Département Informatique
des Systèmes Complexes

Femto-ST

Rapport de stage de Master II Recherche

Génération automatique de tests à partir de modèles UML/ALF

Auteur :
Alexandre Vernotte

Tuteurs :
Fabrice Bouquet
Frédéric Dadeau
Fabien Peureux

24 septembre 2012

Remerciements

Je souhaite remercier Fabrice Bouquet pour avoir su aiguiller mes recherches et m'expliquer de nombreuses notions.

Je remercie aussi Frédéric Dadeau pour sa disponibilité et ses conseils.

Je tiens également à remercier Fabien Peureux pour son aide cruciale, particulièrement dans la précipitation des derniers jours de travail.

Enfin, je remercie Lucie Gobet pour le temps consacré à la relecture de mon mémoire.

Table des matières

Table des figures

Chapitre 1

Introduction

Contents

Alors que la complexité des systèmes informatisés ne cesse de s'accroître, leur part logicielle est de plus en plus conséquente. De nombreuses catastrophes liées au logiciel sont survenues par le passé : le crash de la fusée Ariane 5, le bogue de carte de crédit en Allemagne, etc... Pour empêcher ce genre de catastrophes, la communauté scientifique concentre ses efforts sur l'amélioration des techniques de prévention d'erreurs.

Une de ces techniques est la vérification formelle : l'objectif est d'établir la preuve, à l'aide de logique mathématique, que le système respecte un certain nombres de propriétés. Cependant, la vérification formelle est extrêmement coûteuse (en ressources humaines et matérielles), et n'est justifiable que pour des systèmes critiques (systèmes de transport, systèmes financiers, etc...). Une alternative à la preuve de programme est le test : le but n'est pas d'exécuter un logiciel avec toutes les valeurs d'entrées possibles, mais plutôt d'élire certaines valeurs comme représentants des différents comportements du programme. L'effort est moindre, et pour la plupart des systèmes, la confiance obtenue suffisante.

1.1 Test logiciel

Le test logiciel est une activité très vaste et il existe de nombreux types de test. Une classification élaborée par Tretmans [**?**] propose d'ordonner les différents types de tests selon trois axes, représentés en figure 1.1. Il ne s'agit pas de réaliser un parcours exhaustif de toutes les pratiques existantes, mais de mettre en exergue les principales familles de test connues.

Le premier axe définit les différents niveaux sur lesquels intervient le test logiciel. Le test logiciel est présent sur quatre niveaux différents :

- Le test unitaire et le test de module (ou test de composant) ont pour objectif de détecter un maximum de fautes d'une procédure, d'un module.

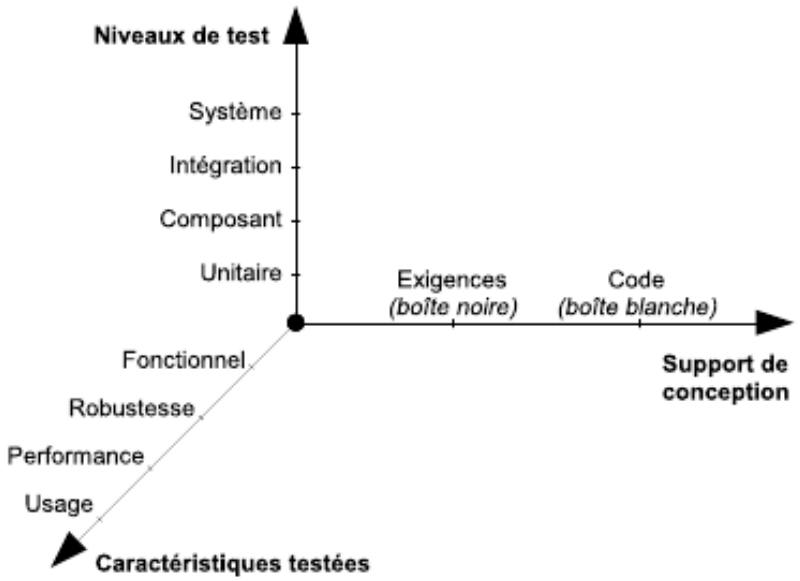

FIGURE 1.1 – Classification tri-dimensionnelle des approches de test

- Le test d'intégration aide à vérifier le bon comportement du système lors de la composition de procédures et modules.
- Le test sous-système et système permettent de vérifier qu'un système respecte ses spécifications initiales.

Le second axe classifie les différents types de tests selon le caractère testé du système :

- Le test fonctionnel vérifie que les différents comportements du système coïncident avec ceux décrits dans ses spécifications fonctionnelles.
- Le test de robustesse consiste à exécuter le système avec des données d'entrée invalides, pour tester son fonctionnement dans des conditions inhabituelles, ou extrêmes d'usage.
- Le test de performance a pour objectif de vérifier que les performances annoncées dans la spécification sont bien atteintes. Il est généralement question de mesurer le temps de réponse du système lorsque celui-ci est soumis à des techniques comme le test de tenue de charge (Load Testing), ou la simulation de crise (Stress Testing).
- Le test d'usage (ou d'utilisabilité) consiste à faire exécuter le système par des utilisateurs. L'objectif de test est généralement d'ordre ergonomique.

Le troisième axe concerne deux grands types de test traditionnels considérés comme complémentaires.

- Le test boîte-noire ignore tout ce qui est lié à l'implémentation du système, et les cas de test sont obtenus en se basant sur les spécifications du système. Généralement, il s'agit de tests fonctionnels qui ont à charge de vérifier que les différents comportements du système sont conformes à ses spécifications. Les cas de test consistent

en un certain ensemble de valeur d'entrées, et vérifient que le résultat (les valeurs ou le comportement de retour) correspond à celui attendu.

- Le test boîte-blanche, contrairement au test boîte-noire, est conçu à partir de la structure interne du système, c'est-à-dire le code source, les algorithmes, les données, etc... L'objectif est d'obtenir une couverture structurelle du système la plus optimale possible. De nombreux critères permettent d'évaluer cette couverture (voir section 3.1).

- On rencontre dans la littérature une troisième notion, nommée test boîte-grise, qui consiste à combiner des tests de type boîte-blanche et des tests de type boîte-noire. En effet, les tests générés sont de type fonctionnel mais la structure interne du système est en partie ou complètement accessible et peut influencer les caractéristiques des tests obtenus.

Nous nous intéresserons dans ce document aux solutions Model-Based Testing. Ces solutions utilisent un modèle du système sous test pour produire des cases de test fonctionnels. D'après les trois axes de classification des types de test présentés en figure 1.1, les solutions MBT produisent des cas de test fonctionnels de type boîte noire, et interviennent à tous les niveaux de tests (d'unitaire à système).

1.2 Le Concept Model-Based Testing

Le test à partir de modèles ou Model-Based Testing (MBT) [85] est une technique de test particulière qui consiste à créer un modèle comportemental du système sous test (ou System Under Test notée SUT) basé sur une liste d'exigences fonctionnelles, dans le but de générer une suite de cas de test. Ces tests sont ensuite concrétisés pour être joués sur le SUT, et les résultats obtenus sont comparés à ceux attendus (à l'aide d'un oracle de test). Cette approche connait une forte croissance parce qu'elle tend à apporter des solutions aux problèmes liés au test traditionnel. D'abord, le fait que la modélisation soit issue d'exigences fonctionnelles permet de pouvoir débuter très tôt dans le cycle de vie du système, voire en parallèle du développement. Ensuite, les techniques MBT sont adaptées aux systèmes qui connaissent de fréquents changements : une modification apportée au SUT peut être facilement répercutée sur le modèle. Si la phase de concrétisation a été automatisée, Il ne reste qu'à régénérer les tests, et de précieuses heures sont économisées. En outre, la production des cas de test étant généralement automatisée et confiée à des algorithmes de génération solides permet de s'affranchir de l'erreur humaine et d'obtenir une suite de test de meilleure qualité. Enfin, il existe plusieurs centaines d'approches classées MBT, et des outillages sont disponibles pour automatiser complètement le mécanisme de test.

Utting, Pretschner et Legeard proposent d'aborder le procédé MBT comme une composition de 4 étapes chronologiques [85]. Les sections suivantes ont pour objectif de décrire chacune de ces étapes. La première étape consiste en l'élaboration du modèle selon une liste d'exigences ou une spécification. La deuxième étape oriente les tests par le choix de critères de sélection de test. La troisième étape réalise la génération de tests. La quatrième étape comprend l'exécution des tests et leur comparaison avec les valeurs attendues.

1ère Etape - Modélisation du SUT

La première étape d'un cycle MBT consiste en la construction d'un modèle de test comportemental du système depuis une liste d'exigences ou une spécification. Il est possible d'utiliser le même modèle à la fois pour la conception du logiciel et la génération des tests : ce type de procédé permet la génération automatique du code, et le type de tests générés sera d'ordre structurel et portera donc sur le code. Cependant, nous n'aborderons pas ce genre d'approche. Il apparaît préférable de dissocier les modélisations, pour d'une part éviter que des erreurs en provenance du développement ne viennent se répercuter sur la génération des tests [75], et d'autre part pour faciliter la validation du modèle par l'abstraction de fonctionnalités, et donc le travail de génération (il doit être néanmoins suffisamment précis pour rester fidèle à la spécification).

Un modèle peut ne spécifier que les données d'entrée. La spécification est plus simple mais les tests générés ne pourront être utilisés comme oracle de test. Ce type de pratique vise généralement à représenter l'environnement d'un système plutôt que son comportement. Lorsque le modèle spécifie les données d'entrée et de sortie du SUT, il lui est possible de prédire les sorties attendues pour chaque entrée. C'est sur ce type de modèle que l'attention sera portée.

De nombreuses notations et paradigmes peuvent être sollicités pour la phase de modélisation. Ces méthodes peuvent êtres classées selon sept catégories :

- **Notations pré/post :** Le système est modélisé comme une collection de variables, avec des opérations qui modifient ces variables. Les opérations sont définies via des conditions pré/post, spécifiées avec Z [82], B [79], VDM [74], JML [9], OCL [54],...

- **Notations basées sur les transitions :** Il s'agit de décrire les transitions entre les différents états du système. On utilise une notation graphique (statecharts de Harel [46], UML State Machines [55], ...) comprenant des *nœuds* (états du système) et des *arcs* (transitions du système).

- **Notations fonctionnelles :** Le système est modélisé comme une collection de fonctions mathématiques, selon la logique du premier ordre, ou d'ordre supérieur. Se référer aux travaux de Gaudel et Legall [35].

- **Notations basées sur l'historique :** Ces méthodes proposent de décrire les traces d'exécutions comportementales dans le temps. Il existe plusieurs types de notions temporelles pour plusieurs types de logiques. Sont par exemple inclus dans ce groupe les Message-Sequence Charts (MSC [47]).

- **Notations opérationnelles :** Le système est modélisé comme une collection de processus exécutables, lancés en parallèle. Cette notation est utilisée pour les systèmes distribués et les protocoles de communication. Les algèbres de processus [27] comme CSS, CSP ou ASP et les réseaux de Petri [76] sont des notations opérationnelles.

- **Notations stochastiques :** Le système est modélisé par un modèle probabilistique des événements et données d'entrée. Ce genre de notation, comme les chaines de markov [28], est généralement utilisé davantage pour les environnements de modèles que pour les SUT.

- **Notations flux de données :** Ce paradigme est davantage orienté sur les données que sur le flux de contrôle (exemple : Lustre [72], diagrammes de block de MATLAB Simulink [33]).

A noter qu'il est possible d'appliquer plusieurs paradigmes avec une seule notation. Ce sera le cas dans ce document, qui traitera du langage UML, permettant l'utilisation du paradigme basé sur les transitions qu'exploitent les statesharts, et le paradigme pré/post qu'exploitent les expressions OCL et les actions Alf.

2ème Etape - Définition des critères de sélection de test

L'objectif du test logiciel est de trouver des erreurs de comportement ou d'accroître la confiance que l'on accorde à un système. L'une des difficultés majeures de cette pratique est de déterminer quand s'arrêter. Il n'est pas réaliste de tester exhaustivement le domaine de valeurs des variables d'entrée du système. Des critères de sélection sont ainsi définis pour permettre aux ingénieurs de test de décider lorsqu'un système a été convenablement testé selon un critère choisi.

Il n'y a pas de "meilleur critère". Chaque type de test implique un certain niveau de confiance sur un aspect du système, et c'est à l'ingénieur validation d'évaluer quel critère est le mieux adapté compte tenu des objectifs de tests (robustesse, sécurité, performance, etc...) et du niveau de confiance requis.

Dans la littérature six critères sont généralement évoqués :

- **Critère de couverture structurelle du modèle :** Ce critère est basé sur la structure du modèle, par exemple les nœuds et arcs d'une machine à états (critères de couverture de graphe [67], comme *tous-les-nœuds* ou *tous-les-arcs*), ou les conditions dans un modèle pre/post, etc... Des critères de couverture de test de type boîte-blanche (voir section 1.1), initialement destinés à la couverture de code, ont été adaptés aux modèles comprenant des variables.

- **Critères de couverture des données :** L'ingénieur validation choisit un nombre restreint de données de test dans un vaste espace de données. Le principe est de découper l'espace de données en groupes, chaque groupe comprenant des données ayant la capacité à détecter les même erreurs, et de choisir un représentant par groupe. Exemple : couverture *Pairwise* et *N-way* [45], partitionnement [87], test aux limites [58], etc...

- **Critères de couverture d'exigences :** Si des exigences sont explicitement liées à des éléments du modèles, il est alors possible de les couvrir (pour s'assurer que les exigences sont respectées).

- **Spécifications de cas de test *Ad Hoc* :** Une spécification vient compléter le modèle pour déterminer quels tests seront générés. Exemple : orienter les tests vers un cas d'utilisation particulier, ou à l'inverse forcer la non-couverture de certains cas.

- **Critères Stochastiques et aléatoires :** Ce type d'orientation est principalement destiné aux modèles environnementaux (l'environnement du modèle lui dicte son comportement). Par exemple, la couverture de valeurs aléatoire selon une distribution nécessite que les données d'entrée d'une suite de test suive une certaine distribution statistique (Gaussienne, de Poisson, uniforme, etc...).

- **Critères basés sur les fautes :** le test basé sur les fautes consiste à déterminer des données de test pour démontrer l'absence de fautes précises. Il peut s'appliquer à presque tous les modèles de SUT, puisque l'objectif principal du test de logiciel est de détecter des éventuelles fautes dans le SUT. Une technique de test répandue pour couvrir ce critère est le test à partir de mutations [34]. Appliquée aux modèles [26, 70], la couverture de mutation consiste à faire muter le modèle à l'aide d'opérateurs qui injectent chacun un certain type de faute. Le but est ensuite de générer une suite de test qui puisse différencier chaque mutant du modèle initial. Le pourcentage de mutants tués représente alors le pouvoir de détection d'erreurs de la suite de test.

Dans le cadre de ce mémoire de projet, nous nous intéressons plus particulièrement à la couverture structurelle des modèles et à la couverture des données.

3ème Etape - Génération Automatique des tests

A partir d'un modèle de test et de spécifications de cas de test, les cas de test peuvent être générer manuellement. Cependant, l'un des avantages du MBT est sa capacité d'automatisation du processus de génération de test. Une suite de test peut être générée automatiquement au moyen d'algorithmes de génération basés sur différentes techniques :

- **Génération aléatoire :** La génération aléatoire consiste à créer un échantillon de l'espace de données. Chaque génération de test basée sur un même modèle peut avoir des caractéristiques différentes. De nombreuses techniques existent comme l'utilisation du partitionnement [69], la modélisation d'un profil d'usage par chaîne de Markov [88], etc...

- **Algorithmes de recherche :** Les algorithmes de recherche pour la génération de test à partir de modèles font l'objet d'un intérêt croissant. Ils incluent les algorithmes issus de la théorie des graphes, comme l'algorithme du postier chinois [59], qui permettent de satisfaire les critères de couverture tous-les-nœuds et tous-les-arcs. Sont aussi inclus les algorithmes de recherche méta-heuristique, les algorithmes évolutionnaires comme les algorithmes génétiques, etc...

- **Model-Checking :** Le principe est de vérifier ou falsifier les propriétés d'un système. D'abord, il s'agit de formuler des spécifications de cas de test comme des propriétés d'atteignabilité (exemple : à un certain moment, on atteint tel état, ou une certaine transition est exécutée), puis l'objectif réside dans la recherche de contre-exemples en vérifiant la négation de la propriété.

- **Exécution symbolique :** cette technique consiste à exécuter le modèle avec des données d'entrée contraintes. Le système est représenté par un ensemble de contraintes sur ses variables d'état. Des solveurs de contraintes permettent de générer des traces symbolique : une trace symbolique représente de nombreuses traces instanciées. Pour cela, une trace symbolique doit être valuée pour obtenir un cas de test valide.

- **Preuve de problèmes déductifs :** Assez similaire au model-checking, mais avec un prouveur de théorèmes à la place du model-checker. Cependant ce procédé est surtout utilisé pour vérifier la satisfaction de *gardes* de transitions dans le cas d'un modèle à état.

- **Résolution de contraintes :** Cette technique est souvent utilisée en conjonction d'une autre technologie (model-checking, exécution symbolique, preuve de théorème, etc...), lorsque d'étroites relations entre les variables sont définies dans les gardes ou les conditions comme des contraintes. Ces contraintes peuvent être résolues par des solveurs de contraintes dédiés. Technique très utile aussi pour l'extraction de données dans un domaine complexe, par exemple pour du test N-wise combinatoire.

Parmi les solutions de génération existantes, nous nous concentrerons sur les algorithmes de recherche et leurs différentes applications dans le domaine MBT.

