow{A}2\}$$

Ainsi, la couverture de l'automate par ce critère implique la couverture de chacune des six transitions de C_α. Il est à noter que la transition $2\xrightarrow{D\setminus\{B,C\}}$e (e est l'état d'erreur) n'est pas considérée car c'est une transition d'erreur.

6.2.2 Critère *all-α-pairs* : toutes les paires d'α-transitions

Ce critère, est proche du critère classique *all-pairs*, et vise à couvrir des paires de transitions consécutives dans l'automate. Similairement au critère précédent, ce critère ne considère que les paires d'α-transitions consécutives activables dans l'automate de propriété.

Définition 29 (Critère *all-α-pairs*)**.** Soit $\mathcal{A}$ l'automate de propriété considéré et soit C_α l'ensemble des $\alpha-$transitions activables de $\mathcal{A}$. Soit, pour une transition t, les fonctions $src(t)$ et $dest(t)$ qui définissent l'état de départ et l'état d'arrivée de la transition t. Alors l'ensemble des paires d'α-transitions activables considérées C_{α^2} est défini par :

$$C_{\alpha^2} = \{(t_1, t_2) \in C_\alpha \times C_\alpha \mid src(t_2) = dest(t_1)\}$$

Une paire de transition (A,B) est couverte par un test si et seulement si le test active la transition A puis la transition B. La propriété est dite couverte vis-à-vis de ce critère pour une suite de tests Ts si chaque paire de transitions de C_{α^2} est couverte par au moins un test de Ts.

Le taux de couverture associé T_{α^2} est défini ainsi :

$$\mathrm{T}_{\alpha^2} = \frac{\text{nombre de paires couvertes par au moins un test de Ts}}{\text{nombre total de paires de } C_{\alpha^2}}$$

Dans notre exemple de propriété de la FIGURE 6.1, C_{α^2} est défini par l'ensemble suivant :

$$\begin{aligned} C_{\alpha^2} \quad = \quad & \{[1\xrightarrow{A}2,\ 2\xrightarrow{C\setminus\{B\}}3], \\ & [1\xrightarrow{A}2,\ 2\xrightarrow{B}4], \\ & [2\xrightarrow{C\setminus\{B\}}3,\ 3\xrightarrow{B}4], \\ & [2\xrightarrow{C\setminus\{B\}}3,\ 3\xrightarrow{D\setminus\{B\}}2], \\ & [3\xrightarrow{B}4,\ 4\xrightarrow{A}2], \\ & [2\xrightarrow{B}4,\ 4\xrightarrow{A}2]\} \end{aligned}$$

Ainsi, la couverture de l'automate par ce critère implique la couverture de chacune des six paires d'$\alpha-$transitions de C_{α^2}.

6.2.3 Critère *k-pattern* : k itérations de `precedes` et `follows`

Ce critère vise à couvrir un certain nombre de fois les évènements des motifs `precedes` et `follows` (et leurs variantes). Plus spécifiquement, ce critère s'intéresse à l'aspect répétitif de ces motifs en visant plusieurs itérations, par la répétition de paires particulières. Dans A `precedes` B, l'évènement A doit arriver avant l'évènement B : la paire d'évènements à considérer est donc (A, B). Dans A `follows` B, c'est l'inverse : A doit arriver après B. Par conséquent la paire d'évènements à considérer pour ce motif est donc (B, A).

Ainsi, ce critère vise la couverture d'au moins k itérations des paires de transitions considérées.

Définition 30 (Critère *k-pattern*)**.** Soient $\mathcal{A}$ un automate de propriété, k le nombre d'activations souhaité pour le motif et (E_1, E_2) la paire d'évènements à couvrir. Soit Ts une suite de test. Soit T_{E_1} (resp. T_{E_2}) l'ensemble des transitions étiquetées par l'évènement censé arriver en premier (resp. en second) dans le motif. Soit C_p^k l'ensemble de séquences de paires à couvrir dans le motif. Il est défini par :

$$\begin{aligned} C_p^1 &= T_{E_1} \times T_{E_2} \\ C_p^k &= (\bigcup_{q \in C_p^1} \bigcup_{s \in C_p^{k-1}} q \cdot s) \cup C_p^{k-1}) \end{aligned}$$

où $q \cdot s$ dénote la concaténation de la paire q avec la séquence de paire s.

$$\mathrm{T}_p^k = \frac{\text{nombre de séquences de } C_p^k \text{ couvertes par Ts}}{\text{nombre total de séquences de } C_p^k}$$

Cette définition fait apparaître que la satisfaction du critère pour *k-pattern* implique nécessairement la couverture du critère *(k-1)-pattern*.

Il est à noter que T_{E_1} et T_{E_2} ne sont pas interchangeables. En effet, dans le cas du motif A `follows` B, T_{E_1} correspond aux transitions étiquetées par B et T_{E_2} celles étiquetées par A. L'évènement A doit arriver *après* l'évènement B car *A suit B*. Dans le cas du `precedes`, c'est l'inverse : T_{E_1} correspond aux transitions étiquetées par A et T_{E_2} celles étiquetées par B car A doit arriver *avant* B.

Dans notre exemple, nous utilisons le motif C `precedes` D. Ainsi, il n'y a que la paire $[(2 \xrightarrow{C\setminus\{B\}} 3, 3 \xrightarrow{D\setminus\{B\}} 2)]$ à considérer. Pour $k = 2$, C_p^2 est alors défini par :

$$\begin{aligned} C_p^2 &= C_p^1 \bigcup C_p^2 \\ &= \{[(2 \xrightarrow{C\setminus\{B\}} 3, 3 \xrightarrow{D\setminus\{B\}} 2)]\} \\ &\quad \bigcup \{[(2 \xrightarrow{C\setminus\{B\}} 3, 3 \xrightarrow{D\setminus\{B\}} 2) ; (2 \xrightarrow{C\setminus\{B\}} 3, 3 \xrightarrow{D\setminus\{B\}} 2)]\} \end{aligned}$$

où $[\mathrm{p}_1 ; \ldots ; \mathrm{p}_n]$ dénote la séquence de paires de transitions $\mathrm{p}_1, \ldots, \mathrm{p}_n$.

6.2.4 Critère *k-scope* : k itérations de `between/and` et `after/until`

Ce critère est spécifique aux portées `between/and`, `after/until` et leurs variantes. En effet, ce sont les seules qui permettent de se répéter. Ce critère est très proche du critère précédent (dans l'esprit) car il vise à couvrir les répétitions de certaines paires de transitions. Dans le cas de ce critère, les paires considérées sont celles composées d'une transition qui ouvre la portée et d'une transition qui la ferme. Ce critère vise donc à s'assurer qu'il est possible d'ouvrir et de fermer plusieurs fois ces portées en choisissant k comme étant le nombre d'itérations souhaité. Le plus souvent, dans la pratique, nous choisirons $k = 2$.

Définition 31 (Critère *k-scope*)**.** Soient $\mathcal{A}$ un automate de propriété, k le nombre d'activation nécessaires pour la portée. Soient T_{op} et T_{cl} deux ensembles représentant respectivement l'ensemble des α-transitions ouvrant (resp. fermant) la portée de $\mathcal{A}$. Soit C_s^k (k-scope) l'ensemble de séquences de paires à couvrir dans la portée. Il est défini par :

$$\begin{aligned} C_s^1 &= T_{op} \times T_{cl} \\ C_s^k &= (\bigcup_{q \in C_s^1} \bigcup_{s \in C_s^{k-1}} q \cdot s) \cup C_s^{k-1} \end{aligned}$$

où $q \cdot s$ dénote la concaténation de la paire q avec la séquence de paire s.

Soit Ts une suite de tests. Le critère est satisfait s'il existe au moins un test de Ts qui couvre au moins k paires de transitions, pas nécessairement distinctes, appartenant à C_s^k. Le taux de couverture associé T_s^k est défini ainsi :

$$\mathrm{T}_s^k = \frac{\text{nombre de séquences de } C_s^k \text{ couvertes par Ts}}{\text{nombre total de séquences de } C_s^k}$$

Cette définition fait apparaître que la satisfaction du critère pour *k-scope* implique nécessairement la couverture du critère *(k-1)-scope*.

Dans notre exemple, pour $k = 2$, C_s^2 est défini ainsi :

$$\begin{aligned} \mathrm{C}_s^2 = \{&[(1 \xrightarrow{A} 2,2 \xrightarrow{B} 4)],[(1 \xrightarrow{A} 2,3 \xrightarrow{B} 4)],\ [(4 \xrightarrow{A} 2,2 \xrightarrow{B} 4)],[(4 \xrightarrow{A} 2,3 \xrightarrow{B} 4)], \\ &[(1 \xrightarrow{A} 2,2 \xrightarrow{B} 4)\,;\,(1 \xrightarrow{A} 2,2 \xrightarrow{B} 4)],\ [(1 \xrightarrow{A} 2,2 \xrightarrow{B} 4)\,;\,(1 \xrightarrow{A} 2,3 \xrightarrow{B} 4)], \\ &[(1 \xrightarrow{A} 2,2 \xrightarrow{B} 4)\,;\,(4 \xrightarrow{A} 2,2 \xrightarrow{B} 4)],\ [(1 \xrightarrow{A} 2,2 \xrightarrow{B} 4)\,;\,(4 \xrightarrow{A} 2,3 \xrightarrow{B} 4)], \\ &[(1 \xrightarrow{A} 2,3 \xrightarrow{B} 4)\,;\,(1 \xrightarrow{A} 2,2 \xrightarrow{B} 4)],\ [(1 \xrightarrow{A} 2,3 \xrightarrow{B} 4)\,;\,(1 \xrightarrow{A} 2,3 \xrightarrow{B} 4)], \\ &[(1 \xrightarrow{A} 2,3 \xrightarrow{B} 4)\,;\,(4 \xrightarrow{A} 2,2 \xrightarrow{B} 4)],\ [(1 \xrightarrow{A} 2,3 \xrightarrow{B} 4)\,;\,(4 \xrightarrow{A} 2,3 \xrightarrow{B} 4)], \\ &[(4 \xrightarrow{A} 2,2 \xrightarrow{B} 4)\,;\,(1 \xrightarrow{A} 2,2 \xrightarrow{B} 4)],\ [(4 \xrightarrow{A} 2,2 \xrightarrow{B} 4)\,;\,(1 \xrightarrow{A} 2,3 \xrightarrow{B} 4)], \\ &[(4 \xrightarrow{A} 2,2 \xrightarrow{B} 4)\,;\,(4 \xrightarrow{A} 2,2 \xrightarrow{B} 4)],\ [(4 \xrightarrow{A} 2,2 \xrightarrow{B} 4)\,;\,(4 \xrightarrow{A} 2,3 \xrightarrow{B} 4)], \\ &[(4 \xrightarrow{A} 2,3 \xrightarrow{B} 4)\,;\,(1 \xrightarrow{A} 2,2 \xrightarrow{B} 4)],\ [(4 \xrightarrow{A} 2,3 \xrightarrow{B} 4)\,;\,(1 \xrightarrow{A} 2,3 \xrightarrow{B} 4)], \\ &[(4 \xrightarrow{A} 2,3 \xrightarrow{B} 4)\,;\,(4 \xrightarrow{A} 2,2 \xrightarrow{B} 4)],\ [(4 \xrightarrow{A} 2,3 \xrightarrow{B} 4)\,;\,(4 \xrightarrow{A} 2,3 \xrightarrow{B} 4)]\} \end{aligned}$$

où $[p_1 ; \ldots ; p_n]$ dénote la séquence de paires de transitions $p_1, \ldots, p_n$.

Les quatre premiers cas (première ligne) correspondent à C_s^1, des séquences d'une paire de transitions. Les lignes suivantes sont les cas spécifiques à $k = 2$. Il s'agit du produit cartésien de toutes les paires permettant d'entrer/sortir de la portée.

Ainsi, la couverture complète de ce critère nécessite la couverture de ces vingt séquences.

6.3 Discussion sur les critères

Dans cette section, nous justifions dans un premier temps la hiérarchisation de nos critères vis-à-vis des critères classiques sur les automates, puis nous abordons l'applicabilité de nos critères sur nos automates.

6.3.1 Hiérarchisation des critères

Nous pouvons établir une hiérarchie de nos critères vis-à-vis de critères de couverture classiques sur les automates. Cette hiérarchie est présentée dans la FIGURE 6.2 où nos critères sont encadrés. Une flèche A→B signifie que si le critère A est couvert, alors le critère B l'est aussi. Les relations en pointillés ne sont possibles que sous certaines conditions. De plus, nous ne considérons pas le critère *all-configuration*. En effet, puisque nous considérons à chaque fois un seul automate, le critère *all-configurations* est identique au critère *all-states*. Nous justifions maintenant la place de chacun de nos critères en montrant successivement leur relation avec les critères classiques.

FIGURE 6.2 – Hiérarchie des nouveaux critères vis-à-vis des critères classiques

Critère *all-*α

Nous plaçons ce critère en dessous du critère classique *all-transitions* et en relation avec *all-states* (sous condition) et *all-*α*-pairs*. Nous montrons maintenant ces trois relations.

all-transitions → all-α. Cette relation se vérifie simplement par construction des éléments à couvrir pour chaque critère. En effet, dans *all-*α, nous couvrons uniquement les α-transitions activables qui forment un sous-ensemble de l'ensemble des transitions de l'automate, considéré dans *all-transitions*. En couvrant toutes les transitions, nous couvrons nécessairement toutes les α-transitions, ainsi nous avons bien *all-transitions* → *all-*α.

FIGURE 6.3 – Automate de A `follows` B `between` C `and` D

FIGURE 6.4 – Exemple de cas spécial pour *all-*α → *all-*α*-pairs*

all-α $\dashrightarrow$ all-states. Pour cette relation, nous distinguons deux cas : les automates avec et sans état d'erreur. Si nous considérons le cas des automates avec un état d'erreur tel celui de la FIGURE 6.3, nous avons *all-α* $\nrightarrow$ *all-states*. En effet, puisque *all-α* ne considère que les α-transitions activables alors l'état d'erreur n'est jamais atteint.

Dans le second cas, où l'automate ne dispose pas d'état d'erreur, alors nous avons *all-α* $\rightarrow$ *all-states*. En effet, les seuls cas où cette relation ne tient pas, est lorsqu'il existe un état de l'automate qui n'est atteignable que par des $\Sigma-$transitions et où toutes les transitions sortantes sont aussi des $\Sigma-$transitions. Cependant, ce cas n'arrive jamais dans nos automates de propriété (il existe toujours une α-transition permettant d'entrer ou de sortir d'un état quelconque de l'automate). Ainsi, nous avons bien *all-α* $\rightarrow$ *all-states*.

all-α-pairs $\rightarrow$ all-α. Par construction, puisque $C_{\alpha^2} = C_\alpha \times C_\alpha$ où C_{α^2} est l'ensemble des paires à couvrir pour *all-α-pairs* et C_α l'ensemble des $\alpha-$transitions à couvrir, alors nous avons bien *all-α-pairs* $\rightarrow$ *all-α*. Les seuls cas où cette relation n'existerait pas est celui où une $\alpha-$transition ne partage pas son état source ni son état destination avec aucune autre $\alpha-$transition tel l'automate montré dans la FIGURE 6.4. Ainsi, à cause de la consécutivité des $\alpha-$transitions d'une paire, les transitions $1\xrightarrow{A}$B et $3\xrightarrow{C}4$ n'apparaitront dans aucune paire du critère *all-α-pairs*. Mais, ce cas n'arrive jamais avec nos automates.

Critère *all-α-pairs*

Nous plaçons ce critère en dessous du critère classique *all-pairs*, en relation avec *all-α* mais distinct de *all-transitions*. La relation avec *all-α* a déjà été montrée précédemment, nous montrons donc les deux autres relations.

all-pairs $\rightarrow$ all-α-pairs. Cette relation est similaire à celle qui lie *all-α* à *all-transitions*. Par construction, les éléments à couvrir pour *all-pairs* est $T \times T$ où T est l'ensemble des transitions de l'automate, et ceux à couvrir pour *all-α-pairs* sont $C_\alpha \times C_\alpha$ où C_α est l'ensemble des $\alpha-$transitions activables de l'automate. Or, $C_\alpha \subseteq T$, donc nous avons $C_\alpha \times C_\alpha \subseteq T \times T$. Donc nous avons bien *all-pairs* $\rightarrow$ *all-α-pairs*.

all-transitions $\nrightarrow$ all-α-pairs. Cette absence de relation est illustrée avec l'exemple de la FIGURE 6.3. Le chemin $[1\xrightarrow{A}2\,;2\xrightarrow{D\setminus\{B\}}4\,;4\xrightarrow{C\setminus\{B\}}2\,;2\xrightarrow{B}3\,;3\xrightarrow{A}2]$ couvre bien toutes les $\alpha-$transitions considérées dans *all-transitions* mais ne permet pas de couvrir la paire d'$\alpha-$transitions $(3\xrightarrow{A}2,2\xrightarrow{B}3)$. Ainsi, nous avons bien *all-transitions* $\nrightarrow$ *all-α-pairs*.

all-α-pairs $\nrightarrow$ all-transitions. Cette absence de relation est simple à vérifier. En effet, *all-α-pairs* ne considère pas les $\Sigma-$transitions, il est donc possible de couvrir toutes les paires d'$\alpha-$ transitions sans couvrir les $\Sigma-$transitions de l'automate, qui, elles, sont couvertes par *all-transitions*. Ainsi, nous avons bien *all-α-pairs* $\nrightarrow$ *all-transitions*.

Critère *k-scope*

Ce critère est indépendant des critères *all-one-loop* et *all-states* mais est en relation avec les critères *all-roundtrip* et *all-α-pairs* sous certaines conditions. Nous montrons

maintenant ces relations.

k-scope $\rightarrow$ all-states. Si nous reprenons l'exemple d'automate de la FIGURE 6.3, les chemins de la forme $[1\xrightarrow{C}2\,;2\xrightarrow{D}3\,;(3\xrightarrow{C}2\,;2\xrightarrow{D}3)^{k-1}]$ ne permettent pas d'atteindre l'état 4. Ainsi, nous avons bien *k-scope* $\nrightarrow$ *all-pairs*.

all-states $\dashrightarrow$ k-scope. Pour cette relation, nous distinguons le cas où $k = 1$ et $k > 1$. Les automates concernés par le critère *k-scope* possèdent tous une portée `between` ou `after until`. Or, dans ces portées, toutes les transitions permettant de sortir de la portée amènent au même état, qui n'est accessible que par ces transitions. Dans notre exemple d'automate de la FIGURE 6.3, il s'agit de l'état 3, accessible uniquement par des transitions portant l'évènement de sortie D. De plus, pour pouvoir effectuer ces transitions de sortie, nous avons nécessairement eu des transitions d'entrée de portée auparavant (sémantique de ces portées). Ainsi, la couverture de tous les états passe nécessairement par la couverture d'une transition d'entrée de portée puis d'une transition de sortie. Dans notre exemple en FIGURE 6.3, pour satisfaire le critère *all-states*, nous devons passer par la transition $2\xrightarrow{D}3$ pour atteindre l'état 3, et nous sommes nécessairement passé par $1\xrightarrow{C}2$. Or la couverture de cette paire de transition est une itération du critère *k-scope*. Ainsi, pour $k = 1$, nous avons bien *all-states* $\rightarrow$ *k-scope*.

Par contre, dans le cas où $k > 1$, nous n'avons plus la garantie qu'un chemin permettant de couvrir tous les états puisse aussi couvrir les itérations suivantes. Ainsi, dans ce cas, nous avons *all-states* $\nrightarrow$ *k-scope*.

all-α-pairs $\dashrightarrow$ k-scope. Nous distinguons les cas où $k = 1$ et $k > 1$. Puisque la couverture d'une itération avec $k = 1$ se résume à la couverture d'une paire d'$\alpha-$transition particulière, alors la couverture de toutes les paires avec le critère *all-α-pairs* implique la couverture des paires de *k-scope* qui en est un sous-ensemble. Dans ce cas, nous avons *all-α-pairs* $\rightarrow$ *k-scope*.

Par contre, dans le cas où $k > 1$, nous n'avons plus la garantie qu'un chemin permettant de couvrir la première paire de l'itération puisse aussi couvrir les paires suivantes. Ainsi, dans ce cas, nous avons *all-α-pairs* $\nrightarrow$ *k-scope*.

k-scope $\nrightarrow$ all-pairs. Cette absence de relation se vérifie avec l'exemple d'automate de la FIGURE 6.3. En effet, les chemins de la forme $[1\xrightarrow{C}2\,;2\xrightarrow{D}3\,;(3\xrightarrow{C}2\,;2\xrightarrow{D}3)^{k-1}]$ permettent de satisfaire le critère *k-scope*. Cependant, ces chemins ne permettent pas de couvrir la paire de transitions $2\xrightarrow{B\setminus\{D\}}4$ et $4\xrightarrow{A\setminus\{B\}}2$, considérée dans *all-pairs*. Ainsi, nous avons bien *k-scope* $\nrightarrow$ *all-pairs*.

all-pairs $\nrightarrow$ k-scope. Ce cas est simple, car nous n'avons pas l'assurance que la couverture de toutes les paires de transitions de nos automates puisse couvrir les k itérations du critère *k-scope*. Ces critères sont sans rapport, ainsi nous avons *all-pairs* $\nrightarrow$ *k-scope*.

FIGURE 6.5 – Automate du motif E `directly precedes` F

all-roundtrip ⇢ k-scope. Nous distinguons les deux cas $k = 1$ et $k > 1$. Le critère *all-roundtrip* nécessite la couverture d'au plus une boucle dans l'automate sans considérer tous les chemins qui permettent d'y arriver. Dans le cas où $k = 1$, cette itération nous permet d'entrer dans la portée puis d'en sortir. Ainsi, dans le plus petit cas nous avons une boucle qui nous ramène à notre point de départ, sinon, nous n'avons qu'une séquence qui est acceptée par *all-roundtrip*. Ainsi, si $k = 1$, nous avons bien *all-roundtrip* $\rightarrow$ *k-scope*.

Le cas $k > 1$ est plus simple, car chaque itération nous amène sur l'état initial du motif (donc une première boucle) et chaque transition sortant de la portée nous amène dans l'état final de la portée. Ainsi, nous avons atteint au moins deux états à deux reprises. Les chemins issus de ces itérations forment alors plusieurs boucles et ne correspondent pas au critère *all-roundtrip*. Ainsi, si $k > 1$, alors nous avons *all-roundtrip* $\not\rightarrow$ *k-scope*.

k-scope ↛ all-one-loop. Nous considérons cette relation pour $k > 1$, le cas où $k = 1$ est déjà établi par la relation *all-roundtrip* ⇢ *k-scope*. L'absence de relation se montre simplement de la manière suivante. Puisque le critère *k-scope* permet de couvrir plusieurs boucles, il est plus fort que le critère *all-one-loop*. Cependant, il est aussi plus faible dans le sens où il ne permet de couvrir que certaines boucles de l'automate et ne nécessite pas tous les préfixes de ces boucles. Ainsi, ces deux critères sont sans rapport et nous avons bien *k-scope* $\not\rightarrow$ *all-one-loop*.

Critère *k-pattern*

Similairement à *k-scope*, ce critère est indépendant des critères *all-one-loop* et *all-states*, mais il est en relation avec *all-α-pairs* et *all-roundtrip* dans le cas où $k = 1$ ou $k < 3$ dans le cas du motif `directly precedes`.

all-states ↛ k-pattern. Si nous reprenons l'exemple d'automate de la FIGURE 6.3, les chemins de la forme $[1 \xrightarrow{C} 2\,;(2 \xrightarrow{B\setminus\{D\}} 4\,;4 \xrightarrow{A\setminus\{D\}} 2)^k]$ ne permettent pas d'atteindre l'état 3. Ainsi, nous avons bien *k-pattern* $\not\rightarrow$ *all-pairs*.

k-pattern ↛ all-states. En reprenant l'exemple d'automate de la FIGURE 6.3, même en ne considérant pas l'état d'erreur, le chemin $1 \xrightarrow{C} 2\,;2 \xrightarrow{D} 3\,;3 \xrightarrow{C} 2\,;2 \xrightarrow{B\setminus\{D\}} 4\,;4 \xrightarrow{A\setminus\{D\}} 2$ permet de couvrir tous les états, mais ne couvre aucune itération du critère *k-pattern*. Ainsi, nous avons bien *all-states* $\not\rightarrow$ *k-pattern*.

all-α-pairs ⇢ k-pattern. Nous distinguons les cas où $k = 1$ et $k > 1$. Puisque la couverture d'une itération avec $k = 1$ se résume à la couverture d'une paire d'$\alpha-$transition, alors la couverture de toutes les paires avec le critère *all-α-pairs* implique la couverture des paires de *k-pattern* qui en est un sous-ensemble. Dans ce cas, nous avons *all-α-pairs* $\rightarrow$ *k-pattern*.

Par contre, dans le cas où $k > 1$, nous n'avons plus la garantie qu'un chemin permettant de couvrir la première paire de l'itération puisse aussi couvrir les paires suivantes. Ainsi, dans ce cas, nous avons *all-α-pairs* $\nrightarrow$ *k-pattern*.

k-pattern $\nrightarrow$ all-pairs. Si nous reprenons l'exemple d'automate de la FIGURE 6.3, les chemins de la forme $[1\xrightarrow{C}2\,;(2\xrightarrow{B\setminus\{D\}}4\,;4\xrightarrow{A\setminus\{D\}}2)^k]$ ne permettent pas d'atteindre l'état 4 et ne traversent donc pas les transitions $2\xrightarrow{D}3$ et $3\xrightarrow{C}2$. Ainsi, nous avons bien *k-pattern* $\nrightarrow$ *all-pairs*.

all-roundtrip $\dashrightarrow$ k-pattern. Pour cette relation, nous distinguons les cas $k = 1$, $k > 1$ et le cas du motif `directly precedes`. Le critère *all-roundtrip* nécessite la couverture de toutes les boucles, sans couvrir tous leurs préfixes possibles (un seul suffit). Dans le cas où $k = 1$, cette itération nous ramène à l'état initial du motif (on en sort avec le premier évènement, on y revient avec le second). Ainsi, nous avons au mieux une boucle sur l'automate.

Le cas $k > 1$ est plus simple à traiter. En effet, une itération d'un motif ou d'une portée nous ramène dans l'état initial du motif pour la répétition. Ainsi, chaque itération supplémentaire, nous ramène dans cet état, créant autant de boucle dans le chemin permettant de toutes les couvrir. Ainsi, pour $k > 1$ nous avons *all-roundtrip* $\nrightarrow$ *k-pattern*.

Le cas particulier du motif `directly precedes` provient de la forme de l'automate, rappelé à la FIGURE 6.5. Il est possible d'itérer deux fois avant de boucler une première fois. Donc le critère *all-roundtrip* est satisfait sur ce motif tant que $k < 3$. Pour les itérations suivantes, nous obtenons bien plus de boucles Ainsi, dans le cas de ce motif, nous avons *all-roundtrip* $\rightarrow$ *k-pattern* si $k < 3$, et *all-roundtrip* $\nrightarrow$ *k-pattern* si $k \geq 3$.

k-pattern $\nrightarrow$ all-one-loop. Cette relation se montre simplement car le critère *all-one-loop* permet de couvrir toutes les boucles au plus une fois avec tous les préfixes, mais le critère *k-pattern* permet de couvrir plus d'une boucle et ne se concentre que sur certaines boucles. Ainsi, ces critères sont sans rapport, d'où *k-pattern* $\nrightarrow$ *all-one-loop*.

6.3.2 Applicabilité des critères

Les critères de couverture *all-α* et $all\text{-}\alpha - pairs$ sont décrits de manière générale pour n'importe quel automate, à condition de retrouver l'information sur les $\alpha-$transitions des motifs. Par contre, les critères *k-scope* et *k-pattern* ne sont applicables que pour certains motifs et certaines portées. Notons que cette information n'a pas été formalisée dans les automates présentés au chapitre 5, mais l'implémentation dispose des moyens pour retrouver cette information à l'aide d'un attribut qui permet de la conserver. Suivant les combinaisons de portées/motifs de l'automate de propriété considéré, il est possible que certains critères ne soient pas applicables à ces combinaisons. La FIGURE 6.6 identifie les critères applicables en fonction des combinaisons de portée et de motif d'une propriété.

Ce tableau montre clairement que certaines combinaisons peuvent être évaluées selon tous les critères. C'est le cas notamment des combinaisons utilisant `between` et `after`

Combinaisons		α	**paires d'α**	**k-portée**	**k-motif**
Portée	Motif				
Globally	Never				
	Always	✓			✓
	(Directly) Follows	✓	✓		✓
	(Directly) Precedes	✓	✓		✓
	Eventually n times	✓*	✓**		
	Eventually at least n times	✓	✓		
	Eventually at most n times	✓*	✓**		
After	Never	✓			
	Always	✓	✓		✓
	(Directly) Follows	✓	✓		✓
	(Directly) Precedes	✓	✓		✓
	Eventually n times	✓	✓*		
	Eventually at least n times	✓	✓		
	Eventually at most n times	✓	✓*		
Before	Never	✓			
	Always	✓	✓		✓
	(Directly) Follows	✓	✓		✓
	(Directly) Precedes	✓	✓		✓
	Eventually n times	✓	✓*		
	Eventually at least n times	✓	✓		
	Eventually at most n times	✓	✓*		
Between (last)	Never	✓	✓	✓	
	Always	✓	✓	✓	
	(Directly) Follows	✓	✓	✓	✓
	(Directly) Precedes	✓	✓	✓	✓
	Eventually n times	✓	✓	✓	
	Eventually at least n times	✓	✓	✓	
	Eventually at most n times	✓	✓	✓	
After (last)	Never	✓	✓	✓	
	Always	✓	✓	✓	
	(Directly) Follows	✓	✓	✓	✓
	(Directly) Precedes	✓	✓	✓	✓
	Eventually n times	✓	✓	✓	
	Eventually at least n times	✓	✓	✓	
	Eventually at most n times	✓	✓	✓	

* : seulement si $n > 0$, ** : seulement si $n > 1$

FIGURE 6.6 – Utilisations possibles des critères par combinaison portée/motif

`until`. A l'inverse, d'autres ne peuvent être évaluées qu'avec quelques critères, voire aucun, notamment la combinaison `globally/never`, dont l'automate est donné à la FIGURE 6.7.

Un autre aspect de l'applicabilité des critères est le choix des valeurs des critères *k-scope* et *k-pattern*. En effet, ces critères ont pour objectif de s'assurer de la répétitivité des

FIGURE 6.7 – Automate de la propriété `never` A `globally`

portées/motifs de la propriété. Ainsi, une valeur k=1 ne suffit pas car elle ne permet que d'illustrer l'entrée puis la sortie des portées/motifs sans retour. Une valeur k=2 permet d'illustrer une entrée après être sorti et permet d'illustrer une répétition, ce que nous cherchons à illustrer avec ces critères. Une valeur de k supérieure à 2 permet de couvrir plusieurs itérations, mais nous pouvons considérer que ces itérations supplémentaires sont superflues (mais pas nécessairement inutiles) pour montrer la répétitivité des portées et des motifs considérés. De plus, lorsque nous avons montré les relations entre ces critères et les critères classiques, nous nous sommes aperçu que l'application des critères pour $k = 1$ était plus faible que des critères classiques (*all-states*/*all-roundtrip* pour *k-scope*, et *all-*α*-pairs*/*all-roundtrip* pour *k-pattern*). Ainsi, nous choisissons une valeur de k au moins égale à 2.

Enfin, un dernier aspect sur l'application des critères concerne la faisabilité des chemins permettant de les couvrir. Certains chemins demandés par les critères peuvent être infaisables à cause du modèle. Par exemple, en considérant un motif A `follows` B, il se peut que la répétition des événement B et A ne puisse se faire qu'un certain nombre de fois. Dans ce cas, avec le critère `k-pattern` et une valeur de k trop grande, certaines séquences du critère ne seront pas nécessairement couvertes. Cette remarque est valable pour les autres critères.

6.4 Synthèse

Nous avons défini dans ce chapitre de nouveaux critères de couverture, inspirés des critères classiques sur les automates que nous avons adaptés aux automates de propriété combinant un motif et une portée. Ces critères s'appuient plus particulièrement sur les éléments qui définissent les propriétés : les transitions qui portent les évènements de la propriété pour les deux premiers critères, et les transitions associées au motif et à la portée pour les deux derniers. Un taux de couverture est associé à chaque critère afin de quantifier la couverture des automates par une suite de tests donnée, pour un critère donné.

Nous avons aussi observé que certaines combinaisons de portées et motifs ne permettent pas l'application de beaucoup de critères. C'est notamment le cas de la combinaison `never` A `globally` qui ne peut être couverte par aucun critère. Cet automate n'exhibe aucune $\alpha-$transition susceptible d'être couverte par les critères définis précédemment. La seule transition potentiellement intéressante est celle qui amène à l'état d'erreur. Le Chapitre 7 adresse ce problème par la définition d'un critère spécifique qui vise à provoquer ces transitions particulières.

Chapitre 7

Critère de robustesse

Sommaire

7.1 Définitions préliminaires 126
7.2 Opérateurs de mutation 128
7.2.1 Suppression de précondition 128
7.2.2 Suppression de tags 129
7.2.3 Suppression de proposition atomique 130
7.2.4 Echange de tags 131
7.2.5 Mutations dans l'exemple de propriété 132
7.3 Critère de robustesse 133
7.3.1 Mutation de l'automate 134
7.3.2 Critère de robustesse 136
7.4 Synthèse 137

Le chapitre précédent montre une utilisation des automates pour la mesure de couverture des propriétés vis-à-vis de critères qui leurs sont spécifiques. Cependant, ces critères ne permettent d'illustrer que des exécutions nominales du système. De plus, nous avons pointé le manque d'applicabilité de nos critères pour certaines combinaisons telles que `globally/never`. Sur cette combinaison particulière, il n'existe qu'un seul élément réellement exploitable sur l'automate et qui n'est pas pris en compte par les autres critères : l'ensemble des transitions d'erreur de la propriété.