4ème Etape - Exécution des tests

Lorsque les tests ont été générés, l'ultime étape consiste à les exécuter. Pour cela, deux possibilités : l'exécution des tests est *Offline*, les tests sont générés complètement avant d'être exécutés, ou elle est *Online*, et dans ce cas les algorithmes de génération de tests peuvent adapter leurs résultats selon les données de sortie du *SUT*. L'exécution Online est parfois nécessaire lorsque le SUT est non-déterministe, cependant l'exécution Offline offre plus de liberté. Il est en effet possible de générer des tests et de les exécuter à plusieurs reprises en utilisant des outils de gestion des tests. Il est aussi possible de diviser les suites des tests pour les exécuter en parallèle sur plusieurs SUT, ou de générer les tests sur une machine et les exécuter sur une autre, avec éventuellement des environnements différents, etc...

L'exécution des tests se fait manuellement (avec un opérateur), et/ou automatiquement. Lorsque manuellement, l'opérateur intéragit avec le SUT en suivant des instructions de test, alors qu'automatiquement, les tests générés sont directement exécutables sous forme de script.

Pour que les tests soient exécutables sur le SUT, il est nécessaire de créer un pont (adapter) pour convertir (concrétiser) les données d'entrées [75].

Dans ce document, nous nous focaliserons sur l'exécution automatique des tests de manière Offline.

1.3 Problématique et contribution

Ce document se place dans le contexte de la génération automatique de tests à partir de modèles UML (Unified modeling langage) / OCL (Object Constraint Language). Il s'inscrit dans les travaux menés par l'équipe VESONTIO du DISC basés sur la génération de tests à partir d'UML, et plus particulièrement les travaux de thèse de Christophe Grandpierre [43], basés justement sur la génération automatique de tests à partir de modèles UML/OCL.

Le langage UML est utilisé depuis quelques années comme notation de modélisation dans le cadre du MBT. Jusqu'à aujourd'hui, UML ne proposait pas de manière standard pour définir le comportement des opérations de classes. Le comportement est pourtant nécessaire à de nombreuses stratégies de générations de tests. Les techniques existantes utilisent des contraintes écrites en OCL, afin de spécifier les pré/postconditions des opérations, pour calculer des tests de type boîte noire.

En 2010, l'OMG (Object Management Group) a émis la spécification d'un langage d'action destiné à UML, nommé Alf (Action Language for Foundational UML), dont la syntaxe est très proche de celle de Java. Cette récente extension au langage UML offre de

nouvelles opportunités pour le test à partir de modèles. Alf permet de définir facilement le comportement des opérations de classe. Sa syntaxe étant proche des langages de programmation objet traditionnels, il devient possible de décliner les tests structurels appliqués à ces langages au langage Alf pour permettre l'analyse et la couverture du comportement des opérations. Notre contribution consiste à intégrer le langage Alf dans la notation UML/OCL existante, et de développer une chaine outillée permettant l'animation de modèles et la génération de test selon cette nouvelle notation.

Le premier objectif de ce document est de faire un état de l'art des différents critères de test (critères de couverture de code, critères de couverture de modèles, critères de couvertures liés à UML, ...) et des principales stratégies de génération automatique de test, dans le but de situer notre choix de modélisation parmi ces approches.

Le second objectif est d'outiller ce choix de modélisation tant que faire se peut, et de développer les composants manquants nécessaires à une chaîne outillée permettant la modélisation et l'animation de modèles UML/OCL/Alf.

Le troisième objectif consiste à valider la chaîne outillée obtenue en implémentant une stratégie de génération automatique de tests.

1.4 Structure du document

Ce mémoire est composé de cinq chapitres.

Le **chapitre 2** présente la notation choisie pour ce mémoire, composée de trois langages : UML, OCL, et Alf.

Le **chapitre 3** introduit les travaux scientifiques à propos des critères de couverture de code source et de modèles, et offre une vue générale sur les algorithmes de recherches et leurs différentes applications dans un contexte de test.

Le **chapitre 4** expose les technologies utilisées pour l'établissement d'une chaîne outillée, ainsi que le développement d'un composant d'animation de modèles dédié à notre notation.

Le **chapitre 5** présente les stratégies de test mises en place, et les illustrent avec une étude de cas.

Le **chapitre 6** conclut les travaux présentés dans ce document, et propose différentes pistes pour la poursuite de ces travaux.

Chapitre 2

Présentation de la notation choisie

Contents

Avant d'aborder en détail les travaux existants liés au concept MBT, il convient dans un premier temps de présenter les langages et notations qui seront utilisés dans cette approche pour modéliser, contraindre et animer un modèle.

D'abord, pour représenter l'architecture d'un système, le choix s'est porté sur Unified Modeling Language (UML) [55]. C'est un langage spécifié par l'OMG dans sa version 2 depuis 2005, qui propose de représenter un système de manière semi-formelle.

Ensuite, un langage issu de la spécification UML permet d'apporter des contraintes à un modèle, Object Constraint Language (OCL) [54].

Enfin, l'OMG a soumis l'année dernière la spécification d'un langage permettant de décrire textuellement le comportement de certains éléments UML, Action Language for Fundational UML (Alf) [52].

La section 2.1 présente la sous-partie UML intégrée à notre notation, ainsi que langage OCL. La section 2.2 introduit le langage Alf et le rôle qu'il occupe dans un contexte MBT. Les technologies existantes permettant de manipuler ces langages sont présentées en section 2.3 Pour illustrer cette combinaison de langage, une étude de cas est réalisée en section 2.4.

2.1 Unified Modeling Language

Unified Modeling Language (UML) [55] est un langage de modélisation graphique daté de 1995 utilisé pour visualiser, spécifier, construire, documenter, et animer les artefacts de systèmes logiciels orientés objet. Il réunit différentes techniques, comme la modélisation de données, la modélisation objet, l'automatisation des processus, et la modélisation de composants. En fait, il est issu des notations Booch, OMT, et OOSE, l'objectif étant de standardiser les méthodes de modélisation. La notation fut adoptée par l'OMG en 1998 qui se charge depuis lors de sa spécification. La version actuelle est 2.4.1, date d'août 2011.

Le fait qu'UML accumule de nombreux avantages justifie son utilisation :

- La notation est extrêmement répandue. Elle est aujourd'hui considérée comme l'une des plus utilisées.
- Son utilisation est graphique, ce qui en fait un langage simple, facile à comprendre.
- UML possède un très grand pouvoir d'expression. Avec quatorze types de diagrammes, il est possible de représenter un système sous de nombreux aspects (structurel, comportemental, interactionnel, etc...).
- Un modèle élaboré à partir d'UML est potentiellement traduisible en langage orienté objet (Java, C++, etc...). De plus, des technologies existent pour automatiser la transformation (Acceleo [1], Xpand [21], JET [10], etc...).
- De nombreux outils supportent la notation UML (voir section 2.3).

UML possède un très grand pouvoir d'expression (voir figure 2.1 ci-dessus). De plus, c'est une notation semi-formelle, qui connaît quelques ambiguïtés. Aussi, il est nécessaire de resteindre le langage et lui donner une sémantique spécifique, de manière à ce qu'il soit juste suffisant dans son niveau d'expressivité et de formalisme pour proposer une modélisation destinée à la génération de tests. Dans les parties suivantes sont présentés les différents diagrammes utilisés, ainsi que leur rôle au sein d'un modèle orienté "Test".

2.1.1 Diagramme de classes

Le diagramme de classes permet de décrire la structure statique d'un système en le déclinant en classes, chaque classe comprenant des attributs, des opérations, et des relations avec les autres classes. En effet, ce diagramme est dit "statique" puisqu'il s'abstrait des aspects temporels et dynamiques.

Typiquement, ce type de diagramme convient parfaitement dans un contexte de test pour déterminer la structure du SUT : ses principales fonctions, ses éventuels attributs, et les différents types d'éléments avec lesquels il est susceptible d'intéragir.

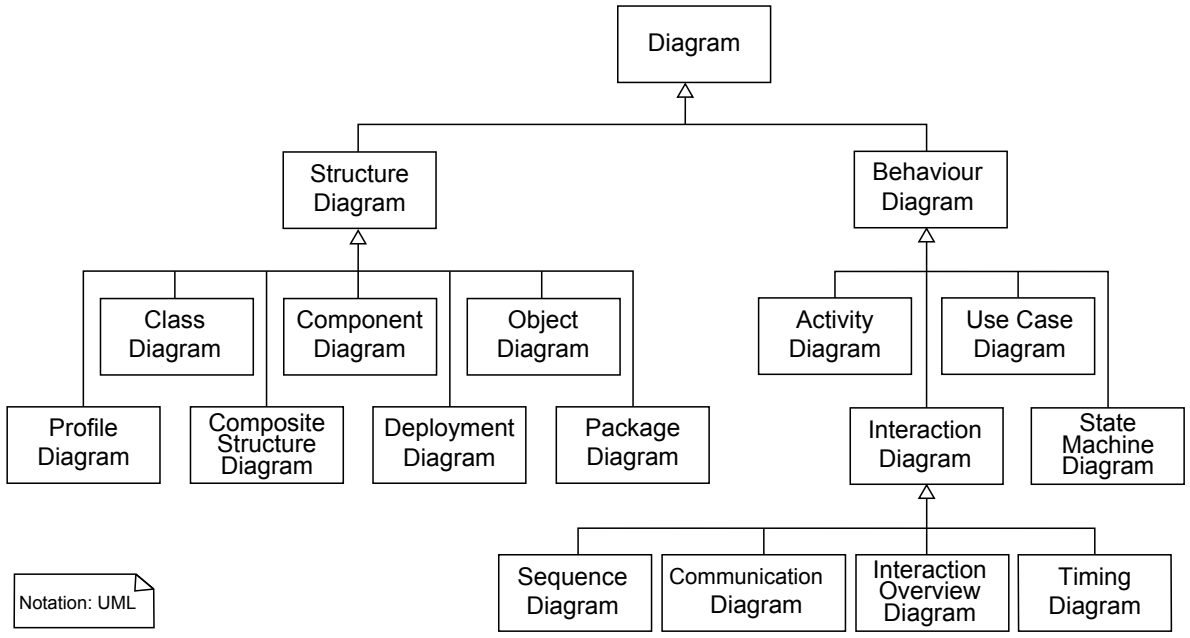

FIGURE 2.1 – Diagrammes proposés par la notation UML

Artefacts principaux

L'élément principal est la classe (Class dans UML). Comme dans tout langage orienté objet, une classe décrit les responsabilités, le comportement et le type d'un ensemble d'objets (qu'on appelle instance de cette classe) au moyen de fonctions (méthodes) et de données. En UML, l'élément représentant une méthode s'appelle Operation, il est lié à une classe et peut contenir des paramètres, de type entrée, sortie, entrée-sortie, ou retour. Les données sont exprimées grâce à l'élément Property, qui lorsque contenu par une classe désigne un Attribut de type primitif ou Enumeration. Un Attribut dont le type est une classe sera contenu par l'élément Association, représenté graphiquement par un "lien" entre les deux classes concernées. Une Enumération correspond aux enum Java, elle permet de définir un type qui n'accepte qu'un ensemble fini d'éléments (EnumerationLiteral).

L'exemple en figure 2.2 est une modélisation sommaire d'une voiture. Une voiture est caractérisée par son état, qui peut prendre quatre valeurs : ETEINT, ALLUME, ARRET, AVANCER, RECULER. La voiture peut démarrer, s'éteindre, s'arrêter, et rouler. Elle peut posséder un volant (caractérisé par son angle d'inclinaison), et jusqu'à 4 roues (qui peuvent être gonflées ou non).

2.1.2 Diagramme d'objets

Un diagramme de classes est une représentation structurelle graphique d'un programme orienté objet, et ce dernier a la faculté d'être instanciable. Pour modéliser le phénomène d'instanciation, UML propose le diagramme d'objets. Il est ainsi possible de définir des éléments qui seront liés à des éléments du diagramme de classes ainsi que la sémantique de leur liaison (appartenance). Ce type de diagramme permet de définir l'état initial d'un système en fournissant un ensemble d'instances de classe à un environnement de test, lui permettant de se focaliser sur la faisabilité des actions en s'abstrayant du problème

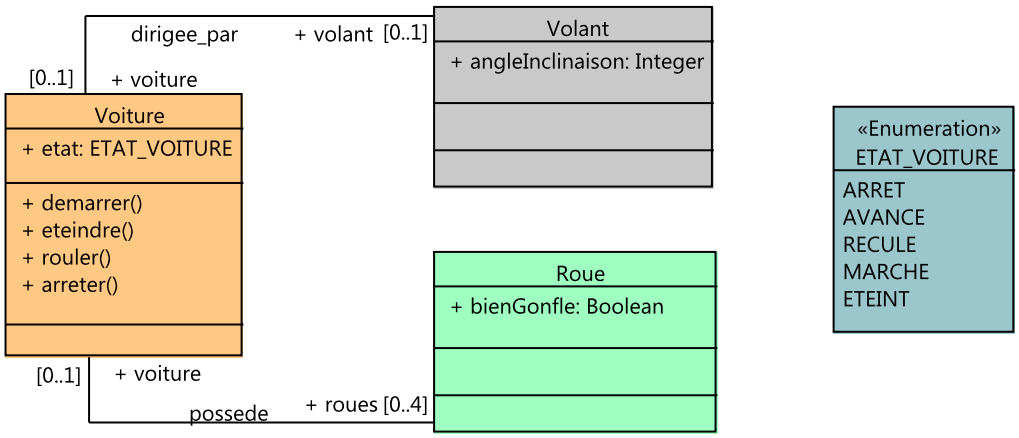

FIGURE 2.2 – Exemple de diagramme de classes

d'instanciabilité des éléments.

Artefacts principaux

Deux éléments sont à retenir. Il y a d'une part InstanceSpecification, possédant un référent (Classifier : un objet pouvant être instancié) et qui d'après la spécification UML peut être l'instance d'une classe, ou d'une association :
"Une InstanceSpecification dont le Classifier est une classe décrit un objet de cette classe, tandis qu'une InstanceSpecification dont le Classifier est une Association décrit un lien de cette Association.".

D'autre part, une InstanceSpecification peut posséder des éléments Slot. Ils permettent de définir une valeur à un Attribut de Classe. S'il s'agit d'un attribut au type primitif, on pourra lui affecter une valeur (booléen, entier) par le biai d'éléments de type Litteral (LitteralBoolean, LitteralInteger). S'il s'agit d'un attribut de type Enumeration, c'est un élément de type InstanceValue qui sera affecté au Slot. InstanceValue requiert un type (ici une énumeration), et une instance (une des valeurs proposées par l'énumeration).

L'exemple en figure 2.3 propose une instanciation du précédent diagramme de classe (Fig. 2.2). Une instance de voiture (306) possède une instance de volant (unVolant), et quatre instances de Roue (R1,R2,R3,R4).

2.1.3 Diagramme d'états-transitions

Le diagramme d'états-transitions, aussi appelé StateChart ou StateMachine, permet de donner une description abstraite du comportement d'une ou plusieurs entités UML, sous la forme d'un automate à états finis (le formalisme UML est une variante des statecharts d'Harel [46]). Typiquement, on l'utilise pour décrire le comportement de classes, mais il peut aussi décrire le comportement d'autres entités comme les cas d'utilisations, les opérations de classe, etc... De fait, il permet de compléter un diagramme de classes en précisant les comportements et les changements d'états du système.

Dans une optique de test, le diagramme d'états-transitions permet de définir le comportement du SUT. En effet, le SUT est représenté par une classe UML, et le diagramme

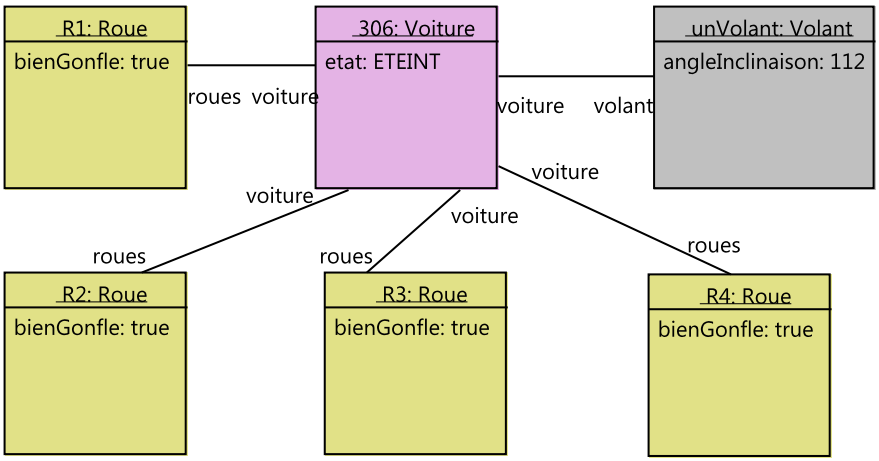

FIGURE 2.3 – Exemple de diagramme d'Objet

d'états-transitions est contenu par cette classe. Les cas de test sont directement liés aux traces d'exécution possibles du statechart.

Artefacts principaux

Un statechart est un type d'automate : il est composé de nœuds (State) reliés entre eux par des transitions. Un état représente le SUT à un moment donné, une transition exprime un changement d'état du SUT. Une transition est déclenchée par un évenement (Event) de différentes natures (appel de méthode, un signal, un timeout, etc...). Il existe aussi des points de choix, qui selon si la garde de la transition est vraie ou fausse, dirige vers des états différents.

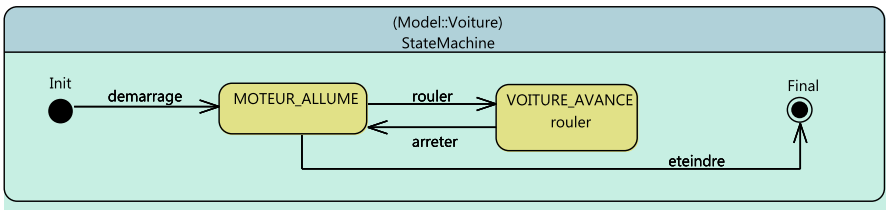

FIGURE 2.4 – Exemple de diagramme d'Etats-Transitions

L'exemple en figure 2.4 permet de définir le comportement du précédent diagramme de classe (Fig. 2.2) concernant une voiture. D'abord, elle ne peut que démarrer. Ensuite, elle peut soit s'éteindre immédiatement, soit rouler indéfiniment, mais devra être à l'arrêt pour pouvoir s'éteindre.

2.1.4 Object Constraint Language

Un diagramme UML à lui seul ne permet pas de modéliser tous les aspects importants d'un système. Le besoin par exemple de définir les contraintes associées aux objets d'un modèle a poussé la spécification UML à se doter d'un langage destiné à l'expression de conditions, Object Constraint Language (OCL) [54]. OCL est un langage formel qui permet

de définir des contraintes sur les modèles UML. Ces contraintes, sous forme d'expressions booléennes, spécifient par exemple des conditions invariantes qu'un modèle doit vérifier, ou des requêtes vers des objets définis dans un modèle. De ce fait, OCL apporte une précision sur la modélisation. La notation, qui se veut facile à lire et à écrire, entre langage naturel et langage mathématique, a été développé par la division Assurance d'IBM, et provient de la méthode Syntropy [31]. La version actuelle est 2.3.1 et date de janvier 2012.

Le langage sera majoritairement utilisé dans un contexte de test pour restreindre d'une part l'atteignabilité des états d'un statechart en précisant une garde, et d'autre part l'exécutabilité des méthodes du SUT par la définition de conditions pre/post.

Subtilités du langage

Primo, OCL est un langage de spécification pur : toute expression OCL est garantie sans effet de bord. Le seul résultat issu d'une expression est une valeur booléenne, aucune modification de l'état du système n'est possible. Secondo, OCL n'est pas un langage de programmation. Il n'est pas possible d'écrire un programme séquenciel, d'invoquer un processus, ou d'activer une opération non-pure via OCL. Tertio, OCL est un langage typé. Il est donc impossible de comparer, par exemple, l'égalité d'une chaîne de caractères (String) avec un entier (Integer). Chaque Classifier définit dans un modèle UML représente un type OCL. Quarto, OCL possède des opérateurs sur les collections d'objets. Il est possible par exemple de vérifier qu'une expression est vraie pour tous les éléments d'une collection avec l'opération forAll, de vérifier qu'une expression est vraie pour au moins un élément d'une collection avec l'opérateur exists, extraire un ou plusieurs éléments d'une collection selon un critère commun avec l'opérateur select, etc...

```
1 context Voiture :: demarrer()
pre : self.etat = ETAT_VOITURE::ARRET & self.roues->forAll(bienGlonfe =
    true)
3 post : self.etat = ETAT_VOITURE::MARCHE
```

L'exemple ci-dessus propose de contraindre la méthode demarrer() de la classe Voiture. Pour pouvoir démarrer, la voiture doit être éteinte (etat = ARRET) et toutes ses roues doivent être bien gonflées. Une fois la méthode demarrer() exécutée, le moteur doit être en marche (etat = MARCHE).

2.2 Action Language for Fundational UML

Action Language for Fundational UML (Alf) [52] est un langage d'action pour UML standardisé par l'OMG en Octobre 2010 qui permet une représentation textuelle d'éléments UML. Son but premier est de se comporter comme une notation de surface pour spécifier des comportements exécutables au sein d'un modèle UML graphique. Il devient possible de spécifier le comportement ou l'effet d'une transition. Ainsi, la spécification d'un système gagne encore en précision.

Auparavant, il n'existait pas de standard concernant la syntaxe concrète d'un langage d'action dédié à UML (comme l'est OCL pour les contraintes), la spécification ne proposant qu'une syntaxe abstraite pour les actions. Pourtant l'approche consistant à placer la modélisation au centre du développement logiciel n'est pas récente. La méthode d'analyse

de Shlaer et Mellor [80] propose de considérer un logiciel comme un ensemble de modèles étroitement reliés, chaque modèle ayant un rôle particulier (modèle d'information, modèle à état, ...). Le but étant de se baser sur cet ensemble de modèles pour générer les squelettes du programme dans un langage d'implémentation quelconque. Cette approche permet de spécifier un système *"au plus haut niveau d'abstraction, permettant ainsi de s'affranchir des décisions spécifiques à un langage de programmation quant à l'organisation d'un logiciel"*.

Ayant la volonté de développer ce concept, Mellor, Tockey, Arthaud et Leblanc souhaitent doter UML d'un langage d'action textuel en soumettant une proposition de sémantique [65]. Un langage d'action aurait la faculté d'affiner considérablement la spécification d'un système, mais permettrait aussi d'effectuer des vérifications (preuves, simulations) très tôt dans le cycle de développement du logiciel, et offrirait la possibilité d'exporter la modélisation vers n'importe quel langage d'implémentation (Java, C++,...) , annulant ainsi les coûts de conversions (avec un outil de génération de code).

Cependant la spécification d'un langage d'action n'est pas une priorité pour l'OMG. Ainsi, on trouve dans la littérature un large panel de propositions non-standardisées de syntaxes concrètes : Shlaer-Mellor Action Language (SMALL) [65], mais aussi Action Semantics Language (ASL) [77], Object Action Language (OAL) [44], Platform-independent Action Language (PAL) [81], Starr's Concise Relational Action Language (SCRALL) [62],That Action Language (TALL) [66], etc...

L'OMG n'émettra pas de RFP (Request For Proposal) concernant un langage d'Action destiné à UML avant 2008, date à laquelle est standardisé le sous-ensemble exécutable UML, baptisé fUML (Foundational Subset for Executable UML Models) [53].

2.2.1 Entre OCL et Java

Alf propose une syntaxe très proche de celle de Java, à ceci près que sa machine virtuelle serait UML, puisqu'il est sémantiquement fidèle au sous-ensemble UML exécutable, fUML. Alf peut ainsi naviguer d'un bout à l'autre des associations de classe d'un modèle. Il possède d'ailleurs toute l'expressivité d'OCL dans l'utilisation et la manipulation des séquences d'objets/valeurs : les opérateurs forAll, exists, select, etc... mais aussi les quotes pour les noms d'éléments comportant des espaces, et les ' : :' pour la navigation dans une Enumération par exemple, ou d'un package vers une classe. Cette expressivité permet entre autres d'éviter les éventuelles incohérences qu'engendrerait l'utilisation d'un langage de programmation, comme Java, en tant que langage d'action.

Supposons que dans un modèle contenant deux classes, Client et Compte, un compte ayant plusieurs soldes de type entier, on souhaite naviguer dans l'association depuis un client (myCustomer) vers son compte et additionner tous les soldes. Les concepts d'association et de navigation sont propres à UML et n'existent pas dans la plupart des langages de programmation. Une approche serait de considérer l'"association-end" Compte comme une collection Java :

```
Boolean flag = true;
for (Account account: myCustomer.accounts) {
  for (Integer balance: account.balance) {
    totalBalance += balance;
  }
}
```

Pourtant, sémantiquement, les association-ends ne sont pas réellement des collections d'objets dans UML, mais davantage des propriétés multivaluées. Le fait que myCustomer.accounts retourne un objet en Java est incohérent avec la sémantique UML. De plus, il y a une forte ambiguïté quant au choix de la collection Java à utiliser (Vector, ArrayList, HashSet,...). D'où l'intérêt d'utiliser un langage d'action. Avec Alf, il est possible de naviguer dans les associations et manipuler correctement les séquences de valeurs :

```
totalBalance = 0;
for (balance in myCustomer.accounts.balance) {
    totalBalance += balance;
}
```

2.2.2 Alf pour le test

Le langage d'action sera utilisé dans le corps des opérations de classe, pour exprimer les comportements de cette opération, et dans l'effet des transitions du statechart, pour spécifier les comportements relatifs au changement d'état dû à la transition. L'objectif est de pouvoir faire évoluer le système et simuler le comportement du SUT, pour générer un jeu de tests cohérent.

Alf apporte une contribution importante dans le domaine du test à partir de modèles UML : il permet de modifier l'état du modèle. La popularité de la modélisation UML exécutable est très récente, et avant Alf aucun standard n'était proposé par la spécification UML. Aucun langage d'action n'était considéré comme référence, et dans le domaine MBT, cela a donné lieu à des solutions très diverses. Par exemple, la solution Smartesting issue de travaux au lifc ([30] et [43]) propose d'utiliser OCL comme langage de contraintes et comme langage d'actions. C'est une double interprétation du langage.

D'autre part, ce langage possède une syntaxe très proche de la syntaxe Java. Ainsi, il devient possible de dériver des tests de type boîte blanche, et ainsi renforcer les tests systèmes généralement produits.

2.3 Technologies UML/OCL/Alf existantes

Ainsi, au vu des langages cités précédemment, il est possible de faire appel à de nombreux outils. Néanmoins, le langage Alf sera peu représenté en raison de sa très récente spécification. La section 2.3.1 propose une liste non exhaustive des technologies dédiées à la modélisation et à l'édition des langages, tandis que la section 2.3.2 concerne l'interprétation.

2.3.1 Modélisation / Edition

Unified Modeling Language UML est une notation qui date de 1997, et pendant ces quinze dernières années une multitude de logiciels et greffons ont été développés dans le but d'exploiter le langage de modélisation. Nous nous restreindrons ici qu'aux principaux outils.

On peut citer Papyrus UML [15] et Top-cased [19], deux outils de modélisation UML développés au sein de la plateforme Eclipse [7]. D'un côté Papyrus UML, développé par le CEA, est un outil open-source qui propose l'élaboration de modèles UML/SysML. C'est un éditeur de qualité mais peu/pas d'outils sont disponibles pour le traitement des modèles. De l'autre Top-Cased, développé par un consortium avec entre autre Airbus France, CNES, et Thalès, est un environnement de modélisation complet, qui possède un éditeur UML (GEF UML Editor), et de nombreux outils liés à l'ingénierie dirigée par les modèles (MDA) comme la simulation d'exécution, le model-checking, la traçabilité des exigences, etc... Cependant, ces deux outils autrefois concurrents ont récemment fusionné pour obtenir Papyrus MDT [13]. En réalité, TopCased qui utilisait auparavant GEF UML Editor intégrera désormais l'éditeur Papyrus MDT pour la création de modèles UML/SysML. Pour l'instant en cours de développement, Papyrus MDT est un logiciel open source très prometteur.

Un des outils les plus utilisés aujourd'hui est Rational Software Architech [16] (anciennement Rational Software Modeler), réalisé par Rational, une division d'IBM. RSA est optimisé pour le développement d'applications JEE, SOA et WebSphere, il permet d'autre part l'édition de modèles UML/SysML. Il est aussi basé sur Eclipse, c'est en revanche un logiciel commercial. Il n'est pas possible par exemple d'installer RSA comme un simple plugin sur une instance d'Eclipse existante : c'est une application de type "client riche" (RCP). Il utilise Eclipse comme noyau, interface graphique, et tire partie de son mécanisme de greffons. Des solutions MBT ont ainsi été réalisées en surcouche de RSA pour permettre l'édition, la vérification, la simulation et l'exportation de modèles UML/OCL. On peut notamment citer Smartesting qui a developpé des extensions pour différents modeleurs dont RSA, le but étant d'exporter un modèle UML/SysML créé via RSA vers un format que pourra traiter la solution de génération automatique de tests CertifyIt [5].

Il existe encore beaucoup d'autres modeleurs UML, comme ArgoUML [3], starUML [17], etc... Ces outils, bien qu'ils puissent être de très bonne qualité, sont généralement moins complets et moins aboutis que ceux cités ci-dessus, et de surcroit sont moins disposés à l'intégration dans une chaîne outillée, contrairement à un modeleur intégré à un IDE tel qu'Eclipse.

Object Constraint Language OCL est un langage textuel reconnu et tous les principaux éditeurs de programmation propose l'expression de contraintes OCL (emacs, Notepad++, Kate, gedit, ...). L'environnement Eclipse propose aussi un ensemble d'éditeurs OCL différenciables par leur niveau de conformité à la spécification du langage.

Action Language for Foundational UML Aujourd'hui, aucun produit finalisé n'existe permettant l'édition de code Alf, néanmoins deux éditeurs sont en cours de développement. D'une part ealf [61], un éditeur créé avec XText et intégré à l'environnement Eclipse, est développé par l'université de Babes-Bolyai, en Roumanie. A terme, une partie compilation doit être ajoutée à l'édition pour transformer du code Alf vers un diagramme d'activité, injecté dans le modèle courant. Ce projet est inactif depuis plusieurs mois. D'autre part, le CEA a soumis début 2011 une version beta d'un éditeur Alf [32], il permet l'ajout d'actions Alf dans le corps des opérations de classe et à terme dans les effets des transitions du diagramme d'états-transitions. Le projet n'est pas terminé et des bugs sont à déplorer, notamment l'impossibilité de résoudre des associations entre les classes, ou de faire appel aux valeurs d'une énumération.

Aucun outil à ce jour ne propose d'utiliser le langage Alf pour développer un modèle fUML complet.

2.3.2 Interprétation

Il y a peu d'outils concernant l'interprétation. OCL est sans-doute le mieux représenté : des fonctionnalités intégrées à l'environnement Eclipse permettent de parser et d'évaluer les contraintes et requêtes UML sur des modèles UML et Ecore. De la même façon, TopCased et RSA intègrent des outils pour l'évaluation de contraintes.

Un outil *stand-alone* nommé USE [20] permet de réaliser des diagrammes de classes et d'objets UML de manière textuelle. Lorsque le diagramme est créé et instancié, des règles invariantes en OCL peuvent être appliquées au modèle. Le logiciel cherche ensuite à vérifier que chaque contrainte est respectée. L'ingénieur validation est informé des contraintes non-satisfaites, et peut visualiser graphiquement les diagrammes de classes et d'objets.

Concernant UML, quelques solutions existent. RSA est doté d'un simulateur qui permet d'animer un statechart, un diagramme d'interaction, ou un diagramme d'activité grâce à une génération automatique du code java correspondant au modèle. Alf n'est pas encore implémenté, mais le projet est prévu. miUML [12] est un projet open source pour fUML. Il est composé de méta-modèles qui décrivent et contraignent fUML et d'APIs qui permettent de créer, éditer, et animer un modèle. Le projet n'est pas terminé, cependant l'implémentation des statesharts et d'un langage d'action sont prévus. Cameo Simulation Toolkit [4] est un simulateur basé sur fUML enrichi avec StateChart XML [18] (SCXML). Il propose l'animation de diagrammes d'activité et de statechart, et permet l'utilisation de différents langages comme langage l'action (Javascript, Ruby, etc...). Alf ne semble cependant pas être implémenté.

Actuellement, aucun outil ne permet d'interpréter le langage Alf. Un parser a été créé [6], mais aucune version finale n'a été livrée. Quelques travaux sur le langage d'action sont en cours [73], mais sont davantage orientés vers la modélisation exécutable.

2.4 Modélisation UML/OCL et Alf par l'exemple : représentation d'un cinéma

Nous nous proposons d'illustrer l'association des différentes notations vues précedemment par la revisite d'un exemple récurrent dans le domaine du test : tester une application web d'achat de billets de cinéma selon une approche MBT. Pour motiver nos choix en matière d'outillage, nous avons utilisé Papyrus MDT pour réaliser ce cas d'exemple. Le plugin Alf proposé par le CEA n'étant pas encore assez stable nous avons utilisé l'éditeur de langage naturel intégré à Papyrus MDT. Cette solution reste malgré tout fonctionnelle bien qu'extrêmement basique.

Les sections 2.4.1 et 2.4.2 offrent une vue générale de l'application et de ses fonctionnalités. La section 2.4.3 définit les exigences fonctionnelles que pourrait exiger un client. La section 2.4.4 propose une modélisation UML/OCL/Alf orientée test.

2.4.1 Contexte

« eCinema » est une application web qui permet à un client d'acheter des billets de cinéma en ligne sans avoir à se rendre physiquement à son cinéma préféré. Cette application est

constituée d'un écran principal qui :

- permet à tout utilisateur de s'identifier par un nom et un mot de passe,
- présente une liste de films et de séances disponibles,
- donne la possibilité de réserver des tickets pour ces séances.

2.4.2 Fonctionnement général

Pour utiliser l'application, un utilisateur devra s'être enregistré dans le système. Pour s'enregistrer, l'utilisateur remplit les champs « Username » et « Password » et clique sur le lien « register ». Un tel enregistrement est valable si l'utilisateur donne un nom inédit et un mot de passe valable (sans espace et d'une taille supérieure ou égale à 4 caractères). La saisie d'un nouvel enregistrement valable implique que l'utilisateur est automatiquement connecté.

Si l'utilisateur est déjà enregistré dans le système, il n'a alors qu'à s'identifier en remplissant les champs « Username » et « Password » puis en cliquant sur le bouton « login ». L'identification est effective si l'association nom/utilisateur est connue du système (i.e. si un enregistrement de cette association a déjà été validé).

Une fois connecté (le bouton « login » se transforme alors en bouton « logout »), l'utilisateur peut acheter des tickets en cliquant sur le lien « buy » situé au bout de la ligne correspondant au film et à la séance sélectionnés. Si des tickets sont encore disponibles, son panier sera crédité d'un ticket correspondant à sa sélection.

Finalement, à tout moment, l'utilisateur peut vérifier son panier en cliquant sur le lien « View bought tickets ». Une fenêtre pop-up s'affiche alors avec l'ensemble des séances et le nombre de tickets sélectionnés par l'utilisateur. Un lien « delete » en face de chaque séance lui permet d'annuler la réservation d'un ticket de la séance en question. Un bouton spécifique « delete all tickets » (situé en bas de la fenêtre pop-up) lui permet également d'annuler toutes ses réservations. Suite à toute annulation de réservation, le nombre de tickets réservés par l'utilisateur et le nombre de tickets disponibles sont alors mis à jour en conséquence.

2.4.3 Exigences Fonctionnelles

Lorsqu'un client commande le développement d'un logiciel, il fournit une liste d'exigences, qu'il souhaite voir apparaître lors de la livraison. Souvent, ces exigences sont divisibles en deux parties : d'une part les exigences générales (voir table 2.1), elle concernent par exemple une fonctionnalité de l'application (l'authentification, l'inscription, ...), et d'autre part des exigences fines (voir table 2.2), qui concernent un cas d'utilisation précis (inscription avec un login vide, etc...). L'objectif est de générer un cas de test pour chaque exigence fine.