Pour illustrer cela, considérons la propriété suivante :

```
never isCalled(sut.buyTicket,
      pre:"self.all_listed_movies->forAll(available_tickets = 0)"
      on "sut",
      including:{@AIM:BUY_Nominal_Success, @AIM:BUY_Last_Ticket_Sold})
globally
```

avec sa représentation en automate montrée à la FIGURE 7.1 où A est l'évènement `isCalled` de la propriété. Cette propriété représente le fait qu'il est impossible d'avoir un achat de ticket avec succès s'il ne reste aucun ticket disponible, quel que soit le film. Cette propriété ne peut être couverte par aucun des critères précédents.

FIGURE 7.1 – Automate de l'exemple `never` A `globally`

Dans le chapitre précédent, nous avons fait une distinction entre les transitions activables de l'automate, considérées dans les critères, et les transitions qui conduisent à l'état d'erreur. En effet, nous considérons que le modèle respecte la propriété et, par conséquent, que tous les chemins qui tenteraient de traverser les transitions d'erreur ne pourraient pas être animées sur le modèle.

Puisque ces évènements particuliers sont interdits, l'idée est de tenter de le provoquer avec des évènements activables sur le modèle ayant une "proximité" avec les transitions d'erreur de la propriété. Ainsi que se passe-t-il si nous essayons d'appeler *buyTicket*, avec la précondition de l'évènement, mais sans forcer le comportement ? Puisque le modèle respecte la propriété, alors nous devrions recevoir un message d'erreur nous indiquant que l'achat est impossible car il ne reste plus de tickets, qu'aucun utilisateur n'est authentifié ou encore que l'utilisateur n'est pas sur la bonne page, i.e. les comportements complémentaires de @AIM:BUY_Nominal_Sucess dans *buyTicket*. Le comportement activé dépend du contexte dans lequel l'opération *buyTicket* est appelée.

Afin de couvrir spécifiquement cet aspect d'exécution dangereuse d'un système, qui relève de la robustesse du système vis-à-vis de la propriété, nous proposons de définir ces évènements *proches* par le biais de règles de mutation sur l'évènement fautif original. Cependant, nous ne considérons pas toutes les transitions d'erreur, mais uniquement les premières transitions d'erreur des chemins fautifs de l'automate. En effet, puisque le modèle respecte la propriété, il n'est pas possible d'atteindre les transitions suivantes.

Dans un premier temps, nous définissons dans la section 7.1 les nouvelles notions utilisées pour la mutation des évènements. Ensuite, nous décrivons dans la section 7.2 les différents opérateurs de mutation. Enfin, nous présentons dans la section 7.3 le critère de couverture associé à ces mutations.

7.1 Définitions préliminaires

Nous donnons dans un premier temps les définitions préalables ainsi que les étapes de préparation nécessaires à certaines mutations.

Un *prédicat* OCL est une expression OCL qui utilise les opérateurs logiques de conjonction, de disjonction (inclusive et exclusive) et de négation entre des *atomes* du langage. Les *atomes* ou *propositions atomiques* sont toutes les autres constructions de type booléen du langage OCL considéré (voir FIGURE 3.9 du chapitre 3).

Les règles de mutations sont définies par rapport aux évènements mis sous une certaine forme, appellée *forme conjonctive*. Des évènements dans cette forme sont des *évènements conjonctifs*.

Définition 32 (Evénement sous forme conjonctive)**.** Un évènement $[op, pre, post, tags]$ est sous *forme conjonctive* si et seulement si pre est une conjonction de propositions atomiques p_k, $pre = p_1 \wedge \ldots \wedge p_n$.

Bien que certaines mutations requièrent des évènements sous forme conjonctive, tous les évènements ne sont pas forcément sous cette forme, et certains possèdent une précondition contenant des disjonctions. Dans un premier temps, nous réécrivons le code OCL de la précondition afin de ne conserver que les opérateurs de conjonction, de disjonction, et de négation. Le code OCL permet l'utilisation de la disjonction exclusive. Nous pouvons éliminer cette construction par la règle suivante :

$$A \oplus B \rightsquigarrow (A \wedge \neg B) \vee (\neg A \wedge B)$$

Les disjonctions sont ensuite éliminées en passant la précondition sous sa forme normale disjonctive (DNF), ce qui a pour effet d'obtenir une disjonction généralisée de conjonctions de propositions atomiques, appellées *clauses*.

Chaque clause de la disjonction est un état différent du système à partir duquel l'évènement peut être activé. Ainsi, l'évènement original peut être vu comme un ensemble d'évènements, chacun possédant sa propre précondition qui est alors une des clauses de la DNF. Par exemple, l'évènement $[op, P \vee P', Q, T]$ est décomposé dans les deux évènements $[op, P, Q, T]$ et $[op, P', Q, T]$, avec P et P' des clauses. L'ensemble des évènements issus de cette décomposition de l'évènement original (disjonctions et liste de tags) est appelé la *forme normale* d'un évènement.

Définition 33 (Forme normale d'un évènement)**.** Soit un évènement $[op, P, Q, T]$ où P est en DNF $P = p_1 \vee \ldots \vee p_m$ avec $p_1 \ldots p_m$ des conjonctions de propositions atomiques. La forme normale de l'évènement est définie par l'ensemble d'évènements sous forme conjonctive défini ainsi :

$$\mathrm{DNF}([op, P, Q, T]) = \bigcup_{i \in 1..m} [op, p_i, Q, T]$$

Il est à noter que cette décomposition est unique pour chaque évènement.

Bien que potentiellement sujette à une explosion combinatoire sur la décomposition des disjonctions, cette opération est acceptable grâce au fait que, en pratique, les préconditions et les postconditions contiennent généralement peu de disjonctions, ce qui permet de limiter ce risque.

Exemple 36 (Exemple de forme normale d'un évènement)**.** Afin d'illustrer les notions introduites dans ce chapitre nous utilisons l'exemple d'évènement fautif suivant :

E_0 = [buyTicket, "current_user.oclIsUndefined() and (in_title=TITLE1 or in_title=TITLE2)" `on` "sut", _, {@AIM :BUY_Nominal_Success, @AIM :BUY_Last_Ticket_Sold}]

En effet, quel que soit le titre du film (TITLE1 ou TITLE2), il est impossible d'activer les comportements avec succès d'achat de ticket s'il n'y a aucun utilisateur connecté.

Dans un premier temps, nous décomposons la disjonction de la précondition en la mettant sous sa forme normale disjonctive. Ainsi, la précondition devient :

(current_user.oclIsUndefined() and in_title=TITLE1) or
(current_user.oclIsUndefined() and in_title=TITLE2)

Cette décomposition crée les deux nouveaux évènements E_1 et E_2 suivants :

E_1 = [buyTicket, "current_user.oclIsUndefined() and in_title=TITLE1" **on** "sut", _, {@AIM :BUY_Nominal_Success, @AIM :BUY_Last_Ticket_Sold}]
E_2 = [buyTicket, "current_user.oclIsUndefined() and in_title=TITLE2" **on** "sut", _, {@AIM :BUY_Nominal_Success, @AIM :BUY_Last_Ticket_Sold}]

Le premier évènement représente l'activation de *buyTicket* sans utilisateur connecté au système, avec TITLE1 en paramètre. Le second évènement est similaire, mais en utilisant TITLE2 en paramètre.

Ces deux évènements sont chacun dans une forme conjonctive. Ainsi, la forme normale de l'évènement d'origine E est la combinaison de ces évènements conjonctifs $E_0 = \{E_1, E_2\}$.

7.2 Opérateurs de mutation

Afin de décrire les évènements *proches* des évènements fautifs considérés, nous fondons notre approche sur des opérateurs de mutations, spécifiques à nos évènements, qui nous permettent de les affaiblir.

Cependant, nous ne pouvons pas muter n'importe quel composante d'un évènement. En effet, certaines composante sont considérées comme *contrôlables* et d'autres, *observables*. Les composantes *contrôlables* nous permettent de définir ce que nous pouvons utiliser comme contrainte pour un évènement. Typiquement, le nom d'opération (et l'instance à partir de laquelle elle est appelée) est considéré comme une composante contrôlable. Il en va de même pour la précondition de l'évènement ou encore l'ensemble des tags. Les composantes *observables* sont ceux que nous pouvons observer à l'issue de l'évènement. C'est typiquement le cas de la postcondition de l'évènement ou de l'ensemble des tags.

Ainsi, nous pouvons considérer l'ensemble des tags comme un composant observable, car nous pouvons déterminer le comportement activé après l'exécution, mais aussi comme une composante contrôlable, car nous pouvons forcer l'activation d'un comportement.

Dans les règles de mutation, les composantes contrôlables seront mutés alors que les composantes observables seront retirés et nous observerons leur valeur après l'exécution de l'évènement. Nous présentons maintenant ces opérateurs de mutation sur des évènements.

7.2.1 Suppression de précondition

Cette mutation affaiblit l'évènement en supprimant sa précondition. Cependant, cette suppression ne donne aucune garantie sur la possibilité d'activer les tags présents, ni sur la satisfiabilité de la postcondition sur l'état suivant. Ainsi, en supprimant la précondition, nous supprimons également la postcondition et la liste des tags de l'évènement.

Définition 34 (Suppression de précondition)**.** Soit l'évènement E=[op, P, Q, T], la mutation de cet évènement est définie ainsi :

$$\mu_{pre^-}(E) = \{[op, \_, \_, \_]\}.$$

Cette règle de mutation génère exactement un évènement.

L'évènement muté est nécessairement activable dans la propriété et sur le modèle. En effet, puisque nous supposons que le modèle est défensif, alors pour chaque opération du modèle, il existe un comportement activable quel que soit l'état à partir duquel l'opération est appelée.

Il est à noter que cette mutation est la plus générale. En effet, toutes les composantes de l'évènement sont retirés, en ne laissant plus que l'opération.

Exemple 37 (Exemple de suppression de précondition)**.** Reprenons l'exemple d'évènement fautif E_0 suivant :

[buyTicket, "current_user.oclIsUndefined() and (in_title=TITLE1 or in_title=TITLE2)" on "sut", _, {@AIM :BUY_Nominal_Success, @AIM :BUY_Last_Ticket_Sold}]

L'évènement muté E est alors simplement décrit par [buyTicket, _, _, _]. Bien que cet évènement englobe l'évènement dont il est issu, le modèle n'autorise pas l'activation de l'évènement fautif. A la place, cet évènement muté pourra correspondre, par exemple, à l'appel de *buyTicket* par un utilisateur authentifié pour le film TITLE1 avec le comportement @AIM :BUY_Nominal_Success.

7.2.2 Suppression de tags

Cette mutation permet de supprimer l'ensemble de tags de l'évènement. Similairement à la mutation précédente, la postcondition a été enlevée. En effet, l'activation d'un tag différent que celui spécifié par l'évènement original ne nous permet pas de garantir la satisfaction de la postcondition, qui est peut être directement liée à l'effet du comportement. Ainsi, dans ce cas, nous ne pourrons observer que la postcondition associée aux comportements qui seront activés.

Définition 35 (Suppression de tags)**.** Soit l'évènement E=[op, P, Q, T], la mutation de cet évènement est définie ainsi :

$$\mu_{tag^-}(E) = \{[op, P, \_, \_]\}.$$

Cette règle de mutation génère exactement un évènement.

Similairement à la mutation précédente, la garantie d'activation sur le modèle provient du fait que le modèle est défensif. Pour un état du système donné (représenté dans l'évènement par la précondition), il existe nécessairement un comportement activable sur le modèle.

Exemple 38 (Exemple de suppression de tags)**.** Reprenons l'évènement E_0 comme exemple. Cette mutation génère l'évènement suivant :

μ_{tag^-}(E) = [buyTicket, "current_user.oclIsUndefined() and (in_title=TITLE1 or in_title=TITLE2)" on "sut", _, _]

Dans ce cas, le seul comportement possible à partir de la précondition est celui étiqueté par @AIM :BUY_Login_Mandatory.

```
1  --@REQ:BASKET_MNGT/BUY_TICKETS
2  if self.currentState = SYSTEM_STATES::WELCOME then
3   if self.all_listed_movies->
4      any(t : Movie | t.title=in_title).available_tickets >= 1 then
5    if self.current_user.oclIsUndefined() then
6     --@AIM:BUY_Login_Mandatory
7     message = MSG::LOGIN_FIRST
8    else
9     let movie : Movie = self.all_listed_movies->
10                 any(t : Movie | t.title=in_title) in
11    let new_ticket : Ticket = (Ticket.allInstances())->
12                       any(owner_ticket.oclIsUndefined()) in
13      if movie.available_tickets = 1 then
14       --@AIM:BUY_Last_Ticket_Sold
15       movie.available_tickets = 0
16      else
17       --@AIM: BUY_Nominal_Success
18       movie.available_tickets = movie.available_tickets-1
19      endif and
20      self.current_user.all_tickets_in_basket->includes(ticket) and
21      movie.all_sold_tickets->includes(ticket) and
22      message = MSG::NONE
23     endif
24  else
25   --@AIM:BUY_Sold_Out
26   message = MSG::ALL_MOVIES_SOLD_OUT
27  endif
28 else
29  --@AIM: BUY_Wrong_Page
30  message = MSG::WRONG_PAGE
31 endif
```

FIGURE 7.2 – Code OCL de l'opération *buyTicket*

7.2.3 Suppression de proposition atomique

Cette mutation nécessite un évènement sous forme conjonctive pour pouvoir être appliquée. Elle permet de supprimer une proposition de la conjonction de la précondition. Cette mutation génère donc plusieurs nouveaux évènements, chacun représentant la suppression d'une des propositions de l'évènement original.

Définition 36 (Suppression de propositions atomiques)**.** Soit l'évènement E conjonctif à muter tel que E=$[op, P, Q, T]$. L'ensemble des mutations μ_{prop^-} est défini par :

$$\mu_{prop^-}(E) = \bigcup_{i \in 1..n} \{[op, \{p_k \in P \mid k \in 1..n \wedge k \neq i\}, \_, \_]\}$$

La mutation μ_{prop^-} génère n nouveaux évènements.

Exemple 39 (Suppression de propositions atomiques)**.** Considérons l'évènement conjonctif E_1. Celui-ci possède deux atomes dans sa précondition. Nous obtenons donc deux mutants pour cet évènement.

$\mu_{prop^-}(E_1)=$ {[buyTicket,"in_title=TITLE1" **on** "sut", _, _],
[buyTicket, "current_user.oclIsUndefined()" **on** "sut", _, _]}

FIGURE 7.3 – Graphe de tags de l'opération *buyTicket*

7.2.4 Echange de tags

L'intuition derrière cette mutation est de se focaliser sur un comportement différent de celui qui est considéré dans l'évènement fautif. Pour filtrer les comportements à considérer, nous utilisons une relation de proximité entre les comportements grâce au graphe de tags d'une opération.

L'intuition derrière cette proximité est la distance que l'on peut observer sur le graphe de tags d'une opération. Certains nœuds sont visuellement proches les uns des autres, alors que d'autres sont plus éloignés. Ce sont ces nœuds proches que nous souhaitons cibler.

Définition 37 (Graphe de tag)**.** Le graphe de tag d'un opération est son graphe de flot de contrôle dont les nœuds sont étiquetés par les tag qui apparaissent au niveau des points de choix.

Exemple 40 (Graphe de tags à partir de code OCL)**.** Pour donner un exemple de graphe de tags, nous prenons le code OCL de l'opération *buyTicket* présenté en FIGURE 7.2. Le graphe de tags correspondant à cette opération est représenté en FIGURE 7.3. Les chemins dans le graphe représentent les différents comportements de l'opération, et les étiquettes de ces nœuds sont les tags associés à ces comportements. Les noms de nœuds correspondent aux tags suivants :

- WP : @AIM :BUY_Wrong_Page
- SO : @AIM :BUY_Sold_Out
- LM : @AIM :BUY_Login_Mandatory
- NS : @AIM :BUY_Nominal_Success
- LT : @AIM :Buy_Last_Ticket_Sold

Chaque transition est étiquetée par l'expression OCL utilisée dans la décision d'un bloc *if-then-else*. Ainsi, les transitions correspondent aux expressions suivantes :

- A = self.currentState = SYSTEM_STATES : :WELCOME
- B = self.all_listed_movies→any(t :Movie|t.title=in_title).available_tickets>=1
- C = self.current_user.oclIsUndefined()
- D = movie.available_tickets = 1

Dans le graphe des tags, les nœuds LT et NS sont *proches*, graphiquement parlant. En effet, il s'agit d'achat de ticket avec succès, donc la seule différence est le comportement lorsqu'il ne reste plus qu'un seul ticket ou plus. Inversement, les nœuds WP et LT sont relativement éloignés.

Afin d'identifier les tags *proches*, nous définissons la fonction $swap(Tags)$ qui permet de les récupérer.

Définition 38 (Fonction swap)**.** Soit T l'ensemble des nœuds portant les tags à considérer. Soient $pred(t)$ la fonction qui dénote, dans le graphe de tags, le prédécesseur du

nœud portant le tag t et $succ(t)$ tous les tags des successeurs du nœud portant le tag t. Ainsi, la fonction $swap(T)$ est définie ainsi :

$$swap(T) = \bigcup_{t \in T} \begin{cases} swap(pred(t)) & \text{si } succ(pred(t)) \backslash T = \emptyset \\ succ(pred(t)) \backslash T & \text{sinon} \end{cases}$$

Il y a deux cas distincts pour cette fonction. Dans le premier cas, elle récupère tous les successeurs du prédécesseur et enlève tous les tags interdits par l'évènement. Si cette restriction donne l'ensemble vide, alors nous remontons dans le graphe avec les prédécesseurs, jusqu'à ce que *swap* ne soit plus vide. Le deuxième cas est le cas simple où il existe des successeurs au prédécesseur du nœud du tag considéré qui ne sont pas interdits par l'évènement.

Exemple 41 (Fonction swap)**.** Reprenons notre exemple de graphe de tags de la FIGURE 7.3 et l'exemple de propriété de la FIGURE 7.1. Dans cet exemple, les tags NS et LT sont interdits par l'évènement du `never`. Or, LT et NS partagent le même prédécesseur, donc $succ(pred(NS)) \backslash \{NS, LT\} = succ(pred(LT)) \backslash \{NS, LT\} = \emptyset$. Dans ce cas, nous remontons d'un cran dans le graphe pour considérer le prédécesseur du prédécesseur de LT et de NS. Donc par définition, $swap(\{LT, NS\}) = swap(pred(LT)) = swap(pred(NS)) = \{LM\}$, où LM est le comportement activé si aucun utilisateur n'est authentifié, identifié par le tag @AIM:BUY_Login_Mandatory et qui est bien le plus proche comportement dans le graphe.

La définition de cette fonction *swap* nous permet de définir la mutation d'échange de tags.

Définition 39 (Echange de tags)**.** Soit l'évènement E=$[op, P, Q, T]$. L'ensemble des mutations μ_{swap} est défini par :

$$\mu_{swap}(E) = \bigcup_{t \in swap(T)} \{[op, P, \_, \{t\}]\}$$

Exemple 42 (Echange de tags)**.** Reprenons l'exemple de graphe de tags de la FIGURE 7.3 et considérons l'exemple de propriété de la FIGURE 7.1. Les comportements interdits par l'évènement A sont ceux d'un achat normal (NS, dans le graphe de tags) et d'un achat du dernier ticket (LT dans le graphe de tags). Avec l'exemple précédent, nous avons déjà $swap(\{LT, NS)\}) = \{LM\}$. Ainsi, l'application de la mutation d'échange de tags donne :

μ_{swap}(A) = {[sut.buyTicket,
"self.all_listed_movies→ forAll(available_tickets = 0)" on "sut",
{@AIM :BUY_Login_Mandatory}]}

7.2.5 Mutations dans l'exemple de propriété

Reprenons l'exemple d'automate de propriété du chapitre précédent, rappelé dans la FIGURE 7.4. Les étiquettes de transition correspondent aux évènements suivants :

A = [login,_,"current_user.all_tickets_in_basket→size()=0" **on** "sut", {@AIM :LOG_Success}]
B = [logout,_,_,{@AIM :LOG_Logout}]
C = [buyTicket, "current_user.all_tickets_in_basket→size()=0" **on** "sut", _, {@AIM :BUY_Success}]
D = [deleteAllTickets, `"not(current_user.oclIsUndefined()) and current_user.all_tickets_in_basket→size()>0" on "sut"`, _, {@AIM :REMALL_Del_All_Tickets}]

FIGURE 7.4 – Automate de l'exemple C`precedes` D `after` A `until` B

Dans cet automate, la transition d'erreur que nous pouvons muter est $2 \xrightarrow{D\setminus\{B,C\}} e$ (e désignant l'état d'erreur). Ceci revient à muter l'évènement D tout en conservant la restriction $\{B, C\}$. En effet, cette restriction permet, en cas de chevauchement des évènements B, C et D, de privilégier la sortie de la portée (cas de la restriction par B) et ensuite, de privilégier l'évènement du motif (cas de la restriction par C). Le vrai évènement fautif est donc ce qu'il reste de D après cette restriction, les mutations nous permettent d'agrandir le périmètre de l'évènement D, mais toujours sous la restriction de B et C, qui eux, restent inchangés.

Dans un premier temps, nous pouvons appliquer la mutation générale de suppression de précondition μ_{pre^-} qui nous donne l'évènement [deleteAllTickets, _, _, _]. Puis, puisque la précondition est déjà sous la forme d'une conjonction de deux propositions atomiques, alors nous pouvons appliquer la mutation μ_{prop^-} :

μ_{prop^-}(D) = {[deleteAllTickets, "not(current_user.oclIsUndefined())" **on** "," _, _], [deleteAllTickets, "current_user.all_tickets_in_basket→size()>0" **on** "sut", _,_]}

Ensuite, puisque cet évènement comporte un tag, nous pouvons appliquer la mutation μ_{tag^-} :

μ_{tag^-}(D) = {[deleteAllTickets, `"not(current_user.oclIsUndefined()) and current_user.all_tickets_in_basket→size()>0" on "sut"`, _, _]}

Nous pouvons aussi définir μ_{swap} qui est alors :

μ_{swap}(D) = {[deleteAllTickets, `"not(current_user.oclIsUndefined()) and current_user.all_tickets_in_basket→size()>0" on "sut"`, _, {@AIM :REMALL_No_Ticket}]}

Ainsi, les mutations sur l'évènement de la transition d'erreur considérée produisent cinq évènements.

7.3 Critère de robustesse

Le processus de mutation peut produire plusieurs nouveaux évènements pour une seule transition d'erreur. Nous mutons l'automate original afin de représenter ces nouveaux évènements dans l'automate et nous définissons le critère de couverture associé à ces évènements à partir de l'automate muté.

7.3.1 Mutation de l'automate

Suivant un ensemble d'évènements mutés $\mathcal{D}_\mu$ et la transition $t_\mathcal{A}$ dont ils sont issus, la mutation d'automate est définie de la manière suivante.

Définition 40 (Mutation d'automate)**.** Soit $\mathcal{A} = \langle Q_\mathcal{A}, F_\mathcal{A}, I_\mathcal{A}, S_\mathcal{A}, R_\mathcal{A}, E_\mathcal{A}, \Sigma_\mathcal{A}, T_\mathcal{A}\rangle$ l'automate original et soit $\mathrm{t}_\mathcal{A} = (\mathrm{q}_\mathcal{A}, \mathrm{L}_\mathcal{A}, \mathrm{q'}_\mathcal{A}) \in T_\mathcal{A}$ la transition d'erreur considérée. Soit $T^\alpha_\mathcal{A}$ l'ensemble des $\alpha-$transitions de $\mathcal{A}$ et soit $T^\Sigma_\mathcal{A} \in T_\mathcal{A}$, l'ensemble des $\Sigma-$transitions de $\mathcal{A}$. Soient $\mathcal{D}_\mu(L))$ l'ensemble des mutations de $\mathrm{L}_\mathcal{A}$ et $\mathrm{L'}_\mathcal{A}$ une mutation parmi ces mutations. Soit $\mathcal{C} = \{\{q_0, p_0\}, \ldots, \{q_n, p_n\}\}$ la fonction qui associe à chaque noeud du chemin fautif partant de $q'_\mathcal{A}$ un nouvel état distinct de ceux de $Q_\mathcal{A}$ et des autres nouveaux états, avec $q_0 = q'_\mathcal{A}$ et $q_n \in E_\mathcal{A}$. Enfin, soit f un nouvel état distinct de tous ceux de $Q_\mathcal{A}$.

L'automate muté est l'automate $\mathcal{A}' = \{Q_{\mathcal{A}'}, \{f\}, I_\mathcal{A}, S_\mathcal{A}, R_\mathcal{A}, E_\mathcal{A}, \Sigma_\mathcal{A}, T_{\mathcal{A}'}\}$ où :

1. $Q_{\mathcal{A}'} = Q_\mathcal{A} \cup \{f\} \cup Img(\mathcal{C})$
2. $q \xrightarrow{L} q' \in T_{\mathcal{A}'}$ si et seulement si :
   - I. $q \xrightarrow{L} q' \in T_\mathcal{A} \wedge \neg(q = q_\mathcal{A} \wedge q \xrightarrow{L} q' \in T^\Sigma_\mathcal{A})$
   - II. $q = q_\mathcal{A} \wedge \exists q'' \in Q_\mathcal{A} \cdot (q'' \neq q'_\mathcal{A} \wedge q \xrightarrow{L'} q'' \in T^\Sigma_\mathcal{A}) \wedge L = L' \backslash L'_\mathcal{A}$
   - III. $q = q_\mathcal{A} \wedge q' = \mathcal{C}(q'_\mathcal{A}) \wedge L = L'_\mathcal{A} \backslash \{$

FIGURE 7.5 – Automate de `never A before B`

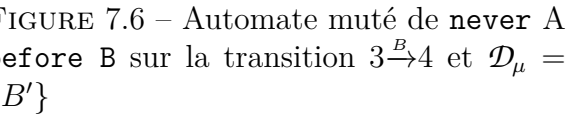


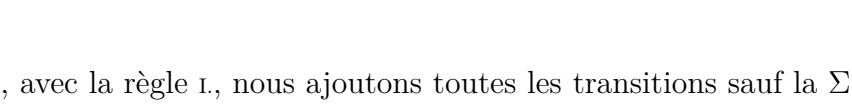
FIGURE 7.6 – Automate muté de `never A before B` sur la transition $3 \xrightarrow{B} 4$ et $\mathcal{D}_\mu = \{B'\}$

En exemple, la FIGURE 7.5 donne l'automate original d'une propriété de la forme `never A before B`. Considérons la transition d'erreur $3 \xrightarrow{B} e$ l'évènement B' comme une mutation de B. La FIGURE 7.6 illustre cette copie des états du chemin fautif et des transitions (en gras) qui le constituent avec la règle 8.IV..

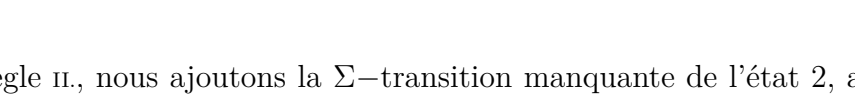
Enfin, la règle 8.V. permet d'ajouter la transition qui porte l'évènement muté de l'état source de la transition d'erreur à l'état *copie* de l'état destination de la transition muté. Ces évènements mutés sont restreints par tous les évènements portés par les $\alpha-$transitions sortant de l'état source afin de garantir une réelle disjonction entre ces évènements. Il est à noter que, bien que les règles de mutations produisent nécessairement des évènements activables (vis-à-vis de la propriété), nous restreignons l'évènement muté par l'évènement fautif afin de garantir un automate déterministe. En cas de chevauchement, cette distinction privilégie l'activation de l'évènement fautif sur son évènement muté.

Exemple 43 (Automate muté de l'exemple)**.** Reprenons l'exemple de propriété décrite dans la FIGURE 7.4. Pour cet exemple de mutation d'automate, nous considérons la transition ses évènements mutés dans l'exemple de la section précédente. Nous notons D' un évènement muté de $\mathcal{D}_\mu$, où $\mathcal{D}_\mu$ est l'ensemble des mutations de l'évènement D.

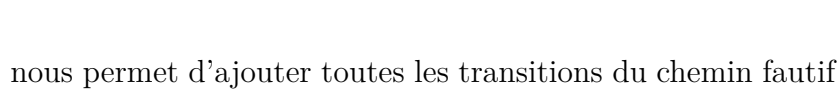
Dans un premier temps, nous appliquons la définition qui permet d'ajouter tous les états, initiaux et d'erreur compris, ainsi que le nouvel état final :



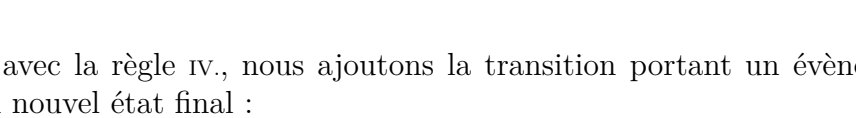
Ensuite, avec la règle I., nous ajoutons toutes les transitions sauf la $\Sigma-$transition qui sort de l'état 2 :



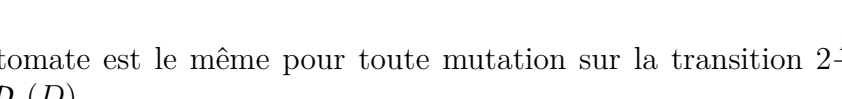
Avec la règle II., nous ajoutons la $\Sigma-$transition manquante de l'état 2, avec l'évènement muté en plus dans sa restriction :



La règle III. nous permet d'ajouter toutes les transitions du chemin fautif sur la copie (ici, la transition réflexive sur l'état final) :



Enfin, avec la règle IV., nous ajoutons la transition portant un évènement muté, de l'état 2 au nouvel état final :



Cet automate est le même pour toute mutation sur la transition $2 \xrightarrow{D \setminus \{B\}} e$, quel que soit $D' \in \mathcal{D}_\mu(D)$.

7.3.2 Critère de robustesse

Avec la mutation d'automate, nous obtenons un nouvel automate qui exhibe de nouvelles $\alpha-$transitions à couvrir : les transitions avec les évènements mutés. Pour nous assurer la couverture de ces transitions, nous définissons un critère spécifique, fondé sur l'automate muté. Ainsi, ce critère se focalise sur l'activation de ces évènements *proches* par des relâchements du contexte d'activation de l'évènement original pour nous permettre d'observer que, malgré cela, l'évènement fautif n'a pas été activé.

Définition 41 (Critère de robustesse)**.** Soit $\mathcal{A}$ l'automate de propriété considéré et soit t=q$\xrightarrow{L}$q' la transition d'erreur considérée. L'ensemble C_μ des mutations de t est défini par :

$$C_\mu(L) = \mu_{pre^-}(L) \cup \mu_{tag^-}(L) \cup \mu_{prop^-}(L) \cup \mu_{swap}(L)$$

Le taux de couverture associé T_μ est défini ainsi :

$$\mathrm{T}_\mu = \frac{\text{nombre d'évènements mutés couverts}}{\text{nombre total d'évènements mutés possibles sur l'automate}}$$

Ce critère vise simplement à couvrir tous les évènements obtenus après l'application des mutations sur l'évènement considéré. Le taux de couverture mesure le nombre d'évènements mutés couvert, quel que soit leur mutation.

Exemple 44 (Critère de robustesse)**.** Si nous reprenons notre exemple de propriété nous avons deux transitions d'erreur qui portent le même évènement : les mutations seront donc identiques pour ces deux transitions. Nous avons déjà calculé les différentes mutations dans la section précédente. Ainsi, pour notre exemple, nous avons donc cinq nouveaux évènements qui peuvent être utilisés sur la nouvelle transition de l'automate muté à partir de l'état 2. Ainsi, nous avons :

$C_\mu(L)$ = {[deleteAllTickets, _, _, _],
[deleteAllTickets, `"not(current_user.oclIsUndefined()) and current_user.all_tickets_in_basket→size()>0" on "sut"`, _, _],
[deleteAllTickets, "not(current_user.oclIsUndefined())" **on** "," _, _],
[deleteAllTickets, "current_user.all_tickets_in_basket→size()>0" **on** "sut", _,_],
[deleteAllTickets, `"not(current_user.oclIsUndefined()) and current_user.all_tickets_in_basket→size()>0" on "sut"`, _, {@AIM :REMALL_No_Ticket}]}

Le premier évènement muté permet d'illustrer des activations de l'opération *deleteAllTickets* sans contraintes obtenu par la mutation μ_{pre^-}.

Le second évènement (μ_{tag^-}) conserve cette précondition. Dans ce cas, nous cherchons à observer le comportement activé en ne laissant que la précondition comme contrainte. A l'état 2, il ne peut pas y avoir de valuation pour cet évènement. En effet, la précondition "current_user.all_tickets_in_basket→size()>0" **on** "sut" nous en empêche et l'activation d'un *buyTicket*, qui permettrait alors de satisfaire cette condition nous amènerait à l'état 3.

La mutation précédente est alors un peu trop restrictive, c'est pourquoi les deux évènements suivants (μ_{prop^-}) sont bien plus intéressants en relachant une des conditions de la précondition. Le premier relâche celle du panier non-vide. Dans ce cas, une activation

de l'évènement à partir de l'état 2 déclenchera le comportement @AIM:REMALL_No_Ticket ou @AIM:REMALL_Wrong_Page, car aucun ticket n'a pu être acheté au préalable (sinon, nous serions dans l'état 3) ou nous n'avons pas changé de page après l'authentification. Le second évènement relâche la condition de l'utilisateur authentifié. Ce cas n'est pas possible car nous sommes justement dans une portée qui spécifie les exécutions entre une authentification et une déconnexion de l'utilisateur. Donc, cette condition est implicitement vraie, à cause de la portée.

Enfin, le dernier évènement spécifie clairement le comportement à activer qui correspond à un des deux comportements activables à ce stade de la propriété (avec le comportement @AIM:REMALL_Wrong_Page). Cet évènement nous permet de nous assurer de sa couverture.

7.4 Synthèse

Les critères présentés au chapitre précédent permettent d'illustrer des exécutions nominales de la propriété. Cependant, ils ne nous permettent pas d'insister sur la robustesse du système vis-à-vis de la propriété. Pour pallier ce manque, nous utilisons un mécanisme de mutation de l'automate en nous focalisant sur les transitions d'erreur.

A cette fin, nous avons défini des opérateurs de mutation qui permettent de générer des évènements proches des évènements d'erreur. L'observation de l'état du système et des comportements activés pendant l'exécution de l'évènement muté permet d'identifier si le système répond conformément à la propriété.

Ensuite, les mutations de ces évènements nécessitent d'adapter l'automate de la propriété pour les faire apparaître. Cette adaptation permet de clairement identifier la transition portant l'évènement muté et permet de forcer son activation en modifiant les états finaux de l'automate.

Enfin, nous avons défini un critère de couverture qui permet de cibler ces transitions. Ce critère permet alors de quantifier, la couverture des évènements mutés par une suite de tests, et nous permettra par la suite de générer les tests qui permettent de les couvrir dans le chapitre suivant.

Chapitre 8

Génération de tests à partir des critères de couverture des automates

Sommaire

8.1 Génération de tests guidée par les critères 140
8.1.1 Rappel de l'exemple 140
8.1.2 Définitions 140
8.1.3 Critère *all-α* 141
8.1.4 Critère *α−pairs* 142
8.1.5 Critères *k-scope* et *k-pattern* 143
8.1.6 Critère de robustesse 145
8.2 Traduction en scénarios 146
8.2.1 Règles de traduction basiques 147
8.2.2 Traduction des restrictions sur un évènement simple 147
8.2.3 Traduction des restrictions sur un évènement sans opération ni comportement 151
8.2.4 Traduction de l'exemple 154
8.3 Synthèse 155

Les deux chapitres précédents ont présenté des critères de couverture spécifiques aux automates de propriétés. Lors de la mesure de couverture, nous avons pu observer qu'une suite de tests ne permet pas nécessairement une couverture à 100%, que le critère soit nominal (voir Chapitre 6) ou de robustesse (voir Chapitre 7). Pour pallier ce manque, nous introduisons une technique de génération de tests, guidée par nos critères présentés précédemment, afin d'améliorer le taux de couverture de la propriété par une suite de tests spécifique.

A cette fin, nous définissons des algorithmes de génération de tests, spécifiques à chaque critère. Les scénarios sont exprimés dans le langage de scénarios décrit au Chapitre 4.

Dans un premier temps, nous présentons dans la section 8.1 la technique de génération de tests mise en œuvre pour chacun des critères introduits dans les deux chapitres précédents. Puis, dans la section 8.2, nous présentons les règles de traduction des expressions

régulières issues de la génération de cas de test dans le langage de scénarios. Enfin, nous appliquons la démarche sur les exemples de propriétés.

8.1 Génération de tests guidée par les critères

Afin de s'assurer de la couverture des éléments définis par nos critères, nous proposons, pour chaque critère, un algorithme qui produit une expression régulière représentant les chemins dans l'automate permettant de couvrir un des éléments du critère.

Pour calculer ces expressions régulières, chaque algorithme utilise l'automate de propriété et ne considère qu'un seul état final à la fois, dans le but de limiter la taille des expressions régulières générées. De plus, cela nous permet d'assurer que le critère est couvert dans différents états de la propriété. Ainsi, il est possible d'obtenir plusieurs scénarios couvrant la même transition/paire de transitions, mais dont le suffixe (l'ensemble des chemins menant à l'état final) diffère.

8.1.1 Rappel de l'exemple

Pour ce chapitre, nous réutilisons l'exemple de propriété décrit dans le Chapitre 6. L'automate de cet exemple est rappelé dans la FIGURE 8.1 avec les étiquettes des transitions qui correspondent aux évènements suivants :

A = [login,_,"current_user.all_tickets_in_basket→size()=0" **on** "sut", {@AIM :LOG_Success}]
B = [logout,_,_,{@AIM :LOG_Logout}]
C = [buyTicket, "current_user.all_tickets_in_basket→size()=0" **on** "sut", _, {@AIM :BUY_Success}]
D = [deleteAllTickets, `"not(current_user.oclIsUndefined()) and current_user.all_tickets_in_basket→size()>0" on "sut"`, _, {@AIM :REMALL_Dell_All_Tickets}]

FIGURE 8.1 – Rappel de l'automate de C`precedes` D `after` A `until` B

8.1.2 Définitions

Dans les algorithmes de génération de tests, nous utilisons les fonctions *chemin(q,q',T)* et *chemin_sbf(q,q',T)* qui permettent de calculer l'expression régulière sur les étiquettes des évènements des transitions couvertes. Nous définissons ces deux fonctions.

Définition 42 (Fonction chemin(q,q',T))**.** La fonction *chemin(q,q',T)* permet de calculer l'expression régulière qui représente tous les chemins de l'état q à l'état q' et qui ne traversent pas les transitions de T.

Définition 43 (Fonction chemin_sbf(q,q',T))**.** La fonction *chemin_sbf(q,q',T)* (le suffixe *sbf* signifie *sans boucle finale*) permet de calculer l'expression régulière qui représente tous les chemins de l'état q à l'état q', qui ne traversent pas les transitions de T et qui ne permettent pas de sortir de l'état q'.

La différence entre ces deux fonctions est que la seconde fonction *chemin_sbf* ne permet pas les boucles qui permettent de sortir de l'état final une fois atteint. Un exemple simple est une boucle réflexive sur l'état final. Elle est considérée dans le *chemin* mais ne l'est pas dans *chemin_sbf*.

Exemple 45 (Fonctions chemin(q,q',T) et chemin_sbf(q,q',T))**.** Prenons l'exemple d'automate de la FIGURE 8.1 et considérons les fonctions de chemins à partir de l'état 1 pour atteindre l'état 2, sans restriction de transitions. Ainsi, nous avons :

$$\begin{aligned} \text{chemin}(1,2,\emptyset) = \; & \Sigma\backslash\{A\}^*.A.((B.\Sigma\backslash\{A\}^*.A) \\ & \mid \Sigma\backslash\{B,C,D\} \\ & \mid (C\backslash\{B,D\}.\Sigma\backslash\{B,D\}^*.(D\backslash\{B\} \mid B.\Sigma\backslash\{A\}^*.A)))^* \\ \text{chemin\_sbf}(1,2,\emptyset) = \; & \Sigma\backslash\{A\}^*.A \end{aligned}$$

Dans le cas de *chemin(1,2,∅)*, après avoir atteint l'état 2 $\Sigma\backslash\{A\}^*.A$, il y a quatre alternatives de boucle pour y revenir s'il est quitté.

- $B.\Sigma\backslash\{A\}^*.A$ permet les boucles de sortie et d'entrée à partir de l'état 2 sans passer par les transitions du motif,
- $\Sigma\backslash\{B,C,D\}$ permet de boucler sur l'état 2,
- $C\backslash\{B,D\}.\Sigma\backslash\{B,D\}^*.(D\backslash\{B\}$ permet de boucler sur les transitions du motif,
- $(C\backslash\{B,D\}.\Sigma\backslash\{B,D\}^*.(B.\Sigma\backslash\{A\}^*.A)))^*$ permet de boucler en passant par C puis par la sortie et l'entrée dans la portée.

Le cas de *chemin_sbf(1,2,∅)* est beaucoup plus simple. En effet, puisque cette fonction n'autorise pas la sortie de l'état 2 après l'avoir atteint, alors le chemin correspondant est celui qui permet d'y mener, à savoir $\Sigma\backslash\{A\}^*.A$.

8.1.3 Critère *all-*α

L'objectif de la génération sur ce critère est d'obtenir un scénario qui permet de couvrir une $\alpha-$transition de l'ensemble C_α (voir la section 6.2.1). La FIGURE 8.2 donne l'algorithme de génération pour couvrir une $\alpha-$transition donnée et pour un état final donné.

```
1  Données : A : automate de la propriété
2            i : état initial considéré (i∈I_A)
3            f : état final considéré (f∈F_A)
4            t = (q,L_α,q') : α−transition à couvrir
5  Résultat : regex : expression régulière du cas de test abstrait
6  Début
7    regex ← chemin(i, q, {t})
8    regex ← regex.L_α
9    regex ← regex.chemin_sbf(q', f, ∅)
10 Fin
```

FIGURE 8.2 – Algorithme de génération - critère *all-*α

Le cas de test abstrait est composé de trois parties. Dans un premier temps nous calculons tous les chemins qui permettent d'atteindre l'état source q de la transition à couvrir (ligne 7) à partir d'un état initial i. Ensuite, puisque ce préfixe permet de

donner tous les chemins possibles qui permettent d'atteindre l'état source de la transition à couvrir, celle-ci est alors activée (ligne 8). Enfin, nous calculons le suffixe à partir de l'état de destination de la transition à couvrir juqu'à l'état final considéré (ligne 9).

Puisque cet algorithme considère un état initial et un état final à la fois pour chaque $\alpha-$transition de C_α, alors pour un automate de propriété donné, nous avons au plus $card(C_\alpha) \times card(F_\mathcal{A}) \times card(I_\mathcal{A})$ expressions régulières possibles, où $F_\mathcal{A}$ (resp. $I_\mathcal{A}$) est l'ensemble des états finaux (resp. initiaux) de l'automate $\mathcal{A}$.

Exemple 46 (Génération d'expressions régulières par le critère *all-*α**).** Reprenons notre exemple de propriété défini au chapitre 6 (dont l'automate est présenté en FIGURE 8.1) et cherchons à couvrir l'$\alpha-$transition $2\xrightarrow{B}4$ en considérant l'état 4 comme état final à atteindre. Ainsi, le préfixe est défini par :

$$\Sigma\backslash\{A\}^*.A.(C\backslash\{B,D\}.\Sigma\backslash\{B,D\}^*.(B.\Sigma\backslash\{A\}^*.A \mid D\backslash\{B\}) \mid \Sigma\backslash\{B,C,D\})^*$$

Ce préfixe permet d'atteindre l'état 2 juste avec $\Sigma\backslash\{A\}^*.A$. Mais l'utilisation de la fonction *chemin* permet aussi d'emprunter les chemins qui permettent de fermer la portée et de l'ouvrir à nouveau pour retourner à l'état 2 (la seconde partie du préfixe). Ce préfixe permet aussi d'effectuer plusieurs itérations du motif avant de terminer sur l'état 2.

Le suffixe est vide. En effet, la traversée de la transition $2\xrightarrow{B}4$ amène immédiatement à l'état final visé. L'utilisation de la fonction *chemins_sbf* ne prend pas en compte les boucles qui sortiraient de l'état final.

8.1.4 Critère $\alpha-$*pairs*

L'algorithme de la FIGURE 8.3 permet de couvrir une paire de transitions de l'ensemble C_{α^2} défini par le critère $\alpha-$*pairs* (voir la section 6.2.2).

```
1  Données : A : automate de la propriété
2            i : état final considéré (i∈I_A)
3            f : état final considéré (f∈F_A)
4            t1 = (q_α1 ,L_α1 ,q'_α1) : première transition à couvrir
5            t2 = (q_α2 ,L_α2 ,q'_α2) : seconde transition à couvrir
6  Résultat : regex : expression régulière du cas de test abstrait
7  Variables : prefix : expression régulière du préfixe
8              suffix : expression régulière du suffixe
9              inter  : expression régulière des chemins intermédiaires
10 Début
11   prefix ← chemin(i, q_α1 ,{t1})
12   inter ← regex.chemin(q'_α1, q_α2 ,{t2})
13   suffix ← regex.chemin_sbf(q'_α2, f, ∅)
14   regex ← prefix.L_α1.inter.L_α2.suffix
15 Fin
```

FIGURE 8.3 – Algorithme de génération - critère $\alpha-$*pairs*

Similairement à l'algorithme précédent, le résultat est une concaténation de plusieurs expressions régulières. Dans un premier temps, nous calculons le préfixe (ligne 8) pour atteindre l'état source $q_{\alpha 1}$ de la première transition de la paire. Ensuite, après la couverture

de la première transition (ligne 9), l'algorithme ajoute tous les chemins intermédiaires entre la première transition et la seconde à couvrir (ligne 10). Enfin, après les chemins intermédiaires, l'algorithme couvre la seconde transition (ligne 11) puis complète le cas de test par tous les chemins menant à l'état final considéré f. Contrairement aux chemins intermédiaires, le suffixe n'a pas de restriction de transitions. En effet, il est peut être nécessaire pour le système d'activer plusieurs fois la paire de transitions avant d'atteindre l'état final.

Puisque cet algorithme considère un état initial et un état final à la fois pour chaque $\alpha-$transition de C_α, alors, pour un automate de propriété donné, nous avons au plus $card(C_{\alpha^2}) \times card(F_{\mathcal{A}}) \times card(I_{\mathcal{A}})$ expressions régulières possibles, où $F_{\mathcal{A}}$ (resp. $I_{\mathcal{A}}$) est l'ensemble des états finaux (resp. initiaux) de l'automate $\mathcal{A}$.

Exemple 47 (Génération d'expressions régulières par le critère des paires d'$\alpha-$transitions)**.** Reprenons notre exemple de propriété (dont l'automate est représenté en FIGURE 8.1) et cherchons à couvrir la paire d'$\alpha-$transitions $\{1\xrightarrow{A}2,2\xrightarrow{C\backslash\{B\}}3\}$ en considérant l'état 4 comme état final à atteindre. Ainsi, le préfixe est simplement défini par $\Sigma\backslash\{A\}^*$ qui permet de rester dans l'état 1, source de la transition $1\xrightarrow{A}2$. Nous calculons ensuite tous les chemins intermédiaires entre les deux transitions à couvrir, i.e. les boucles sur l'état 2, car nous nous interdisons, pour l'instant, la traversée de la seconde transition. Ces chemins intermédiaires sont définis par :

$$(B.\Sigma\backslash\{A\}^*.A \mid \Sigma\backslash\{B,C,D\})^*$$

qui sont en fait les boucles permettant de sortir et d'entrer dans la portée.

Enfin, après la couverture de la seconde transition de la paire, il ne reste plus qu'à décrire les chemins permettant d'atteindre l'état 4, l'état final considéré, à partir de l'état 3. Enfin, le suffixe est décrit de la manière suivante :

$$(D\backslash\{B\}.\Sigma\backslash\{B,C,D\}^*.C\backslash\{B,D\} \mid \Sigma\{B,D\})^*.(D\backslash\{B\}.\Sigma\backslash\{B,C,D\}^*.B \mid B)$$

8.1.5 Critères *k-scope* et *k-pattern*

Cet algorithme permet de générer des tests pour couvrir les critères *k-pattern* et *k-scope* de la portée (voir les sections 6.2.3 et 6.2.4). En effet, pour les deux critères, il s'agit de couvrir des successions de paires de transitions.

La FIGURE 8.4 donne l'algorithme qui permet de générer un cas de test permettant de couvrir une séquence issue de l'un de ces deux critères. Nous notons $first(p)$ et $second(p)$ la première (resp. la seconde) transition de la paire p, et $src(t)$ et $dest(t)$ qui dénotent l'état source (resp. destination) de la transition t.

Similairement aux algorithmes précédents, celui de la FIGURE 8.4 décrit le résultat sous forme d'expressions régulières et est constitué de plusieurs parties. Dans cet algorithme, nous considérons un état initial i et un état final f, une séquence Seq de paires de transitions à couvrir, le nombre k d'itérations considérées par la séquence (qui est identique à la taille de Seq). Enfin, nous considérons l'ensemble des transitions T et qui

```
1  Données : A : automate de la propriété
2            i : état final considéré (i∈I_A)
3            f : état final considéré (f∈F_A)
4            Seq : séquence des paires de transitions à couvrir
5            T : ensemble des transitions portant les évènements
6                des premières et secondes transitions de Seq
7            k : nombre d'itérations (= taille de Seq)

9  Résultat : regex : expression régulière du cas de test abstrait

11 Début
12   regex ← chemin(i,src(first(Seq[0])),{first(Seq[0])})
13   pour j de 0 à k−2 faire
14     regex ← regex.first(Seq[j])
15     regex ← regex.chemin(dest(first(Seq[j])), src(second(Seq[j])), T)
16     regex ← regex.second(Seq[j])
17     regex ← regex.chemin(dest(second(Seq[j])), src(first(Seq[j+1])), ∅)
18   fin
19   regex ← regex.first(Seq[k−1])
20   regex ← regex.chemin(dest(first(Seq[k−1])), src(second(Seq[k−1])), T)
21   regex ← regex.second(Seq[k−1])
22   regex ← regex.chemin_sbf(dest(second(Seq[k−1])), f, T)
23 Fin
```

FIGURE 8.4 – Algorithme de génération - critères *k-scope* et *k-pattern*

portent les évènements considérés par les premières et les secondes transitions des paires de *Seq*. Cet ensemble est nécessaire afin de restreindre les transitions à emprunter lors du calcul des chemins entre les transitions d'une même paire. En effet, sans cela, nous pourrions couvrir par accident une autre paire que celle considérée.

La première permet de rejoindre l'état source de la première transition de la première paire de la séquence à partir de l'état initial i (ligne 13). Ensuite (lignes 14 à 19) permet de calculer les chemins pour chaque élément de la séquence *Seq* à partir de la précédente. Ainsi, nous ajoutons en premier lieu la première transition de la paire actuellement considérée (ligne 15), puis nous tentons d'atteindre l'état source de la deuxième transition de la paire (ligne 16). Ensuite, nous couvrons cette seconde transition (ligne 17) pour enfin calculer le chemin qui nous permet d'atteindre la paire suivante dans la séquence (ligne 18). L'algorithme itère sur toutes les paires de la séquence à l'exception de la dernière qui est ajoutée après (lignes 20 à 22) pour terminer le chemin sur l'état final f de l'automate considéré (ligne 23).

Puisque cet algorithme considère un état initial et un état final à la fois pour chaque séquence issue des critères *k-scope* et *k-pattern*, alors pour un automate de propriété donné, nous avons au plus $card(C_s^k) \times card(F_{\mathcal{A}}) \times card(I_{\mathcal{A}})$ expressions régulières possibles pour *k-scope* et au plus $card(C_p^k) \times card(F_{\mathcal{A}}) \times card(I_{\mathcal{A}})$ expressions régulières possibles pour *k-pattern*, où $F_{\mathcal{A}}$ (resp. $I_{\mathcal{A}}$) est l'ensemble des états finaux (resp. initiaux) de l'automate $\mathcal{A}$.

Exemple 48 (Génération d'expressions régulières pour le critère *k-scope*)**.** Reprenons notre exemple de propriété pour couvrir 2 itérations de la portée et considérons l'état 4 comme état final. Considérons la séquence de paires de transitions à couvrir suivante (pour $k = 2$) :

$$(1 \xrightarrow{A} 2,3 \xrightarrow{B} 4)\,;(4 \xrightarrow{A} 2,2 \xrightarrow{B} 4)$$

Dans notre exemple d'automate, l'ensemble T est défini de la manière suivante :

$$\mathrm{T} = \{1 \xrightarrow{A} 2,\ 4 \xrightarrow{A} 2,\ 3 \xrightarrow{B} 4,\ 2 \xrightarrow{B} 4\}$$

La première étape est le calcul du préfixe pour atteindre l'état 1 (l'état source de la transition $1 \xrightarrow{A} 2$) qui correspond simplement à $\Sigma\backslash\{A\}^*$. Ensuite, après la traversée de la transition $1 \xrightarrow{A} 2$, nous devons atteindre l'état 3 (source de $3 \xrightarrow{B} 4$). Cependant, la restriction de *chemin* ne nous permet pas d'emprunter les transitions $2 \xrightarrow{B} 4$ et $3 \xrightarrow{B} 4$, ce qui nous laisse comme possibilité les chemins qui bouclent sur le motif. Ainsi, ces chemins sont

$$\Sigma\backslash\{B,C,D\}.C\backslash\{B,D\}.(\Sigma\backslash\{B,D\}^* \mid D\backslash\{B\}.\Sigma\backslash\{B,C,D\}^*.C\backslash\{B,D\})^*$$

Ceux-ci nous permettent d'atteindre l'état 3 pour activer la transition $3 \xrightarrow{B} 4$. Ensuite, nous devons calculer les chemins entre $3 \xrightarrow{B} 4$) et $4 \xrightarrow{A} 2$, qui se résument à $\Sigma\backslash\{A\}^*$. Il nous reste maintenant les chemins entre les transitions de la dernière paire $4 \xrightarrow{A} 2$ et $2 \xrightarrow{B} 4$ décrits par :

$$(\Sigma\backslash\{B,C,D\} \mid C\backslash\{B,D\}.\Sigma\backslash\{B,D\}^*.D\backslash\{B\})^*.$$

Il ne nous reste plus qu'à terminer avec la dernière transition $2 \xrightarrow{B} 4$ et le chemin amenant à l'état final qui, dans ce cas, est déjà atteint. L'utilisation de *chemin_sbf* ne permet pas de considérer la boucle réflexive.

Ainsi, l'expression régulière finale, qui correspond aux chemins permettant de couvrir la séquence de paire de transitions, est :

$$\Sigma\backslash\{A\}^*.A.\Sigma\backslash\{B,C,D\}.C\backslash\{B,D\}.(\Sigma\backslash\{B,D\}^* \mid D\backslash\{B\}.\Sigma\backslash\{B,C,D\}^*.C\backslash\{B,D\})^*.$$
$$B.\Sigma\backslash\{A\}^*.A.(\Sigma\backslash\{B,C,D\} \mid C\backslash\{B,D\}.\Sigma\backslash\{B,D\}^*.D\backslash\{B\})^*.B$$

8.1.6 Critère de robustesse

L'objectif de cet algorithme est la couverture du critère des évènements limites. Il permet de couvrir une transition qui porte un évènement muté issu des règles de mutation (voir la section 7.2) sur un automate muté exhibant cette transition (voir la section 7.3). Cet algorithme est le même que pour le critère des $\alpha-$*transitions*, à la différence des paramètres. En effet, dans cet algorithme, nous considérons l'automate muté et nous cherchons à couvrir la transition mutée. L'algorithme permettant de couvrir une des mutations qui est présentée en FIGURE 8.5.

Similairement aux autres algorithmes, la première étape est le calcul d'un préfixe qui permet d'atteindre l'état source q de la transition à couvrir (ligne 7) avant de couvrir la transition mutée considérée (ligne 8). Il ne reste ensuite plus qu'à atteindre l'unique état final de l'automate pour compléter le cas de test (ligne 9). Il est à noter que dans cet algorithme, nous ne donnons pas l'état final à considérer. En effet, par construction, l'automate muté (voir la section 7.3) ne comporte plus qu'un seul état final nommé f_A.

```
1  Données : A : automate muté de la propriété dont f_A est l'état final
2            i : état final considéré (i∈I_A)
3            t = (q,L_μ,q') : la transition mutée de l'évènement d'erreur à couvrir

5  Résultat : regex : expression régulière du cas de test abstrait
6  Début
7    regex ← chemin(i, q, {t})
8    regex ← regex.L_μ
9    regex ← regex.chemin_sbf(q', f_A, ∅)
10 Fin
```

FIGURE 8.5 – Algorithme de génération - critère de robustesse

Puisque cet algorithme considère un état initial et un état final à la fois pour chaque $\alpha-$transition de C_α, alors pour un automate de propriété donné, nous avons au plus $card(C_\mu) \times card(I_\mathcal{A})$ expressions régulières possibles, où $F_\mathcal{A}$ (resp. $I_\mathcal{A}$) est l'ensemble des états finaux (resp. initiaux) de l'automate $\mathcal{A}$.

Exemple 49 (Génération d'expressions régulières par le critère des mutations)**.** Reprenons notre exemple de propriété et considérons l'automate muté généré dans le chapitre précédent et rappelé en FIGURE 8.6.

FIGURE 8.6 – Automate de la FIGURE 8.1 muté

Dans un premier temps, nous calculons le préfixe permettant d'atteindre l'état source de la transition mutée $2 \rightarrow f_A$ qui est l'état 2 dans notre cas. Ainsi le préfixe est défini par :

$$\Sigma\backslash\{A\}^*.A.(\Sigma\backslash\{B,C,D,D'\}^*.(B.\Sigma\backslash\{A\}^*.A \mid C\backslash\{B,D\}.\Sigma\backslash\{B,D\}^*.(D\backslash\{B\} \mid B.\Sigma\backslash\{A\}^*.A))^*)^*$$

Le calcul de la fin est immédiat : il s'agit du chemin de l'état 2 à l'état final qui est simplement défini par $D'\backslash\{B,C,D\}$, la transition réflexive n'est pas considérée car nous utilisons la fonction *chemin_sbf*.

8.2 Traduction en scénarios

Dans le processus de génération de tests guidée par les critères, le résultat de la génération est une expression régulière représentant des cas de test abstraits. Cependant, ces expressions définissent tous les chemins permettant de couvrir un critère mais sont difficilement lisibles et inexploitables en l'état. Cette difficulté de lecture peut être un frein pour un ingénieur de test, en particulier s'il souhaite le modifier pour répondre à ses besoins ou l'optimiser (en spécifiant plus précisément les constructions Σ par exemple). Nous proposons donc de traduire les expressions régulières dans le langage de scénarios introduit au Chapitre 4.

Le langage de scénarios permet à l'ingénieur de test d'identifier clairement les différents chemins proposés dans le cas de test et éventuellement de modifier le scénario ou d'en supprimer des parties. Enfin, cette traduction permet l'utilisation des différentes directives

de sélection des chemins qui permettent à l'ingénieur de test d'adapter le scénario pour limiter l'explosion combinatoire de son dépliage en fonction de son expertise et de sa connaissance du modèle.

8.2.1 Règles de traduction basiques

Les constructions des expressions régulières trouvent une correspondance directe dans le langage de scénarios avec les règles suivantes, avec E et F des expressions régulières :

$$\begin{aligned} \tau(\mathrm{E.F}) &\rightsquigarrow \tau(\mathrm{E})\,;\,\tau(\mathrm{F}) \\ \tau(\mathrm{E}_1 \mid \ldots \mid \mathrm{E}_n) &\rightsquigarrow \textbf{choice}\ \{\tau(\mathrm{E}_1)\}\textbf{or}\ \ldots\ \textbf{or}\ \{\tau(\mathrm{E}_n)\} \\ \tau(\mathrm{E}^*) &\rightsquigarrow \textbf{unfold between}\ 0\ \textbf{and}\ \mathrm{N}\ \textbf{times}\ \{\tau(\mathrm{E})\} \\ \tau(\mathrm{E}^+) &\rightsquigarrow \textbf{unfold between}\ 1\ \textbf{and}\ \mathrm{N}\ \textbf{times}\ \{\tau(\mathrm{E})\} \end{aligned}$$

Dans le cas de répétitions non bornées (+ et *), il est nécessaire de fixer une borne maximale N afin d'assurer le dépliage du scénario. Cette borne doit être choisie par l'ingénieur validation en fonction de la connaissance qu'il a du modèle.

Les évènements simples trouvent aussi une correspondance directe dans le langage de scénarios :

$$\begin{aligned} \tau([\text{instance.op, P, Q,}\{\mathrm{t}_1, \ldots, \mathrm{t}_n\}]) &= \textbf{assert}(\mathrm{P})\,; \\ &\quad \text{instance.op}\ \textbf{including}\ \{\mathrm{t}_1, \ldots, \mathrm{t}_n\}\,; \\ &\quad \textbf{assert}(\mathrm{Q})\,; \\ \tau([\_, \mathrm{P, Q,}\{\mathrm{t}_1, \ldots, \mathrm{t}_n\}]) &= \textbf{assert}(\mathrm{P})\,; \\ &\quad \$\mathrm{OP}\ \textbf{including}\ \{\mathrm{t}_1, \ldots, \mathrm{t}_n\}\,; \\ &\quad \textbf{assert}(\mathrm{Q})\,; \end{aligned}$$

Les assertions avant et après l'appel de l'opération traduisent la précondition et la post-condition de l'évènement.

8.2.2 Traduction des restrictions sur un évènement simple

Bien que la traduction des évènements simples (sans restriction) dans le langage de scénarios soit immédiate, la traduction d'évènements avec restrictions, tel que l'évènement C\{B,D} de la transition (2→3), est plus complexe. L'exclusion d'opération est possible dans le cas d'un appel d'opération quelconque avec la construction $OP, mais il ne permet pas la restriction d'évènements plus fins en fonction d'une précondition par exemple.

Puisque ce genre de construction n'existe pas dans le langage, nous devons déplier tous les cas possibles qui permettent de représenter cette restriction. Nous décrivons en premier lieu les simplifications qui peuvent être faites avant de déterminer tous les cas. Ensuite, nous donnons d'abord l'intuition sur un exemple avec une restriction, puis avec deux, avant de donner une généralisation de cette traduction.

Simplification

Cette étape de simplification permet d'éliminer de la restriction les évènements qui n'ont rien en commun avec l'évènement de base, en ne gardant que les évènements qui

ont une possible intersection avec l'évènement de base. Ainsi, nous limitons le nombre de cas à considérer.

La première simplification s'effectue sur le nom de l'instance utilisée et sur le nom d'opération. Prenons par exemple, l'évènement [sut.buyTicket, _, _, _]\[sut.login, _, _,_] est exactement équivalent à l'évènement [buyTicket, _, _, _], car restreindre un appel de *buyTicket* par des appels de *login* n'a pas de sens car il empêche les comportements de *login* lors d'un appel de *buyTicket*.

Cette simplification peut être plus fine en se fondant sur les tags des évènements. Prenons l'exemple suivant :

[buyTicket, _, _, {@AIM :BUY_Nominal_Success}]\[buyTicket, _, _, {@AIM :BUY_Login_Mandatory}]

La restriction @AIM:BUY_Nominal_Success par @AIM:BUY_Login_Mandatory n'a pas non plus de sens. Enfin, nous avons un dernier cas avec les évènements de la forme [_,P,Q,_] : cet évènement représente potentiellement l'activation de n'importe quelle opération avec n'importe quel comportement. Cependant, les activations qui nous intéressent sont celles qui utilisent l'opération de l'évènement de base et ses tags.

Prenons l'exemple de la restriction suivante :

[buyTicket, _, _, {@AIM :BUY_Nominal_Success}]\[_, not(currentUser.oclIsUndefined()), currentUser.oclIsUndefined(),_]

qui représente un achat avec succès de ticket (évènement de base) mais cela ne doit pas déconnecter l'utilisateur (évènement de restriction). Cette restriction peut être simplifiée en ne considérant que l'opération et les tags de l'évènement de base :

[buyTicket, _, _, {@AIM :BUY_Nominal_Success}]\[buyTicket, not(currentUser.oclIsUndefined()), currentUser.oclIsUndefined(),{@AIM :BUY_Nominal_Success}]

Le cas [_,P,Q,T] n'appartient pas au cas précédent. En effet, puisque les tags autorisés sont connus, alors il devient possible de retrouver l'opération à laquelle ils appartiennent et nous pouvons spécifier l'opération de l'évènement.

Définition 44 (Simplification des restrictions)**.** Soit l'évènement de base E=[op,P,Q,T] et sa restriction R=$\{E_1, \ldots, E_n\}$ avec E_k=[op_k,P_k,Q_k,T_k], $k \in 1..n$. La simplification de E\R est définie ainsi :

$$E\backslash R = \{E' | E_k \in R \wedge k \in 1..n \wedge E' = E_k \wedge (op_k = op \wedge T \cap T_k \neq \emptyset) \vee (op_k = \_ \wedge (T_k = \_ \vee T_k \cap T \neq \emptyset))\}$$

Une seule restriction

Intuitivement, un évènement restreint E\{E_1}, avec E=[op,P,Q,T] et E_1=[op,P_1,Q_1,T_1], est activé si et seulement si l'évènement E est activé et si E_1 ne l'est pas. Pour cela, il est nécessaire qu'au moins un des composants de E_1 ne soit pas satisfait par l'appel d'opération pour que l'évènement restreint soit activé. La FIGURE 8.7 présente les différents *cas de restriction* :

	Précondition	**Comportements**	**Postcondition**
0	$P \wedge \neg P_1$	op including T	Q
1	$P \wedge P_1$	op including $T \backslash T_1$	Q
2	$P \wedge P_1$	op including T	$Q \wedge \neg Q_1$

FIGURE 8.7 – Enumération des cas pour une restriction $E \backslash E_1$

	Cas E_1	**Cas E_2**	**Précondition**	**Comportement**	**Postcondition**
0	0	0	$P \wedge \neg P_1 \wedge \neg P_2$	op including T	Q
1	1	0	$P \wedge P_1 \wedge \neg P_2$	op including $T \backslash T_1$	Q
2	2	0	$P \wedge P_1 \wedge \neg P_2$	op including T	$Q \wedge \neg Q_1$
3	0	1	$P \wedge \neg P_1 \wedge P_2$	op including $T \backslash T_2$	Q
4	1	1	$P \wedge \neg P_1 \wedge P_2$	op including T	$Q \wedge \neg Q_2$
5	2	1	$P \wedge P_1 \wedge P_2$	op including $T \backslash \{T_1 \cup T_2\}$	Q
6	0	2	$P \wedge P_1 \wedge P_2$	op including $T \backslash T_1$	$Q \wedge \neg Q_2$
7	1	2	$P \wedge P_1 \wedge P_2$	op including $T \backslash T_2$	$Q \wedge \neg Q_1$
8	2	2	$P \wedge P_1 \wedge P_2$	op including T	$Q \wedge \neg Q_1 \wedge \neg Q_2$

FIGURE 8.8 – Enumération des cas pour $E \backslash \{E_1, E_2\}$

0. ce cas spécifie que l'opération *op* de E avec les comportements T peut être appelée d'un état qui ne vérifie pas P_1. Il est alors possible d'appeler l'opération sans avoir à restreindre la postcondition ni les comportements.
1. ce cas spécifie l'appel de l'opération mais à partir d'un état potentiellement dangereux car il satisfait P_1. Afin d'éviter l'évènement restreint, nous n'autorisons pas les comportements interdits T_1 de E_1. Ainsi, il n'est pas nécessaire de restreindre la postcondition.
2. ce cas spécifie l'appel d'opération à partir d'un état qui satisfait P_1 et active les comportements T_1 de E_1. Ainsi, la postcondition ne doit pas être satisfaite. Dans le cas contraire, l'évènement active E_1, ce que l'on souhaite justement éviter.

Deux restrictions

Nous pouvons étendre la méthode pour prendre en compte deux évènements dans la restriction. Soit la soustraction $E \backslash \{E_1, E_2\}$ avec $E=[op,P,Q,T]$, $E_1=[op,P_1,Q_1,T_1]$ et $E_2=[op,P_2,Q_2,T_2]$, nous pouvons la représenter par l'ensemble d'évènements, numérotés de 0 à 8 dans la FIGURE 8.8.