D'abord, voici les exigences générales :

Requirement ID	Requirement name	Requirement description	Criticity
ACCOUNT-MNGT	Account-Management	Capacité du système à gérer plusieurs comptes	1
Suite à la page suivante			

Table 2.1 – continued from previous page			
Requirement ID	**Requirement name**	**Requirement description**	**Criticity**
ACCOUNT-MNGT / LOG	Log	Capacité du système à autoriser/interdire l'accès au compte utilisateur	1
ACCOUNT-MNGT / REGISTRATION	Registration	Capacité du système à gérer les comptes utilisateurs	2
BASKET-MNGT	Basket-Management	Capacité pour tout utilisateur authentifié de gérer son panier	2
BASKET-MNGT / BUY-TICKETS	Buy-Tickets	Capacité pour tout utilisateur authentifié d'acheter des tickets	2
BASKET-MNGT / DISPLAY-TICKETS	Display-Tickets	Capacité du système à afficher le panier d'un utilisateur authentifié	3
BASKET-MNGT / REMOVE-TICKETS	Remove-Tickets	Capacité pour tout utilisateur authentifié de supprimer un ou plusieurs de ses tickets	2
NAVIGATION	Navigation	Capacité du système à proposer une navigation par le biai d'une Interface Graphique	3

TABLE 2.1: Classification des exigences fonctionnelles générales

Ces exigences sont détaillées dans le tableau ci-dessous :

Aim Id	Aim name	Aim description	Criticity
Buy-Login-Mandatory	Buy-Login-Mandatory-to-buy	Pour selectionner un billet, un utilisateur doit s'authentifier en entrant un couple login/password valide	3
BUY-Sold-Out	Ticket-Sold-Out	Si tous les billets d'une session sont vendus, un message "sold out" devra être affiché pour le film, et le bouton d'achat devra être désactivé	2

Suite à la page suivante

Table 2.2 – Suite de la page précédente			
Aim Id	**Aim name**	**Aim description**	**Criticity**
BUY-Success	Buy	Quand un utilisateur choisi un billet pour une session le nombre de billets devra être incrémenté et le nouveau nombre de billets devra être mis à jour	2
DIS-Check-Basket	Check-Basket	Un utilisateur peut vérifier le contenu de son panier	2
DIS-Login-Mandatory	Login-Mandatory-to-show	Pour visualiser ses précédants achats, un utilisateur doit s'authentifier en offrant un couple login/password valide	3
LOG-Empty-User-Name	Login-Empty-User-Name	Un login vide est interdit	2
LOG-Invalid-Password	Log-Invalid-Password	Un login avec un password invalide est interdit	3
LOG-Logout	Logout	Un utilisateur peut se déconnecter. La page d'accueil sera alors affichée	1
LOG-Success	Login	Pour une inscription valide, un utilisateur doit fournir un login et un password afin de créer un compte	2
NAV-Go-To-Home	Go-To-Home	un utilisateur peut retourner à la page d'accueil	1
REG-Empty-User-Name	Empty-User-Name	Un login vide est interdit	2
REG-Empty-Password	Empty-Password	Un password vide est interdit	2
REG-Go-To-Register	Register	Un utilisateur doit pouvoir créer un compte	2
REG-Login-Already-Used	Login-Already-Used	Pendant l'inscription, si l'utilisateur fournit un login qui fait déjà l'objet d'une inscription, un message d'erreur doit être affiché	3
REG-Success	Register	Un utilisateur doit pouvoir créer un compte	3
REG-Unregister	Unregister	Un utilisateur doit pouvoir se désinscrire de l'application	2
Suite à la page suivante			

Table 2.2 – Suite de la page précédente			
Aim Id	**Aim name**	**Aim description**	**Criticity**
REM-Del-All-Tickets	Del-All-Tickets	Lorsqu'il vérifie le contenu de son panier, un utilisateur peut effacer tous ses billets. Le nombre de billets disponible pour le film en question devra être mis à jour en conséquence.	2
REM-Del-Ticket	Del-Ticket	Lorsqu'il vérifie le contenu de son panier, un utilisateur peut effacer un de ses billets. Le nombre de ses billets restants ainsi que le nombre de billets disponibles pour le film en question devront être mis à jour en conséquence	3

TABLE 2.2: Liste de l'ensemble des exigences fonctionnelles fines

2.4.4 Une représentation UML/OCL/Alf de ECinema

Dans cette partie nous proposons une modélisation de ECinema pour le test. Premièrement, la section 2.4.4 présente les trois diagrammes UML. Deuxièmement, la section 2.4.4 liste toutes les expressions OCL rattachées au gardes et les actions Alf rattachées aux effets des transitions.

Modélisation UML

La figure 2.5 montre le diagramme de classes associé au système ECinema. La classe ECinema est liée à la classe User de deux manières : l'association "knows" représente les utilisateurs inscrits sur l'application, l'association 'has' représente l'utilisateur connecté. Un seul utilisateur ne peut être connecté à la fois. ECinema est aussi liée à la classe Movie, elle peut posséder une infinité de films. La classe Ticket est liée à la fois à la classe Movie et à la classe User : un utilisateur peut posséder plusieurs tickets, et plusieurs tickets peuvent être associés au même film.

La classe principale est ECinema, elle sera considérée comme le SUT. C'est une particularité de la modélisation destinée au test, la classe représentant le SUT est l'élément central, et contient toutes les opérations et les évènements qui peuvent être observés sur le SUT. Les autres éléments du diagramme ne sont que des classes et des attributs nécessaires au test du SUT.

Le diagramme de classe montre aussi l'utilisation massive d'énumérations pour définir des ensembles restreints de valeurs. En général, une énumération possède un type de valeur pour chaque cas répertorié dans les exigences. Par exemple, un login peut être classé selon

3 catégories : inconnu de l'application (UNREGISTERED_USER), inscrit à l'application (REGISTERED_USER), ou vide (INVALID_USER).

La figure 2.6 montre le diagramme d'objets associé au système ECinema. Une instance du SUT est proposée, elle possèdera à l'état initial 2 films, et un utilisateur enregistré. Un utilisateur non-inscrit est instancié pour pouvoir tester le mécanisme d'inscription. De la même manière, 10 instances de tickets sont présentes pour pouvoir s'assurer que l'achat et la suppression de billets réagissent comme attendu. Le diagramme d'objet permet de s'affranchir du mécanisme d'instanciation en mettant à disposition un "parc" d'instances.

La figure 2.7 montre le diagramme d'états-transitions associé au système ECinema. De nombreuses séquences d'actions sont possibles. Une fois arrivé dans l'état welcome, deux possibilités : soit s'authentifier soit s'inscrire. Si on choisit de s'inscrire, plusieurs transitions gèrent les différents cas d'erreur et de bon fonctionnement. Si on choisi de se loguer, on a alors accès à l'achat de billets et à l'état displayTickets, qui permet de visualiser les tickets achetés, et d'éventuellement en supprimer. Il est alors possible de retourner à l'état welcome et se déloguer, ou se désinscrire.

Contraintes OCL / Actions Alf

Nous décrivons ici en détail les gardes et les effets de chaque transition. Toutes les actions ci-dessous correspondent à une transition du stateshart. La garde contient une expression OCL, et l'effet une action Alf.

```
action goToRegister()
guard: self.current_user.oclIsUndefined()
effect: this.message = 'MSG'::REGISTER;
```

```
action logout
guard: self.current_user.oclIsUndefined() = false
effect: this.message = 'MSG'::BYE;
this.current_user = null;
```

```
action login
guard: self.current_user.oclIsUndefined()
effect: if (in_userName == 'USER_NAMES'::INVALID_USER) {
  this.message= 'MSG'::EMPTY_USERNAME;
} else if (!all_registered_users->exists(name == in_userName)) {
  this.message= 'MSG'::UNKNOWN_USER_NAME_PASSWORD;
} else {
  let user_found:User = all_registered_users->select u (u.name ==
      in_userName)->first();
  if (user_found.password == in_userPassword) {
    this.current_user = user_found;
    this.message = 'MSG'::WELCOME;
  } else {
    this.message = 'MSG'::WRONG_PASSWORD;
  }
}
```

```
action unregister
guard: self.current_user.oclIsUndefined() = false
effect: for(film in this.all_listed_movies) {
  film.available_tickets += this.current_user.all_tickets_in_basket->size()
      ;
}
this.current_user.all_tickets_in_basket = null;
this.all_registered_users->remove(current_user);
this.current_user = null;
this.message = MSG::BYE;
```

```
action buyTicket
guard: self.all_listed_movies->any(t : Movie | t.title = in_title).
    available_tickets >=1
effect: if (this.current_user.isEmpty()) {
  this.message = 'MSG'::LOGIN_FIRST;
} else {
  let targeted_movie : Movie = this.all_listed_movies->select m (m.title ==
      in_title)->first();
  Ticket unallocated_ticket = Ticket.allInstances()->select t (t.
     owner_ticket == null)->first();
  this.current_user.all_tickets_in_basket->add(unallocated_ticket);
  this.targeted_movie.all_sold_tickets->add(unallocated_ticket);
  this.targeted_movie.available_tickets--;
  if (this.targeted_movie.available_tickets == 0) {
    this.message= MSG::NO_MORE_TICKET;
  } else {
    this.message= MSG::NONE;
  }
}
```

```
action registration(in_userName, in_userPassword) REG_Success
guard: in_userName > USER_NAMES::INVALID_USER and
not all_registered_users->exists(user | user.name=in_userName)
effect: let user_found : User = User.allInstances()->select u (u.name ==
    in_userName && u.password == in_userPassword)->first();
this.all_registered_users->add(user_found);
this.current_user = user_found;
```

```
action registration(in_userName,in_userPassword) - REG_Uncomplete
guard: (in_userName=USER_NAMES::INVALID_USER or  in_userPassword=PASSWORDS
    ::INVALID_PWD) = true
effect: if (in_userName == 'USER_NAMES'::INVALID_USER) {
  this.message= 'MSG'::EMPTY_USERNAME;
} else {
  this.message= 'MSG'::EMPTY_PASSWORD;
}
```

```
action registration(in_userName,in_userPassword) - REG_Existing_UserName
guard: self.all_registered_user->exists(user | user.nale = in_userName)
```

```
effect: this.message = 'MSG'::EXISTING_USER_NAME;
```

```
action showBoughtTickets() - Logged in
guard: self.current_user.oclIsUndefined() = false
effect: this.message= MSG::NONE;
```

```
action deleteAllTickets(in_title)
guard: self.current_user.all_tickets_in_basket->select(movie.title =
    in_title)->size() >= 1
effect: Collection<Ticket> all_targeted_tickets = this.current_user.
    all_tickets_in_basket->select t (t.movie.title == in_title);
for(tick in all_targeted_tickets) {
  this.deleteTicket(in_title);
}
```

```
action deleteOneTicket(in_title) - REM_Del_Ticket
guard: self.current_user.all_tickets_in_basket->select(movie.title =
    in_title)->size() >= 1
effect: Ticket targeted_ticket = this.current_user.all_tickets_in_basket->
    select t (t.movie.title == in_title);
this.deleteTicket(in_title);
```

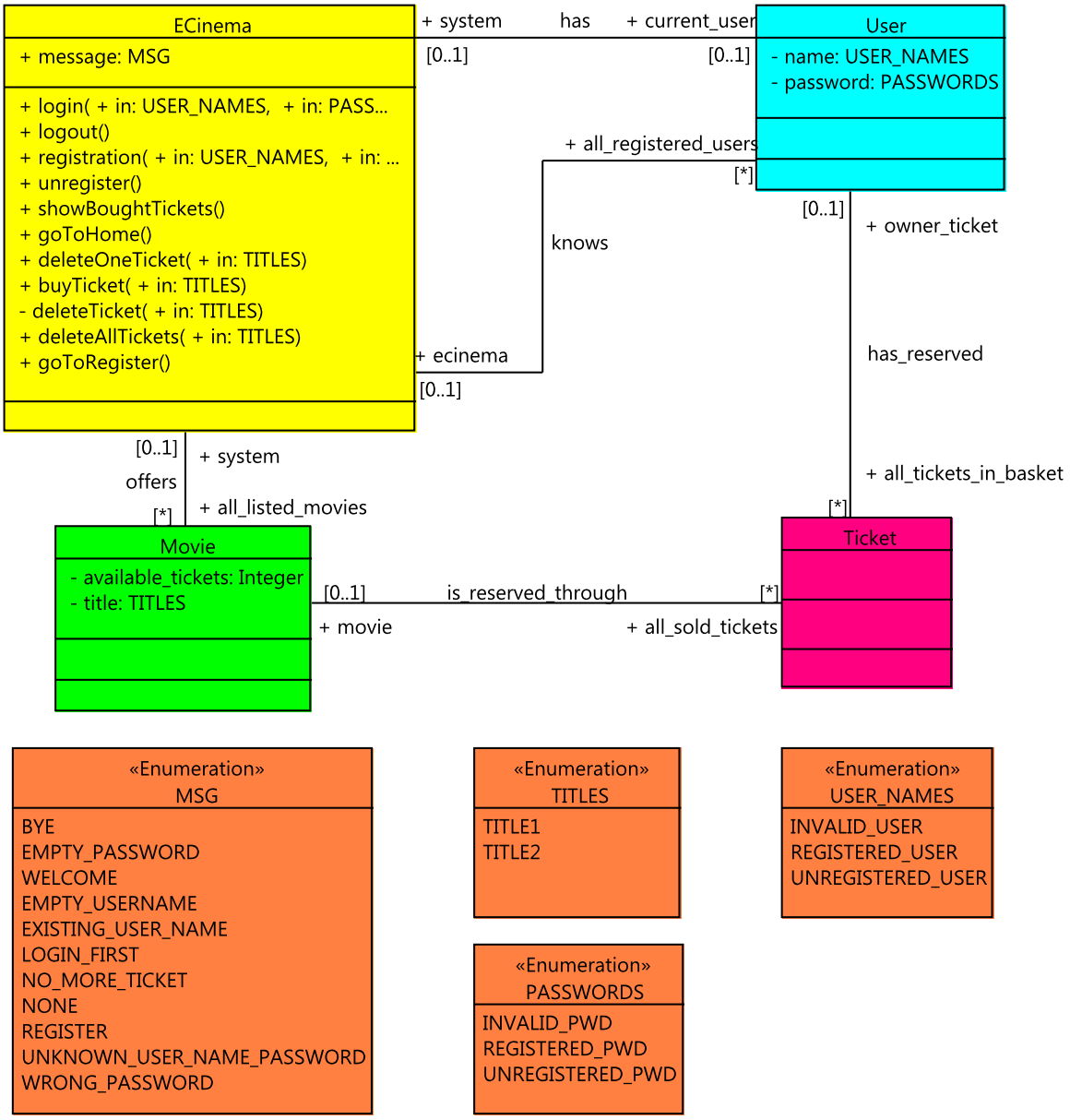

FIGURE 2.5 – Diagramme de classes de ECinema

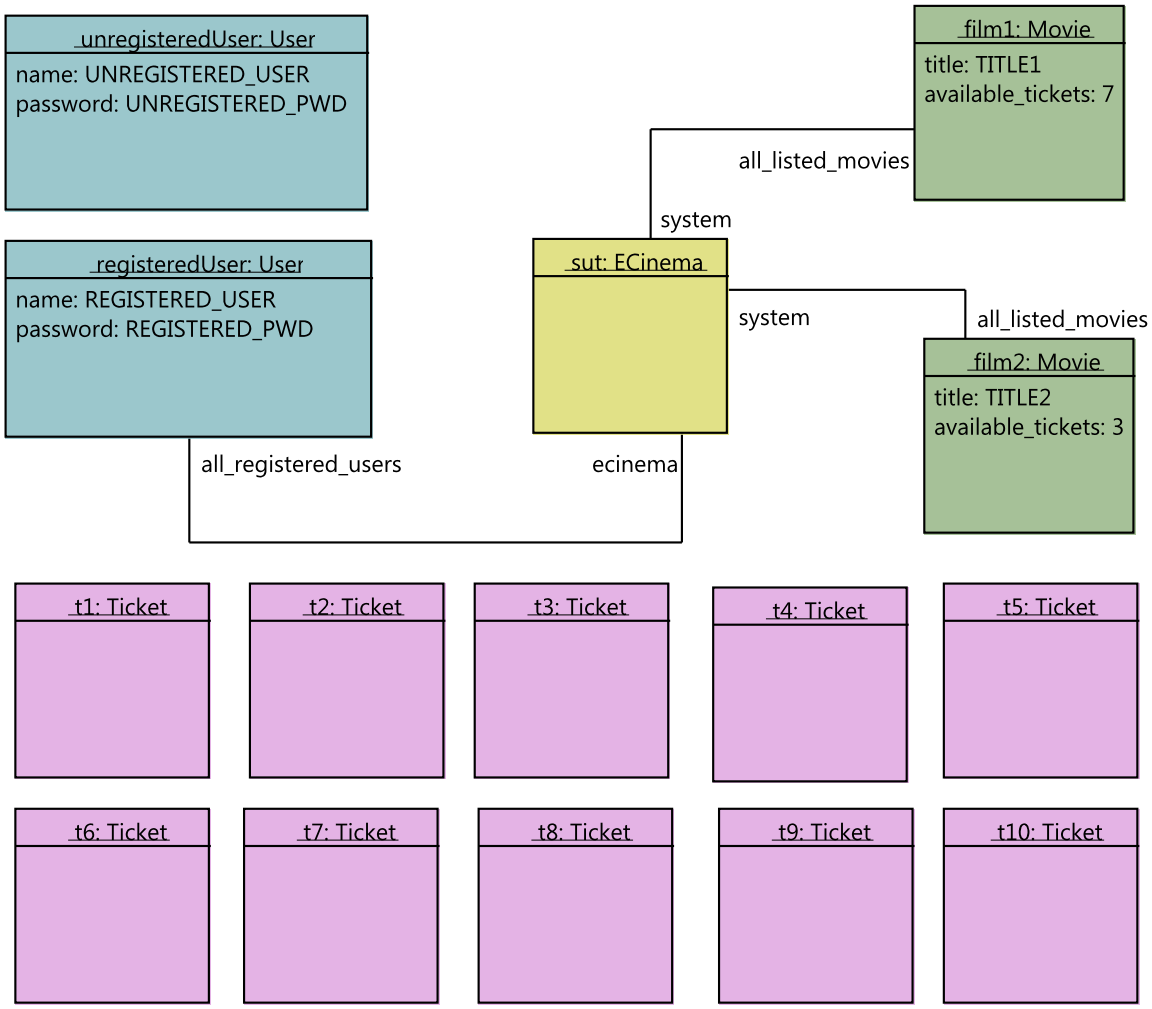

FIGURE 2.6 – Diagramme d'objets de ECinema

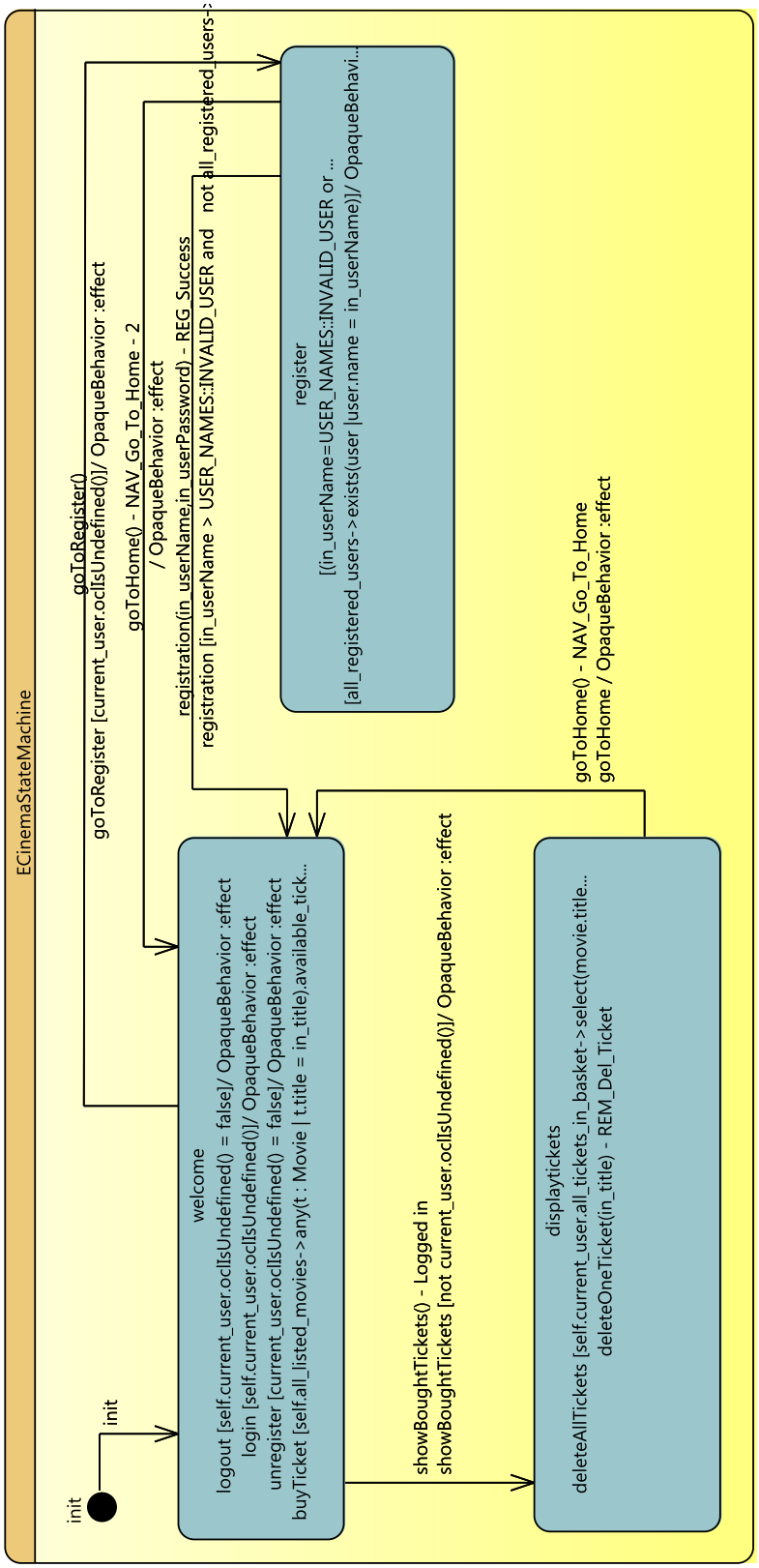

FIGURE 2.7 – Statechart de ECinema

2.5 Synthèse

Notre choix de notation pour la modélisation destinée à la génération automatique de tests est constituée de trois langages :

- UML permet de définir la structure du modèle (diagramme de classes), ainsi que son comportement général (statechart).
- OCL est utilisé pour apporter des précisions à la modélisation UML en contraignant le modèle.
- Alf a pour intérêt de permettre d'exprimer la dynamique du modèle.

Les langages UML et OCL sont très utilisés, et de nombreux outils existent permettant la création de modèles UML/OCL. Concernant l'interprétation, puisque le langage Alf est très récent, les solutions pour l'interprétation de modèles UML annoté avec Alf sont rares. Quelques outils existent, mais aucun n'est véritablement finalisé.

Pour illustrer la combinaison des trois langages, un exemple fil rouge a été réalisé : la modélisation d'une application web d'achat de billets de cinéma. Cet exemple a été entièrement réalisé avec Papyrus MDT. Toute la modélisation a été possible, des améliorations sur l'ergonomie du modeleur sont cependant nécessaires, notamment lors de l'ajout d'actions Alf.

Chapitre 3

Etat de l'art : approches applicables avec UML/OCL/Alf

Contents

Parmi les trois langages inclus dans la notation, UML et OCL ont déjà fait l'objet de nombreuses recherches, et différentes solutions MBT les exploitent. En revanche, l'ajout du langage Alf est une particularité de ce document. La syntaxe Alf étant très proche d'un langage de programmation classique tel que Java, l'intégration du langage d'action dans un contexte MBT rend possible la génération de tests de type boîte blanche, en plus des tests de type boîte noire traditionnels.

Ainsi, ce chapitre a pour objectif d'offrir une vue synthétique des principaux critères de couverture, aussi bien pour les tests de type boîte blanche que pour les tests de type boîte noire. Le second objectif est d'établir un état de l'art des algorithmes évolutionnaires, et plus particulièrement l'algorithme génétique, lorsque utilisés pour la génération de tests basées sur les critères mentionnés.

La section 3.1 traite des critères de couverture de type boîte blanche, tandis que la section 3.2 traite des critères de couverture de modèles et des approches exprimées dans la littérature pour satisfaire ces critères. Dans la section 3.3, nous présentons les algorithmes de recherche, et plus particulièrement l'algorithme génétique, ainsi que ses différentes utilisations pour la génération automatique de tests.

3.1 Critères pour la Couverture de code source

Lorsque le code source d'un programme est connu et que le souhait est de tester son comportement, une approche consiste à s'appuyer sur la structure même du code. Cette approche appelée test structurel (ou test de type "boîte blanche") s'oppose au test fonctionnel (ou test de type "boîte noire"), qui n'a pas accès au code source et ne peut que comparer les données de sorties avec les données d'entrées soumises. Les premiers travaux concernant le test structurel date des années 70 ([51] et [39]).

Le test structurel peut se décomposer en deux étapes. La première consiste à repérer les blocs d'instructions dont l'exécution est nécessaire à la satisfaction de critères de couverture (voir section 3.1.2). La seconde concerne la génération de données de test garantissant l'exécution des blocs d'instructions.

Dans la littérature, de nombreux travaux portent sur la génération de données de test. Gotlieb, Botella, et Rueher ([40],[41] et [42]) proposent d'utiliser des techniques de résolution de contraintes pour la génération de données de tests. Le but est de produire un cas de test pour un point choisi dans le programme, c'est-à-dire pour une instruction donnée. Une autre approche est avancée par Godefroid, Klarlund et Sen [36] grâce à l'outil DART, qui repose sur sur la combinaison d'une génération automatique aléatoire de cas de test, et d'une analyse dynamique à-la-volée du programme face à un cas de test, pour trouver un comportement alternatif (si le cas de test ne passe pas dans le then d'un if, l'analyse dynamique permet de trouver une solution qui satisfait la condition du if). Néanmoins, nous n'aborderons pas plus en détail les techniques de génération liées au test de code source.

La première étape du test structurel s'applique comme suit. Par l'analyse du code source il est possible d'établir des tests. Pour pouvoir guider la génération de tests, il est possible de leur fixer un objectif à atteindre, soumettre à des critères de couvertures basés sur différents aspects d'un programme.

Il existe trois types de critères de couverture :

- basé sur **le graphe de contrôle** : le code source est représenté par un graphe de contrôle, et les critères associés nécessitent de parcourir le graphe.

- basé sur **le flot des données** : il s'agit de se concentrer sur la déclaration et l'utilisation des variables. La représentation sous forme de graphe de flot de contrôle est aussi utilisée en intégrant les informations nécessaires liées aux variables.

- basé sur **les fautes** : le but est d'orienter le choix des données tests pour repérer les fautes connues (dans la programmation de composants électronique par exemple). Un exemple de critère est la mutation, qui consiste à créer un modèle de fautes composé de versions mutantes du code (par injection de fautes syntaxiques, etc...).

Le but étant de produire des Données de Test (DT) qui exécuteront un ensemble de comportements du programme satisfaisant l'un ou les critères choisis.

Nous nous intéresserons plus particulièrement aux critères de couvertures basés sur le graphe de contrôle et sur les données.

La section 3.1.1 explique plus en détails comment représenter un bloc d'instructions par un graphe de contrôle, la section 3.1.2 est consacrée à la présentation des différents critères de couverture.

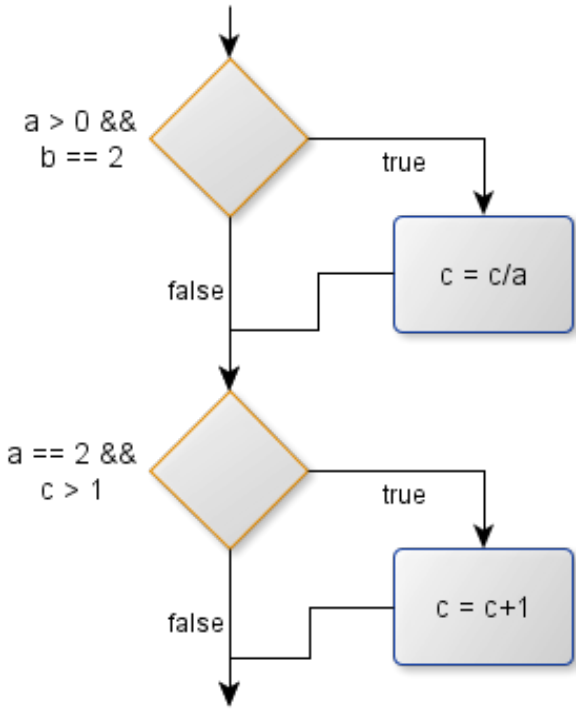

FIGURE 3.1 – Graphe de contrôle de la méthode foo

3.1.1 Représentation d'un algorithme par un graphe de contrôle

L'objectif est de s'assurer de la bonne structure du code. Un fragment de code source n'étant qu'une suite d'instructions dictées par d'éventuelles conditions/itérations, une solution consiste à convertir ce fragment vers une structure connue : un graphe orienté connexe. Dans ce graphe, chaque nœud représente un bloc d'instructions, et chaque arc représente la possibilité de transfert de l'exécution d'un nœud à un autre.

Pour illustrer ce procédé, observons la méthode foo ci-dessous :

```
public void foo(int a, int b, int c) {
  if (a>0 && b==2) {
    c=c/a;
  }
  if (a==2 || c>1) {
    c=c+1;
  }
}
```

La méthode foo possède deux blocs d'instructions (composés chacun d'une instruction), et deux branchements conditionnels. Le graphe équivalent à la méthode foo est visible en figure 3.1.

3.1.2 Critères de couverture

De nombreux documents traitent des critères de couverture ([67], [90], et [60]). Nous nous appuierons sur ces derniers pour présenter les différents critères.

Bien que les critères de couvertures soit décrits séparément, il n'y a pas de "meilleur critère" applicable à tous les programmes. Généralement la satisfaction d'un critère peut engendrer la satisfaction d'autres critères, mais il existe quelques exceptions qui ne vérifient pas cette hiérarchie. Ainsi, chaque critère possède ses avantages et ses inconvénients et une couverture rigoureuse du code source nécessite, dans la mesure du possible, de satisfaire une combinaison de critères la plus complète possible.

Un critère intuitif : tous-les-nœuds

Le critère de couverture tous-les-nœuds propose de parcourir chaque bloc d'instructions au moins une fois. Ce type de couverture est relativement intuitif et paraît cohérent, parce qu'il permet de s'assurer que tous les blocs d'instructions ont été exécutés. Cependant, ce critère est considéré comme peu fiable de part son faible taux de détection d'erreurs.

Lorsque des données de test permettent de couvrir tous les nœuds du graphe, on dit qu'elles satisfont TER1=1 ou TER1 (Test Effectiveness Ratio 1) [89]. Lorsque TER1 = 1, alors tous les nœuds du graphe ont été parcourus au moins une fois, et le critère tous-les-nœuds est satisfait. En effet, le taux de couverture est calculé selon la règle suivante : $TER1$ = nb de nœuds couverts / nb de nœuds total.

Dans l'exemple en section 3.1.1, il est possible de satisfaire le critère tous-les-nœuds avec les données de test (a=2 ; b=2 ; c=4). Si la première décision avait été définie comme la disjonction de deux conditions plutôt que comme leur conjonction, le critère tous-les-nœuds serait toujours satisfait, et la division potentielle de c par 0 ne serait pas détectée.

Critères basés sur les décisions

Cette catégorie de critères se focalise sur la couverture des décisions et des conditions. La hiérarchie des critères basés sur les décisions est observable en figure

Couverture tous-les-arcs Pour satisfaire le critère de couverture tous-les-arcs (ou toutes-les-décisions), il faut déterminer des données de test qui permettent de parcourir chaque arc du graphe au moins une fois, ce qui signifie que chaque décision a été évaluée au moins une fois à vrai et au moins une fois à faux. La plupart du temps le critère tous-les-arcs inclut le critère tous-les-nœuds, à quelques exceptions près, par exemple lorsqu'il n'y a aucune condition. Dans la pratique, le critère tous-les-arcs inclut le critère tous-les-nœuds pour remédier à ces exceptions.

Lorsque des données de test permettent de couvrir tous les arcs du graphe, on dit qu'elles satisfont TER2=1 ou TER2 [89]. Lorsque TER2 = 1, alors tous les arcs du graphe ont été parcourus au moins une fois, et le critère tous-les-arcs est satisfait. En effet, le taux de couverture est calculé selon la règle suivante : $TER2$ = nombre d'arcs Couverts / nombre d'arcs Total.

Même si ce critère est plus fort que le critère tous-les-nœuds, il ne détecte pas certaines erreurs. Dans l'exemple en section 3.1.1, il est possible de satisfaire le critère tous-les-arcs avec les données de test (a=2 ; b=2 ; c=4) et (a=0 ; b=2 ; c=0) . Si dans la seconde décision, la condition $c > 1$ avait été victime d'une erreur telle qu'elle soit définie comme $c < 1$, le critère serait tout de même satisfait, et l'erreur resterait non détectée puisque

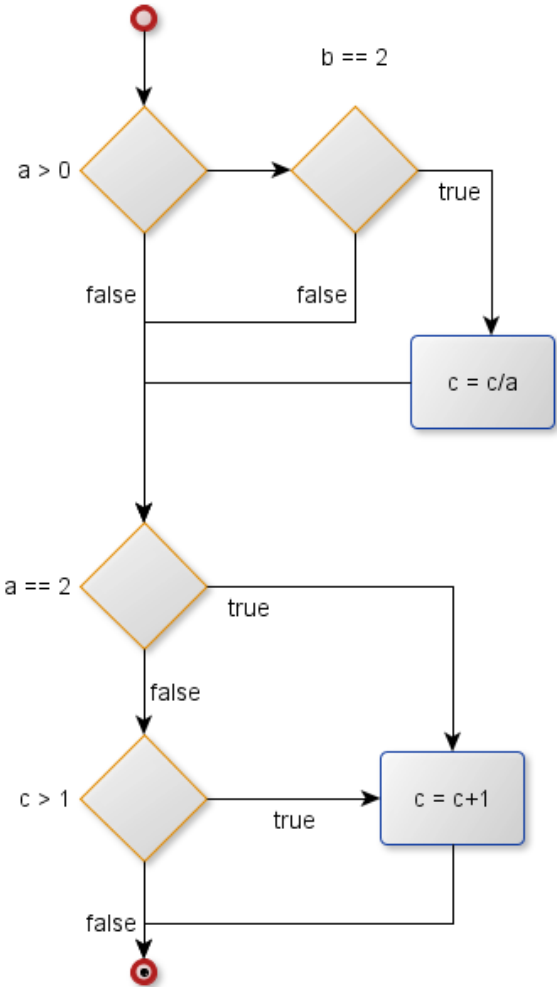

FIGURE 3.2 – Graphe de contrôle de la méthode foo vue comme un processeur

qu'avec les données de test choisies, c'est uniquement la condition $a == 2$ qui détermine la valeur de la décision.

Couverture toutes-les-conditions Le critère de couverture toutes-les-conditions est un peu plus fort que le critère tous-les-arcs : il est satisfait lorsque chaque condition d'une décision a été évaluée au moins une fois à vrai et au moins une fois à faux. Ce critère est proche du critère tous-les-arcs, il permet cependant de détecter certaines erreurs que ne détecte pas le critère tous-les arcs. Dans l'exemple en section 3.1.1, il est possible de satisfaire le critère toutes-les-conditions avec les données de test (a=1 ; b=2 ; c=4), et (a=2 ; b=1 ; c=0).