Dans ce cas, nous faisons la combinaison de tous les *cas de restriction* de chaque évènement de la restriction. Les numéros de la colonne de chaque évènement donne le *cas de restriction* utilisé pour l'énumération de cas considérée. Ainsi, la 6^e énumération représente les *cas de restriction* 0 pour l'évènement E_1 et 2 pour l'évènement E_2.

Généralisation

Nous généralisons l'intuition précédente à plusieurs restrictions pour définir des parties de scénario représentant ces différentes possibilités. Pour cela, nous définissons la fonction ϕ qui permet de récupérer pour la i^{ième} énumération de la restriction, le cas de restriction du k^{ième} évènement de la restriction. Puisque nous avons 3 cas possibles par évènement et que nous faisons la combinatoire de tous ces cas pour le n évènements de la restriction, alors nous avons au total 3^n cas possibles. Ainsi, nous avons $i \in 0..3^n - 1$.

Définition 45 (Fonction de détermination de cas)**.** Soit i le numéro d'énumération des cas de la restriction et k le numéro d'ordre de l'évènement de la restriction considéré. La fonction de sélection de cas ϕ est définie ainsi :

$$\phi(i,k) = \frac{i}{3^{k-1}} \ mod\ 3$$

Exemple 50 (Exemple de détermination de cas)**.** Reprenons l'exemple de la FIGURE 8.8. La colonne de gauche représente l'énumération des restrictions (toutes les combinaisons possibles de restriction). Ce sont ces valeurs qui sont utilisées dans le paramètre i de ϕ. Dans l'exemple, il y a deux évènements dans la restriction, donc le paramètre k de ϕ sera 1 et 2.

Si nous souhaitons obtenir le *cas de restriction* pour l'évènement E_1 ($k = 1$) pour la 7^e énumération de restriction ($i = 7$) alors $\phi(7, 1) = 1$ qui correspond au *cas de restriction* 1 de E_1 où la précondition est satisfaite et les tags doivent être différents. Cette valeur est celle de la case à l'intersection de la ligne portant le numéro 7 et de la colonne de E_1 dans la FIGURE 8.8.

Le *cas de restriction* pour l'évènement E_2 ($k = 2$) pour cette même énumération est donné par $\phi(7, 2) = 2$ qui correspond au cas où la précondition est satisfaite, les tags interdits sont activés, mais la postcondition n'est pas satisfaite. C'est le cas à l'intersection de la ligne portant le numéro 7 et de la colonne de E_2 dans la FIGURE 8.8.

La fonction de détermination de cas étant définie, nous l'utilisons maintenant pour déterminer la traduction en scénario de chaque énumération de cas. Cette traduction permet de combiner les différents *cas d'énumération* des évènements de la restriction pour une énumération donnée en un sous-scénario.

Définition 46 (Traduction du i^{ème} cas de la restriction)**.** Soit i un entier dans $[0..3^n - 1]$, F=[op,P,Q,T]\{[op$_1$,P$_1$,Q$_1$,T$_1$],...,[op$_n$,P$_n$,Q$_n$,T$_n$]}, avec $op \neq \_$, le scénario qui traduit le i^{ième} cas de la restriction F est défini ainsi par $\mathcal{C}$(i, F) :

$$\mathcal{C}(\text{i, F}) = \left(\begin{array}{l} \texttt{assert(P}\wedge \bigwedge_{k\in 1..n \wedge \phi(i,k)\neq 0} \text{P}_k \wedge \bigwedge_{k\in 1..n\wedge\phi(i,k)=0} \neg \text{P}_k\texttt{)} ; \\ \\ \text{op } \texttt{including T}\backslash \bigcup_{k\in 1..n\wedge\phi(i,k)=1} \text{T}_k ; \\ \\ \texttt{assert(Q}\wedge \bigwedge_{k\in 1..n\wedge\phi(i,k)=2} \neg \text{Q}_k\texttt{)} ; \end{array} \right)$$

```
choice{
  assert(P∧¬P1);
  op including T;
  assert(Q)
} or {
  assert(P∧P1);
  op including T\T1;
  assert(Q);
} or {
  assert(P∧P1);
  op including T;
  assert(Q∧¬Q1);
}
```

FIGURE 8.9 – Traduction de la restriction de l'exemple 51

Le domaine $[0..3^n - 1]$ de la variable i permet d'énumérer toutes les configurations possibles de la soustraction (première colonne de la FIGURE 8.7 et de la FIGURE 8.8)

Enfin, nous pouvons maintenant définir la règle de traduction de ce type d'évènement pour l'évènement restreint dans sa globalité.

Définition 47 (Règle de traduction d'une restriction avec opération)**.** Soit la restriction suivante F=[op,P,Q,T]\{[op$_1$,P$_1$,Q$_1$,T$_1$],...,[op$_n$,P$_n$,Q$_n$,T$_n$]} et $op \neq \_$, nous avons :

$$\tau(\mathrm{F}) \rightsquigarrow \textbf{choice}\{\mathcal{C}(0, F)\}\textbf{or}\ldots\textbf{or}\{\mathcal{C}(3^n - 1, F)\}$$

qui représente un choix entre chaque cas de la soustraction.

Exemple 51 (Exemple de traduction d'une restriction d'évènement)**.** Soient les deux évènements E=[op,P,Q,T], E$_1$=[op,P$_1$,Q$_1$,T$_1$]. La traduction de l'évènement E\{E$_1$} dans le langage de scénarios est montré dans la FIGURE 8.9. Pour simplifier l'écriture, la notation `assert`(p$_1$∧... ∧ p$_n$) est une réécriture du scénario `assert`(p$_1$) ; ... ; `assert`(p$_n$) ;.

Ce scénario permet bien d'activer tous les évènements autorisés par E\{E$_1$}

8.2.3 Traduction des restrictions sur un évènement sans opération ni comportement

La dernière règle de traduction concerne les restrictions dont l'évènement de base ne spécifie pas d'opération ni de tags. C'est le cas des évènements issus des *becomesTrue* et des évènements des $\Sigma-$transitions. Il est à noter que si l'évènement possède une liste de tags, alors nous pouvons déduire les opérations qui les définissent et utiliser la règle précédente.

Puisque l'évènement de base autorise potentiellement n'importe quel comportement de n'importe quelle opération, nous rassemblons les évènements de la restriction par opération. L'intuition est de déterminer, pour chaque opération apparaissant dans les évènements restrictifs, les évènements autorisés. Cette traduction rejoint la règle précédente

mais la généralise à toutes les opérations qui apparaissent dans les évènements de la restriction. Nous donnons d'abord une intuition de cette traduction avant de la généraliser.

Intuition

Considérons l'évènement restreint [_, P, Q, _]\{[op_1, P_1, Q_1, T_1]}. Cet évènement représente tous les évènements à partir d'un état satisfaisant P et qui amène dans un état Q. Cependant, si c'est l'opération op_1 qui est appelée, alors nous devons veiller à ce que soit :

— l'état de départ ne satisfait pas P_1,
— l'état d'arrivée ne satisfait pas Q_1,
— aucun tag de T_1 n'a été activé,
— une combinaison des trois précédents a été satisfaite.

Ainsi, l'évènement restreint peut se réécrire de la manière suivante :

$$[\_, P, Q, \_]\backslash\{[op_1, P_1, Q_1, T_1]\} \rightsquigarrow \begin{array}{l} ([op_1, P, Q, \_]\backslash\{[op_1, P_1, Q_1, T_1]\}) \\ \mid [op, P, Q, \_] \end{array}$$

où *op* est n'importe quelle opération du système sauf op_1. Nous avons déjà déterminé la manière de traduire une restriction d'évènement lorsque l'évènement de base porte une opération dans la sous-section précédente.

Nous pouvons étendre à deux évènements dans la restriction. Considérons l'évènement [_, P, Q, _]\{[op_1, P_1, Q_1, T_1], [op_2, P_2, Q_2, T_2]}. Il peut se réécrire de la manière suivante :

$$[\_, P, Q, \_]\backslash\{[op_1, P_1, Q_1, T_1], [op_2, P_2, Q_2, T_2]\} \rightsquigarrow \begin{array}{l} ([op_1, P, Q, \_]\backslash\{[op_1, P_1, Q_1, T_1]\}) \\ ([op_2, P, Q, \_]\backslash\{[op_2, P_2, Q_2, T_2]\}) \\ \mid [op, P, Q, \_] \end{array}$$

Avec cette traduction, nous examinons donc toutes les restrictions, opération par opération, en fonction de celles qui apparaissent dans les évènements de la restriction.

Généralisation

Dans un premier temps, nous définissons une fonction qui nous permet de grouper les évènements de la restriction par opération, ce que nous avons fait plus haut.

Définition 48 (Groupement des évènements de la restriciton avec la même opération)**.** Soit l'ensemble d'évènements suivant {E_1, …, E_n}, avec E_k=[op_k, P_k, Q_k, T_k] et k∈1..n. La fonction de collecte pour une opération *op* est définie ainsi :

$$\mathcal{R}(\{E_1, \ldots, E_n\}, op) = \{E_m \mid m \in 1..n \wedge op_m = op\}$$

Nous pouvons maintenant définir la règle de traduction pour des évènements sans opération ni comportement.

Définition 49 (Traduction des restrictions sur un évènement sans opération ni comportement)**.** Soit $\{E_1, \ldots, E_m\}$ avec E_k=[op_k, P_k, Q_k, T_k] et k∈1..m et soit $op_{m+1}, \ldots, op_n$ l'ensemble des opérations qui n'apparaissent pas dans les évènements E_k. Soit E=[_, P, Q, _]. La soustraction F=E\{$E_1, \ldots, E_n$} est traduite de la manière suivante :

$$\begin{aligned}
\tau(F) = &\textbf{choice}\{\\
&\quad \tau([op_1, P, Q, _] \backslash \mathcal{R}(E_1, \ldots, E_n, op_1))\\
&\}\textbf{or}\{\\
&\quad \ldots\\
&\}\textbf{or}\{\\
&\quad \tau([op_m, P, Q, _] \backslash \mathcal{R}(E_1, \ldots, E_n, op_n))\\
&\}\textbf{or}\{\\
&\quad \tau([op_{m+1}, P, Q, _])\\
&\}\textbf{or}\{\\
&\quad \ldots\\
&\}\textbf{or}\{\\
&\quad \tau([op_n, P, Q, _])
\end{aligned}$$

Exemple 52 (Traduction de l'évènement $\Sigma\backslash\{B, D\}$)**.** Reprenons notre exemple de propriété et considérons la transition réflexive sur l'état 3, $3 \xrightarrow{\Sigma\backslash\{B,D\}} 3$. L'évènement $\Sigma\backslash\{B, D\}$ peut être réécrit de la manière suivante :

$$\begin{aligned}
&\Sigma\backslash\{B, D\} \rightsquigarrow\\
&\quad [logout, _, _, _] \backslash [logout, _, _, \{@AIM : LOG_Logout\}]\\
&\quad | \ [deleteAllTickets, _, _, _] \backslash [deleteAllTickets, \texttt{"not(current_user.oclIsUndefined()) and}\\
&\qquad \texttt{current_user.all_tickets_in_basket} \rightarrow \texttt{size()>0" on "sut"},\\
&\qquad _, \{@AIM : REMALL_Dell_All_Tickets\}]
\end{aligned}$$

La FIGURE 8.10 donne le scénario issu de la traduction précédente. Ce scénario spécifie qu'il est possible d'activer n'importe quelle opération sauf *deleteAllTickets*, sous certaines conditions, et *logout* avec succès. Dans le cas du *logout*, il n'y a aucune contrainte quant à son activation, à part le comportement. Ainsi, nous pouvons appeler *logout* uniquement si le tag activé n'est pas @AIM :LOG_Logout (première alternative). Dans le cas de l'opération *deleteAllTickets*, nous pouvons l'appeler uniquement si la précondition de D n'est pas satisfaite au moment de l'appel (seconde alternative) ou, dans le cas où elle est satisfaite,

```
scenario sigmaBD(){
  choice{
    logout excluding {@AIM:LOGOUT_Success};
  } or {
    assert("not(not(current_user.oclIsUndefined()) and
    current_user.all_tickets_in_basket→size()>0)" on "sut");
    deleteAllTickets including {@AIM:REMALL_Dell_All_Tickets};
  } or {
    assert("not(current_user.oclIsUndefined()) and
    current_user.all_tickets_in_basket→size()>0" on "sut");
    deleteAllTickets excluding {@AIM:REMALL_Dell_All_Tickets};
  } or {
    $OP\{logout, deleteAllTickets};
  }
}
```

FIGURE 8.10 – Traduction de l'évènement $\Sigma\backslash\{B, D\}$

```
scenario sigmaA(){
  choice{
    login excluding {@AIM:LOG_Success};
  } or {
    login including {@AIM:LOG_Success};
    assert("current_user.all_tickets_in_basket→size()=0" on "sut")
  } or {
    $OP\{login};
  }
}
```

FIGURE 8.11 – Traduction de l'évènement $\Sigma\backslash\{A\}$

l'appel est correct uniquement si le comportement @AIM :REMALL_Del_All_Tickets n'est pas activé (troisième alternative). Enfin, il est possible d'appeler n'importe quelle opération du modèle à l'exclusion de *logout* et *deleteAllTickets* (dernière alternative).

8.2.4 Traduction de l'exemple

Reprenons le cas de test produit dans l'exemple 46. Par souci de concision, nous ne traduisons que le préfixe, que nous rappelons ici :

$$\Sigma\backslash\{A\}^*.A.(\Sigma\backslash\{B, D\}^*.(B.\Sigma\backslash\{A\}^*.A)^*)^*$$

Nous traduisons, dans un premier temps, les évènements simples :

A ⇝ login including {@AIM :LOG_Success} ;
assert("current_user.all_tickets_in_basket→size()=0" **on** "sut") ;

B ⇝ logout including {@AIM :LOG_Logout} ;

Ensuite, nous traduisons les évènements restreints $\Sigma\backslash\{A\}$ en FIGURE 8.11 et $\Sigma\backslash\{B, D\}$ en FIGURE 8.10 (que nous avons déjà traduit en fin de section précédente). Prenons la

```
scenario prefixe(){
  unfold between 0 and N times{
    sigmaA();
  };
  login including {@AIM:LOG_Success};
  assert("current_user.all_tickets_in_basket→size()=0" on "sut");
  repeat between 0 and N times{
    unfold between 0 and N times{
      sigmaBD();
    };
    repeat between 0 and N times{
      logout including {@AIM:LOG_Logout};
      repeat between 0 and N times{
        sigmaA();
      };
      login including {@AIM:LOG_Success};
      assert("current_user.all_tickets_in_basket→size()=0" on "sut");
    };
  };
}
```

FIGURE 8.12 – Scénario du préfixe de l'exemple

traduction de l'évènement $\Sigma \backslash \{A\}$ où l'on autorise toutes les opérations exceptée *login*, les appels de *login* qui n'utilisent pas le tag @AIM :LOG_Success ou encore les appels de login avec succès mais avec un panier non-vide. Ce choix garantit l'expression de tous les évènements qui ne correspondent pas à l'évènement A.

Les évènements sont traduits, il ne nous reste plus qu'à traduire la séquence décrite par l'expression régulière. La traduction complète du préfixe du cas de test asbtrait est présentée dans la FIGURE 8.12.

8.3 Synthèse

Nous avons présenté dans ce chapitre la technique de génération guidée par les critères définis dans les chapitres précédents. Nous avons associé à chaque critère un algorithme qui permet de générer un scénario de test permettant de satisfaire un critère ($k-$itération du motif/de la portée) ou l'ensemble des éléments à couvrir pour un critère ($\alpha-$transitions, paires d'$\alpha-$transitions et robustesse). Ces cas de tests sont exprimés sous forme d'expressions régulières qui sont ensuite traduites dans le langage de scénarios décrit au chapitre 4. Cette traduction permet à l'ingénieur de tests, à l'aide d'un formalisme proche d'un langage de programmation, de spécifier plus précisément des parties du scénario en fonction de ses besoins de tests et de son expertise. Il est à noter que ce processus de traduction n'ajoute pas de directives de pilotage au scénario, qui doivent alors être rajoutées par l'ingénieur à la suite de la génération des scénarios, avant la génération de tests à partir de ceux-ci.

Cette dernière étape de génération de cas de tests abstraits permet de terminer la

chaîne de la démarche proposée dans ce document. En effet, nous avons maintenant le processus qui, à partir des propriétés temporelles formalisées à partir de la spécification puis transformées en automates, nous permet de générer des scénarios de tests à partir des critères de sélection sur les automates de propriété. Puis, à partir de ces scénarios, nous pouvons générer des tests avec la méthode présentée au chapitre 4.

Troisième partie

Evaluation expérimentale des contributions

Chapitre 9

Etude de cas : ECinema

Sommaire

9.1 Propriétés considérées 160
9.2 Mesure de couverture 164
9.2.1 Remarques générales 164
9.2.2 Couverture des propriétés 166
9.3 Capacité de détection 167
9.3.1 Protocole expérimental 168
9.3.2 Mutations considérées 170
9.3.3 Analyse 174
9.4 Discussion 175
9.5 Conclusions 177

Ce chapitre décrit l'expérimentation menée sur le modèle eCinema. Nous proposons ici d'évaluer les suites de test générées par notre approche à celles générées par l'outil CertifyIt. Pour cela, nous fondons notre expérimentation sur deux aspects. Le premier est celui de la couverture des automates de propriétés vis-à-vis de nos critères et le second repose sur l'évaluation de la capacité de détection de fautes des suites respectives. A cette fin, nous utilisons un ensemble de mutants sensés représenter des erreurs que l'on pourrait retrouver sur une implémentation. Nous mesurons alors le nombre de mutants éliminés par les suites respectives mais aussi quels mutants ont été éliminés uniquement par l'approche CertifyIt ou la nôtre.

Dans un premier temps, dans la section 9.1, nous décrivons les propriétés considérées dans le cadre de cette expérimentation. Dans la section 9.2, nous comparons les suites de tests en fonction des critères de couverture introduits dans les Chapitres 6 et 7. Ensuite, dans la section 9.3, nous analysons la capacité de détection des mutants de nos suites et et celui de la suite de CertifyIt. Nous décrivons d'abord le protocole expérimental utilisé pour conduire cette expérimentation, puis les différentes familles de mutants considérés avant d'exposer les résultats de cette analyse. Enfin, nous terminons dans la section 9.4 avec un bilan et une discussion sur ce qui ressort de ces expérimentations.

FIGURE 9.1 – Automate de la propriété P_1

9.1 Propriétés considérées

Pour cette expérimentation, nous considérons les exemples de propriétés suivantes. Elles permettent d'illustrer la majorité des constructions proposées dans le langage de propriétés.

Propriété 1

Considérons la propriété suivante : "Entre une authentification réussie et une déconnexion, il y a toujours un utilisateur courant". Elle peut être décrite dans le langage de propriétés de la manière suivante :

```
always ("not(current_user.oclIsUndefined())" on "sut")
after  isCalled(login, including:{@AIM:LOG_Success})
until  isCalled(logout, including:{@AIM:LOG_Logout})
```

L'automate de la sémantique de cette propriété est représenté dans la FIGURE 9.1 avec, pour les évènements A, B et C :

- A = [login, _, _, {@AIM :LOGIN_Success}]
- B = [logout, _, _, {@AIM :LOGOUT_Logout}]
- C = [_,"not(current_user.oclIsUndefined())" on "sut", "not(current_user.oclIsUndefined())" on "sut", _]

Propriété 2

Considérons la propriété suivante : "Pour qu'un utilisateur puisse supprimer avec succès des tickets, il en a nécessairement acheté au moins un auparavant". Cette propriété s'écrit ainsi :

```
eventually isCalled(buyTicket, including:{@AIM:BUY_Nominal_Success
                                          @AIM:BUY_Last_Ticket_Sold})
at least 1 times
before isCalled(deleteTicket, including:{@AIM:REM_Del_Ticket})
```

L'automate de propriété représentant la sémantique est présenté à la FIGURE 9.2 avec, pour les évènements A et B :

- A = [deleteTickets, _, _, {@AIM :REM_Del_Ticket}]
- B = [buyTicket, _, _, {@AIM :BUY_Nominal_Success, @AIM :BUY_Last_Ticket_Sold}]

FIGURE 9.2 – Automate de la propriété P_2

FIGURE 9.3 – Automate de la propriété P_{31}

FIGURE 9.4 – Automate des propriétés P_{32}, P_{33} et P_4

Propriété 3

Considérons la propriété suivante : "Il n'est possible de faire un achat de ticket avec succès que lorsqu'un utilisateur est authentifié". Cette propriété peut s'interpréter de trois manières différentes. La première (P_{31}) correspond à l'achat impossible de tickets lorsqu'aucun utilisateur n'a été authentifié sur le système.

```
never isCalled(buyTicket, including:{@AIM:BUY_Nominal_Success
                                     @AIM:BUY_Last_Ticket_Sold})
before isCalled(login, including:{@AIM:LOG_Success})
```

L'automate de cette propriété est présenté à la FIGURE 9.3 avec :
- A = [login, _, _, {@AIM :LOGIN_Success}],
- B = [buyTicket, _, _, {@AIM :BUY_Nominal_Success, @AIM :BUY_Last_Ticket_Sold}],

La seconde (P_{32}) correspond à cas où un utilisateur est authentifié et où le déni d'achat ne peut pas provenir d'un défaut d'authentification :

```
never isCalled(buyTicket, including:{@AIM:BUY_Login_First})
after isCalled(login, including:{@AIM:LOG_Success})
until isCalled(logout, including:{@AIM:LOG_Logout})
```

La dernière interprétation (P_{33}) correspond au cas où il y a eu un utilisateur authentifié qui s'est ensuite déconnecté. Dans ce cas, l'achat avec succès est impossible tant qu'un utilisateur ne se reconnecte pas avec succès.

```
never isCalled(buyTicket, including:{@AIM:BUY_Nominal_Success
                                     @AIM:BUY_Last_Ticket_Sold})
after isCalled(logout, including:{@AIM:LOG_Logout})
until isCalled(login, including:{@AIM:LOG_Success})
```

Les automates de propriété des deux dernières interprétations sont identiques et sont représentés par l'automate de la FIGURE 9.4 avec, pour les évènements A, B et C de la propriété P_{32} :
- A = [login, _, _, {@AIM :LOGIN_Success}],
- B = [logout, _, _, {@AIM :LOGOUT_Logout}],
- C = [buyTicket, _, _, {@AIM :BUY_Login_First}],

et pour P_{33}, nous avons :
- A = [logout, _, _, {@AIM :LOGOUT_Logout}],
- B = [login, _, _, {@AIM :LOGIN_Success}],
- C = [buyTicket, _, _, {@AIM :BUY_Nominal_Success, @AIM :BUY_Last_Ticket_Sold}].

Propriété 4

Considérons la propriété suivante : "Tant que l'utilisateur est connecté, s'il n'a acheté que des tickets pour le film TITLE1, alors il ne peut pas supprimer avec succès des tickets pour le film TITLE2". Cette propriété se traduit de la manière suivante :

```
never isCalled(deleteTicket,
    pre:"current_user.all_tickets_in_basket→forAll(title=TITLE1) and
    not(current_user.all_ticket_in_basket→isEmpty()) and
    in_title = TITLE2" on "sut", including:{@AIM:REM_Del_Ticket})
after isCalled(login, including:{@AIM:LOG_Success})
until isCalled(logout, including:{@AIM:LOG_Logout})
```

L'automate de la sémantique de cette propriété est représenté dans la FIGURE 9.4 avec, pour les évènements A, B et C :

- A = [login, _, _, {@AIM :LOGIN_Success}]
- B = [logout, _, _, {@AIM :LOGOUT_Logout}]
- C = [deleteTicket, "current_user.all_tickets_in_basket→forAll(title=TITLE1) and not(current_user.all_ticket_in_basket→isEmpty()) and in_title = TITLE2" on "sut", _, {@AIM :REM_Del_Ticket}]

Propriété 5

Considérons la propriété suivante : "Tant qu'un utilisateur est authentifié, celui-ci a la possibilité (mais pas l'obligation) d'acheter des tickets". Cette propriété se traduit de la manière suivante :

```
eventually isCalled(buyTicket, including:{@AIM:BUY_Nominal_Success
                                          @AIM:BUY_Last_Ticket_Sold})
at least 0 times
after isCalled(login, including:{@AIM:LOG_Logout})
until isCalled(logout, including:{@AIM:LOG_Success})
```

L'automate de la sémantique de cette propriété est représenté dans la FIGURE 9.5 avec, pour les évènements A, B et C :

- A = [login,_,_,{@AIM :LOG_Logout}]
- B = [logout,_,_,{@AIM :LOG_Success}]
- C = [buyTicket,{@AIM :BUY_Nominal_Success,@AIM :BUY_Last_Ticket_Sold}]

Propriété 6

Considérons la propriété suivante : "Pour se déconnecter avec succès, un utilisateur doit s'être connecté auparavant". Cette propriété se traduit de la manière suivante :

```
          isCalled(login, including:{@AIM:LOG_Success})
precedes  isCalled(logout, inclusing:{@AIM:LOG_Logout})
globally
```

L'automate de la sémantique de cette propriété est représenté dans la FIGURE 9.6 avec, pour les évènements A et B :

- A = [login,_,_,{@AIM :LOG_Logout}]
- B = [logout,_,_,{@AIM :LOG_Success}]

FIGURE 9.5 – Automate de la propriété P_5

FIGURE 9.6 – Automate de la propriété P_6

Propriété 7

Considérons la propriété suivante : “Lorsque tous les tickets ont été achetés, il est nécessaire d'en supprimer avant de pouvoir en acheter à nouveau”.

Nous pouvons exprimer de deux manières différentes le fait que *tous les tickets ont été achetés*. La première consiste à fonder notre observation sur le nombre de tickets restants indiqués par l'attribut `available_tickets` de chaque film. La seconde consiste à se fonder sur les objets de la classe *Ticket*. L'inconvénient de cette seconde possibilité est que le modèle ne dispose pas d'opération permettant d'observer la quantité de tickets associé à un utilisateur. C'est à cette seconde formulation que nous nous intéressons ici.

Nous pouvons tirer deux interprétations de cette propriété. La première (P_{71}) est la plus intuitive, c'est à dire qu'il y a au moins un appel à *deleteTicket* ou à *deleteAllTickets* avec succès avant un nouvel achat avec succès. Cette interprétation s'exprime de la manière suivante :

```
after isCalled(buyTicket,
  post:"Ticket.allInstances()→forAll(not(owner_ticket.oclIsUndefined()))
  and Ticket.allInstances()→forAll(not(movie.oclIsUndefined()))"
  on "sut", including:{@AIM:BUY_Last_Ticket_Sold})
until isCalled(buyTicket,
  including:{@AIM:BUY_Last_Ticket_Sold,@AIM:BUY_Nominal_Success})
eventually isCalled(deleteTicket, including:{@AIM:REM_Del_Ticket})
  || isCalled(deleteAllTicket, including:{@AIM:REM_Del_All_Tickets})
at least 1 times
```

L'automate de cette interprétation est donné en FIGURE 9.7 avec, pour les évènements A, B, C et D :

- A = [buyTicket,_,"Ticket.allInstances()→forAll(not(owner_ticket.oclIsUndefined()) and Ticket.allInstances()→forAll(not(movie.oclIsUndefined()))" on "sut"
- B = [buyTicket,_,_,{@AIM :BUY_Last_Ticket_Sold,@AIM :BUY_Nominal_Success}]
- C = [deleteTicket,_,_,{@AIM :REM_Del_Ticket}]
- D = [deleteAllTicket,_,_,{@AIM :REM_Del_All_Tickets}]

La seconde interprétation (P_{72}) met plus en lumière le rôle de *deleteTicket* et de *deleteAllTickets* dans un possible échec de la propriété : si tout les tickets ont été achetés et que l'utilisateur courant possède des tickets (sinon, l'échec de la suppression est légitime), alors le fait que le système ne libère pas les tickets supprimés est une violation de la propriété. Nous pouvons exprimer cela de la manière suivante :

```
after isCalled(buyTicket,
   post:"Ticket.allInstances()→forAll(not(movie.oclIsUndefined())) and
   Ticket.allInstances()→forAll(not(owner_ticket.oclIsUndefined()))"
   on "sut", including:{@AIM:BUY_Last_Ticket_Sold})
until isCalled(buyTicket,
   including:{@AIM:BUY_Last_Ticket_Sold,@AIM:BUY_Nominal_Success})
never isCalled(deleteTicket,
   pre:"current_user.all_tickets_in_basket>0" on "sut",
   post:"Ticket.allInstances()→forAll(not(movie.oclIsUndefined())) and
```

FIGURE 9.7 – Automate de la propriété P_{71}

FIGURE 9.8 – Automate de la propriété P_{72}

```
Ticket.allInstances()→forAll(not(owner_ticket.oclIsUndefined()))"
on "sut", including:{@AIM:REM_Del_Ticket})
  ||isCalled(deleteAllTicket,
pre:"current_user.all_tickets_in_basket>0" on "sut",
post:"Ticket.allInstances()→forAll(not(movie.oclIsUndefined())) and
Ticket.allInstances()→forAll(not(owner_ticket.oclIsUndefined()))"
on "sut", including:{@AIM:REM_Del_All_Tickets})
```

L'automate de cette interprétation est donné en FIGURE 9.8 avec, pour les évènements A, B, C et D :

- A = [buyTicket,_,"Ticket.allInstances()→forAll(not(movie.oclIsUndefined())), and Ticket.allInstances()→forAll(not(owner_ticket.oclIsUndefined()))" on "sut", {@AIM :BUY_Last_Ticket_Sold}]
- B = [buyTicket,_,_,{@AIM :BUY_Last_Ticket_Sold,@AIM :BUY_Nominal_Success}]
- C = [deleteTicket,"current_user.all_tickets_in_basket>0" on "sut", "Ticket.allInstances()→forAll(not(movie.oclIsUndefined())) and Ticket.allInstances()→forAll(not(owner_ticket.oclIsUndefined()))" on "sut", {@AIM :REM_Del_Ticket}]
- D = [deleteAllTicket,"current_user.all_tickets_in_basket>0" on "sut", "Ticket.allInstances()→forAll(not(movie.oclIsUndefined())) and Ticket.allInstances()→forAll(not(owner_ticket.oclIsUndefined()))" on "sut", {@AIM :REM_Del_All_Ticket}]

9.2 Mesure de couverture

Nous étudions dans cette section la couverture des différentes suites de tests issues de notre méthode et celle de CertifyIt.

9.2.1 Remarques générales

Dans cette section et la suivante, nous utilisons les tests générés par notre approche afin d'évaluer leur couverture. Pour cela, nous ne considérons que le premier test issu du dépliage de chaque scénario pour deux raisons. La première est le temps de génération. En effet, il n'y a pas besoin de générer plusieurs tests qui permettent la couverture des mêmes éléments. La seconde est la simplicité des tests ce qui permet de limiter les appels à des opérations qui ne sont pas nécessaires. Dans la plupart des cas, cela nous évite le dépliage des répétitions et des évènements de type Σ. De plus, cela permet de réduire l'appel à des opérations qui n'ont aucun rapport avec les propriétés considérées, limitant ainsi la taille des tests.

Lors de la mesure de couverture des mutations du critère de robustesse pour chaque propriété, nous considérons systématiquement l'automate muté qui illustre cette mutation. Cela nous permet de capturer l'activation de l'évènement muté par une transition spécifique de l'automate muté au lieu d'une transition portant un évènement plus général de l'automate original (le plus souvent, une Σ-transition).

Enfin, il est à noter que pour la couverture des k-itérations de la portée, deux métriques sont données : la première donne la couverture de toutes les séquences que propose le critère, la seconde, donnée entre parenthèses, donne la couverture des séquences effectivement réalisables sur l'automate. Par exemple, en reprenant l'automate de la FIGURE 9.4,

Propriétés	**Critères**	α	α^2	**2-***scope*	**2-***pattern*	**Robustesse**	**CertifyIt**
P_1	α	4/4	4/4	4/4			3/4
	α^2	5/7	7/7	7/7			3/7
	1s	1/2 (1/1)	1/2 (1/1)	1/2 (1/1)			1/2 (1/1)
	2s	1/6 (1/2)	2/6 (2/2)	2/6 (2/2)			1/6 (1/2)
P_2	α	3/3	3/3			1/3	2/3
	α^2	4/4	4/4			0/4	1/4
	μ	0/2	0/2			2/2	2/2
P_{31}	α	1/1				1/1	1/1
	α^2	2/2				2/2	2/2
P_{32}	α	3/3	3/3	3/3		1/3	2/3
	α^2	2/3	3/3	3/3		0/3	1/3
	1s	1/2 (1/1)	1/2 (1/1)	1/2 (1/1)		0/2 (0/1)	1/2 (1/1)
	2s	1/6 (1/2)	2/6 (2/2)	2/6 (2/2)		0/6 (0/2)	1/6 (1/2)
	μ	2/3	2/3	2/3		3/3	3/3
P_{33}	α	3/3	3/3	3/3		1/3	1/3
	α^2	2/3	3/3	3/3		0/3	0/3
	1s	1/2 (1/1)	1/2 (1/1)	1/2 (1/1)		0/2 (0/1)	0/2 (0/1)
	2s	1/6 (1/2)	2/6 (2/2)	2/6 (2/2)		0/6 (0/2)	0/6 (0/2)
	μ	2/2	2/2	2/2		2/2	0/2
P_4	α	3/3	3/3	3/3		1/3	2/3
	α^2	2/3	3/3	3/3		0/3	1/3
	1s	1/2 (1/1)	1/2 (1/1)	1/2 (1/1)		0/2 (0/1)	1/2 (1/1)
	2s	1/6 (1/2)	2/6 (2/2)	2/6 (2/2)		0/6 (0/2)	1/6 (1/2)
	μ	0/6	0/6	0/6		6/6	0/6
P_5	α	4/4	4/4	4/4			3/4
	α^2	6/6	6/6	6/6			3/6
	1s	1/2 (1/1)	1/2 (1/1)	1/2 (1/1)			1/2 (1/1)
	2s	1/6 (1/2)	2/6 (2/2)	2/6 (2/2)			1/6 (1/2)
P_6	α	2/2	2/2		2/2	0/3	2/2
	α^2	1/2	2/2		2/2	0/3	1/2
	1p	1/1	1/1		1/1	0/1	1/1
	2p	1/2	2/2		2/2	0/2	1/2
	μ	2/2	2/2		2/2	2/2	2/2
P_{71}	α	5/5	5/5	5/5		1/5	1/5
	α^2	5/7	7/7	7/7		0/7	0/7
	1s	1/2 (1/1)	1/2 (1/1)	1/2 (1/1)		0/2 (0/1)	0/2 (0/1)
	2s	1/6 (1/2)	1/6 (1/2)	2/6 (2/2)		0/6 (0/2)	0/6 (0/2)
	μ	0/2	0/2	0/2		2/2	0/2
P_{72}	α	3/3	3/3	3/3		1/3	1/3
	α^2	2/3	3/3	3/3		0/3	0/3
	1s	1/2 (1/1)	1/2 (1/1)	1/2 (1/1)		0/2 (0/1)	0/2 (0/1)
	2s	1/6 (1/2)	2/6 (2/2)	2/6 (2/2)		0/6 (0/2)	0/6 (0/2)
	μ	0/6	0/6	0/6		6/6	0/6

FIGURE 9.9 – Résultats de la couverture

nous nous apercevons que la séquence $[(3 \xrightarrow{A} 2,2 \xrightarrow{B} 3)\,;\,(1 \xrightarrow{A} 2,2 \xrightarrow{B} 3)]$ n'est pas possible car il n'y a aucun moyen de retourner à l'état 1 après l'avoir quitté.

9.2.2 Couverture des propriétés

Nous présentons maintenant la couverture de chaque critère associé à chaque propriété en fonction des suites de tests produites par notre démarche et celle de CertifyIt. De plus, seules les suites issues de la propriété considérée sont utilisées pour la couverture (en plus de celle de CertifyIt). Le tableau de la FIGURE 9.9 présente le résultat de la couverture de chaque propriété par les suites des tests.

En examinant cette couverture, nous nous apercevons que la suite de CertifyIt couvre partiellement, et de manière inégale, les critères de nos propriétés. Ainsi, la propriété P_{31} est entièrement couverte, alors que la couverture de la propriété P_{32} se révèle être très faible. Dans la plupart des cas, c'est l'évènement de connexion après une déconnexion réussie qui fait défaut. L'unique test de CertifyIt impliquant une déconnexion réussie avec l'opération *logout* ne permet pas cette reconnexion. En effet, puisque ces tests activent les chemins les plus courts pour atteindre un comportement donné, alors l'activation du comportement de connexion avec succès de l'opération *login* peut être déclenché dès l'état initial. Ainsi, le générateur n'essaiera pas d'atteindre ce comportement après une déconnexion.

Un effet similaire est observé avec les deux interprétations de la propriété 7, P_{71} et P_{72}. Dans ce cas, il s'agit du manque de couverture des opérations après avoir atteint le comportement @AIM:BUY_Sold_Out de *buyTicket*. En effet, seule la transition représentant l'activation de la transition $1 \xrightarrow{A} 2$ des automates 9.7 et 9.8) de ce comportement est couverte, mais l'unique test qui l'active n'appelle aucune opération après cela. Ainsi, les autres transitions de l'automate ne sont jamais couvertes.

En général, ce manque de couverture est principalement dû à la technique de génération de tests de CertifyIt. Lorsque celui-ci tente d'atteindre un comportement, le moteur de génération cherche le chemin le plus court. Cet effet est surtout visible par la couverture de la propriété P_5. Bien que les tests de CertifyIt couvrent trois des quatre α-transitions de l'automate, ils ne permettent de couvrir que la moitié des paires.

La non couverture d'une de ces $\alpha-$transitions entraîne nécessairement un manque de couverture pour le critère *all-α-pairs*, ce que nous pouvons observer en particulier sur la suite de CertifyIt. Il en va de même avec le critère *k-scope* pour les propriétés P_1, P_{32}, P_{33}, P_4, P_5 et P_6 dont la reconnexion n'est jamais activée, empêchant la couverture de la seconde itération.

Les suites issues des critères permettent de couvrir les manques laissés par CertifyIt, offrant ainsi une complémentarité à cette suite de tests vis-à-vis des propriétés considérées. En plus de la complémentarité de nos tests pour la suite de CertifyIt, nous avons une complémentarité entre les suites issues des critères d'une même propriété. La propriété P_{71} en est un exemple. En effet, la suite issue de *all-α* couvre bien sûr toutes les $\alpha-$transition, mais pas toutes les paires, ni toutes les itérations de la portée. La suite issue de *all-α-pairs* permet la couverture des paires manquantes, sans toutefois couvrir la seconde itération de la portée qui est, elle, couverte par la suite issue du critère *k-scope*.

Concernant les mutations d'évènements, les évènements issus des règles μ_{tag-} et μ_{pre-}

sont des affaiblissements très forts sur un évènement. Ainsi, la plupart du temps ces évènements mutants sont couverts par les suites nominales et celle de CertifyIt, ce qui est notamment illustré avec les propriétés P_{32}, P_{33} et P_6.

Les autres mutations, et plus particulièrement μ_{prop-}, illustrées dans les propriétés P_4 et P_{72}, ne sont couvertes que par les suites de robustesse des propriétés. En effet, ces nouveaux évènements représentent des appels d'opération dans des contextes très particuliers (représentés par les préconditions). Les suites nominales et celle de CertifyIt ne permettent pas nécessairement l'activation des évènements dans ces contextes particuliers. Les suites de robustesse offrent ainsi une complémentarité supplémentaires en visant des évènements qui nécessitent un contexte particulier.

9.3 Capacité de détection

Nous analysons dans cette section la capacité de détection des fautes par des suites de tests vis-à-vis des propriétés définies précédemment.

Cette analyse se fonde sur l'utilisation de modèle mutants sur lesquels sont animés les tests. Si, lors de l'animation de ces tests, les résultats obtenus ne correspondent pas à ceux attendus, alors une non-conformité est détectée. Nous avons choisi d'effectuer cette analyse par mutation du modèle original au lieu d'une comparaison avec des implémentations fautives pour deux raisons. La première est que nous restons au même niveau d'abstraction tout le long du processus. Nous évitons ainsi les problèmes liés aux possibles biais dus à la concrétisation des tests d'une part, et aux possibles différences des capacités d'observation entre le système réel et le modèle.

La seconde est que les mutations sur le modèle permettent de représenter des modèles de faute facilement transposable à un système de test réel. Le fait d'effectuer cette analyse sur le modèle nous permet de nous abstraire des possibles fautes liées à des détails d'implémentation qui ne sont pas représentés dans le modèle et qui sont donc plus difficilement détectables.

Similairement à l'étape d'évaluation de la couverture de la section précédente, nous ne considérons ici que le premier test issu du dépliage de chaque scénario. Cela nous permet de ne pas ajouter de *bruit* lors de la détection, i.e. en détectant des mutations avec des évènements qui n'ont aucun rapport avec la propriété.

Concernant les observations, nos tests font systématiquement appels à chaque opération d'observation du modèle après chaque appel d'opération. La suite de CertifyIt, quant à elle, fait appel aux opérations d'observation de façon plus intelligente en appellant uniquement les opérations d'observation dont la variable d'état observée a pu être modifiée par l'opération précédente.

Enfin, nous faisons l'hypothèse que le modèle respecte toujours les spécifications des propriétés. Par extension, le modèle respecte les propriétés décrites dans le langage de propriétés si celles-ci sont l'expression des propriétés de la spécification. C'est-à-dire que s'il y a non conformité entre le modèle original et un test issue d'une des propriétés, alors la faute se situe au niveau de la propriété.

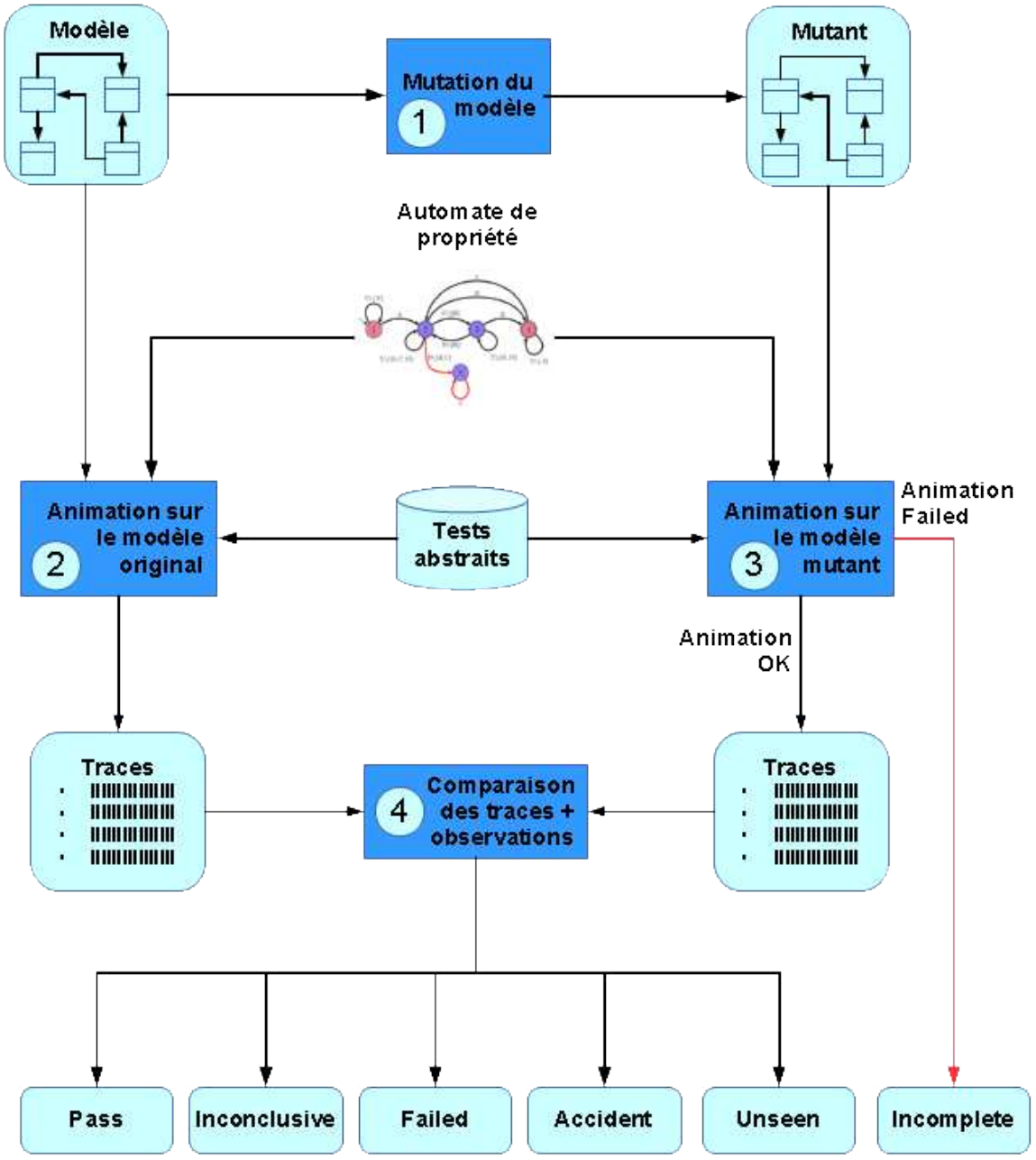


FIGURE 9.10 – Processus du protocole expérimental

9.3.1 Protocole expérimental

Afin d'effectuer la comparaison entre une suite de tests fonctionnels (celle de CertifyIt dans notre étude) et celles produites par nos travaux, nous proposons une approche qui vise à quantifier la capacité de détection de notre approche vis-à-vis de celle de CertifyIt sur nos propriétés. Pour cela, nous proposons le processus expérimental décrit à la FIGURE 9.10.

Notre approche se repose sur l'utilisation de modèles mutants dérivés du modèle original dont les règles de mutations sont présentées dans la sous-section 9.3.2. Ces mutants sont le reflet de fautes que nous pourrions retrouver dans une implémentation.

Pour la comparaison, notre approche se fonde sur la comparaison de traces d'animation des tests sur le modèle original d'une part, et sur un modèle muté d'autre part. Une trace d'animation est définie comme une séquence d'éléments définis par :

— l'opération appellée avec ses paramètres,
— le comportement activé dans cette opération,
— la transition de l'automate de propriété traversée par cet appel d'opération,
— les valeurs de retour des opérations d'observation qui suivent cette opération.

Les traces d'animation étant définies, nous pouvons maintenant donner le protocole. Dans un premier temps (point 1), nous générons un ensemble de mutants à partir du modèle original avec les opérateurs de mutations définis à la sous-section 9.3.2. Ensuite au point 2, nous animons tous les tests dont nous disposons sur le modèle original afin d'en extraire, pour chaque test, la séquence des transitions activées sur l'automate de propriété et l'oracle associé. Il est à noter que les tests peuvent provenir de différentes origines, telle qu'une suite de tests manuels ou une suite générée automatiquement par un autre outil. Dans notre cas, nous considérons les suites de tests issues de notre démarche pour chaque propriété et la suite issue de CertifyIt. Ensuite, nous procédons exactement de la même manière pour le point 3, mais en utilisant un modèle muté au lieu de l'original. Avec certains mutants, l'animation peut échouer sans pouvoir terminer l'exécution du test en cours. Ces cas de tests sont classés séparément dans la catégorie *incomplete*. Pour les autres, au point 4, nous comparons pour chaque propriété et chaque test les traces obtenues sur les modèles mutants (point 3) avec celles obtenues sur le modèle original (point 2). En plus de comparer les traces, nous comparons aussi les valeurs de retour des points d'observation. Ces points d'observations sont les opérations d'observation définies par l'ingénieur de test dans le modèle (paragraphe *Opérations de classe*, section 3.1.1.0 du Chapitre 3) et doivent être le reflet des capacités d'observation attendues sur le système sous test réel.

Pour effectuer la comparaison de ces traces, nous nous appuyons sur cinq critères :

A : l'identité des séquences de transitions traversées,
B : l'identité des des valeurs de retour des opérations d'observation,
C : le passage par un état final,
D : l'arrivée dans un état d'erreur,
E : l'identité des comportements activés pour chaque opération.

Nous pouvons ensuite établir un verdict de la comparaison entre deux traces vis-à-vis de ces critères. Pour cela, nous distinguons cinq verdicts, en plus du verdict *incomplete* décrit précédemment :

— **Pass** (A∧B∧C∧¬D∧E) : ce verdict représente le cas où les traces sont identiques, à la fois au niveau des transitions traversées par les différents appels d'opération, des valeurs de retour des opérations d'observation et des comportement activés. De plus, au moins un état final a été traversé et l'état d'erreur n'a pas été atteint. Ce verdict indique que ce test ne permet pas de détecter le mutant mais qu'il permet d'illustrer la propriété.

— **Inconclusive** (A∧B∧¬C∧¬D∧E) : ce verdict est similaire au précédent, mais dans ce cas, aucun état final n'a été atteint. Ainsi, le test n'est pas considéré comme pertinent pour la propriété et ne permet pas de détecter le mutant. C'est typiquement le cas des tests qui bouclent sur l'état initial de l'automate tel que celui de la propriété P_1.
— **Failed** (¬A∧¬B∧D) : ce verdict indique qu'une non conformité a été détectée au niveau des valeurs de retour des opérations d'observation et l'état d'erreur a été atteint (ce qui implique implicitement que les séquences de transitions divergent). Les tests qui présentent ce verdict sont non seulement considérés pertinents pour la propriété mais permettent de détecter une des fautes relative à cette propriété.
— **Accident** (¬B∧¬D) : ce verdict indique qu'une non conformité a été détectée entre les deux traces au moins au niveau des points d'observations sans toutefois arriver dans un état d'erreur. Le passage par un état final et l'identité des séquences de transitions ne sont pas considérés dans ce verdict. Un test qui présente ce verdict à l'exécution détecte bien une non conformité mais cette détection est néanmoins plus faible qu'avec le verdict précédent, car elle représente la détection d'une faute qui n'est pas directement liée à la propriété.
— **Unseen** : ce verdict correspond à tous les autres cas. En particulier, les tests où aucune non conformité n'est relevé au niveau des opérations d'observation mais dont les séquences de transitions divergent ou les comportements activés ne sont pas identiques. En général, ce cas peut être imputé à un manque de points d'observation fournis par le système sous test (le modèle dans notre cas).

9.3.2 Mutations considérées

Nous décrivons maintenant les différents opérateurs de mutation sur le modèle pour la génération de mutants. Pour cela, nous nous inspirons de des mutations décrites par Black *et al.* dans [BOY00] et nous ajoutons d'autres mutations spécifiques aux constructions du langage OCL.

Remplacement d'opérateurs arithmétiques (AOR) : Ces mutations permettent de remplacer les opérateurs arithmétiques sur les entiers par d'autres opérateurs arithmétiques. Il est à noter que nous ne considérons pas toutes les combinaisons possibles : ces règles représentent les erreurs les plus fréquentes dans l'écriture de code OCL.

Le tableau de la FIGURE 9.11 donne les différentes règles pour cette catégorie de mutations.

Remplacement d'opérateurs relationnels (ROR) : Ces mutations permettent de remplacer les opérateurs relationnels par les opérateurs relationnels opposés. Cette mutation est à mettre en rapport avec la mutation *Relational Operator Replacement (RRO)* décrite par Black *et al.*.

Le tableau de la FIGURE 9.11 donne les différentes règles pour cette catégorie de mutations.

Remplacement d'opérateurs relationnels aux limites (CBOR) : Ces mutations permettent de remplacer les opérateurs relationnels par l'opérateur qui inverse la compréhension de la borne. Par exemple, dans $x < 2$, la borne 2 n'est pas comprise dans ces cas, et cette expression est alors transformée en $x \leq 2$ qui maintenant considère

la borne. Cette mutation est à mettre en rapport avec la mutation *Relational Operator Replacement (RRO)* décrite par Black *et al.*.

Le tableau de la FIGURE 9.11 donne les différentes règles pour cette catégorie de mutations.

Remplacement d'opérateurs ensemblistes simples (SSOR) : Ces mutations permettent de permuter les opérateurs ensemblistes *simples* d'OCL (*includes*, *excludes*, *includesAll* et *excludesAll*). Il est à noter que les règles de mutation impliquant l'opérateur *excludesAll* ne sont applicables que dans un contexte d'évaluation, *excludesAll* ne pouvant pas apparaître dans un contexte d'action.

Le tableau de la FIGURE 9.11 donne les différentes règles pour cette catégorie de mutations.

Remplacement d'opérateurs ensemblistes itératifs (ISOR) : Ces mutations permettent de permuter les opérateurs ensemblistes *itératifs* d'OCL (*exists* et *forAll*). Il est à noter que ces règles ne s'appliquent que dans un contexte de contrainte, l'opérateurs *exists* ne pouvant pas être utilisé dans un contexte d'action.

Le tableau de la FIGURE 9.11 donne les différentes règles pour cette catégorie de mutations.

Remplacement d'opérateurs logiques (LOR) : Ces mutations permettent la permutation des opérateurs logiques. Il est à noter que ces règles ne s'appliquent que dans un contexte de contrainte. Bien que l'opérateur *and* apparaisse dans un contexte d'action, sa sémantique est totalement différente.

Le tableau de la FIGURE 9.11 donne les différentes règles pour cette catégorie de mutations.

Remplacement d'opérandes (OR) : Ces mutations permettent de remplacer les littéraux apparaissant dans le code par d'autres littéraux. Dans le cas d'un littéral d'énumération, celui-ci est remplacé par un autre littéral de la même classe d'énumération. Dans le cas d'un booléen, celui-ci est remplacé par son opposé.

Négation de condition (SNO) : Ces mutations permettent la négation d'une proposition atomique dans une expression booléenne. Par exemple, l'application de cette mutation sur l'expression *(A or B) and C* produira les trois mutants *(not(A) or B) and C*, *(A or not(B)) and C* et *(A or B) and not(C)*. Cette mutation correspond à la mutation *Simple Expression Negation Operator (SNO)* de Black *et al.*.

Condition manquante (MC) : Cette mutation permet la suppression d'une proposition atomique d'une expression booléenne dans un contexte de contraintes. Elle est comparable à la mutation *Missing Condition Operator (MCO)* de Black *et al.*.

Blocage à 0 (SA0) : Cette mutation permet de remplacer une expression entière par 0 (la valeur de l'expression est *bloquée* à 0). Par exemple, l'expression $a + b$ produira les mutants $a + 0$, $0 + b$ et 0.

Blocage à 1 (SA1) : Cette mutation, similaire à la précédente, permet de remplacer une expression entière par 1 (la valeur de l'expression est *bloquée* à 1). Par exemple, l'expression $a + b$ produira les mutants $a + 1$, $1 + b$ et 1.

Blocage à false (SAF) : Cette mutation permet de remplacer une expression booléenne par *false* (la valeur de l'expression est *bloquée* à *false*). Par exemple, l'expression *a and b* produira les trois mutants *false and b*, *a and false* et *false*. Cependant, cette mutation ne peut s'appliquer que dans un contexte de contraintes. Puisque les actions

Mutations	Opérateur original	Opérateur muté
AOR	+	-
	-	+
	*	/
	/	*
ROR	>	<
	≥	≤
	<	>
	≤	≥
CBOR	>	≥
	≥	>
	<	≤
	≤	<
ISOR	exists	forAll
	forAll	exists

Mutations	Opérateur original	Opérateur muté
SSOR	includes	excludes
	includesAll	excludesAll
	excludes	includes
	excludesAll	includesAll
	union	intersection
	intersection	union
	isEmpty	notEmpty
	notEmpty	isEmpty
LOR	and	or
	and	xor
	or	and
	or	xor
	xor	and
	xor	or

FIGURE 9.11 – Règles de mutation des remplacements d'opérateurs

sont considérées comme des expressions booléennes, alors cette mutation entraînerait nécessairement une action *false* qui rendrait le comportement inactivable.

Blocage à true (SAT) : Cette mutation, similaire à la précédente permet de remplacer une expression booléenne par *true* (la valeur de l'expression est *bloquée* à *true*). Par exemple, l'expression *a and b* produira les trois mutants *true and b*, *a and true* et *true*. Nous ne considérons cette mutation que dans un contexte de contraintes. Puisque les actions sont considérées comme des expression booléennes, alors cette mutation entraînerait nécessairement une action *true* qui consiste à supprimer l'action, ce qui n'est pas le but de cette mutation.

Suppression d'action (AD) : Cette mutation permet de supprimer une action d'un comportement. Elle ne s'applique que dans un contexte d'action et permet de remplacer une action par *true*. De plus, elle ne permet pas de supprimer les actions qui affectent une valeur au paramètre de retour de l'opération. Dans le cas contraire, le modèle ne serait alors plus valide.

Nous avons volontairement considéré toutes ces catégories, dont nous savons que certaines ne permettent pas la violation des propriétés proposées, dans le but de ne pas biaiser l'expérimentation. Les tableaux de la FIGURE 9.12 récapitulent le nombre de mutations par catégorie pour chaque opération. Nous pouvons remarquer que la mutation de remplacement d'opérandes (OR) produit un grand nombre de mutants. Un grand nombre de ces mutants remplace un opérande de type *MESSAGE* par une des 15 autres valeurs de l'énumération. Ainsi, pour chaque apparition d'un opérande de ce type dans une opération, l'opérateur OR crée 15 mutants, d'où l'explosion du nombre de mutants produits par cet opérateur.

Opérations \ Mutants	**AOR**	**ROR**	**CBOR**	**SSOR**	**ISOR**	**LOR**	**OR**	**Total**
buyTicket	1	1	1	2	0	0	62	67
deleteAllTickets	1	0	0	1	0	0	62	64
deleteTicket	1	0	0	1	0	0	62	64
goToHome	0	0	0	0	0	0	21	21
goToRegister	0	0	0	0	0	0	34	34
showBoughtTickets	0	0	0	0	0	0	34	34
login	0	0	0	0	1	0	94	95
logout	0	0	0	0	0	0	47	47
register	0	0	0	1	1	2	83	87
unregister	1	0	0	1	0	0	47	49
Total	4	1	1	6	2	2	546	562

Opérations \ Mutants	**SNO**	**MC**	**SA0**	**SA1**	**SAT**	**SAF**	**AD**	**Total**
buyTicket	7	0	4	4	7	7	4	33
deleteAllTickets	3	0	2	2	3	3	5	18
deleteTicket	6	0	2	2	6	6	3	25
goToHome	2	0	0	0	2	2	1	7
goToRegister	1	0	0	0	1	1	1	4
showBoughtTickets	1	0	0	0	1	1	1	4
login	7	0	0	0	7	7	1	22
logout	2	0	0	0	2	2	1	7
register	10	4	0	0	10	10	2	36
unregister	3	0	3	3	2	2	5	18
Total	42	4	11	11	41	41	24	174

FIGURE 9.12 – Nombres de mutations par opération et par type de mutations

Suites	**Nôtre**						**CertifyIt**					
Verdicts / Mutants	P	I	A	F	U	N	P	I	A	F	U	N
AOR			4						4			
ROR			1						1			
CBOR			1						1			
SSOR	2		1	1	2		4		1	1		
ISOR	2						2					
LOR	2		1			1	2		1			1
OR	304		242				2		542		2	
SNO	3		25	2		4			28		2	4
MC	4						2		2			
SA0			10						10			
SA1			10						10			
SAT	18		4				6		16			
SAF	6		25	1		9			31	1		9
AD	7		12		4		15		8			

FIGURE 9.13 – Verdicts d'élimination des mutants par famille de mutations et par suite de tests

9.3.3 Analyse

Dans un soucis de place et de lisibilité, nous abrégeons les verdicts *Pass* par P, *Inconclusive* par I, *Accident* par A, *Failed* par F, *Unseen* par U et *Incomplete* par N (non-animable).

Dans une suite de tests, deux tests peuvent être étiquetés par deux verdicts différents pour le même mutant. Afin d'établir un verdict global pour la suite de tests considérée, nous proposons la relation d'ordre totale stricte sur les verdicts définie de la manière suivante :

$$N > F > I > A > U > P > I$$

Pour un mutant donné, le verdict associé à la suite de tests considérée est le plus *grand* verdict associé aux tests de la suite. Cette relation signifie, par exemple, que si dans la suite de tests considérée et pour un mutant donné il existe un test dont le verdict est *Accident* et un autre test obtient un verdict *Pass*, alors la suite obtient le verdict *Accident* pour le mutant considéré car $A > P$.

La FIGURE 9.13 montre la classification de chaque mutant en fonction de leur famille et du verdict obtenu par les suites de tests obtenues par nos suites de tests d'une part, et par la suite de CertifyIt d'autre part. Cette représentation nous permet d'identifier une première tendance sur les liens entre les familles de mutants et les suites de tests considérées.

La suite de CertifyIt, grâce à sa stratégie fondée sur le critère DC/C, détecte très bien

les mutants qui interviennent au niveau des décisions des opérations, tant que le modèle fournit des opérations d'observation qui permettent d'établir la relation de conformité. En effet, dans le modèle eCinema, la plupart des mutants détectés le sont par l'observation du message retourné par chaque opération. Ainsi, les mutants issus des règles OR qui modifient le message de retour, et les mutants SNO, SAT et SAF (lorsqu'ils sont appliqués sur les conditions des constructions `if-then-else`) qui modifient le comportement, et par conséquent le message associé, sont tous détectés par CertifyIt.

Toutefois, la suite de CertifyIt est faible lorsque les mutants portent sur des actions où les objets modifiés ne sont pas utilisés dans les décisions. C'est typiquement le cas des objets de la classe *Ticket* où les objets sont associés au film et à un utilisateur lors de l'achat d'un ticket et isolés lorsqu'ils sont supprimés. Ces objets et les collections qui les utilisent (*all_tickets_in_basket* de *User* et *all_sold_tickets* de *Movie*) ne sont pas utilisés dans les décisions des opérations. De plus, le fait que le modèle ne fournisse pas de moyen d'observer ces objets ou ces collections rend la détection d'autant plus difficile.

C'est avec la famille de mutants AD (suppression d'action) que nos suites de tests s'illustrent comme le montre les 12 mutants détectés et 4 mutants détectés mais qui n'ont pas pu être observés, alors que CertifyIt n'en détecte que 8 sur les 23. Les mutations SSOR ou SNO, SAT et SAF, lorsqu'elles sont appliquées sur des conditions d'opérateurs ensemblistes, sont aussi détectées par nos suites où quatre d'entre elles permettent la violation des propriétés (2 pour SNO, 1 pour SSOR et 1 pour SAF).

Cette différence vis-à-vis de la suite de CertifyIt s'explique par le fait que nos suites de tests se focalisent sur les enchaînements d'opérations, les répétitions d'opérations *login* et *logout* en est un exemple récurrent. Par exemple, si l'utilisateur n'est pas réellement déconnecté du système malgré une déconnexion réussie (suppression de l'action `current_user.oclIsUndefined()` de *logout*), alors il n'est pas possible d'effectuer une connexion avec succès car l'utilisateur est toujours présent dans le système.

9.4 Discussion

Nous proposons maintenant une discussion sur différents aspects de ces expérimentations et ce qu'il en ressort.

Couverture des propriétés

Dans la section 9.2, nous avons étudié la couverture des suites de tests sur un ensemble de propriétés vis-à-vis des critères de couverture définis dans le Chapitre 6 et le Chapitre 7. La suite de CertifyIt couvre de manière inégale les différents critères sur les propriétés. Sur certaines propriétés, cette suite permet une couverture totale des critères, alors que les taux de couverture peuvent être très faibles sur d'autres propriétés.

Dans la majorité des cas de notre expérimentation, les manques de couverture sont dûs aux absences de répétitions de certaines séquences. Dans eCinema, le cas de la reconnexion après une déconnexion en est un exemple typique avec la suite de CertifyIt. En effet, la stratégie du générateur est d'atteindre au moins une fois chaque cible de test avec le chemin le plus court possible. Ainsi, les chemins qui ne permettent pas de faire évoluer

l'état du système vers un état à partir duquel la cible peut être appelée ne sont pas pris en compte.

Les suites issues des critères nominaux (voir Chapitre 8) permettent de palier ces manques de couverture et offrent une complémentarité à la suite de CertifyIt pour la couverture des éléments manquants. De plus, même si la suite de CertifyIt permet de couvrir certains critères, nos suites permettent de renforcer cette couverture par des tests supplémentaires.

Détection des mutants

Dans la section 9.3, nous avons étudié la capacité de détection de fautes par les différentes suites de tests vis-à-vis de chaque propriété sur des modèles mutés. Nous avons montré que les suites extraites des critères nominaux utilisées pour illustrer les propriétés permettent la détection de fautes introduites dans le modèle, accidentellement dans la grande majorité des cas. Toutefois, c'est la suite issue du critère de robustesse qui permet de détecter les fautes qui permettent la violation des propriétés. Les mutations d'évènements permettent de cibler les cas proches du cas d'erreur et ce sont les tests générés pour cibler ces évènements mutés qui permettent la détection de fautes permettant la violation de la propriété.

Toutefois, en restant au niveau du modèle, nous pouvons détecter par animation des non conformités entre le modèle et une propriété grâce à l'animation en parallèle sur l'automate de propriété, ce que nous avons illustré avec la propriété P_{72}. En effet, dans le modèle eCinema, nous ne disposons pas de moyen d'observer les tickets *physiquement* vendus (représentés par les objets de la classe *Ticket*), mais le fait que certains tests atteignent l'état d'erreur de la propriété nous a permis de nous rendre compte de la violation de la propriété. Cependant, cet indicateur n'est disponible qu'au niveau du modèle et nous n'avons aucune garantie de pouvoir l'observer au niveau de l'implémentation.

Ce problème d'observabilité est récurrent dans toute technique de génération de tests à partir de modèles. Sans observation pour rendre compte d'une faute dans le système, une faute peut ne jamais apparaître ou être observée de manière tellement indirecte qu'il peut être difficile de l'identifier. Les verdicts que nous avons définis permettent de refléter cela. Ainsi, les tests qui permettent la violation de la propriété et qui ont été observés sont classés dans *Fail*. En revanche, ceux qui ne peuvent pas être observés sont classés dans *Unseen*. Avec cette seconde catégorie, nous rendons compte du fait que les tests détectent bien les fautes liées à la propriété mais que le modèle ne dispose pas du moyen de les observer. Connaître les tests dont le résultat d'animation est *Unseen* peut constituer un indice pour l'ingénieur de test afin de lui permettre d'identifier les fautes potentiellement difficiles à observer.

Validation du modèle

Nous avons évoqué dans le Chapitre 4 le fait que notre approche puisse être utilisée à des fins de validation du modèle lors de sa conception en utilisant les scénarios comme un moyen de décrire des *test unitaires* au niveau modèle. Nous pouvons étendre cette possibilité avec les tests générés à partir de nos propriétés. Ainsi, l'animation des tests

produits sur le modèle permet de détecter des non conformités entre les propriétés et leur mise en œuvre dans le modèle comportemental, ce qui a été le cas avec la propriété P_1 lors de la détection des mutants.

Le cas de la propriété P_6 peut nous en convaincre plus facilement. En effet, nous avons noté lors de la couverture de cette propriété dans la section 9.2 que certains dépliages des scénarios détectent une non-conformité entre la propriété et le modèle, i.e. il est possible de se déconnecter avec succès sans un appel à l'opération *login*. Cela est rendu possible par l'opération *registration* qui, après l'enregistrement avec succès d'un nouvel utilisateur, connecte immédiatement celui-ci dans le système. Ainsi, la propriété P_6 telle que nous l'avons écrit est incomplète et devrait plutôt s'écrire :

```
          isCalled(login, including:{@AIM:LOG_Success})
          || isCalled(register, including:{@AIM:REG_Success})
precedes  isCalled(logout, inclusing:{@AIM:LOG_Logout})
globally
```

Cela s'applique aussi dans l'autre sens. Si la propriété est bien décrite mais que l'implémentation dans le modèle de l'opération *registration* ne connecte pas l'utilisateur après un enregistrement réussi, alors il y a non conformité et il est donc nécessaire de corriger le modèle.