Néanmoins, le critère toutes-les-conditions n'inclut pas le critère tous-les-arcs. Typiquement, pour la décision `a==2 || c>1`, et avec le jeu de tests (a=2 ; b=0 ; c=1) et (a=1 ; b=0 ; c=2), toutes les conditions ont été une fois vraies une fois fausses, pourtant la décision a toujours été évaluée à vrai, et le critère tous-les-arcs n'est pas satisfait.

Couverture toutes-les-conditions/décisions Le critère toutes-les-conditions/décisions est la combinaison des critères tous-les-arcs et toutes-les-conditions. Pour être satisfait, chaque décision doit avoir été évaluée au moins une fois à vrai et

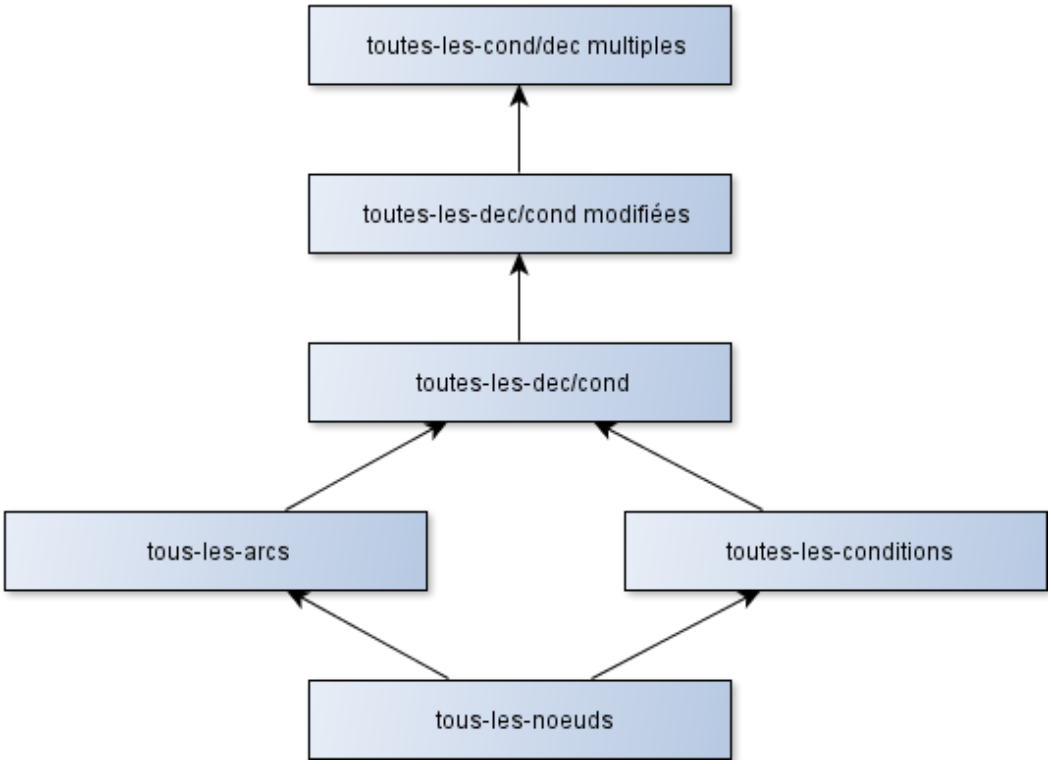

FIGURE 3.3 – Hiérarchie des critères basés sur les décisions

au moins une fois à faux, et chaque condition d'une décision doit avoir été évaluée au moins une fois à vrai et au moins une fois à faux. Typiquement, ce critère permet de faire l'union entre les erreurs détectées par le critère tous-les-ars et les erreurs détectées par le critère toutes-les-conditions.

Dans l'exemple en section 3.1.1, il est possible de satisfaire le critère toutes les conditions/décisions avec les données de test (a=0 ; b=1 ; c=1) et (a=2 ; b=2 ; c=4) . Toutes les décisions et toutes les conditions sont évaluées à vrai et à faux.

Ce critère a cependant une faiblesse : il ne détecte pas réellement toutes les conditions, car lorsque le programme est exécuté, des conditions sont fréquemment masquées par d'autres. Ce phénomène est visible sur la figure 3.2, représentant le graphe de contrôle de la méthode foo. Les conditions multiples ont été divisées en conditions simples, de la même manière qu'un compilateur générerait le code machine équivalent à un programme ([25]). Ainsi, une conjonction de deux conditions sera évaluée à faux si sa première condition est fausse, et une disjonction de deux conditions sera évaluée à vrai si sa première condition est vraie. On ne peut donc pas s'assurer que les deuxièmes conditions ont été évaluées au moins une fois à vrai et au moins une fois à faux avec ce critère.

Couverture toutes-les-conditions/décisions modifiées Le critère toutes les conditions/décisions modifiées est basé sur la norme D0-178B [23]. Il véhicule la philosophie suivante : pour tester un système, un ingénieur doit fournir des données de test pour chaque clause de chaque prédicat. Ainsi, pour satisfaire ce critère, chaque condition doit avoir été évaluée une fois à vrai et une fois à faux, et la valeur de la décision doit être directement corrélée à la valeur de la condition.

Dans l'exemple en section 3.1.1, il est possible de satisfaire le critère full predicate

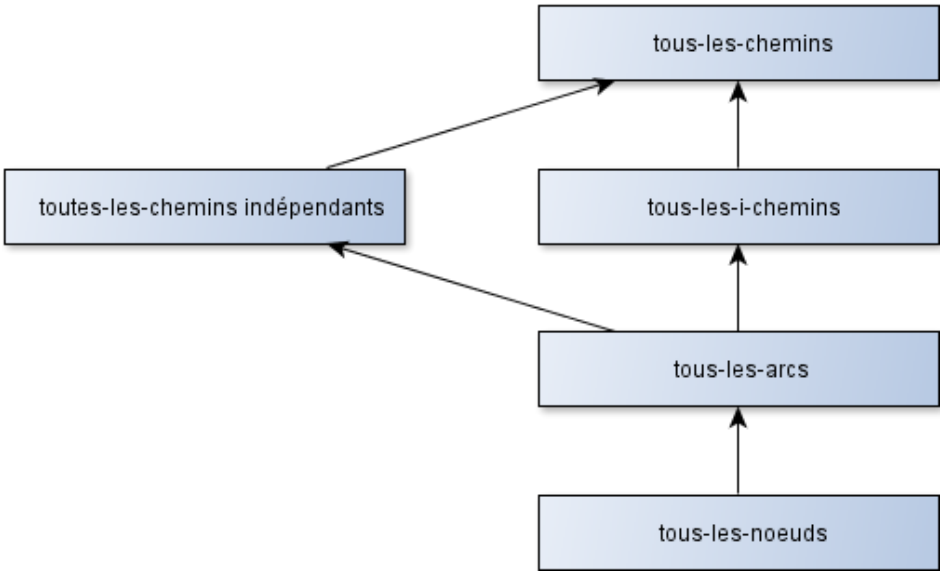

FIGURE 3.4 – Hiérarchie des critères basés sur les chemins

avec les données de test (a=0 ; b=1 ; c=1) et (a=2 ; b=2 ; c=4) . Toutes les décisions et toutes les conditions sont évaluées à vrai et à faux, et chaque résultat de décision est en corrélation avec chaque résultat d'une condition.

Couverture toutes-les-conditions multiples Le critère toutes-les-conditions multiples apporte une solution au problème rencontré avec le critère toutes les conditions/décisions. Pour satisfaire ce critère, il faut supporter toutes les combinaisons de conditions pour chaque décision. Cela signifie que pour n conditions, 2^n données de test sont requises pour la satisfaction de ce critère.

Dans l'exemple en section 3.1.1, il est possible de satisfaire le critère toutes-les-conditions multiples avec les données de test (a=2 ; b=2 ; c=4), (a=2 ; b=1 ; c=2), (a=0 ; b=2 ; c=4), DT_{10}= (a=0 ; b=1 ; c=1) . Toutes les combinaisons de conditions sont couvertes et tous les arcs ont été traversés.

Critères basés sur les chemins

Ce type de critère requière la couverture de certains types de chemins d'exécution. La hiérarchie des critères basés sur les chemins est visible en figure 3.4.

Critère tous-les-chemins indépendants Le critère tous-les-chemins indépendants peut être perçu comme le renforcement du critère tous-les-arcs. Ce critère est satisfait lorsque tous les arcs ont été parcourus, et ce dans chaque configuration possible. La tâche est compliquée puisque pour chaque arc, il faut prendre en compte les arcs parcourus avant et les arcs parcourus après.

Pour évaluer le nombre de chemins indépendants dans un programme [64], on peut calculer ce qu'on appelle le *Nombre de McCabe*, noté $V(G)$, où G est un graphe. Il est obtenu de la manière suivante :

$V(G)$ = Nombre d'arcs - Nombre de nœuds + 2

Ainsi, la procédure de calcul des chemins indépendants peut être effectuée comme suit :

1. Calcul de $V(G)$.
2. Production au hasard d'une donnée de test couvrant le maximum de nœuds de décisions du graphe
3. Production d'une donnée de test qui modifie le chemin obtenu avec la première donnée de test
4. Vérification de l'indépendance du chemin par rapport aux précédents chemins obtenus
5. Recommencer les étapes 3 et 4 jusqu'à la couverture de tous les arcs
6. S'il n'y a plus de décisions à étudier sur le chemin initial et que le nombre de chemins trouvés est inférieur à $V(G)$, il s'agit alors de prendre en compte les chemins secondaires déjà trouvés

Critère tous-les-chemins Le critère tous-les-chemins est le plus fort de tous les critères. Sa satisfaction requière une couverture exhaustive de tous les chemins du graphe de contrôle. Par exemple, pour couvrir le graphe de contrôle en figure 3.2, douze chemins sont possibles, il est donc nécessaire d'avoir douze données de test.

Bien que le critère tous-les-chemins ne garantisse toujours pas l'exactitude du fragment de code testé, ce critère est rarement envisageable du fait de l'explosion combinatoire du nombre de chemins liée à la complexité du code et à la présence de boucles.

Le critère tous-les-i-chemins est une alternative. Dans ce cas, le critère est satisfait lorsque tous les chemins sans cycle possibles sont couverts et, si le fragment de code comporte des boucles, les chemins supplémentaires à couvrir sont ceux qui traversent la boucle entre 0 et i fois. Cette alternative permet de contenir l'explosion combinatoire issue du critère tous-les-chemins.

Critères basés sur les données

Les critères basés sur le flot de données s'attardent davantage à couvrir certaines propriétés liées aux données, comme leur définition ou leur utilisation. On peut représenter le flot de données en ajoutant certaines informations supplémentaires au graphe de contrôle, concernant notamment la manipulation des variables. On dit qu'une variable est définie lorsque sa valeur est modifiée et qu'elle est utilisée lorsqu'une instruction requière l'accès à sa valeur.

La hiérarchie des critères basés sur les données est visible en figure 3.5.

Pour illustrer ces différents critères, nous utiliserons cet exemple, dont le graphe de contrôle annoté peut être observé en figure 3.6 :

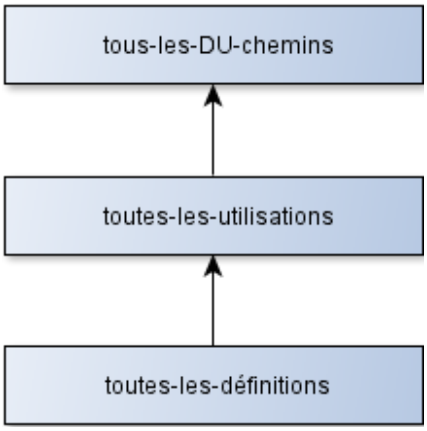

FIGURE 3.5 – Hiérarchie des critères basés sur les données

```
public int foo() {
  int x = random();
  int y = random();
  int z = 0;
  while (x >= y) {
    x = x - y;
    z++;
  }
  return z;
}
```

Critère toutes-les-définitions Le critère le plus faible est toutes-les-définitions. Il est satisfait lorsque les données de test couvrent au moins un chemin faisable entre la définition d'une variable et son utilisation.

Dans l'exemple en figure 3.6, pour la définition de x en A1, le chemin (A1,A2) permet de satisfaire le critère. Pour la définition de z en A1, le chemin (A1,A2,A4) permet de satisfaire le critère.

Critère toutes-les-utilisations Le critère toutes-les-utilisations est plus fort : pour être satisfait, il faut vérifier que toutes les paires définitions/utilisations sont faisables, c'est à dire que les données de tests doivent couvrir tous les chemins faisables depuis une définition vers chacune de ses définitions.

Dans l'exemple en figure 3.6, pour la définition de x en A1, les chemins (A1,A2) et (A1,A2,A3,A1,A2,A4) permettent de satisfaire le critère. Pour la définition de z en A1, les chemins (A1,A2,A4) et (A1,A2,A3,A1,A2,A4) permettent de satisfaire le critère.

A noter que la différenciation entre les utilisations destinées à un calcul et les utilisations destinées à l'établissement d'un prédicat a donné lieu à d'autres critères [78].

Critères tous-les-DU-chemins Le critère le plus fort pour la couverture du flot de données est le critère tous-les-DU-chemins. Pour satisfaire ce critère, il faut d'une part satisfaire le critère toutes-les-utilisations, et d'autre part couvrir tous les chemins sans cycle entre une définition et chacune de ses utilisations.

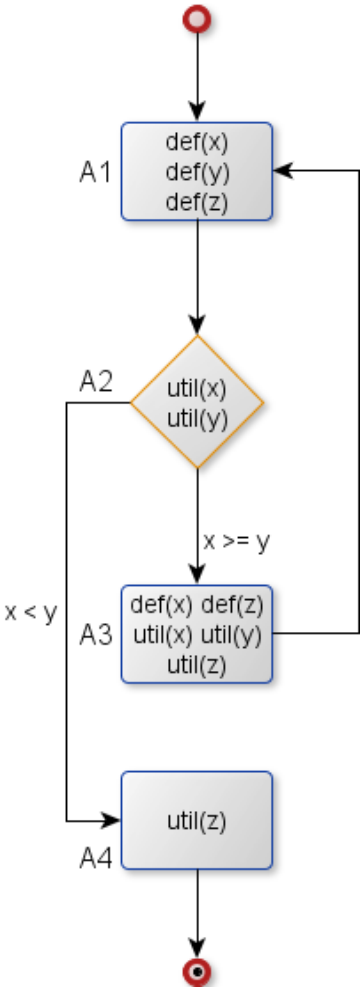

FIGURE 3.6 – Graphe de contrôle annoté destiné à la couverture des données

Ce critère est difficilement applicable parce qu'il requiert un nombre conséquent de données de test pour être satisfait.

3.2 Modèles Pre/Post et Transition-based

Nous avons établi en section 1.2 les différents types de notations permettant la modélisation de systèmes destinée au test. Nous nous intéresserons ici à deux types en particulier : les notations de type pre/post et les notations basées sur les transitions (transition-based).

Dans la section 3.2.1, nous approfondirons ces deux types de notations pour nous permettre de positionner notre approche. Ensuite, dans la section 3.2.2 nous présenterons les critères dédiés à la couverture de modèles.

3.2.1 Notations de modélisation

Différentes notations sont utilisées dans une optique MBT [84], nous nous intéresserons ici à deux d'entre elles : les notations de type pré/post et les notations à base de transitions.

Un modèle de type pré/post représente un système comme une collection de variables, dont le rôle est de représenter l'état du système. Des opérations sont alors définies pour

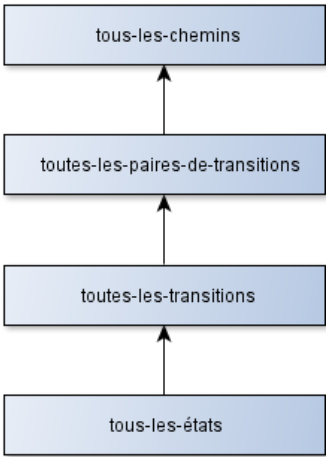

FIGURE 3.7 – Hiérarchie des critères basés sur les modèles états/transitions

permettre la modification de ces variables et ainsi faire évoluer l'état du système. Il est possible d'appliquer des contraintes à ces opérations sous la forme de pré et postconditions, d'une part pour empêcher le modèle d'atteindre un état illégal (dans le cas d'une application, empêcher un utilisateur lambda d'accéder au panneau de configuration), et d'autre part pour s'assurer que les opérations ont correctement réalisé la tâche qui leur incombe. Les notations B [24], Z [82], VDM [74], UML/OCL et d'autres permettent de réaliser des modèles de type pré/post.

Les notations à base de transitions sont davantage focalisées sur les transitions entre les états, par exemple les machines à états finis, les statecharts de Harel, les statecharts UML, les FSM, les EFSM, etc... Ces modèles sont généralement représentés graphiquement : chaque nœud correspond à un état du système, et chaque arc est une transition. Une transition correspond à un événement survenant sur le système comme l'appel d'une fonctionnalité venant de l'extérieur ou un déclenchement régulier dans le temps. Dans ce type de notation, il est possible de contraindre le déclenchement des transitions par l'ajout d'une garde, similaire à la postcondition pour une opération. L'ajout d'effet permet de renseigner le changement qu'apporte le déclenchement de la transition au système.

Le choix des notations dans ce document (UML, OCL, Alf) permet de faire la combinaison de ces deux représentations. En effet, on peut comparer la classe UML caractérisant le SUT à une machine B, puisque toutes deux sont composées de variables, et d'opérations pour modifier les variables. Cependant, UML dispose aussi d'un diagramme d'états-transitions, il peut donc être aussi catégorisé selon les notations transition-based.