9.5 Conclusions

Dans ce chapitre, nous avons défini un protocole expérimental pour comparer notre approche à partir de propriétés temporelles et les tests issus de l'outil CertifyIt. Nous avons procédé de deux manières. Dans la première, nous avons comparé la couverture respective des suites de tests de notre approche et celle de CertifyIt sur chaque propriété, vis-à-vis des critères de couverture introduits au Chapitre 6 et au Chapitre 7. Avec cette étape, nous avons pu identifier des manques de couverture de la suite de CertifyIt qui, dans la majorité des cas, peuvent être imputés à la stratégie de l'outil qui applique un autre critère de sélection de tests qui consiste à atteindre chaque comportement par sélection du chemin le plus court.

La seconde expérimentation porte sur la capacité de détection de fautes par animation des tests sur des modèles mutés. Bien que les suites détectent beaucoup de mutants lors de leurs exécutions, cette étape, nous a permis d'identifier les mutants qui permettent la violation des propriétés, les autres détections étant alors considérées comme *accidentelles*.

Enfin, nous avons terminé par une discussion sur les différents aspects de cette expérimentation. Ainsi, il en ressort que les tests générés par notre approche à base d'automates sont complémentaires d'autres approches de génération de tests (CertifyIt dans notre cas), tant sur la couverture que sur la détection de fautes, et ciblent les aspects dynamiques du système. De plus, cette expérimentation fait ressortir une possible synergie entre le modèle et les propriétés lors de la conception du modèle. En effet, les tests générés à partir des propriétés étant du même niveau d'abstraction, nous pouvons les animer sur le modèle. La détection de non conformités sur l'automate de propriété permet alors à un ingénieur d'identifier des fautes dans le modèle ou la propriété et de les corriger, garantissant une plus grande confiance dans le modèle lors de sa conception.

Chapitre 10

Mise en œuvre et application dans le cadre du projet ANR TASCCC

Sommaire

10.1 Présentation 180
10.1.1 Les acteurs du projet 181
10.1.2 Détail des sous-projets 181
10.2 L'étude de cas : GlobalPlatform 183
10.2.1 Gestion du contenu de la carte 184
10.2.2 Chargement d'une application 184
10.3 Prototype logiciel 185
10.3.1 Interface générale 185
10.3.2 Définition des éléments de traçabilité 187
10.3.3 Mesure de couverture d'une suite de tests 187
10.3.4 Génération de tests 188
10.3.5 Rapport de génération de tests 190
10.4 Evaluation de l'approche 191
10.4.1 Point de vue de l'évaluateur Critères Communs 191
10.4.2 Evaluation utilisateur de la méthode 193
10.5 Synthèse 194

Les travaux décrits dans cette thèse représentent une partie du processus du projet ANR TASCCC [20] (**T**est **A**utomatique basé sur des **SC**énarios et évaluation **C**ritères **C**ommuns) qui vise à accélérer l'activité de test de produits embarqués, notamment dans le cadre d'une évaluation Critères Communs [21]. Ce chapitre présente le projet, l'étude de cas utilisée et la part d'implémentation qui correspondent aux travaux décrits dans ce document et l'évaluation de la méthode du projet par les acteurs industriels.

Nous présentons d'abord dans la section 10.1 les différents sous-projets qui constituent le projet TASCCC puis nous abordons dans la section 10.2 l'étude de cas qui a servi de

20. ANR-09-SEGI-014 - `http://disc.univ-fcomte.fr/TASCCC`
21. Site Critères Communs : `http://www.commoncriteriaportal.org.` (2013)

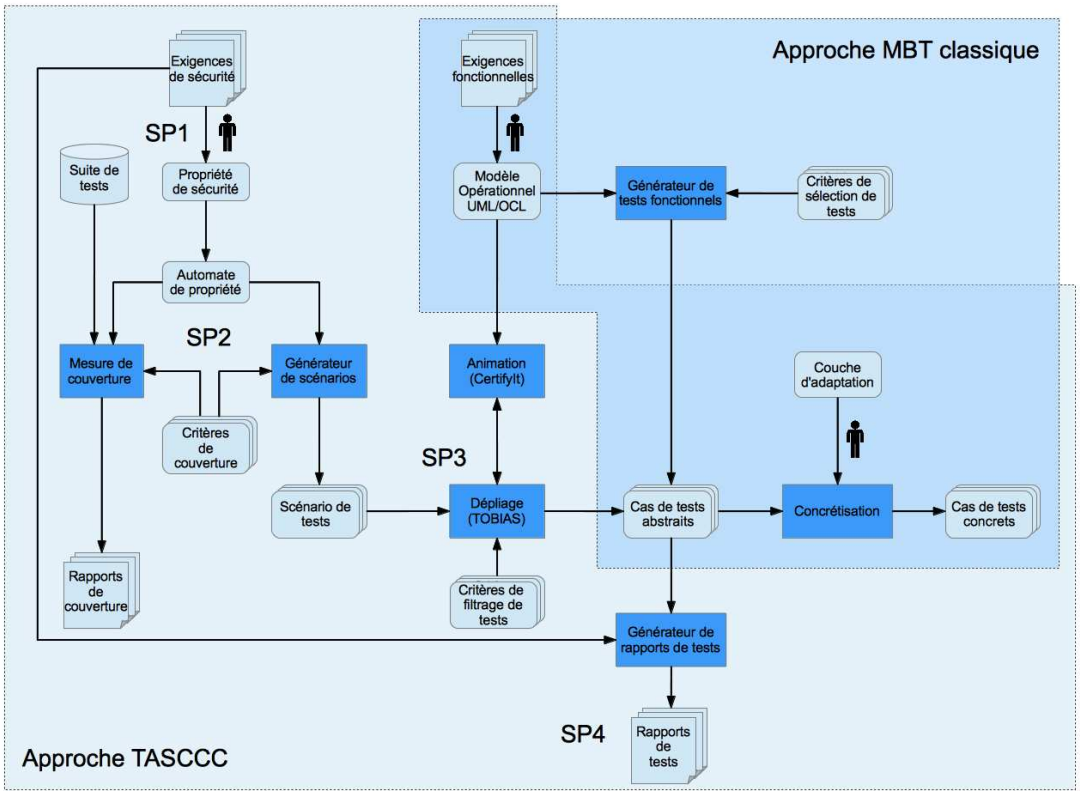


FIGURE 10.1 – Schéma de l'approche proposée par le projet TASCCC

support d'expérimentation. Nous présentons ensuite dans la section 10.3 le démonstrateur logiciel issu du projet. Enfin, dans la section 10.4, nous terminons par les évaluations du processus et de l'outil par deux partenaires industriels du projet, du point de vue d'un évaluateur des Critères Communs et du point de vue d'un utilisateur de la méthode.

10.1 Présentation

La FIGURE 10.1 présente le schéma de la démarche introduite par le projet TASCCC. L'approche classique de génération de tests fonctionnels à partir d'un modèle et de critères de couverture structurels est schématisée dans la partie droite de la figure. L'ingénieur validation décrit un modèle formel qui est donné en entrée d'un générateur de tests produisant des tests abstraits en fonction d'un ou plusieurs critères de sélection des tests. Ces tests abstraits sont ensuite concrétisés par le biais d'une couche d'adaptation qui fait le lien entre les éléments du modèle et les éléments réels du système sous test et définit ainsi une relation de conformité. Cette concrétisation permet d'obtenir des tests concrets qui peuvent être ensuite exécutés sur un banc de test ou exportés dans divers formats.

La démarche TASCCC, dans la partie gauche, propose une approche qui permet de compléter les tests fonctionnels couvrant le modèle par des tests des exigences de sécurité. Par exemple, pour valider les aspects *temporels* des exécutions d'un système, l'ingénieur

validation doit décrire une propriété et choisir un besoin de test au moyen de patrons de propriétés qui lui sont proposés. De ces derniers sont issues des stratégies de test qui permettent au générateur de scénarios d'engendrer des scénarios de test à partir de ces éléments. Ces scénarios sont ensuite dépliés et valués à partir de critères de couverture par des techniques de dépliage combinatoire mises en œuvre dans l'outil Tobias [LDdB+07] du LIG. A partir de ce point, la chaîne TASCCC réutilise le processus de concrétisation des cas de tests par l'intermédiaire d'une couche d'adaptation. Parallèlement à la génération, nous pouvons aussi évaluer la couverture d'une suite de tests, qu'elle soit issue d'un outil ou d'une campagne manuelle, sur une propriété. Cette mesure permet d'informer de la couverture déjà existante de la propriété et d'identifier ce que la génération de tests peut amener pour compléter cette couverture.

10.1.1 Les acteurs du projet

Le projet TASCCC s'appuie sur un partenariat équilibré entre membres industriels et académiques. Chaque partenaire amène son expertise théorique et/ou technologique au projet. Ainsi, *Supélec* de Gif-sur-Yvette s'intéresse à la modélisation des propriétés de sécurité par le biais d'un langage *ad hoc*. Le département *FEMTO/DISC* de Besançon s'intéresse au traitement des propriétés pour la production de scénarios de test. Le *LIG*[22], apporte sa technologie de dépliage combinatoire de scénarios et de réduction des suites de tests issus du dépliage. La société *Smartesting*, basée à Besançon, fournit la technologie d'animation de modèles UML/OCL au travers de son logiciel CertifyIt. *Serma Technologies*[23], localisée à Bordeaux, apporte son expertise dans le domaine de la certification Critères Communs et évalue les apports de la méthode proposée par TASCCC dans le contexte d'une évaluation. Enfin, *Trusted-Labs*, située à Versailles, porte le projet et est rattachée à *Gemalto*, localisée à Meudon, qui fournit l'étude de cas sur laquelle sera appliquée et validée la démarche.

10.1.2 Détail des sous-projets

Toute la démarche est réalisée en vue de son industrialisation, et implique des notions de traçabilité, obligatoires pour répondre aux exigences des Critères Communs. Le projet est composé de 5 sous-projets que nous décrivons maintenant.

SP1 - Formalisation de politiques de sécurité. Ce sous-projet, placé en amont dans la chaîne de TASCCC, se focalise sur la formalisation des politiques de sécurité. Son objectif est de définir un langage d'expression de propriétés de sécurité qui doit être utilisable par des non experts.

La définition du langage décrit dans le Chapitre 5 est un résultat de ce sous-projet. Le langage issu de ce sous-projet a été implémenté en tant qu'extension du langage OCL pour être intégré dans les modèles UML/OCL.

SP2 - Génération pilotée de scénarios de test. Ce sous-projet s'intéresse à la production automatique de scénarios de tests à partir des propriétés définies dans le SP1

22. Laboratoire d'Informatique de Grenoble : `http://www.liglab.fr`
23. Site Serma Technologies : `http://www.serma-technologies.com` (2013)

qui représentent des séquences permettant d'illustrer ou de mettre à l'épreuve ces propriétés. Ces scénarios de tests doivent aussi répondre à des besoins de tests qui capturent l'intention de l'ingénieur validation.

C'est dans ce sous-projet que s'inscrivent la plupart des travaux introduits dans cette thèse. En effet, ce sous-projet a pour but de générer des scénarios de tests dans le langage de scénarios, nommé TSLT de l'outil Tobias (**T**est **S**chema **L**anguage for **T**obias), à partir du langage de propriétés du SP1 (Chapitre 5) et à partir de stratégies de sélection de tests qui sont représentés par les critères de couverture présentés dans Chapitres 6 et 7. Enfin, la génération des scénarios est décrite au Chapitre 8 mais dans le langage TSLT, propre à TOBIAS. Il est à noter que les deux langages, bien que très proches, diffèrent au niveau syntaxique et sur la manière de déplier. TOBIAS déplie tous les tests et les filtre à *posteriori* alors que notre langage est interprété et la sélection se fait au moment du dépliage.

SP3 - Dépliage des scénarios et génération de tests. Ce sous-projet s'occupe de l'instanciation des scénarios issus du SP2 en cas de tests abstraits. L'instanciation consiste donc à déplier ces scénarios en valuant les paramètres des opérations. Cependant, ces dépliages peuvent donner lieu à de nombreux cas de tests et il est donc nécessaire de recourir à des techniques de réduction de suites de tests pour contrôler l'explosion combinatoire. De plus, un dépliage combinatoire brutal sans vérification de sa faisabilité sur le modèle peut générer des cas de test qui peuvent ne pas être cohérents avec la représentation formelle de la spécification.

Le dépliage combinatoire est effectué par l'outil Tobias fourni par le LIG couplé à la technologie de génération de cas de tests en utilisant le moteur d'animation de CertifyIt de Smartesting. La collaboration entre ces deux outils permet à Tobias de générer des cas de tests cohérents vis-à-vis du modèle par l'animation pas-à-pas du dépliage sur ce modèle. De plus l'implémentation de différentes techniques [TLdB$^+$12, LVTB12, TdBL12] de réduction de suites de tests permet d'obtenir un ensemble restreint de cas de tests sans nuire à la pertinence de la suite de tests vis-à-vis du scénario dont ils sont issus.

Afin de disposer d'un outillage complet des travaux présentés dans cette thèse, nous avons développé une solution similaire avec un langage de scénarios *ad hoc* et la technique de dépliage combinatoire associé dans le Chapitre 4.

SP4 - Traçabilité et qualité des tests produits. L'objectif de ce sous-projet est la mise en place d'un système de traçabilité entre les exigences de la spécification informelle et les cas de tests produits tout au long du processus TASCCC. En effet, il est nécessaire d'identifier précisément quelles fonctionnalités sont testées et quels tests permettent de les couvrir. Cette traçabilité permet de nous assurer que les exigences prises en compte dans le projet ont non seulement été couvertes, mais ont aussi été *suffisamment* exercées.

La finalité de cette sous-tâche est la production automatique de rapports de génération de tests indiquant, pour une propriété donnée, les exigences qu'elle permet de satisfaire, quels sont les tests qui la couvrent et quelles ont été les stratégies mises en œuvre pour produire ces tests. Ce sous-projet est important du point de vue de l'industriel car il lui permet d'avoir un retour immédiat sur les tests produits et la couverture des propriétés considérées. Les rapports de test sont aussi importants du point de vue des Critères Communs car ils permettent à l'évaluateur d'avoir une vue d'ensemble sur la couverture des différentes exigences de la spécification.

SP5 - Application à l'étude de cas. Ce sous-projet correspond à l'application et l'évaluation de la méthode proposée par le projet sur l'étude de cas fournie par Gemalto avec les exigences Critères Communs. Il a pour but de traduire les exigences de sécurité de la spécification dans le formalisme introduit dans le SP1 puis d'appliquer la démarche du projet. De plus, ce sous-projet comporte un aspect d'évaluation de la méthode, tant du point de vue Critères Communs que du point de vue utilisateur de la méthode.

Le résultat de ce sous-projet correspond à deux livrables portant sur l'évaluation de la méthode d'un coté par un évaluateur Critères Communs [Rou12] et de l'autre par un utilisateur de la chaîne outillée sur l'étude de cas GlobalPlatform [BG13] que nous présentons maintenant.

10.2 L'étude de cas : GlobalPlatform

GlobalPlatform est un standard industriel de gestion de ressources pour les cartes à puce multi-applicatives. Il décrit un ensemble de fonctionnalités et d'interfaces permettant de gérer tous les aspects d'administration d'une carte tout au long de sa vie. Les implémentations de ce standard sont extrêmement répandues, en particulier sur les cartes bancaires répondant au standard EMV (Eurocard-Mastercard-Visa) et sur les cartes SIM et USIM pour téléphones portables, mais aussi sur un grand nombre de cartes d'identité, de passeports électroniques, de cartes Vitale, etc.

Cependant, l'étude de cas du projet TASCCC ne considère pas tous les aspects de la spécification GlobalPlatform et ne se concentre que sur les fonctionnalités suivantes :

— gestion sécurisée du cycle de vie de la carte,
— authentification des entités hors carte et de la carte,
— communications sécurisées avec les entités hors carte,
— gestion du contenu de la carte, en particulier des applications,
— routage des commandes vers les différentes applications.

Un aspect important du standard GlobalPlatform est qu'il est conçu pour permettre à plusieurs acteurs distincts (opérateurs de téléphonie, organisations bancaires, opérateurs de transport, etc.) de coexister sur une carte.

Cela se fait au travers de la notion de domaine de sécurité (SD : Security Domain). Un SD est la représentation sur carte d'une entité ayant un rôle dans son fonctionnement. Plus précisément, un SD est une application système au travers de laquelle toutes les interactions avec GlobalPlatform se déroulent. En pratique, chaque SD est caractérisé par :

— son AID (Application IDentifier), un identifiant unique sur la carte ;
— un ensemble de privilèges définissant le périmètre de ses prérogatives sur le reste de la carte. Par exemple, un SD peut avoir le droit (ou pas) de bloquer la carte, d'installer de nouvelles applications, d'effacer des applications existantes, etc ;
— un ensemble de clés cryptographiques partagées avec l'entité à laquelle il appartient. De cette façon, seule cette dernière peut s'authentifier auprès de lui, et éventuellement chiffrer le canal de communication.

La commande GobalPlatform SELECT permet de sélectionner le SD avec lequel on désire communiquer. Toute interaction avec GlobalPlatform passe par le biais des SD. Il

existe nécessairement sur toute carte GlobalPlatform 2.2 un SD particulier, l'ISD (Issuer Security Domain), associé à l'émetteur de la carte. Ce SD est nécessairement privilégié pour permettre, entre autres, de procéder à toute opération de gestion du contenu de la carte et de son cycle de vie.

En outre, le composant central d'administration de GlobalPlatform est désigné par le sigle OPEN (Open Platform ENvironement). Il est chargé d'effectuer toutes les tâches de vérification et de mise à jour qui échappent au contrôle des seuls SD. Ses interfaces ne sont publiées qu'aux seuls SDs, et sont inaccessibles par une entité externe (sauf par invocation indirecte au travers d'un SD).

A titre d'exemple, nous donnons les détails de la gestion du contenu de la carte et du chargement d'une application dans GlobalPlatform 2.2.

10.2.1 Gestion du contenu de la carte

La commande INSTALL est une commande multi-usages, permettant de réaliser toutes les fonctions de gestion de contenu de la carte et permet les actions suivantes :

- **Load** : permet d'initier la session de chargement du code d'une application,
- **Install** : permet d'installer une application dont le code a été chargé au préalable,
- **Make Selectable** : permet de marquer une application comme pouvant être sélectionnée par une application externe (par le biais de la commande SELECT),
- **Extradition** : permet d'associer une application à un autre SD,
- **Registry Update** : permet de mettre à jour le registre GlobalPlatform (une mise à jour des privilèges par exemple),
- **Personalization** : permet d'initier une session de personnalisation de l'application.

Du point de vue d'une évaluation Critères Communs, nous pouvons considérer qu'il existe soit une seule opération INSTALL qui rassemble toutes les actions précédentes, ou nous pouvons considérer les six opérations séparément. C'est cette seconde solution qui a été choisie dans le cadre de TASCCC.

10.2.2 Chargement d'une application

Le chargement d'une application sur une carte conforme au standard GlobalPlatform 2.2 se déroule en cinq étapes. Dans un premier temps, il est nécessaire de sélectionner le SD auquel la nouvelle application doit être rattachée avec la commande SELECT. Ensuite, les commandes INITIALIZE_UPDATE et EXTERNAL_AUTHENTICATE permettent à l'application externe et au SD de s'authentifier mutuellement puis d'ouvrir un canal de communication sécurisé. Ce canal créé, il est alors possible d'initialiser la session de chargement avec la commande INSTALL[for Load] avant de charger le code de l'application avec la commande LOAD. Enfin, le système procède à l'installation de la nouvelle application avec la commande INSTALL[for Install].

Il est à noter que les commandes SELECT, INITIALIZE_UPDATE, et EXTERNAL_AUTHENTICATE ne sont pas spécifiques à GlobalPlatform et sont définies dans le standard ISO 7816-4 [Int05].

10.3 Prototype logiciel

Le projet TASCCC a donné lieu à un prototype [DCL+13] mettant en œuvre la démarche du projet. Il s'agit d'une suite de modules pour Eclipse[24] version Juno, chacun intégrant les résultats des sous-projets précédents. Les modules principaux sont les suivants :

- un éditeur de propriétés, développé par Supélec qui permet à l'ingénieur de spécifier ses propriétés qui seront ensuite enregistrées dans un fichier .tocl,
- un module de génération de scénarios, développé au DISC qui correspond à l'implémentation des critères des Chapitres 6 et 7. Il permet, à partir des propriétés saisies, de générer les scénarios correspondant aux critères sélectionnés. Contrairement au Chapitre 8, les expressions régulières sont traduites dans le langage TSLT propre à TOBIAS,
- un module de mesure de couverture, qui permet de mesurer la couverture d'une suite de tests externe sur les automates vis-à-vis des critères. Ce module correpond au Chapitres 6,
- un module de génération de tests, développé par le LIG, qui, à partir des fichiers TSLT produits par le module de génération de scénarios, permet de générer les tests avec TOBIAS et CertifyIt,
- deux modules de rapports, développés par Smartesting, pour aider à la visualisation des données de couverture et de génération de tests.

Nous détaillons maintenant les différents aspects de ce prototype.

10.3.1 Interface générale

L'interface générale, présentée à la Figure 10.2, comporte plusieurs vues. Une première vue en haut à gauche rassemble la liste des projets TASCCC. Chaque projet nécessite plusieurs fichiers :

- un fichier .smtmodel qui représente le modèle de test dans le format de l'outil CertifyIt,
- un fichier .uml qui donne une représentation du modèle dans le format UML2.2. Ce fichier est nécessaire pour la complétion dans l'éditeur de propriétés.
- un fichier .txt qui regroupe les éléments de traçabilité à considérer. Dans le cadre du projet TASCCC, nous traçons les SFR (Security Functional Requirements, les différentes catégories d'exigences de sécurité) et les TSFI (Target of evaluation Security Functional Interface), qui correspondent aux opérations du système sous test.
- un ou plusieurs fichiers .tocl qui regroupent les propriétés temporelles écrites dans le langage défini dans le SP1. Chaque fichier peut contenir plusieurs propriétés et chaque propriété peut être annotée afin d'indiquer les exigences de traçabilité qui lui sont associées.
- un ou plusieurs fichiers de tests .xml dans le format de sortie de l'outil CertifyIt. Ce fichier est utilisé lors de la mesure de la couverture des propriétés.

24. Eclipse Foundation : http ://www.eclipse.org

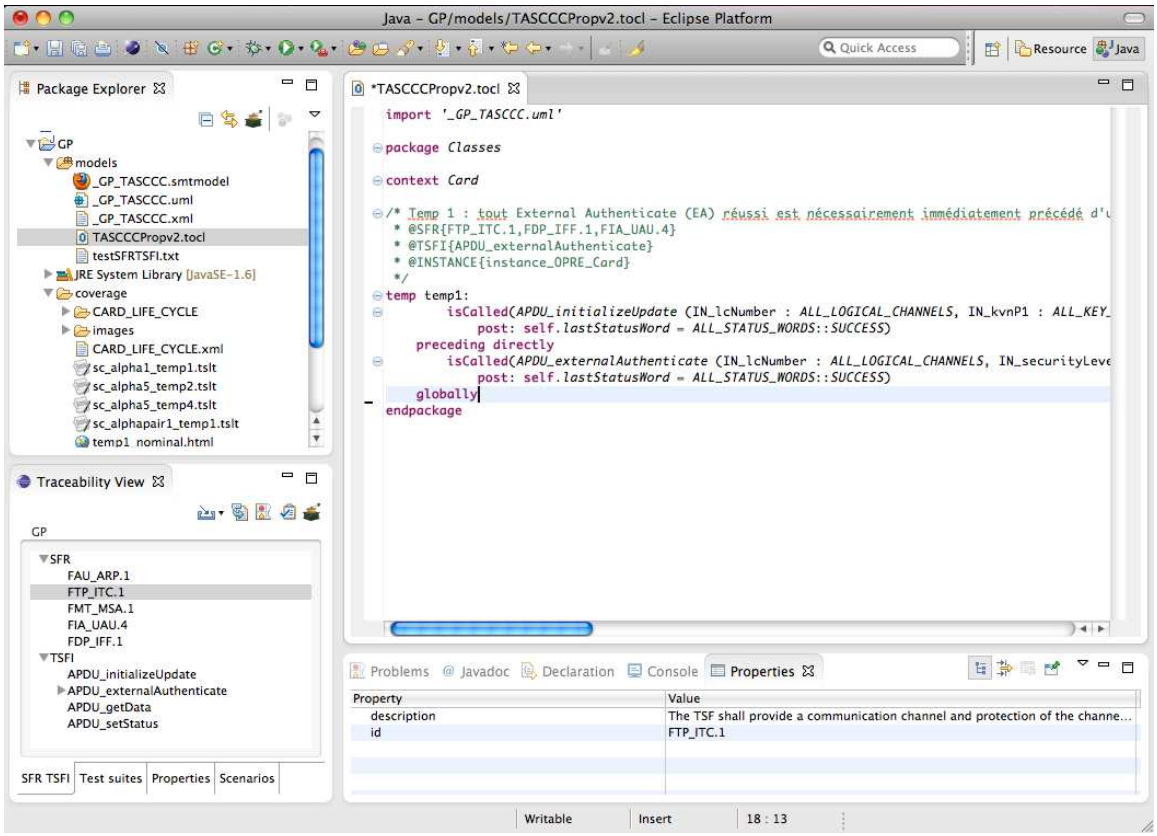


FIGURE 10.2 – Interface générale du prototype TASCCC

Une seconde vue, en bas à gauche récapitule tous les éléments de traçabilité dans différents onglets. Un premier onglet “SFR TSFI” liste les SFR et les TSFI considérées. Ces éléments peuvent être référencés lors de l'écriture des propriétés. L'onglet “Test suites” récapitule les suites de tests et les tests associés qui sont utilisés pour la mesure de couverture. L'onglet “Properties” récapitule toutes les propriétés TOCL chargées dans le projet. Enfin, le quatrième onglet récapitule les scénarios générés pour chaque propriété. Le nommage de chaque scénario permet de connaître le critère dont il est issu.

Cette seconde vue propose une liste de boutons qui correspondent aux actions possibles dans le projet TASCCC. Le premier bouton permet de charger les différents fichiers à considérer (suites de tests, traçabilité et propriétés). Le second bouton permet de générer les scénarios pour la propriété sélectionnée, le critère est ensuite choisi dans une boite de dialogue. Le troisième bouton permet de mesurer la couverture d'une propriété par la suite de tests sélectionnée et de générer le rapport de couverture. Le quatrième bouton permet de générer le rapport de génération de tests et des exigences à tracer. Enfin, le dernier bouton permet de générer les scénarios de tests dans le format TSLT.

La troisième vue, en haut à droite, est l'éditeur de propriété. Cet éditeur propose une coloration syntaxique et une complétion sur les éléments du modèle pour faciliter la saisie. L'éditeur permet la saisie de plusieurs propriétés dans un même fichier et chaque propriété peut être annotée pour ajouter une description et les SFR et TSFI associées à

```
TRACEABILITY ::= ELEMENT*
ELEMENT      ::= @TYPE{identifier(,KEY=VALUE)*}
TYPE         ::= sfr|tsfi|action
KEY          ::= @TYPE|key
VALUE        ::= "value"|{VALUE*,}|value
```

FIGURE 10.3 – Grammaire du fichier des éléments de traçabilité

la propriété.

Enfin, la dernière vue, en bas à droite permet de visualiser les propriétés des différents éléments. Il est ainsi possible de voir les descriptions des SFR/TSFI, pour chaque test les propriétés couvertes, la propriété d'origine et les critères dont est issu un scénario particulier.

10.3.2 Définition des éléments de traçabilité

Les éléments de traçabilité sont définis dans un fichier dédié qui permet de définir les éléments de traçabilité avec la syntaxe décrite dans la FIGURE 10.3.

Un fichier dédié permet de définir les éléments de traçabilité qui sont identifiés par un identifiant unique (identifier) et par un type qui correspond aux éléments de traçabilité tel que ceux utilisés par les Critères Communs. Chaque élément possède un ensemble de couples clef-valeur (key-value) pour associer différentes informations à chaque élément. Les clefs peuvent faire référence à des éléments déjà définis dans la traçabilité tels que les SFR avec la clef @sfr. Les valeurs peuvent être des chaînes de caractères (telles que des descriptions) ou un ensemble de valeurs.

Dans l'exemple de fichier de la FIGURE 10.4, nous avons décrit trois SFR (FTP_ITC.1, FDP_IFF.1, FIA_UAU.4) qui possèdent chacune une description. Nous avons aussi une TSFI identifiée par *APDU_setStatus* et elle est nommée "SetStatus". Enfin, le fichier décrit l'action *checkInitUpdateBefore* qui possède une description et est rattachée aux trois SFR définies précédemment (par la clef @sfr) ainsi que des tags des comportements du modèle.

10.3.3 Mesure de couverture d'une suite de tests

Le rapport de couverture est issu de la mesure de couverture et montre, pour une propriété particulière, la couverture d'une suite de tests sur cette propriété. Cette partie du prototype est l'intégration des travaux décrits dans le Chapitre 5 et le Chapitre 6 appliqués à la mesure de couverture. Pour évaluer la couverture, il est nécessaire de fournir une suite de tests dans le format XML utilisé par CertifyIt pour publier les tests par l'intermédiaire du premier bouton de la vue de traçabilité. La liste des tests inclus dans la suite est ensuite ajoutée dans l'onglet "Test Suite". Ensuite, il faut spécifier le dossier dans lequel le rapport de couverture doit être exporté puis, il est nécessaire de spécifier quel critère nous souhaitons couvrir. Un choix *Nominal* indique que l'on souhaite la couverture de tous les critères nominaux et un choix *Robustesse* indique une couverture des transitions mutées de l'automate.

testSFRTSFI.txt

```
@sfr{FTP_ITC.1,
    description="The TSF shall provide a communication channel and protection of the channel data fr
}
@sfr{FDP_IFF.1,
    description="the TSF shall enforce the secure channel protocol information flow control policy"
}
@sfr{FIA_UAU.4,
    description="The TSF shall prevent reuse of authentication data related to the authentication me
}
@action{checkInitUpdateBefore,
    description = "Check that the previous operation that was requested was actually InitializeUpdat
    @tags = { (@REQ:APDU_EXTERNAL_AUTHENTICATE_SUCCESS),
              (@AIM:NO_PREVIOUS_INITIALIZE_UPDATE, @REQ:APDU_EXTERNAL_AUTHENTICATE_ERROR_MISSING_PRE
    @sfr = {FTP_ITC.1, FDP_IFF.1, FIA_UAU.4,}
}
@tsfi{APDU_setStatus,
    name="SetStatus"
}
```

FIGURE 10.4 – Exemple de fichier contenant les éléments de traçabilité

La FIGURE 10.5 montre un exemple de rapport de couverture issu de la mesure sur le modèle eCinema. Il se présente sous la forme d'une page HTML qui récapitule la propriété utilisée et sa représentation en automate (avec la légende qui l'accompagne).

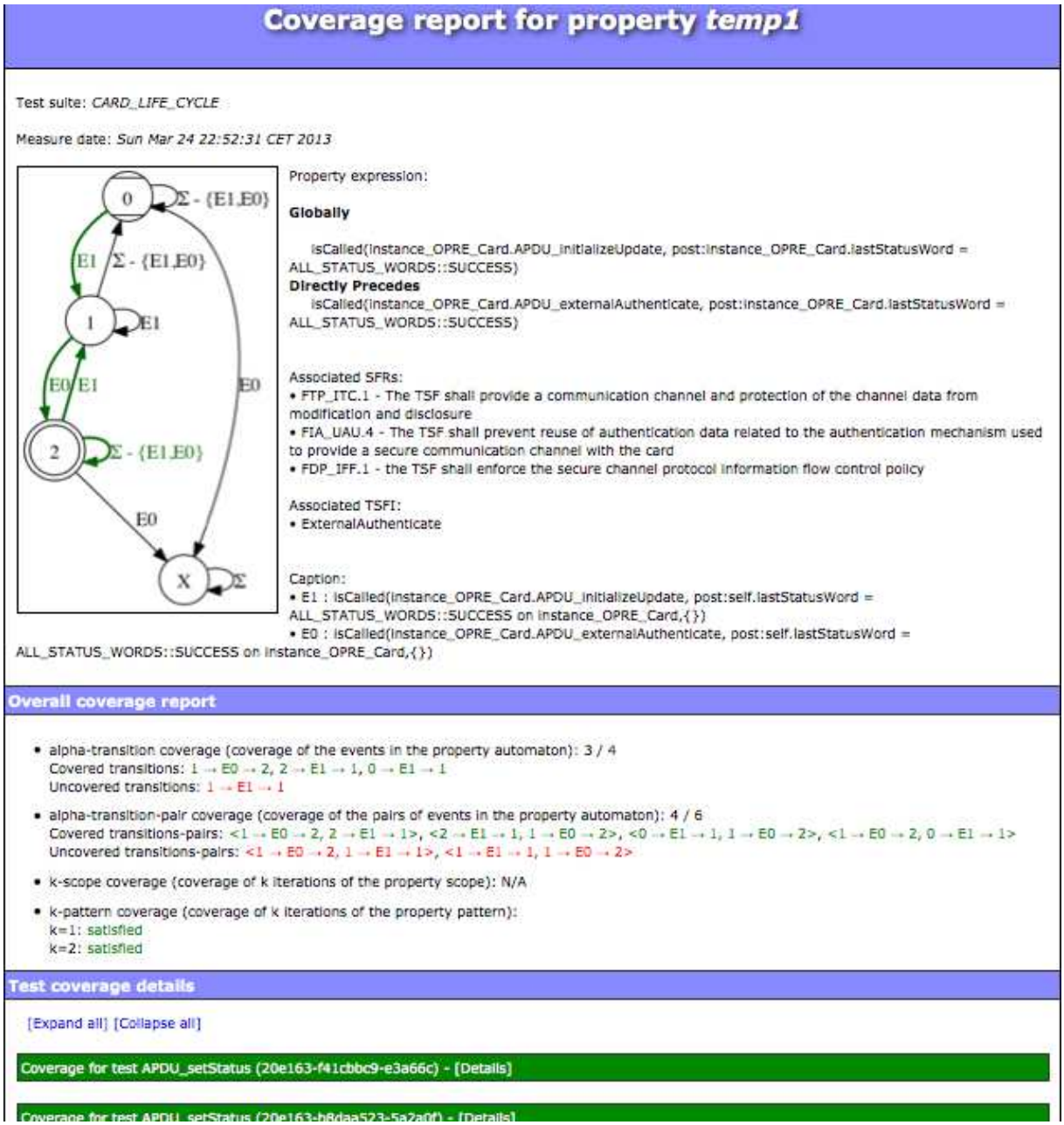
Dans un premier temps, la page récapitule la propriété considérée et sa représentation en automate, la suite de tests considérée pour la couverture et les éléments de traçabilité associés à la propriété. L'image de l'automate permet de se rendre compte de la couverture globale des transitions de la propriété. Sur cette page, les transitions colorées en vert (telle que la transition $0 \xrightarrow{E_0} 1$) indiquent qu'elles ont été couvertes par au moins un test, une transition rouge indique la couverture d'une transition d'erreur de l'automate et les transitions noires n'ont pas été couvertes.