3.2.2 Critères de couverture de modèles

Les critères de couverture de code basés sur les décisions sont directement applicables aux modélisations de type pré/post. Au lieu de tester les décisions d'une instruction if par exemple, il s'agit ici de mesurer les préconditions et postconditions appliquées aux opérations du modèle. En revanche, la singularité des systèmes à base de transitions a donné lieu à de nouveaux critères ([68] et [84]), permettant d'évaluer la couverture des transitions du système. Cette section propose de présenter ces nouveaux critères. Leur hiérarchie est observable en figure 3.7.

Critère tous-les-états A l'instar du critère tous-les-nœuds pour le code source (voir section 3.1.2), le critère tous-les-états est intuitif puisque sa non-satisfaction permet de détecter des états inateignables. En effet, pour satisfaire ce critère, chaque état du système doit avoir été traversé au moins une fois par un des cas de test.

Dans l'exemple en figure 3.8, tous les états sont couverts avec un seul cas de test : le chemin (A,E) par exemple.

Il arrive que dans certains cas une partie des états s'exécute parallèlement plutôt que séquenciellement. Une variante du critère tous-les-états existe alors : le critère toutes-les-configurations. Il impose de couvrir toutes les configurations d'états possibles.

Le critère tous-les-états et sa variante toutes-les-configurations restent faibles, parce qu'ils ne sont pas centrés sur les transitions. Dans l'optique MBT, ils sont même obsolètes puisque ils ne contribuent pas à l'objectif de test : tester les fonctionnalités d'un système (qui sont caractérisées par les transitions).

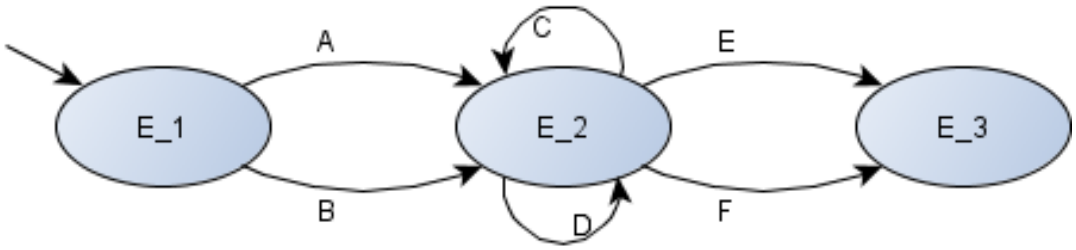

FIGURE 3.8 – Exemple d'un système de transitions

Critère toutes-les-transitions Le critère toutes-les-transitions peut être apparenté au critère tous-les-arcs (voir section 3.1.2) puisque graphe de contrôle et système de transitions ont une construction comparable. Ce critère est satisfait si chaque transition d'un modèle est traversée au moins une fois par l'un des cas de test produits. Ce critère permet de satisfaire le critère tous-les-états par effet de bord.

Dans l'exemple en figure 3.8,toutes les transitions sont couvertes avec les cas de test suivant : les chemins (A,D,E) et(B,C,F).

Le critère toutes-les-transitions est considéré comme le critère minimum qu'un ensemble de cas de test doit satisfaire. Il permet de solliciter chaque précondition au moins une fois.

Critère toutes-les-paires-de-transitions Le critère de couverture toutes-les-transitions teste chaque transition indépendamment, mais ne teste pas les séquences de transitions. Pourtant, le système peut ne pas fonctionner comme entendu parce qu'une certaine séquence de transitions est permise ou parce qu'une séquence valide n'est pas réalisable. Le critère toutes les paires de transitions permet de vérifier ce genre de fautes : toutes les paires de transitions adjacentes doivent avoir été traversées au moins une fois. En d'autres termes, dans un état donné e, chaque transition entrante de e doit être poursuivie au moins une fois par chaque transition sortante de e.

Dans l'exemple en figure 3.8, l'état E_2 possède 2 transitions entrantes, et 4 transitions sortantes. Cela signifie que 8 chemins sont nécessaires pour que cet état seulement satisfasse le critère toutes les paires de transitions.

Critères basés sur les chemins Un système de transitions étant similaire à un graphe de contrôle concernant la représentation du flux, les critères basés sur les chemins vus en section 3.1.2 peuvent être adaptés. Les critères tous-les-chemins indépendants, tous-les-i-chemins, et tous-les-chemins sont ainsi applicables à un statechart. Bien entendu, les problèmes liés à l'explosion combinatoire pour le critère tous-les-chemins sont valables aussi ici.

Critères propres à UML Parce que la notation UML est utilisée depuis quelques années pour la modélisation de la structure et du comportement d'un SUT, de nouveaux critères ont été établis [63] pour exploiter la sémantique propre à ce langage. Comme nous n'utilisons dans ce document qu'un ensemble restreint d'UML, seuls deux critères nécessitent d'être évoqués.

D'une part, un critère a été établi pour mesurer la couverture de la multiplicité des associations de classe, nommé critère AEM (Assocation-End Multiplicity coverage). Pour que ce critère soit complètement satisfait, tous les types de multiplicités possibles entre deux doivent avoir été créés au moins une fois. Typiquement, si l'on reprend l'exemple en figure 2.2 à la section 2.1.1, l'association *possede* n'est satisfaite que si au moins une instance de l'ensemble des instances de la classe voiture (il n'y aura qu'une seule instance dans une modélisation orientée test si la classe Voiture représente le SUT) n'a été associée à aucune instance de Roue, qu'au moins une instance a été associée avec (1..3) instance(s) de Roue, et qu'au moins une instance a été associée avec quatre instances de Roue.

D'autre part, la couverture des attributs de classe peut être mesurée par le critère CA (Class Attribute coverage). Ce critère est satisfait lorsque l'ensemble prédéfini de combinaisons de valeurs des attributs a été couvert, et ce pour chaque classe du diagramme. Ce critère est en réalité un méta-critère, parce que l'ensemble des combinaisons de valeurs est défini par un autre critère de couverture.

Par exemple, si la classe A possède deux attributs e et f qui, selon l'ensemble prédéfini, peuvent respectivement prendre deux et trois valeurs différentes , il faudra 6 configurations différentes pour atteindre toutes les combinaisons de valeurs et satisfaire le critère.

Ces critères sont particulièrement intéressant pour tester la structure d'une modélisation réalisée avec UML.

3.3 Algorithmes de recherche pour la génération de tests

Nous allons désormais présenter la stratégie que nous avons choisie pour notre solution concernant l'étape de génération de tests, à savoir utiliser les algorithmes de recherche. Nous commencerons par clarifier les notions de recherche aléatoire et d'optimisation dans la section 3.3.1. Nous présenterons ensuite un type d'algorithme de recherche, l'algorithme génétique en section 3.3.2. Les informations que nous délivrerons sont basées sur l'ouvrage *Practical Genetic Algorithms*, de Randy et Sue Ellen Haupt [49].

3.3.1 Recherche aléatoire et Optimisations

La solution la plus naïve pour effectuer une recherche dans un espace de donnée est d'effectuer le parcours de cet espace aléatoirement, et à chaque valeur ou combinaisons de valeurs obtenues, vérifier si celles-ci constituent une solution optimale face au problème

que l'on cherche à résoudre. L'avantage d'une telle technique est qu'elle est généralement peu complexe à mettre en place. Pourtant, l'absence totale de contrôle sur la méthode de recherche expose ce type d'algorithme à une durée d'exécution extrêmement longue. Pour accroître la convergence d'une recherche aléatoire, on utilise des mécanismes d'optimisation.

La notion d'optimisation, d'un manière générale, est le fait de rendre quelque chose plus efficace, meilleur. Optimiser signifie faire varier un concept initial, c'est-à-dire une solution initiale à un problème, et utiliser les informations obtenues pour améliorer celui-ci. Dans la vie quotidienne, nous sommes constamment confrontés à de nombreuses opportunités d'optimisation : quel itinéraire choisir pour arriver sur son lieu de travail au plus vite, sachant que la route principale est en travaux ? A quelle heure aller au supermarché afin d'éviter de faire la queue à la caisse ? Optimiser consiste donc à ajuster les entrées ou les caractéristiques d'un appareil, d'un processus mathématique ou d'une expérience pour obtenir le résultat souhaité. Les entrées ou caractéristiques sont des variables, et l'appareil, processus ou expérience est apparenté à une fonction de coût, ou fonction de fitness. Dans le cas d'une expérience, les variables sont des éléments physiques faisant partie de l'expérience.

Haupt *et al.* proposent de différencier les différentes optimisations selon six critères. A noter que ceux-ci ne sont pas nécessairement exclusifs.

Optimisations Essai-Erreur / Fonctionnelles La méthode essai-erreur consiste à ajuster les variables qui influencent le résultat, sans connaître le processus qui permet d'arriver à ce résultat. Typiquement, la méthode essai-erreur peut être utilisée pour optimiser un logiciel qualifié de "boite noire", i.e. dont le code source n'est pas connu. Au contraire, une fonction mathématique est décrite par une formule, donc connue du mathématicien, qui peut manipuler la fonction pour obtenir le résultat escompté.

Optimisations à variable unique / variables multiples S'il n'y a qu'une seule variable à optimiser, l'optimisation est dite unidimensionnelle. Au contraire, un problème composé d'un ensemble de variable nécessite une optimisation multidimensionnelle. Ajuster le variateur d'une lampe pour obtenir la luminosité souhaitée est une optimisation unidimensionnelle, alors que la configuration d'une table de mixage est une optimisation multidimensionnelle. A noter que la difficulté d'optimisation accroît avec le nombre de dimensions, aussi de nombreuses approches proposent, lorsque cela est possible, de généraliser une optimisation multidimensionnelle en un ensemble d'optimisations unidimensionnelles. Il est en effet plus abordable de résoudre un ensemble de sous-problèmes peu complexes, plutôt qu'un seul problème d'une grande complexité.

Optimisation Statique / Dynamique Une optimisation dynamique implique que le résultat soit fonction du temps, tandis qu'une optimisation statique implique que le résultat soit indépendant du temps. Par exemple, il existe de nombreuses routes pour effectuer le trajet Besançon-Belfort. Du point de vue de la distance, le problème est statique, et il suffit d'emprunter la route la plus courte. Cela dit, résoudre ce problème est bien plus complexe et de nombreux facteurs sont à prendre en compte : l'heure de la journée, la météo, etc... Intégrer la notion de temps accroit la difficulté d'optimiser ce genre de problème.

Optimisation à variable discrète / continue On distingue deux types d'optimisations selon si les variables sont discrètes ou continues, c'est-à-dire si leur espace de donnée est fini ou non. Trouver la valeur minimale d'une fonction mathématique nécessite une optimisation de variable continue. A l'inverse, déterminer dans quel ordre accomplir un ensemble de tâches nécessite une optimisation de variables discrètes, ou optimisation de combinaisons, puisque la solution optimale consiste en une certaine combinaison de valeurs issues de l'ensemble fini des combinaisons de valeurs possibles.

Optimisation à variable contrainte / non-contrainte Souvent, des contraintes sont attachées aux variables d'un problème. Dans ce cas, on utilise une optimisation contrainte, qui intègrent les tests d'égalités et d'inégalités de variables dans la fonction fitness, contrairement à une optimisation non-contrainte qui permet aux variables de prendre n'importe quelle valeur. La plupart des routines d'optimisations numériques ont plus de difficultés pour manipuler des variables contraintes, aussi ces variables sont généralement transformées en variables non-contraintes. Prenons l'exemple d'une fonction $f(x)$ que l'on souhaiterait minimiser sur l'intervalle $-1 \leq x \leq 1$. Plutôt que de faire varier $f(x)$, il est alors plus intéressant d'introduire la variable non-contrainte u définie par $x = sin(u)$, et de faire varier $f(sin(u))$ pour toute valeur de u. L'optimisation porte donc sur une variable non-contrainte, et est de fait moins complexe à réaliser.

Optimisations aléatoires / minimisation Certaines algorithmes tentent de déterminer le minimum/maximum d'une fonction de fitness à partir d'un ensemble initial de valeurs de variables. On distingue deux types d'algorithmes. Les algorithmes d'optimisations dits traditionnels sont basés sur des méthodes de calculs. Ce genre d'algorithme est potentiellement rapide mais tend à rester coincé dans un minimum local. En revanche, les méthodes aléatoires déterminent un ensemble de variables au moyen de calculs probabilistiques. Ils sont généralement plus lents, mais plus efficaces pour trouver le minimum global.

Il existe de nombreux algorithmes de minimisation. Le fonctionnement général reste identique : parcourir l'espace de données pour trouver la solution ayant meilleur fitness, c'est-a-dire la solution la plus proche de la solution idéale, selon la fonction de fitness. Dans ce document, nous nous intéressons à un algorithme qui tend à réduire la probabilité de converger vers un minimum local, l'algorithme génétique.

3.3.2 L'algorithme génétique

L'algorithme génétique est une technique de recherche et une optimisation basé sur les principes de la génétique et de la sélection naturelle. Développé par JH Holland [50] et DE Goldberg [37], le principe de l'algorithme génétique est comme suit : l'état initial est composé d'un ensemble d'individus, une population, que l'on fait évoluer au moyen de différentes règles de sélection et d'évolution, de manière à ce que la valeur de chaque individu maximise ou minimise la fonction de fitness (selon le type de problème à résoudre). Ainsi, pour un problème donné, en considérant les combinaisons de valeurs d'entrée possible comme une population et le résultat souhaité comme fitness à atteindre, il est possible de faire converger les individus vers cette valeur de fitness. Des problèmes très complexes ont pu être résolus grâce à ce type d'algorithme, comme l'optimisation du

contrôle de transmission de gaz naturel dans un réseau de pipelines [38].

L'algorithme génétique possède certains avantages :

- il peut gérer un vaste espace de donnée
- il retourne non pas une solution unique mais un ensemble de solutions
- il fonctionne avec des variables discrètes ou continues
- il est très adapté au parallélisme

L'algorithme génétique ne constitue pas une solution adaptée à tous les types de problèmes. Certains problèmes incluant peu de variables sont généralement résolus rapidement avec des méthodes traditionnelles. Lorsque exécutés simultanément, l'algorithme génétique n'a pas fini de constituer sa population initiale qu'une méthode basée sur le calcul a terminé et trouvé la solution. De fait, l'implémentation d'un tel algorithme se justifie lorsque le problème à résoudre est complexe, par exemple si l'espace de données des variables liées au problème est vaste.

Fonctionnement général

L'algorithme génétique est un algorithme d'optimisation : il commence par définir les variables d'optimisations, la fonction de fitness, et le coût, et il termine par tester une éventuelle convergence. Le fonctionnement entre ces deux étapes est propre à l'algorithme génétique. De nombreuses variantes ont vu le jour ces dernières décennies, néanmoins elles gardent toujours une base identique. L'algorithme 3.1 présente le fonctionnement d'un algorithme génétique basique.

Algorithme 3.1 Algorithme génétique basique

```
Début
initialiser population
evaluer population
TantQue minimum non-trouve Faire
  selection naturelle
  effectuer croisement & mutation
  evaluer population
FinTantQue
Fin
```

Une des particularités de l'algorithme génétique est qu'il dispose de nombreuses solutions au problème posé. C'est ce qu'on appelle une population. C'est en faisant évoluer cette population que l'algorithme permet de converger vers la solution optimale. La population initiale est généralement générée de manière aléatoire, on peut cependant envisager d'autres techniques pour constituer l'ensemble des individus. Les générations successives d'individus sont représentées par une boucle while, où chaque itération de la boucle constitue la création et l'évaluation d'une génération. Ainsi, tant qu'aucun individu n'est évalué comme la solution optimale, une nouvelle génération est obtenue, grâce au mécanisme de sélection naturelle, qui permet de ne garder que les individus les plus forts (i.e. dont le coût

est le plus faible), et les opérateurs génétiques de croisement et de mutation, qui créent de nouveaux individus à partir de ceux issus de la sélection naturelle selon la fonction de fitness.

Dans les paragraphes suivants, nous expliquons le rôle de chacun des composants de l'algorithme génétique.

Population initiale La première étape de l'algorithme génétique consiste à initialiser la population de manière aléatoire. Cette population est constituée d'individus. Chaque individu représente une solution au problème que l'algorithme tente de résoudre. Un individu regroupe donc les valeurs de chaque variable d'optimisations, et ces valeurs sont représentés par des chromosomes. Une approche répandue est d'utiliser la représentation binaire des valeurs des variables, et de concaténer les séries de bits obtenues pour former un seul ensemble. Cet ensemble constitue un individu. Cette représentation est pratique puisqu'elle facilite la tâche des opérateurs génétiques.

Evaluation l'évaluation de la population est effectuée avec une fonction de fitness. Ce peut être une fonction mathématique, un appareil, une expérience. L'essentiel est de pouvoir déterminer l'efficacité, le coût de la combinaison de valeurs des variables d'optimisations pour chaque individu, de manière à pouvoir les comparer et éventuellement déterminer si l'un d'eux représente la solution optimale.

Sélection Naturelle L'objectif de la sélection naturelle est de supprimer individus les plus faibles de la population, selon leur valeur de fitness. Ne garder que les individus avec le meilleur fitness permet de faire évoluer la population vers la solution optimale. Pour cela, les individus sont classés selon leur fitness, et les plus faibles sont supprimés jusqu'à ce que la taille de la population soit égale au pourcentage de sélection, défini au préalable et qui précise le pourcentage d'individus à supprimer . Cette procédure fait directement écho au principe réel de sélection naturelle propre à la théorie de l'évolution, où seules survivent les espèces les plus adaptées à l'environnement dans lequel elles se trouvent.

Opérateurs génétiques de croisement et de mutation Ces opérateurs reflètent le principe même de l'évolution : à partir d'un ou plusieurs individus, obtenir un ou plusieurs individus avec un bagage génétique légèrement différent. Lorsque couplés à la sélection naturelle, ils contribuent à "améliorer" la population. En effet, seuls les individus les plus forts sont disponibles pour ces opérateurs. L'algorithme dispose de deux opérateurs, le croisement et la mutation.