Ensuite, la page récapitule la couverture de la propriété vis-à-vis des critères de couverture. Cette section permet de se rendre compte de la qualité de la couverture pour chaque critère.

Enfin, la dernière section permet de visualiser la couverture individuelle des tests. Chaque sous-section donne l'image de l'automate et les transitions que couvre le test considéré ainsi que la liste des opérations qui composent le test. La couleur du bandeau indique l'état de la couverture : en vert, le test atteint un état final (tel que les deux tests APDU_setStatus en bas), en gris, il n'atteint pas d'état final et en rouge, il a atteint un état d'erreur.

10.3.4 Génération de tests

La génération de test est un processus en deux étapes. Dans un premier temps, nous générons les scénarios à partir des critères de couverture. Cette étape, développée par FEMTO/DISC, est l'intégration des travaux décrits dans les Chapitre 5 et 6 dans une optique de génération de tests et le Chapitre 8 dans une optique de génération de tests. Ces scénarios sont exprimés dans le langage TSLT [TLdB+12] et sont enregistrés dans des



Coverage report for property *temp1*

Test suite: *CARD_LIFE_CYCLE*

Measure date: *Sun Mar 24 22:52:31 CET 2013*

Property expression:

Globally

isCalled(instance_OPRE_Card.APDU_initializeUpdate, post:instance_OPRE_Card.lastStatusWord = ALL_STATUS_WORDS::SUCCESS)
Directly Precedes
isCalled(instance_OPRE_Card.APDU_externalAuthenticate, post:instance_OPRE_Card.lastStatusWord = ALL_STATUS_WORDS::SUCCESS)

Associated SFRs:
- FTP_ITC.1 - The TSF shall provide a communication channel and protection of the channel data from modification and disclosure
- FIA_UAU.4 - The TSF shall prevent reuse of authentication data related to the authentication mechanism used to provide a secure communication channel with the card
- FDP_IFF.1 - the TSF shall enforce the secure channel protocol information flow control policy

Associated TSFI:
- ExternalAuthenticate

Caption:
- E1 : isCalled(instance_OPRE_Card.APDU_initializeUpdate, post:self.lastStatusWord = ALL_STATUS_WORDS::SUCCESS on instance_OPRE_Card,{})
- E0 : isCalled(instance_OPRE_Card.APDU_externalAuthenticate, post:self.lastStatusWord = ALL_STATUS_WORDS::SUCCESS on instance_OPRE_Card,{})

Overall coverage report

- alpha-transition coverage (coverage of the events in the property automaton): 3 / 4
  Covered transitions: 1 → E0 → 2, 2 → E1 → 1, 0 → E1 → 1
  Uncovered transitions: 1 → E1 → 1
- alpha-transition-pair coverage (coverage of the pairs of events in the property automaton): 4 / 6
  Covered transitions-pairs: <1 → E0 → 2, 2 → E1 → 1>, <2 → E1 → 1, 1 → E0 → 2>, <0 → E1 → 1, 1 → E0 → 2>, <1 → E0 → 2, 0 → E1 → 1>
  Uncovered transitions-pairs: <1 → E0 → 2, 1 → E1 → 1>, <1 → E1 → 1, 1 → E0 → 2>
- k-scope coverage (coverage of k iterations of the property scope): N/A
- k-pattern coverage (coverage of k iterations of the property pattern):
  k=1: satisfied
  k=2: satisfied

Test coverage details

[Expand all] [Collapse all]

Coverage for test APDU_setStatus (20e163-f41cbbc9-e3a66c) - [Details]

Coverage for test APDU_setStatus (20e163-b8daa523-5a2a0f) - [Details]

FIGURE 10.5 – Rapport de couverture

FIGURE 10.6 – Rapport de génération de tests - vue TSFI

fichiers séparés qui portent l'extension .tslt.

A l'issue de cette première étape, l'ingénieur a la possibilité d'éditer le fichier de scénario afin de préciser certains points trop génériques du scénario. De plus, le langage TSLT dispose de clefs de filtrage et d'un dépliage optimisé pour la réduction de l'explosion combinatoire qui sont l'objet de la thèse de Taha Triki au LIG.

La seconde étape, développée par le LIG, est la génération de tests à partir de ces scénarios à proprement parler. Ces cas de tests sont alors enregistrés dans une suite de tests particulière qui peut enfin être exportée dans un format XML.

10.3.5 Rapport de génération de tests

Le rapport de génération est aussi un document se présentant sous la forme de plusieurs pages HTML, chacune représentant une vue des éléments de traçabilité. La FIGURE 10.6 montre la vue "TSFI" qui donne les informations de couverture pour chaque TSFI. Ainsi, pour chaque action (qui correspondent aux différents comportements d'une opération) qui définit une TSFI (colonne *TSFI*), cette page donne les SFR associées (colonne *SFR*) ainsi que les tags du modèle correspondant à ces exigences (colonne *Tags*). Pour chaque action, nous pouvons observer dans la colonne *Tests* les tests qui contribuent à couvrir les tags du modèle associé. Enfin, dans la colonne TSFI, une barre de progression récapitule le taux de couverture de la TSFI en fonction de ses actions couvertes.

La FIGURE 10.7 montre la vue "SFR↔Propriété", centrée sur les SFR, qui donne les correspondances entre les SFR à considérer et les propriétés qui permettent de les couvrir. La dernière colonne récapitule, pour chaque propriété, les tests permettant de couvrir chaque SFR.

La FIGURE 10.8 montre la vue "Tests", centrée sur les tests de la suite de tests considérée. Pour chaque test, il est possible de voir les propriétés couvertes (avec le type de critère *nominal* ou *robustesse*) ainsi que tous les éléments spécifiques aux Critères Communs tels que les SFR, les TSFI et les actions qui sont couvertes.

La dernière vue, illustrée en FIGURE 10.9, montre la vue de détail d'un test. Cette page récapitule les SFR, les TSFI, les actions et les propriétés couvertes par le test considéré. De plus, elle détaille la succession des opérations qui composent le test et les sous-opérations

FIGURE 10.7 – Rapport de génération de tests - vue SFR↔Propriété

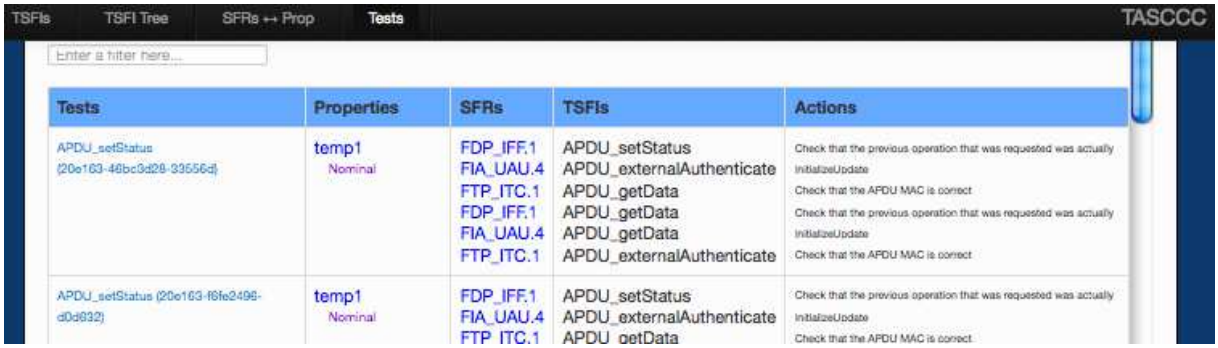


FIGURE 10.8 – Rapport de génération de tests - vue Tests

d'une opération plus large. Dans la figure, les opérations *initializeUpdate* et *externalAuthenticate* sont appelées lors de l'appel de *openSecureSession*, l'opération plus générale.

L'ensemble des pages de ce rapport permet à un évaluateur Critères Communs de facilement visualiser les tests qui permettent de couvrir les exigences attendues dans le cadre d'une évaluation. Ainsi, ces différentes pages donnent à l'ingénieur validation et à l'expert Critères Communs un aperçu rapide des exigences couvertes par les tests générés.

10.4 Evaluation de l'approche

Dans le projet TASCCC, l'évaluation de la méthode est faite de deux points de vue différents. Le premier est celui de l'industriel utilisateur de la méthode, Gémalto, qui présente son évaluation dans le livrable [BG13]. Le second est l'évaluation du côté des Critères Communs présentée dans le livrable [Rou12].

10.4.1 Point de vue de l'évaluateur Critères Communs

La norme des Critères Communs a été conçue pour évaluer la sécurité des systèmes d'informations. Le terme *Critères Communs* reflète la volonté de définir des critères reconnus par tous les acteurs de la sécurité et permettant de comparer des résultats obtenus lors d'évaluations sécuritaires indépendantes. L'objectif de ces évaluations est de fournir

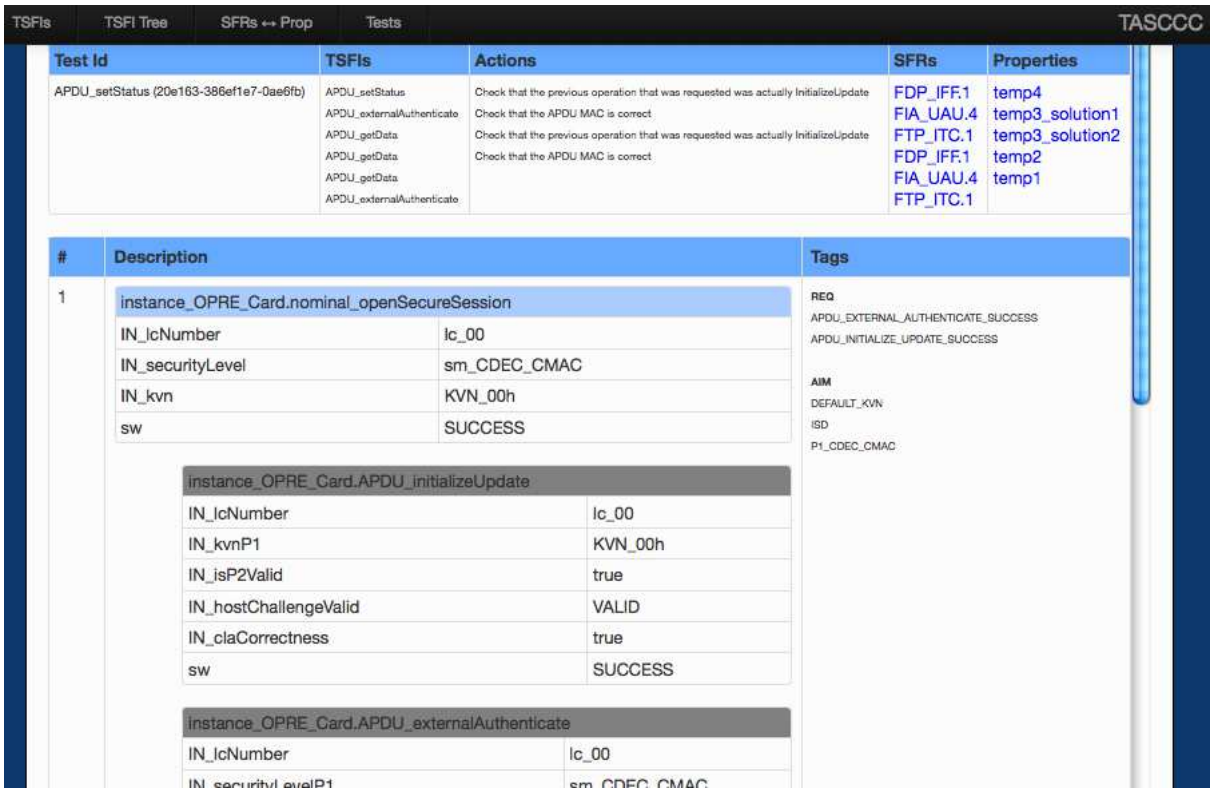


FIGURE 10.9 – Rapport de génération de tests - détail des tests

l'assurance que ces exigences sont correctement implémentées dans le produit et que les mécanismes qui les mettent en œuvre peuvent résister à des attaques dont le niveau est préalablement défini.

Les Critères Communs définissent des niveaux de confiance nommés *EAL* (Evaluation Assurance Level) au nombre de 7, du niveau 1, le plus bas, indiquant l'assurance d'un produit testé fonctionnellement au niveau 7, le plus élevé, assurant une conception vérifiée formellement et un système testé. Afin de définir ces niveaux de confiance l'évaluation est décomposée en plusieurs *classes d'assurance*, elles-mêmes décomposées en *familles*, chacune d'elle possèdant différents niveaux (en général de 1 à 3) de plus en plus exigeants. Ainsi, la classe *ATE*, qui se focalise sur l'aspect test d'un logiciel, est composée de quatres familles, mais seules deux d'entres elles sont considérées dans le projet TASCCC :

— ATE_COV : cette famille intègre tous les aspects du test concernant la couverture des tests. Ainsi, elle permet d'évaluer l'effort de test sur le logiciel. Plus le niveau est élevé, plus la démonstration de la couverture doit être rigoureuse.

— ATE_FUN : cette famille comprend tous les aspects de tests fonctionnels au sens large. Cette famille ne comprend que deux niveaux : le premier concerne l'existence des tests et la seconde concerne l'ordonnancement des différents tests.

La démarche mise en place dans le cadre de TASCCC permet d'atteindre les exigences ATE_COV2 et ATE_FUN1 qui permettent de satisfaire, pour ces familles, les exigences visées pour un niveau EAL5, qui correspond au niveau minimum pour des systèmes cri-

tiques.

L'évaluation des experts Critères Communs conclut sur un avis très positif concernant la méthode proposée par le projet TASCCC dans le livrable [Rou12]. Dans un premier temps, il valide l'adéquation des tests générés vis-à-vis des attentes Critères Communs en indiquant ceci dans le livrable :

> *Il a ainsi été validé que les tests produits satisfont pleinement les critères d'évaluation habituellement appliqués pour le type de produit visé. Un critère important est la pertinence des tests, en particulier lorsque des outils automatiques sont utilisés. L'étude montre que les tests de la campagne TASCCC ont le même niveau de pertinence que ceux qui seraient produits manuellement par un ingénieur de validation. Un avantage de l'approche TASCCC est de produire davantage de tests et de solliciter ainsi le produit dans des contextes plus variés.*

Un autre point de son évaluation met en lumière le fait que dans certains cas l'approche TASCCC permet une meilleure traçabilité entre les tests et les exigences qu'une approche manuelle :

> *L'approche est apte à répondre aux attentes des Critères Communs. Mais elle va au delà puisqu'elle permet d'établir une correspondance directe entre les exigences de sécurité et les tests, ce que l'approche manuelle ne parvient pas à faire de façon satisfaisante sans passer par une correspondance avec le design.*

Enfin, il souligne le fait que la méthode simplifie grandement le travail de l'ingénieur Critères Communs, grâce aux différents rapports produits par l'outil :

> *Enfin le travail d'évaluation est simplifié dans la mesure où l'aspect formel et automatique de l'outillage donne une garantie sur la qualité des tests produits. L'évaluateur n'a alors pas à passer en revue les centaines de cas tests produits mais peut se limiter à valider que la traduction des exigences en propriétés est correcte et juste.*

10.4.2 Evaluation utilisateur de la méthode

L'évaluation utilisateur consiste en l'évaluation du prototype issu du projet par un ingénieur validation.

L'approche permet à un utilisateur de déplacer la charge de travail de la mise au point manuelle de tests à la description, à un niveau plus abstrait, des propriétés temporelles et d'un modèle fonctionnel dans un langage dédié. Bien qu'une certaine connaissance métier des exigences de sécurité soit nécessaire afin de décliner les cibles de sécurité en propriétés, l'automatisation de ce processus permet un gain de temps et une couverture mieux maîtrisée des tests générés et exécutés. De plus, l'existence d'une suite de tests permet de s'affranchir des incohérences entre le modèle et les propriétés.

Dans le rapport de couverture, la visualisation sous forme d'automate permet une première vérification du comportement de la propriété. De plus, la mesure de couverture permet d'aider à déterminer si une propriété est suffisamment testée par une suite de tests, et de décider s'il faut les compléter au moyen de l'outil de génération automatique.

Dans le rapport de tests, l'ensemble des vues proposées par le rapport de tests permet de répondre aux exigences ATE_FUN et ATE_COV des Critères Communs. En effet, il permet de montrer les relations qui existent tant au niveau de la spécification entre les SFR et les TSFI qu'au niveau de la génération de tests entre ces derniers et les tests générés.

Dans le livrable TASCCC [BG13] qui est le rapport d'évaluation, les évaluateurs soutiennent que l'approche est pertinente dans le cadre de leur activité :

> *La démarche proposée semble pertinente et surtout efficace. Elle permet en utilisant un haut niveau d'abstraction d'obtenir des bases de test complètes et ciblant mieux un périmètre de résultats attendus plus large.*

De plus, ils soulignent l'importance de l'aspect traçabilité du projet dans un cadre industriel :

> *La vue de la traçabilité des tests vers les TSFI et celle vers les SFR est tout à fait exploitable pour un projet industriel. Un coordinateur d'évaluation est à priori à même d'utiliser ces informations pour les revoir et les fournir à un évaluateur. On obtient aisément la liste des propriétés testées, la liste des SFR associées ainsi que la liste des TSFI et des actions associées. Cela permet de répondre aux exigences d'exhaustivité et de traçabilité.*

10.5 Synthèse

Nous avons présenté dans ce chapitre le projet financé dans lequel s'inscrit le cœur des travaux de cette thèse. Les contributions de ces travaux sont surtout rattachées au SP2 et aux liaisons avec le SP1 et le SP3.

Les partenaires évaluateurs de la méthode, Serma Technologies du point de vue Critères Communs et Gemalto du point de vue de l'utilisation du processus, évaluent positivement le projet TASCCC et ce qu'il apporte aux différents acteurs du processus de certification.

Du côté des évaluateurs Critères Communs, ce projet apporte une réelle traçabilité entre les éléments de la spécification informelle, leur traduction dans un langage formel et tout ce qui en découle jusqu'à la génération des tests. Cette traçabilité fait partie des exigences des Critères Communs mais restait une tâche manuelle. Le projet TASCCC donne cette traçabilité à partir des propriétés et des cibles de sécurité qui lui sont rattachés.

Du point de vue des utilisateurs, Gemalto met en avant la simplicité d'utilisation du prototype et le gain de temps dans la mise au point des tests pour les cibles de sécurité. De plus, les vues proposées dans le prototype rendent bien compte de la traçabilité qui est renforcée par le rapport de génération de tests. Le rapport de couverture permet, dans un premier temps, de se rendre compte de la qualité d'une suite de tests vis-à-vis d'une propriété mais aussi de s'assurer de la couverture complète de la propriété à la fin du processus proposé par TASCCC.

Quatrième partie

Conclusions et perspectives

Chapitre 11

Conclusions et perspectives

Sommaire

11.1 Processus 198
11.2 Le langage de propriétés 200
11.3 Les critères de couverture 202
11.4 Le langage de scénarios 203
11.5 Expérimentations 204
11.6 Autres pistes 205

Nous présentons dans ce chapitre les conclusions sur le travail effectué au cours de cette thèse et les pistes de recherche qui en découlent.

Nous avons présenté une technique de génération de tests à partir de propriétés temporelles sur des modèles exprimés en UML/OCL, dont le processus est illustré dans la FIGURE 1.2 de la page 9.

Notre approche permet de répondre aux questions posées lors de l'exposition de notre problématique à la section 1.2, page 5 :

Q I. "Comment exprimer des propriétés sur la dynamique du système ?" Nous apportons deux moyens d'exprimer ces propriétés. Le premier est un langage de scénarios qui permet à l'ingénieur de tests de spécifier de manière abstraite des séquences d'opérations. Le second est le langage de propriétés qui nous permet de systématiser l'écriture de propriétés temporelles sur le système. Ce langage, inspiré des patrons de propriété de Dwyer *et al.*, propose un ensemble restreint de constructions qui permettent cependant l'expression d'une grande majorité de propriétés temporelles couramment utilisées. De plus, nous avons associé à ce langage une sémantique à base d'automates, un formalisme visuel qui permet à l'ingénieur de test une première validation de sa propriété.

Q II. "Dans quelle mesure peut-on tester le système vis-à-vis de propriétés sur sa dynamique ?" Les propriétés en elles-mêmes ne sont pas suffisantes. Pour cela, nous avons défini plusieurs critères, répartis en deux groupes. Le Chapitre 6 présente des critères spécifiques à nos automates de propriétés qui s'inspirent des critères classiques sur les automates. Ces critères nous permettent d'illustrer des exécutions nominales de la propriété dans différentes situations. Dans le Chapitre 7, nous avons défini un autre critère mais qui se focalise sur la robustesse du système vis-à-vis de la propriété. A cette

fin, des mutations nous permettent de produire des évènements différents mais proches d'un évènement fautif de la propriété afin de se rapprocher d'un cas d'erreur et de le provoquer. Ainsi, notre approche permet d'une part d'illustrer des exécutions nominales, et, d'autre part, des exécutions plus problématiques vis-à-vis du système.

Q III. "Dans quelle mesure est-il possible d'automatiser un tel processus ?" Le processus que nous proposons rend possible l'automatisation de notre approche. A partir de l'écriture manuelle des propriétés et de leur représentation en automates qui en découle, nous pouvons appliquer automatiquement les critères sélectionnés par l'ingénieur pour mesurer la couverture d'une suite de tests ou pour générer les scénarios. Le dépliage des scénarios est aussi automatisé, rendant ainsi toute la chaîne automatique, bien que l'ingénieur puisse intervenir à différents points du processus (notamment, l'optimisation des scénarios en vue du dépliage combinatoire).

11.1 Processus

Lors de la présentation des différentes *briques* de notre technique de génération de tests, nous avons donné plusieurs pistes quant à des utilisations possibles de certaines parties. Ainsi, le langage de scénarios ne sert pas uniquement comme langage intermédiaire entre les automates et les tests abstraits, mais aussi comme un moyen de spécifier des séquences d'opérations dans un but de validation du modèle. L'animation des tests produits permet de s'assurer de la cohérence entre le modèle et les propriétés. Il en va de même pour les critères qui nous permettent de générer les tests. La mesure de couverture permet à l'ingénieur d'identifier les faiblesses de couverture d'une suite de tests.

La FIGURE 11.1 présente le processus de génération de tests dans lequel s'intègre nos travaux, les personnages au-dessus signifient que ces étapes sont manuelles. Ce processus comprend deux aspects différents. Le premier est un aspect de validation du modèle et des propriétés. Le second est la génération de tests à partir de ces propriétés.

Dans ce processus, nous avons en parallèle les tâches de **conception du modèle** et de *développement du système*, propres à l'approche MBT. A cela s'ajoute une troisième tâche qui est l'**écriture des propriétés** propres à notre démarche. De ces propriétés, écrites dans un langage *ad hoc*, nous en tirons une représentation sous forme d'**automate de propriété** qui est la sémantique de la propriété. Avec cet automate, l'ingénieur **sélectionne les critères de couverture** qu'il souhaite appliquer pour la **génération de scénarios**. L'ingénieur de test a ensuite la possibilité de modifier les scénarios produits dans le but **d'optimiser** leur dépliage. Enfin, le **dépliage** combinatoire de ces scénarios permet la génération de tests abstraits, au même niveau d'abstraction que le modèle.

Pour l'aspect de vérification du modèle, c'est à ce moment où le processus de conception du modèle peut profiter des tests abstraits générés précédemment. Les tests (qui peuvent aussi provenir d'une source externe) peuvent être **animés sur le modèle**. Si un test échoue, alors l'ingénieur de test doit **analyser la cause de l'échec**. Ainsi, il doit déterminer si cet échec provient d'une erreur dans la propriété, auquel cas il doit **corriger la propriété** et générer les tests une nouvelle fois, ou si la cause provient du modèle, auquel cas c'est une **correction du modèle** qui est nécessaire. Dans les deux cas, cette étape permet de renforcer la confiance que l'on peut avoir dans le modèle et les

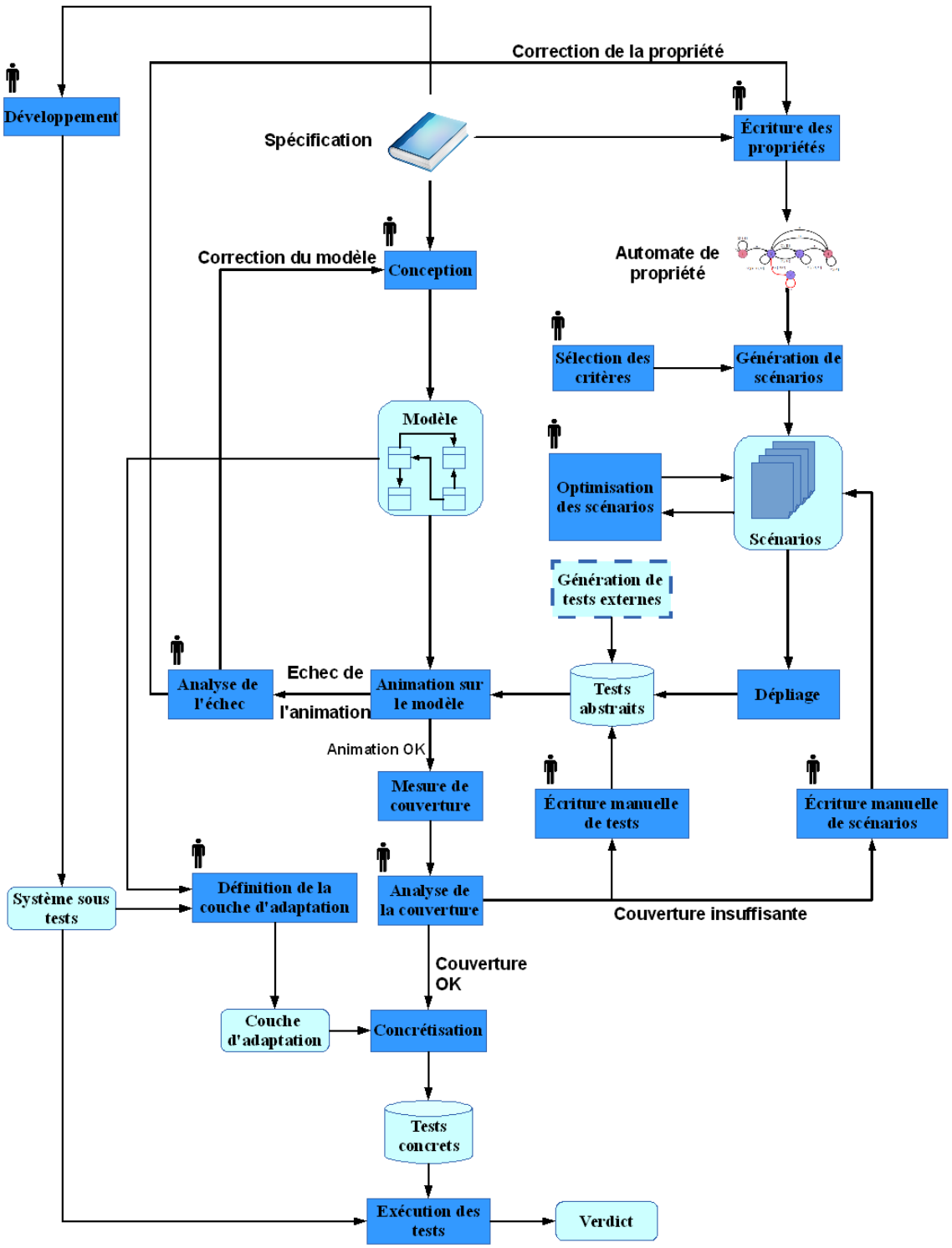


FIGURE 11.1 – Processus itératif de génération de tests et de validation de modèle

propriétés.

Pour l'aspect de génération de test, c'est au moment où les tests sont animés avec succès sur le modèle que celui-ci débute. Nous pouvons **mesurer la couverture** de ces tests sur les propriétés et l'ingénieur doit ensuite en **analyser les résultats**. S'il estime que la couverture est insuffisante, alors il a la possibilité d'**écrire ses tests** ou d'**écrire des scénarios** dans le langage proposé, en choisissant les directives de pilotage. S'il estime, par contre que la couverture est suffisante, alors nous reprenons la fin du processus de génération de tests du MBT. L'ingénieur **définit la couche d'adaptation** entre les éléments du modèle et les éléments du système sous test. Cette couche d'adaptation est ensuite utilisée pour la **concrétisation** des tests abstraits en tests concrets, qui sont enfin **exécutés** sur le système sous test.

Orthogonalement à ce processus, nous avons aussi un aspect itératif pour la conception du modèle et l'écriture des propriétés qui rejoint une pratique courante du test à partir de modèles. Au lieu d'appliquer la méthode sur un modèle déjà complet et un ensemble de propriétés, l'ingénieur de test peut concevoir un modèle plus petit accompagné de propriétés qui ciblent ce modèle. L'ingénieur peut ensuite générer les tests à partir de ces propriétés et valider son modèle. Si le modèle est validé, alors l'ingénieur peut agrandir le modèle et écrire de nouvelles propriétés pour prendre en compte d'autres exigences de la spécification. Ce processus se poursuit itérativement jusqu'à la couverture de toutes les exigences de la spécification. Cette approche à l'avantage d'enrichir progressivement la base de suites de tests, évitant alors de générer tous les tests en une fois, et ainsi, d'assurer une non-régression entre les différentes itérations.

Enfin, nous rappelons que notre approche se fonde exclusivement sur des modèles décrits de manière défensive. En effet, cette propriété sur les modèles considérés nous permet de nous assurer qu'à tout moment, une opération est activable, quel que soit l'état du système courant. Cette pratique permet de modéliser tous les comportements de l'opération, en particulier les comportements exceptionnels, qui sont potentiellemnt intéressants dans le cadre du test à partir de modèles.

11.2 Le langage de propriétés

Nous avons présenté dans le Chapitre 5 un langage de propriétés inspiré des patrons de propriété décrits par Dwyer *et al.* [DAC99] dans le but de faciliter le travail de spécification des exigences pour la génération de tests. Ce langage permet de définir une propriété temporelle comme un *motif* qui doit être satisfait à l'intérieur d'une *portée* (une portion d'exécution du système). Il permet donc d'exprimer simplement un large panel de propriétés temporelles avec un ensemble restreint de constructions. De plus, nous avons ajouté des variantes de motifs et de portées afin d'améliorer l'expressivité du langage par rapport au langage original.

Pour des raisons d'extensibilité et de lisibilité pour l'ingénieur de test, nous avons associé à ce langage une sémantique à base d'automates [CDJT11] par composition des portées et des motifs du langage. Nous avons défini des automates, pour chaque portée et chaque motif, dont chaque transition est étiquetée par un évènement, directement en rapport avec ceux de la propriété. Puis, par un mécanisme de composition de ces

automates, nous obtenons un automate qui représente alors la sémantique de la propriété.

L'avantage de cette sémantique est de donner une visualisation graphique de la propriété à l'ingénieur de test (les automates ont une taille raisonnable, avec au plus 7 états pour la plus grande combinaison), ce qui lui permet une première étape de validation de la propriété qu'il a décrit [BG13].

Enfin, cette approche par composition d'automates nous a permis de relever des incohérences dans les sémantiques LTL proposées dans [DAC99]. La comparaison entre notre sémantique et celle de Dwyer *et al.* a été publiée dans [CCDJ+13].

Perspectives sur le langage de propriétés

Variables. Une piste intéressante est l'utilisation de variables globales dans les propriétés. Ces variables permettraient alors d'utiliser les mêmes données pour des évènements différents. Considérons la propriété suivante : "Un utilisateur ne peut supprimer un ticket pour un film que s'il en a déjà acheté pour ce film auparavant". Dans l'état actuel de notre langage, il n'est pas possible d'exprimer le fait que l'on considère un film quelconque, mais qu'il faut que ce soit le même dans les deux évènements. Pour cela, nous pouvons avoir la construction suivante :

```
let _title : TITLES in
        isCalled(sut.buyTicket, pre:"in_title = _title" on "sut",
                 including:{@AIM:BUY_Nominal Success}
precedes isCalled(sut.deleteTicket, pre:"in_title = _title" on "sut",
                 including:{@AIM:REM_Del_Ticket})
globally
```

Ainsi, le titre *_title* considéré dans les deux évènements doit être le même. Cette amélioration est apparue comme étant une nécessité lors du projet OSeP [25] et permettrait d'ajouter de l'expressivité à notre langage. Cette piste a déjà fait l'objet d'une intégration dans le prototype pour le projet OSeP.

Chaîne d'évènements. Notre langage en l'état ne permet de pas de considérer toutes les constructions du langage de Dwyer *et al.*, en particulier les chaînes d'évènements. Cette amélioration peut se faire à plusieurs endroits. Le langage de propriétés tel que proposé dans le livrable TASCCC [KT12], duquel provient notre langage, dispose déjà de ces constructions. Ainsi, il nous faudrait alors donner la sémantique de ces chaînes dans le cadre de nos automates.

La définition de nouveaux critères ou une adaptation des critères du Chapitre 6 pourrait être nécessaires pour assurer la couverture de ces chaînes d'évènements. Celles-ci peuvent être constituées d'évènements qui se suivent mais pas nécessairement directement. Dans ce cas, il peut être alors intéressant de couvrir des séquences où certaines opérations sont intercalées entre ces évènements de la chaîne.

Ouverture/fermeture des intervalles de portée. Une autre amélioration concerne les chevauchements entre un évènement de la portée et un évènement du motif. Nous avons fait le choix d'occurences d'évènements toujours distinctes. Mais considérons par exemple la propriété `after` A `until` B `eventually A`. Pour illustrer la propriété, il nous faudrait

25. Site OSeP : `http://osep.univ-fcomte.fr/` (2013)

déjà entrer dans la portée avec A, effectuer la transition obligatoire du `eventually A`, avant de terminer par B, soit la séquence A ; A ; B. Avec la fermeture de la portée à gauche, alors l'évènement d'ouverture de la portée et celui de `eventually A` peuvent être confondus et un seul appel à A est alors nécessaire pour couvrir les deux transitions. Il en va de même avec les transitions de fin de portée.

Une piste serait donc de donner les moyens à un ingénieur de test de spécifier plus finement l'ouverture/fermeture des portées. Cette approche a déjà fait l'objet d'une première étude dans [CCDJ+13].

Borne sur les itérations des portées/motifs. Nous pouvons améliorer l'expressivité des motifs `(directly) precedes` et `(directly) follows` et les portées `between (last)` et `after (last) until` en donnant la possibilité de spécifier le nombre maximum d'itérations. Ainsi, nous pourrions écrire la propriété suivante :

```
eventually A
between B and C at most 3 times
```

Ainsi, la propriété possède une information supplémentaire sur la portée qui peut être exploitée par les critères de couverture, par exemple, en limitant (ou en fixant) le nombre d'itérations des critères *k-pattern* et *k-scope*, réduisant ainsi le nombre de chemins infaisables.

Propriété sous forme d'automate. Notre langage ne permet pas la définition de toutes les propriétés possibles, ce qui a été montré avec les travaux de Dwyer *et al.*. Ainsi, une autre piste concerne la possibilité de définir les propriétés directement sous la forme d'automate. Cela permettrait alors à l'ingénieur de test de décrire des propriétés impossibles à décrire avec le langage de propriétés. Par exemple, il serait possible de définir des portées à plusieurs états de substitution ou d'imbriquer des portées. Ces aspects sont déjà prévus dans les définitions de nos automates et de leur composition.

11.3 Les critères de couverture

Nous avons décrit au Chapitre 6 quatre critères de couverture spécifiques à nos propriétés, adaptés des critères de couverture classiques sur les automates. Ces critères permettent de sélectionner les exécutions intéressantes vis-à-vis de la propriété considérée en s'appuyant sur les informations spécifiques aux automates de propriété telles que les évènements, la portée et le motif.

Dans le Chapitre 7, nous avons défini un critère de robustesse sur la propriété. Contrairement aux critères précédents qui permettent d'illustrer différents aspects de la propriété, ce critère permet d'illustrer des exécutions potentiellement dangereuses du système vis-à-vis de la propriété considérée. Pour cela, nous utilisons des opérateurs de mutation spécifiques sur les évènements dangereux afin de générer des évènements qui partagent une certaine proximité avec ces évènements.

Perspectives sur les mutations d'évènements

Les quatre mutations décrites dans le Chapitre 7 sont très simples et permettent de générer des évènements proches mais avec une proximité trop faible, dans le sens où ce sont

FIGURE 11.2 – Exemple de graphe de tags

des affaiblissements très forts de l'évènement d'origine. Un exemple de critère que nous pourrions développer est celui de l'échange de tags. Ce critère permet de considérer les comportements voisins dans le graphe. Considérons le graphe de tags de la FIGURE 11.2. Nous nous apercevons que les nœuds T6 et T3 partagent deux décisions sur les trois de leur gardes comme entre T6 et T5, mais T3 et T5 n'en partagent qu'une. Une piste serait d'établir une relation de proximité entre les tags fondée sur les décisions qui composent sa garde. Là où, en considérant le tag T5, la mutation d'échange de tags ne donnait que le tag T6, cette relation nous permettrait d'obtenir l'ensemble {T3,T6}. Cette relation doit cependant prendre en compte le fait que nous utilisons un langage d'action et que les décisions peuvent différer à cause de la forme SSA.

11.4 Le langage de scénarios

Nous avons décrit dans le Chapitre 4 un langage de scénarios textuel qui permet à un ingénieur de test d'écrire ses propres scénarios de tests abstraits qui correspondent à un objectif de test. Ce langage, inspiré de celui décrit dans [Tis09] à l'occasion du projet POSE, se rapproche d'un langage de programmation tel que Java pour une prise en main plus rapide que son prédécesseur [CB11]. Il intègre des directives de pilotage sur les instructions pour guider le dépliage et limiter l'explosion combinatoire.

Pour le dépliage des scénarios, nous avons opté pour un dépliage combinatoire qui prend en compte les directives de pilotage pour réduire les chemins à explorer et limiter le nombre de tests générés.

Enfin, puisque ce langage est au niveau d'abstraction du modèle, nous pouvons l'utiliser pour écrire des scénarios de test abstraits dans le but de valider le modèle, dans un esprit du test unitaire au niveau abstrait.

Perspectives sur le langage de scénarios

Restriction d'évènements. Nous avons vu que la traduction des restrictions est une étape très lourde et qui pose problème. Ce dépliage est cependant nécessaire car le langage de scénarios ne dispose pas de construction permettant d'écrire simplement une restriction d'évènement.

Une piste pour remédier à cela est l'utilisation d'une construction *ad hoc*. L'analyse de la faisabilité de l'évènement serait décalée au moment du dépliage, en l'analysant à la volée lors de l'animation. Ainsi, la tâche de l'assurance de la faisabilité d'un évènement restreint est délégué à l'animation et non plus à la traduction.

Dépliage symbolique. Un second point problématique dans notre langage de scénarios est le dépliage combinatoire qui n'est clairement pas optimal pour au moins deux raisons. La première est le temps d'exécution, qui peut être très long en fonction des directives de pilotage choisies et de la longueur des scénarios. La seconde concerne les

tests générés. Ces tests sont issus d'une exploration de l'arbre, toujours dans le même ordre. Des chemins potentiellement intéressants peuvent ne pas être considérés car laissés de côté par les directives de pilotage.

Le première piste à envisager pour le langage de scénarios est l'amélioration de la technique de dépliage. En effet, le nœud du problème lors du dépliage combinatoire est celui posé par la construction $OP et les répétitions non bornées + et *. Ces constructions sont génératrices d'une grande combinatoire. Pour remédier à cela, nous avons proposé dans [CB11] une technique de génération de tests à partir de scénarios en utilisant le moteur d'animation symbolique de CertifyIt. Cette approche nous permet de déterminer la faisabilité des chemins du scénario (par détermination de l'atteignabilité des chemins du scénario), au lieu d'attendre l'exploration complète de l'arbre, très coûteuse en temps. Cette approche est implémentée dans le prototype pour le projet OSeP en traduisant les expressions régulières dans le langage de *stories*, langage interne du moteur de résolution symbolique de CertifyIt.

Une autre approche est celle utilisée dans PUT [TS05a], que nous pourrions appliquer pour le dépliage de nos scénarios en tests abstraits.

Directives de pilotage. La seconde piste pour améliorer le langage de scénarios est la définition de directives de pilotage plus fines que celles dont nous disposons actuellement. Par exemple, nous pourrions nous inspirer des critères de réduction de suites de tests définis dans [TLdB+12, LVTB12, TdBL12] pour le projet TASCCC. Par exemple, l'un de ces critères filtre une suite de tests en ne conservant qu'un test sur deux. Nous pourrions appliquer le même raisonnement pour le choix des valuations lors du dépliage.

11.5 Expérimentations

Nous avons conduit une expérimentation sur notre modèle fil rouge eCinema afin d'évaluer la pertinence de notre approche sur deux aspects : la couverture des propriétés et le pouvoir de détection de fautes. Pour cela, nous avons généré des suites de tests pour chaque critère et chaque propriété et nous avons utilisé la suite de tests générée par CertifyIt pour effectuer la comparaison.

La mesure de couverture nous a permis d'identifier des manques de couverture des propriétés par la suite de CertifyIt vis-à-vis de critères de couverture, qui ont été comblés par nos suites de tests. Ainsi, notre approche est complémentaire vis-à-vis d'autres techniques de génération de tests, telle que CertifyIt dans notre cas.

La seconde partie d'expérimentation sur l'évaluation du pouvoir de détection des fautes nous a permis de détecter des fautes qui ne l'étaient pas par la suite de CertifyIt. Nous avons classé les verdicts pour identifier les tests qui détectent des fautes, et plus particulièrement, ceux qui permettent la violation de la propriété.

Enfin, cette expérimentation a fait ressortir un fait intéressant : la possibilité d'utiliser cette approche comme une première étape de validation du modèle et des propriétés. L'une d'elle nous a permis d'identifier une non-conformité avec le modèle. Il s'est avéré que la propriété était mal décrite ce qui nous a permis de la corriger. Cela nous a permis d'avoir une plus grande confiance dans le modèle et les propriétés, et par conséquent, dans les tests générés par leur biais.

11.6 Autres pistes

Au-delà des améliorations proposées pour les points clefs de notre approche, nous pouvons envisager d'autre pistes plus exploratoires.

Test en ligne. Concernant l'utilisation des automates de propriétés pour la génération de tests, nous nous sommes attaché à leur utilisation pour du test *hors ligne*. Une des pistes serait donc d'utiliser nos automates en tant que moniteurs d'un système pour du test *en ligne*. Une telle approche a déjà été étudiée avec l'utilisation des patrons de propriété de Dwyer *et al.* pour la définition de propriétés dans [Dor11]. Cette approche permet de surveiller l'évolution des reconfigurations de composants dans un système par des automates représentant la propriété. A chaque instant, cette approche permet de déterminer si la propriété est *vraie* lorsque la propriété est vérifiée et *fausse* lorsqu'il y a eu violation. Entre ces deux cas, il existe deux cas intermédiaires. Ainsi, la propriété peut être *potentiellement vraie* qui signifie que le système est dans un état temporaire qui respecte la propriété ou elle peut encore être *potentiellement fausse* si la configuration des composants se rapproche trop d'une configuration fautive.

Nous pouvons faire un parallèle entre ces verdicts et les états de nos automates. Ainsi, nous avons le verdict *vrai* lorsque nous sommes dans un état final, *faux* lorsque l'on atteint l'état d'erreur. Les verdicts intermédiaires sont représentés par les états inconclusifs de nos automates. Le verdict *potentiellement faux* fait référence à un état inconclusif qui possède une transition menant à l'état d'erreur et le verdict *potentiellement vrai* concerne les autres états inconclusifs.

Cette approche utilise la sémantique LTL donnée par Dwyer *et al.* et non pas nos automates de propriété. L'avantage de notre approche nous permet d'observer, en plus de l'évolution de la satisfaction de la propriété via l'automate, la couverture des critères en direct.

Aspect temporisé. Une autre amélioration envisageable sur les automates de propriété peut être la prise en compte d'un aspect temporisé dans les évènements. Ainsi, cet ajout pourrait prendre la forme d'une nouvelle composante ajoutée aux types d'évènements *isCalled* et *becomesTrue* ou d'un nouveau type d'évènement, dépendant d'horloges. Ainsi, cela nous permettrait de prendre en considération des propriétés telles que "Après une connexion réussie, s'il n'y a aucune activité de la part de l'utilisateur au bout de 30 minutes, il est automatiquement déconnecté".

L'ajout d'une contrainte de temps aux évènements impliquerait la définition de nouveaux critères de couverture et/ou l'adaptation des critères existants. Enfin, nous pourrions définir d'autres opérateurs de mutation sur les évènements pour le critère de robustesse.

Bibliographie

[ABC+02] Fabrice Ambert, Fabrice Bouquet, Sébastien Chemin, Sébastien Guenaud, Bruno Legeard, Fabien Peureux, and Nicolas Vacelet. Projet BZ-Testing-Tools - Génération de tests aux limites à partir d'un modèle formel B ou Z - Annexes Techniques. Compte rendu d'avancement au 30 avril 2002, ANVAR, May 2002. 150 pages.

[ABM97] Lionel Van Aertryck, Marc V. Benveniste, and Daniel Le Métayer. CASTING : A Formally Based Software Test Generation Method. In *ICFEM*, pages 101–, 1997.

[Abr96] J.-R. Abrial. *The B-book : assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B : System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[AD06] Bernhard K. Aichernig and Carlo Corrales Delgado. From faults via test purposes to test cases : on the fault-based testing of concurrent systems. In *Proceedings of the 9th international conference on Fundamental Approaches to Software Engineering*, FASE'06, pages 324–338, Berlin, Heidelberg, 2006. Springer-Verlag.

[AWW08] Bernhard K. Aichernig, Martin Weiglhofer, and Franz Wotawa. Improving Fault-based Conformance Testing. *Electron. Notes Theor. Comput. Sci.*, 220(1) :63–77, December 2008.

[BB87] Tommaso Bolognesi and Ed Brinksma. Introduction to the iso specification language lotos. *Comput. Netw. ISDN Syst.*, 14(1) :25–59, March 1987.

[BBFM99] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Météor : A Successful Application of B in a Large Project. In *Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume I - Volume I*, FM '99, pages 369–387, London, UK, UK, 1999. Springer-Verlag.

[BDGJ06] Fabrice Bouquet, Frédéric Dadeau, Julien Groslambert, and Jacques Julliand. Safety property driven test generation from JML specifications. In *Proceedings of the First combined international conference on Formal Approaches to Software Testing and Runtime Verification*, FATES'06/RV'06, pages 225–239, Berlin, Heidelberg, 2006. Springer-Verlag.

[BDL06] Fabrice Bouquet, Frédéric Dadeau, and Bruno Legeard. Automated boundary test generation from JML specifications. In *Proceedings of the 14th*

international conference on Formal Methods, FM'06, pages 428–443, Berlin, Heidelberg, 2006. Springer-Verlag.

[BDLU05] Fabrice Bouquet, Frédéric Dadeau, Bruno Legeard, and Mark Utting. JML-Testing-Tools : a symbolic animator for JML specifications using CLP. In N. Halbwachs and L. Zuck, editors, *Procs of the 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, Tool session (TACAS'05)*, volume 3440 of *LNCS*, pages 551–556, Edinburgh, United Kingdom, April 2005. Springer.

[BFM+09] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL : ANSI/ISO C Specification Language, version 1.4*, 2009. `http://frama-c.cea.fr/acsl.html`.

[BG13] Julien Bernet and François Guérin. Livrable TASCCC 5.5 : Rapport sur l'utilisation industrielle du processus outillé. Technical report, Gemalto, february 2013.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *J. ACM*, 31(3) :560–599, June 1984.

[Bin99] Robert V. Binder. *Testing object-oriented systems : models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[BJE94] K. Burroughs, A. Jain, and R.L. Erickson. Improved quality of protocol testing through techniques of experimental design. In *Communications, 1994. ICC '94, SUPERCOMM/ICC '94, Conference Record, 'Serving Humanity Through Communications.' IEEE International Conference on*, pages 745–752 vol.2, 1994.

[BLP02] Fabrice Bouquet, Bruno Legeard, and Fabien Peureux. Clps-b - a constraint solver for b. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '02, pages 188–204, London, UK, UK, 2002. Springer-Verlag.

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system : an overview. In *Proceedings of the 2004 international conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 49–69, Berlin, Heidelberg, 2005. Springer-Verlag.

[BMdB+01] P. Bontron, O. Maury, L. du Bousquet, Y. Ledru, C. Oriat, and M.-L. Potet. TOBIAS : un environnement pour la création d'objectifs de test à partir de schémas de test. In *14th International Conference Software & Systems Engineering and their Applications - ICSSEA'2001*, Paris, France, 2001.

[BOY00] Paul E. Black, Vadim Okun, and Yaacov Yesha. Mutation operators for specifications. In *Proceedings of the 15th IEEE international conference on Automated software engineering*, ASE '00, pages 81–, Washington, DC, USA, 2000. IEEE Computer Society.

[CB11] Kalou Cabrera Castillos and Julien Botella. Scenario based test generation using test designer. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '11, pages 79–88, Washington, DC, USA, 2011. IEEE Computer Society.

[CCDJ+13] Kalou Cabrera Castillos, Frédéric Dadeau, Jacques Julliand, Bilal Kanso, and Safouan Taha. A compositional automata-based semantics for property patterns. In E.B. Johnsen and L. Petre, editors, *iFM 2013, 10th International Conference on integrated Formal Methods*, volume 7940 of *LNCS*, pages 316–330, Turku, Finland, june 2013. Springer.

[CD10] Kalou Cabrera Castillos and Frédéric Dadeau. Livrable TASCCC 2.1 : Définition d'un langage d'expression de scénarios. Technical report, FEMTO/-DISC, september 2010.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE : unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

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

[CDHR02] James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. Expressing checkable properties of dynamic systems : the Bandera Specification Language. *STTT*, 4(1) :34–56, 2002.

[CDJT11] Kalou Cabrera Castillos, Frédéric Dadeau, Jacques Julliand, and Safouan Taha. Measuring test properties coverage for evaluating uml/ocl model-based tests. In *Proceedings of the 23rd IFIP WG 6.1 international conference on Testing software and systems*, ICTSS'11, pages 32–47, Berlin, Heidelberg, 2011. Springer-Verlag.

[CDPP96] David M. Cohen, Siddhartha R. Dalal, Jesse Parelius, and Gardner C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 13(5) :83–88, 1996.

[CGP+08] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE : Automatically Generating Inputs of Death. *ACM Trans. Inf. Syst. Secur.*, 12(2) :10 :1–10 :38, December 2008.

[CJRZ01] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. STG : a tool for generating symbolic test programs and oracles from operational specifications. *SIGSOFT Softw. Eng. Notes*, 26(5) :301–302, September 2001.

[CKK+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C : a software analysis perspective. In *Proceedings of the 10th international conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE : a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, pages 178–188, New York, NY, USA, 1987. ACM.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 411–420, New York, NY, USA, 1999. ACM.

[DAC11] M. Dwyer, G. Avrunin, and J. Corbett. A system of specification patterns, March 2011.

[DC12] Frédéric Dadeau and Kalou Cabrera Castillos. Livrable TASCCC 2.2 : Définition du processus de génération de tests à partir de propriétés et de besoins de tests. Technical report, FEMTO/DISC, may 2012.

[DCL+13] Frederic Dadeau, Kalou Cabrera Castillos, Yves Ledru, Taha Triki, German Vega, Julien Botella, and Safouan Taha. Test generation and evaluation from high-level properties for common criteria evaluations – the tasccc testing tool. In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, ICST '13, pages 431–438, Washington, DC, USA, 2013. IEEE Computer Society.

[Dor11] Julien Dormoy. *Contributions à la spécification et à la vérification des reconfigurations dynamiques dans les systèmes à composants.* PhD thesis, LIFC, Université de Franche-Comté, 16 décembre 2011.

[EH86] E. Allen Emerson and Joseph Y. Halpern. s̈ometimesänd n̈ot neverr̈evisited : on branching versus linear time temporal logic. *J. ACM*, 33(1) :151–178, January 1986.

[FABA10] Michael Felderer, Berthold Agreiter, Ruth Breu, and Álvaro Armenteros. Security Testing by Telling TestStories. In Gregor Engels, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung*, volume 161 of *LNI*, pages 195–202. GI, 2010.

[FBCO+09] Michael Felderer, Ruth Breu, Joanna Chimiak-Opoka, Michael Breu, and Felix Schupp. Concepts for model-based requirements testing of service oriented systems. *Proceedings of the IASTED International Conference*, 642 :018, 2009.

[FMFR08] Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. j-post : a java toolchain for property-oriented software testing. *Electron. Notes Theor. Comput. Sci.*, 220(1) :29–41, December 2008.

[FOB+11] Elizabeta Fourneret, Martin Ochoa, Fabrice Bouquet, Julien Botella, Jan Jurjens, and Parvaneh Yousefi. Model-Based Security Verification and Testing for Smart-cards. In *Proceedings of the 2011 Sixth International Conference on Availability, Reliability and Security*, ARES '11, pages 272–279, Washington, DC, USA, 2011. IEEE Computer Society.

[FW88] P. G. Frankl and E. J. Weyuker. An Applicable Family of Data Flow Testing Criteria. *IEEE Trans. Softw. Eng.*, 14(10) :1483–1498, October 1988.

[FW06] Gordon Fraser and Franz Wotawa. Property relevant software testing with model-checkers. *SIGSOFT Softw. Eng. Notes*, 31(6) :1–10, November 2006.

[GG06] Alain Giorgetti and Julien Groslambert. Jag : Jml annotation generation for verifying temporal properties. In Luciano Baresi and Reiko Heckel, editors, *Fundamental Approaches to Software Engineering*, volume 3922 of *Lecture Notes in Computer Science*, pages 373–376. Springer Berlin Heidelberg, 2006.

[Gil62] A. Gill. *Introduction to the Theory of Finite State Machines*. McGraw-Hill, New York, 1962.

[Got09] Arnaud Gotlieb. Euclide : A constraint-based testing framework for critical c programs. In *ICST*, pages 151–160. IEEE Computer Society, 2009.

[Gra08] C. Grandpierre. *Stratégies de génération automatique de tests à partir de modèles comportementaux UML/OCL*. PhD thesis, LIFC, Université de Franche-Comté, 17 juillet 2008.

[HCL+03] Hyoung Seok Hong, Sung Deok Cha, Insup Lee, Oleg Sokolsky, and Hasan Ural. Data flow testing as model checking. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 232–242, Washington, DC, USA, 2003. IEEE Computer Society.

[HD01] John Hatcliff and Matthew B. Dwyer. Using the Bandera Tool Set to Model-Check Properties of Concurrent Java Software. In *Proceedings of the 12th International Conference on Concurrency Theory*, CONCUR '01, pages 39–58, London, UK, UK, 2001. Springer-Verlag.

[HLN+88] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and a. Shtul-Trauring. Statemate : a working environment for the development of complex reactive systems. In *Proceedings of the 10th international conference on Software engineering*, ICSE '88, pages 396–406, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[Hoa85] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[HRR91] Kirsten M. Hansen, Anders P. Ravn, and Hans Rischel. Specifying and verifying requirements of real-time systems. In *Proceedings of the conference on Software for citical systems*, SIGSOFT '91, pages 44–54, New York, NY, USA, 1991. ACM.

[HU00] Olaf Henniger and Hasan Ural. Test Generation Based On Control And Data Dependencies Within Multi-Process SDL Specifications. In Edel Sherratt, editor, *SAM*, pages 189–202. VERIMAG, IRISA, SDL Forum, 2000.

[Int05] International Standard ISO/IEC 7816–4, second edition, 2005. Available at www.iso.org.

[ITU02] ITU-T. *ITU-T Rec. Z.100 – Formal description techniques (FDT) – Specification and Description Language (SDL)*, 2002. Más información en `www.sdl-forum.org`.

[JJ05] Claude Jard and Thierry Jéron. TGV : theory, principles and algorithms : A tool for the automatic synthesis of conformance test cases for

non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf.*, 7(4) :297–315, August 2005.

[KHBC99] Young G. Kim, Hyoung S. Hong, Doo H. Bae, and Sung D. Cha. Test cases generation from uml state diagrams. *Software, IEE Proceedings- [see also Software Engineering, IEE Proceedings]*, 146(4) :187–192, 1999.

[KT12] Bilal Kanso and Safouan Taha. Livrable TASCCC 1.2 : Définition des patrons de conception de propriétés. Technical report, Supélec, march 2012.

[Lam00] Axel van Lamsweerde. Formal specification : a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 147–159, New York, NY, USA, 2000. ACM.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML : A Notation for Detailed Design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[LDdB+07] Yves Ledru, Frédéric Dadeau, Lydie du Bousquet, Sébastien Ville, and Elodie Rose. Mastering combinatorial explosion with the tobias-2 test generator. In R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer 0002, editors, *ASE*, pages 535–536. ACM, 2007.

[LK83] J. W. Laski and B. Korel. A Data Flow Oriented Program Testing Strategy. *IEEE Trans. Softw. Eng.*, 9(3) :347–354, May 1983.

[LL96] Gerard Le Lann. The Ariane 5 Flight 501 Failure - A Case Study in System Engineering for Computing Systems. Rapport de recherche RR-3079, INRIA, 1996. Projet REFLECS.

[LPU02] Bruno Legeard, Fabien Peureux, and Mark Utting. Automated Boundary Testing from Z and B. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*, FME '02, pages 21–40, London, UK, UK, 2002. Springer-Verlag.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2 :219–246, 1989.

[LT93] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7) :18–41, July 1993.

[LVTB12] Yves Ledru, German Vega, Taha Triki, and Lydie du Bousquet. Test suite selection based on traceability annotations. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 342–345, New York, NY, USA, 2012. ACM.

[Mar95] Bruno Marre. LOFT : A Tool for Assisting Selection of Test Data Sets from Algebraic Specifications. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT*, volume 915 of *Lecture Notes in Computer Science*, pages 799–800. Springer, 1995.

[McM92] Kenneth Lauchlin McMillan. *Symbolic model checking : an approach to the state explosion problem.* PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209.

[Mil82] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[MLdB03] O. Maury, Y. Ledru, and L. du Bousquet. Intégration de TOBIAS et UCASTING pour la génération. In *16th International Conference Software and Systems and their applications-ICSSEA*, Paris, 2003.

[MP92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems.* Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[MRK+97] L. E. Moser, Y. S. Ramakrishna, G. Kutty, P. M. Melliar-Smith, and L. K. Dillon. A graphical environment for the design of concurrent real-time systems. *ACM Trans. Softw. Eng. Methodol.*, 6(1) :31–79, January 1997.

[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, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, The Netherlands, 1997.

[MS04] Glenford J. Myers and Corey Sandler. *The Art of Software Testing (2. ed.* John Wiley & Sons, 2004.

[Nta84] Simeon C. Ntafos. On Required Element Testing. *IEEE Trans. Software Eng.*, 10(6) :795–803, 1984.

[OA99] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723, pages 416–429. Springer, 1999.

[Obj12] Object Management Group. Unified Modeling Language (UML) infrastructure specification, version 2.4.1. OMG Document formal/2012-05-06, May 2012.

[Obj13] Object Management Group. Action Language for Foundational UML (Alf) Concrete Syntax for a UML Action Language. OMG Document formal/2013-03-03, March 2013.

[OXL99] A Jefferson Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for Generating Specification-Based Tests. In *Proceedings of the 5th International Conference on Engineering of Complex Computer Systems*, ICECCS '99, pages 119–, Washington, DC, USA, 1999. IEEE Computer Society.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[PP04] Alexander Pretschner and Jan Philipps. Methodological Issues in Model-Based Testing. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 281–291. Springer, 2004.

[RBJ00] Vlad Rusu, Lydie du Bousquet, and Thierry Jéron. An Approach to Symbolic Test Generation. In *Proceedings of the Second International Conference on Integrated Formal Methods*, IFM '00, pages 338–357, London, UK, UK, 2000. Springer-Verlag.

[RG99] Johannes Ryser and Martin Glinz. A Scenario-Based Approach to Validating and Testing Software Systems Using Statecharts. In *In 12th International Conference on Software and Systems Engineering and their Applications (ICSSEA'99)*, page 7, 1999.

[RJB04] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2^{nd} edition, 2004. ISBN 0321245628.

[Rou12] Davy Rouillard. Livrable TASCCC 5.4 : Rapport sur la prise en compte des exigences ATE. Technical report, Serma Technologies, november 2012.

[RTC82] RTCA. DO-178B : Software Considerations in Airborne Systems and Equipment Certification, 1982.

[RW85] Sandra Rapps and Elaine J. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Trans. Softw. Eng.*, 11(4) :367–375, April 1985.

[SHJ11] Rupert Schlick, Wolfgang Herzner, and Elisabeth Jöbstl. Fault-based generation of test cases from uml-models : Approach and some experiences. In Francesco Flammini, Sandro Bologna, and Valeria Vittorini, editors, *Computer Safety, Reliability, and Security*, volume 6894 of *Lecture Notes in Computer Science*, pages 270–283. Springer Berlin Heidelberg, 2011.

[Spi92] J. M. Spivey. *The Z notation : a reference manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1992.

[TB03] Jan Tretmans and Ed Brinksma. TorX : Automated Model-Based Testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering*, pages 31–43, December 2003.

[TdBL12] Taha Triki, Lydie du Bousquet, and Yves Ledru. Réduction de suites de tests avec des critères d'équivalence basés sur la couverture structurelle. In *Actes de la Conférence AFADL'12 : Approches Formelles dans l'Assistance au Développement de Logiciels*, pages 120–134, Grenoble, January 2012.

[TH02] Kerry Trentelman and Marieke Huisman. Extending JML Specifications with Temporal Logic. In *Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*, AMAST '02, pages 334–348, London, UK, UK, 2002. Springer-Verlag.

[Tis09] Regis Tissot. *Contribution à la génération automatique de tests à partir de modèles et de schémas de test comme critères de sélection synamiques*. PhD thesis, Université de Besançon, Besançon, France, 2009.

[TLdB+12] Taha Triki, Yves Ledru, Lydie du Bousquet, Frédéric Dadeau, and Julien Botella. Model-Based Filtering of Combinatorial Test Suites. In Juan de Lara and Andrea Zisman, editors, *FASE*, volume 7212 of *Lecture Notes in Computer Science*, pages 439–454. Springer, 2012.

[Tre96a] Jan Tretmans. Conformance testing with labelled transition systems : implementation relations and test generation. *Comput. Netw. ISDN Syst.*, 29(1) :49–79, December 1996.

[Tre96b] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3) :103–120, 1996.

[Tre04] Jan Tretmans. Model-based testing : Property checking for real, 2004. Keynote address at the International Workshop for Construction and Analysis of Safe Secure and Interoperable Smart Devices, 2004. Available : `http://www-sop.inria.fr/everest/events/cassis04`.

[TS05a] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 253–262, New York, NY, USA, 2005. ACM.

[TS05b] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests with unit meister. In Michel Wermelinger and Harald Gall, editors, *ESEC/SIGSOFT FSE*, pages 241–244. ACM, 2005.

[TSL04] Li Tan, Oleg Sokolsky, and Insup Lee. Specification-based testing with linear temporal logic. In Du Zhang, Éric Grégoire, and Doug DeGroot, editors, *IRI*, pages 493–498. IEEE Systems, Man, and Cybernetics Society, 2004.

[UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing : A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[USW00] H. Ural, K. Saleh, and A. Williams. Test generation based on control and data dependencies within system specifications in SDL. *Comput. Commun.*, 23(7) :609–627, March 2000.

[Win09] Judd Winick. Batman, Juin 2009.

[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.

[WM01] Jeremiah Wittevrongel and Frank Maurer. Scentor : Scenario-based testing of e-business applications. In *Proceedings of the 10th IEEE International Workshops on Enabling Technologies : Infrastructure for Collaborative Enterprises*, WETICE '01, pages 41–48, Washington, DC, USA, 2001. IEEE Computer Society.

[WMMR05] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. Pathcrawler : Automatic generation of path tests by combining static and dynamic analysis. In Mario Dal Cin, Mohamed Kaâniche, and András Pataricza, editors, *EDCC*, volume 3463 of *Lecture Notes in Computer Science*, pages 281–292. Springer, 2005.

Résumé

Les travaux proposés dans cette thèse, effectuée dans le cadre du projet ANR TASCCC, présentent une technique de génération de tests à partir de modèles comportementaux en UML/OCL et de propriétés temporelles. Pour cela, nous décrivons un langage de propriétés temporelles inspiré des patrons de propriétés introduits par M. Dwyer *et al.*. Une propriété est définie comme la combinaison d'une *portée*, qui représente les exécutions du système dans laquelle un *motif* doit être satisfait. Nous associons à chaque portée et motif une sémantique à base d'automates particuliers, les automates de substitution. Par un mécanisme de substitution d'un automate de motif dans un automate de portée, nous obtenons un automate représentant la sémantique de la propriété.

Nous avons ensuite défini des critères de couverture nominaux, inspirés des critères de couverture classiques sur les automates, spécifiques à nos automates de propriété. Ces critères se concentrent sur les informations supplémentaires apportées par la propriété originale, telles que ses évènements, sa portée et son motif. En complément, nous avons défini un critère de couverture qui, par le biais de mutation d'évènements de certaines transitions, permet de cibler des exécutions potentiellement dangereuses du système en tentant de provoquer les évènements interdits de la propriété.

Ensuite, nous avons défini pour chaque critère un algorithme qui permet d'extraire des chemins dans l'automate, ciblant les éléments du critère considéré. Ces chemins sont traduits en scénarios dans un langage que nous avons défini. Enfin, un dépliage combinatoire de ces scénarios, éventuellement guidé par des directives de pilotage intégrées à celui-ci, permet la génération de cas de tests abstraits.

Finalement, cette approche a été validée par une expérimentation sur une étude de cas dans ce document et sur GlobalPlatform, l'étude de cas de taille industrielle dans le cadre du projet TASCCC.

Mots-clés: test à partir de modèles, UML/OCL, patrons de propriétés, scénarios

Abstract

In the work presented in this thesis, supported by the ANR TASCCC project, we propose a test generation technique using behavioral models in UML/OCL and temporal properties. To this end, we describe a temporal property langage based from the property patterns introduced by M. Dwyer *et al.*. A property is a combination of a *scope*, representing the considered execution pathes in the system, and a *pattern*, a property that has to be satisfied inside the *scope*. We then give to each scope and pattern a specific automata-based semantics, called substitution automata. By combining a scope automaton with a pattern automaton, we obtain an automaton representing the semantics of the property.

Next, we described nominal coverage criteria, based on classical coverage criteria over automata, specific to our property automata. These criteria focus on informations from the originating property, such as the property events, its scope and its pattern. We complemented this approach with another criterion that, by mutations over events held by certains transitions, aims at activating potentially faulty executions of the system by provoking forbidden events in the automata.

We then described for each criterion an algorithm that aims at extracting paths in the automata with respect to the considered criterion. These paths are then translated to scenarios in an *ad hoc* langage that we defined. Finally, an unfolding process over these scenarios, with the help of driving commands embedded in the scenarios, allows the generation of abstract test cases.

Finally, our approach has been validated on the case study presented in this document and on GlobalPlatform, an industrial-sized case study in the TASCCC project.

Keywords: Model-based testing, UML/OCL, property patterns, scenarios