La forme la plus commune de l'opérateur de croisement est représentée en figure 3.9. Deux des individus ayant le meilleur fitness sont sélectionnés pour produire deux descendants. Pour mieux comprendre, expliquons le mécanisme à l'aide d'une représentation binaire, où l'ensemble des chromosomes d'un individu est une suite de bits. Un point de croisement est déterminé, qui a pour objectif de séparer la suite de bits, et chaque descendant recevra une des deux portions obtenues. Considérons les individus pères A et B, dont les chromosomes forment une suite de 10 bits. Le point de croisement est défini à 4. Cela signifie que les 4 premiers bits de l'individu A constituent les 4 premiers bits du descendant n° 1, et les 4 premiers bits de l'individu B constitue les 4 premiers bits du descendant n° 2. On inverse alors la correspondance père-fils, et les 6 derniers bits de l'individu A sont affectés à l'individu n° 2, et les 6 derniers bits de l'individu B sont affectés à l'individu n° 1. Les deux descendants sont ensuite incluent dans la population,

et s'il s'avère que leur coût est plus faible que la moyenne des individus de la population, ils seront conservés selon le processus de sélection naturelle, éventuellement au profit de leur "ancêtres".

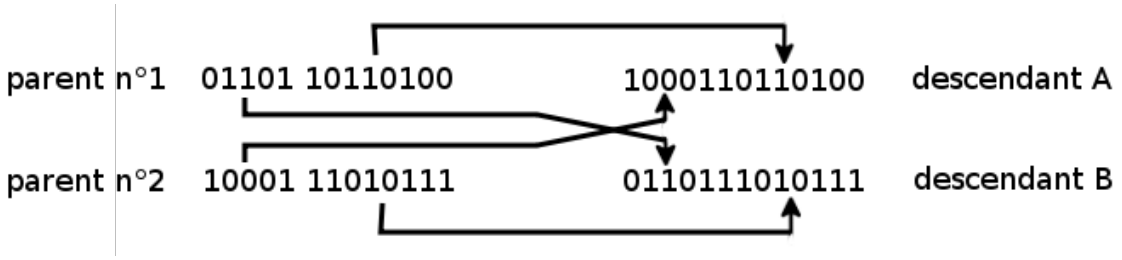

FIGURE 3.9 – Opérateur génétique de croisement

Le fonctionnement d'un algorithme de recherche traditionnel reste globalement le même : depuis un point de départ arbitraire, l'objectif est de se déplacer en direction de la solution au meilleur fitness. De fait, ce type d'algorithme a de fortes de chances de trouver un minimum local. L'opérateur de mutation, représenté en figure 3.10, est une des raisons pour lesquelles l'algorithme génétique a moins tendance à trouver un minimum local plutôt qu'un minimum global. L'intérêt d'un tel opérateur est d'intégrer de nouvelles données génétiques à un individu. Représentons l'espace de solutions au problème posé comme une surface avec des collines et des vallées, les collines représentants les solutions dont le coût est le plus élevé, et les vallées les solutions dont le coût est le plus faible. L'opérateur de mutation peut potentiellement obtenir une solution qui est géographiquement diamétralement opposée à la population actuelle, dont les individus s'orientent vers un nombre limité de vallées. Son principe est simple, apporter une modification au bagage génétique d'un individu. Dans le cas d'une représentation binaire, il s'agit de changer la valeur d'un des bits.

FIGURE 3.10 – Opérateur génétique de mutation

Algorithme génétique pour la génération de tests

A ce jour, aucun travaux ne proposent d'utiliser l'algorithme génétique pour générer des tests à partir de modèles UML. Cependant, UML est un langage de modélisation graphique pour représenter des systèmes orientés objet, nous pouvons donc nous inspirer des travaux de recherche sur le génération de tests de programmes orientés objet. De plus, l'objectif de test de ces approches est adapté à notre solution, puisque la génération de tests est évaluée selon des critères de couverture structurelle des méthodes de classe sous test.

Tonella [83] propose une approche intéressante, où les individus sont représentés comme un tableau d'appel de constructeurs et de méthodes, avec leur paramètres. Il a introduit de nouveaux opérateurs génétiques, qui agissent à différents niveaux : sur les constructeurs, sur les méthodes, ou sur les paramètres. Ils consistent principalement à

insister sur la modification du contenu d'un individu pour accroître les chances d'obtenir un chemin d'exécution nouveau. Il utilise tous-les-arcs comme critère de couverture.

Wappler et Lammermann [86] utilise une forme traditionnelle de l'algorithme génétique pour le test de programmes orientés objets. Dans leur solution, un individu contient des informations à propos des méthodes appelées, ainsi que leur valeurs de sortie et leurs paramètres. Ils proposent des techniques pour encoder et décoder ces trois éléments pour former un individu. Les critères de couvertures utilisés sont basés sur les décisions.

Les travaux de Ghiduc sont directement inspirés de Tonella et de Pargas *et. al* [71], qui propose une approche génétique pour générer des tests de programmes procéduraux. Ghiduc utilise une représentation des individus et des opérateurs génétiques relativement identiques à ceux proposés par Tonella. La valeur ajoutée de ses travaux réside dans l'utilisation d'un graphe de dominance pour représenter la méthode de classe sous-test, obtenu à partir d'un graphe de flot de contrôle selon la technique présentée par Harrold et Rothermel [48]. L'utilisation d'une telle structure permet à la génération de test de couvrir des critères basés sur les données, comme toutes-les-definitions.

Même si leur fonctionnement diffère, les stratégies citées ci-dessus ont toujours le même but : générer des tests pour une méthode de classe. Dans ce mémoire, nous nous situons à un niveau d'abstraction plus élevé, et l'objectif et non pas de tester une mais toutes les méthodes du SUT. Il s'agit donc de s'inspirer des travaux existants pour développer une solution originale.

3.4 Bilan

Pour déterminer si un fragment de code, un programme, ou un système a été correctement testé, des critères de couverture ont été établis. On peut trouver quatre catégories principales de critères : les critères basés sur les décisions, les critères basés sur les chemins, les critères basés sur les données, et les critères basés sur les modèles à transitions.

La notation exposée dans ce document est composée de trois langages, sur lesquels on peut appliquer les critères suivants :

- Alf : il peut être soumis à tous les critères destinés au test de code source (décisions, chemins, données).
- UML/OCL : cette combinaison peut satisfaire tous les critères basés sur les transitions. Une adaptation des critères basés sur les décisions et les chemins est possible.

Pour la génération automatique de tests, nous avons décidé de nous intéresser à l'algorithme génétique. En effet, notre stratégie principale est de type aléatoire, et nous utilisons ce type d'algorithme pour optimiser cette stratégie. Cet algorithme de type évolutionnaire est basé sur les principes de la génétique et de la sélection naturelle. Adaptée à une problématique de génération de test, l'algorithme génétique dispose d'une population initiale de séquences de test générées aléatoirement, chacune de ces séquences est constituée d'appels d'opérations et de valeurs de paramètres. Il s'agît alors de faire évoluer cette population par mutation et reproduction des séquences, de manière à les faire converger vers l'objectif de test : le respect d'un critère de test.

Il n'existe pas, à ce jour, de travaux basés sur l'implémentation d'algorithme génétique pour la génération automatique de tests à partir de modèles UML. Nous devrons donc

nous inspirer des différentes approches proposer pour le test de programmes impératifs et orientés objets, pour en déduire une solution adapté à notre objectif.

Chapitre 4

Environnement de modélisation et d'animation

Contents

Ce chapitre est consacré à l'implémentation de notre solution MBT nommée UML-Alf-TST. L'environnement développé pour ce mémoire se décline en deux étapes.

Il est d'abord nécessaire, pour représenter un système sous test selon les langages UML/OCL/Alf, de disposer d'un modeleur. De nombreux outils existent, aussi la première étape consiste à choisir les technologies les plus adaptées à notre solution parmi celles existantes (voir section 2.3.1). Le modeleur nécessite d'être pleinement fonctionnel, intégrable dans une chaîne outillée, et adaptable selon une utilisation spécifique (dans notre cas, modéliser un système pour générer des tests).

Ensuite, il faut pouvoir parser et interpréter un bloc de code Alf, dans l'objectif d'exécuter de simuler l'exécution d'une opération, et de mettre à jour le modèle en conséquence. Aucun outil ne permet à ce jour d'animer un modèle dont le corps des opérations est renseigné avec Alf. La seconde étape consiste donc à développer un composant d'animation de modèles. L'ingénieur validation doit pouvoir exécuter manuellement des opérations, et observer directement les changements sur le système. L'outil doit aussi permettre à un module tiers d'exécuter une suite d'opérations de manière automatique.

Le chapitre est structuré comme suit. La section 4.1 définit les technologies utilisées pour la modélisation de système avec notre notation. Le développement du module

d'animation de modèles est expliqué à la section 4.2. Nous illustrons les procédés de modélisation et d'animation avec une étude de cas en section 4.3.2. La section 4.4 propose un bilan sur le développement.

4.1 Outils pour la modélisation

De nombreux logiciels permettent la création de modèles UML (voir section 2.3.1), cependant les quelques travaux concernant le langage Alf sont intégrés à Eclipse [7], et plus particulièrement au modeleur Papyrus MDT [13]. Même si le modeleur est toujours en cours de développement, sa qualité ne cesse d'augmenter. Il est suffisamment abouti pour être utilisable en l'état. Papyrus MDT est ainsi l'outil de modélisation dédié à notre solution.

Papyrus MDT est un modeleur principalement développé par le CEA datant de 2008, dont l'origine est la fusion de deux autres modeleurs, Papyrus UML [15] et TopCased [19]. Ce n'est pas un logiciel "stand-alone", mais une extension constituée d'un ensemble de greffons branchés sur l'environnement Eclipse. D'ailleurs, Papyrus est un des projets Eclipse inclus dans l'ensemble "Modeling Tools", ce qui signifie qu'il est livré par défaut avec la version d'Eclipse orientée modélisation.

Dans l'état actuel des choses, la phase de modélisation est en grande partie réalisable. Il est possible d'ajouter des contraintes OCL et des actions Alf au sein des opérations de classe. Le CEA développe actuellement un éditeur Alf permettant de spécifier le comportement des opérations et des transitions via un simple clic droit sur l'élément. Pourtant, l'éditeur n'est pas finalisé, et son utilisation complète est impossible. Au niveau ergonomique, certaines parties de la grammaire ne sont pas prises en compte, et l'analyseur syntaxique a tendance à souligner la majeur partie du code, compromettant sa lisibilité. Au niveau fonctionnel, l'ajout d'action Alf aux transitions d'un diagramme d'Etats-Transitions n'est pas encore implémenté.

L'ajout de contraintes OCL, bien que peu ergonomique, est toutefois possible. Cependant il n'existe pas, à l'heure actuelle, d'évaluateur de contraintes OCL lié au modeleur Papyrus MDT. Il n'y a aucun moyen de vérifier une contrainte OCL, et adapter un évaluateur OCL de type "stand-alone" n'est pas faisable rapidement, puisque le contexte d'une contrainte est établi par le modèle UML lui-même (et non avec l'utilisation du mot-clé "context").

Face à ces problèmes, la combinaison UML-OCL-Alf (définie en chapitre 2) a été révisée. Le diagramme d'Etats-Transitions issu d'UML a été retiré, puisqu'il n'est pas possible d'en tirer parti. Sa réintégration pourra alors faire l'objet de travaux ultérieurs. Notre solution permet ainsi la création de modèles de type Pré/Post. Concernant les contraintes OCL, nous avons décidé de les traduire en Alf, de manière à pouvoir utiliser le composant d'animation pour évaluer les contraintes. En effet, puisque le langage Alf possède l'expressivité d'OCL, toute contrainte OCL est exprimable en Alf, et ce avec une très faible variation grammaticale. Ainsi, toutes les constructions Alf utilisées dans ce document pour définir une précondition respectent le pouvoir d'expression OCL.

4.2 Développement du composant d'animation

Comme vu dans la section 2.3.2, aucun outil pour l'animation de modèles UML ne supporte aujourd'hui la totalité de l'approche présentée dans document. La principale raison

est qu'aucun ne propose d'utiliser le langage Alf pour réaliser les changements d'états du modèle. L'une des principales tâches de ce projet a donc été de développer un composant d'animation dédié aux langages UML et Alf, à intégrer au modeleur Papyrus MDT. Cet outil doit pouvoir effectuer une analyse lexicale et syntaxique d'un bloc Alf, interpréter la signification de ce bloc, et mettre à jour le modèle ou une représentation du modèle en conséquence.

La section 4.2.1 explique le fonctionnement général du composant d'animation, la section 4.2.2 est consacrée à la transformation de la partie statique du modèle, la section 4.2.3 explique en détails la transformation de la structure dynamique.

4.2.1 Fonctionnement du module

Les différentes étapes nécessaires à l'animation de modèles peuvent être observées en figure 4.1. Avant tout, l'ingénieur validation doit avoir créé un modèle UML composé d'un diagramme de classes et d'un diagramme d'objets. Le SUT doit être représenté par une classe, et ses fonctionnalités par des opérations. Le comportement des opérations est renseigné avec Alf.

D'abord, les informations relatives à la modélisation doivent être traduites, transformées vers un format de données compréhensible pour l'animateur. D'une part, la structure statique du modèle, à savoir les diagrammes de classes et d'objets, est transformée à l'aide d'ATLAS transformation language (ATL [56, 57]) vers une instance de notre métamodèle UML4TST. Ce métamodèle représente le sous-ensemble UML utilisé dans ce mémoire. D'autre part le contenu dynamique, c'est-à-dire les actions Alf, est analysé par ANother Tool for Language Recognition (ANTLR [2]) et l'arbre de syntaxe abstraite (ou AST) de chaque action est construit. A partir de ces arbres, ANTLR est sollicité à nouveau pour générer leur graphe de flot de contrôle associé. Typiquement, un graphe de flot de contrôle est composé de nœuds d'instructions, contenant une liste de statements sous la forme d'un sous-arbre issu de l'AST initial, et de nœuds de décisions, représentants les boucles "if" et "for", dont la condition est également un sous-arbre issu de l'AST initial. Les graphes obtenus sont alors intégrés à l'instance UML4TST, en établissant un lien vers l'opération à laquelle l'action Alf est rattachée.

Enfin, nous disposons donc d'une instance UML4TST, où chaque opération possède un graphe de flot de contrôle. Il est alors possible, soit de piloter manuellement l'exécution d'une ou de plusieurs opérations et d'observer les changements apportés au système, ou d'utiliser un module tiers pour exécuter une suite d'opérations de manière automatique.

4.2.2 Transformation de la structure statique

Dans le cadre de ce projet, l'animation consiste à exécuter des méthodes de classe dont le fonctionnement est décrit avec Alf, et de mettre à jour le modèle en conséquence. Il n'est cependant pas envisageable de modifier directement le modèle UML, puisqu'il permet d'établir l'état initial du SUT. De plus la complexité du métamodèle UML est telle qu'il est préférable, autant pour l'animation d'un modèle que pour la génération de tests, d'avoir un métamodèle ad hoc restreint.

Métamodèle UML4TST

Issu du métamodèle UML4MBT créé par Smartesting et basé sur des travaux menés au LIFC [30], le métamodèle UML4TST (aussi issu du LIFC) a été élaboré pour répondre à

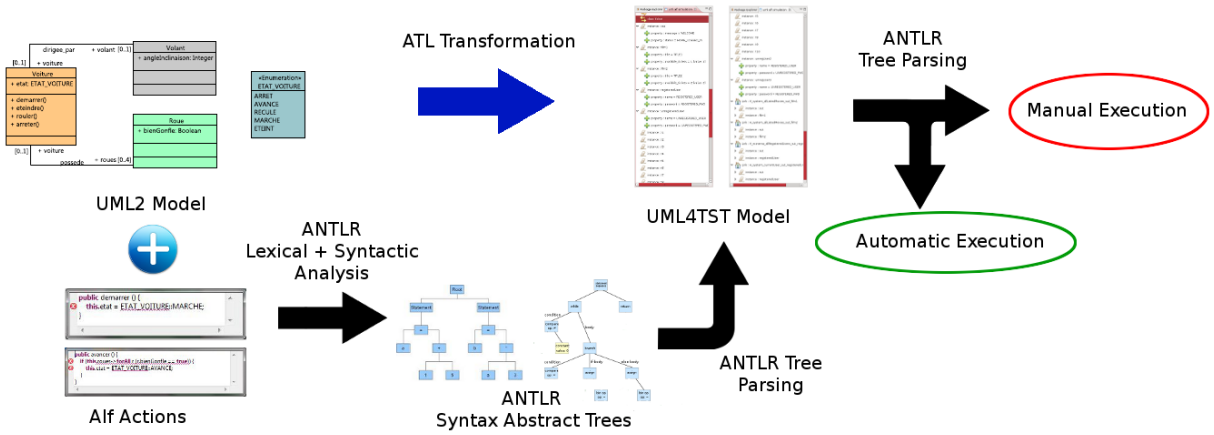

FIGURE 4.1 – Chaine d'exécution pour l'animation

plusieurs problématiques :

- Posséder un métamodèle du sous-ensemble UML dédié pour le test
- Réduire le nombre de transformation entre les métamodèles pour la modélisation et les métamodèles pour la génération de tests en servant de pivot.
- Posséder un métamodèle (UML4MBT est la propriété de Smartesting) potentiellement utilisable pour toute nouvelle approche MBT, comme celle présentée dans ce document.

Comme le sous-ensemble UML utilisé dans ce projet (voir section 2.1), UML4TST est composé de trois diagrammes pour la modélisation du SUT : un diagramme de classes pour décrire la structure statique, un diagramme d'objets pour définir l'état initial en fournissant un ensemble d'instances de classe, et un diagramme d'états-transitions pour représenter le comportement abstrait.

Nous utilisons donc ce métamodèle pour l'animation, et par extension la génération automatique de tests. Pour cela, il est nécessaire d'effectuer la conversion d'un modèle UML2 vers son équivalent UML4TST, en utilisant la technologie ATL.

Transformation avec ATL

La transformation d'un modèle UML2 vers un modèle UML4TST est effectuée avec ATL, un langage outillé pour la transformation de modèle. Simplement, il permet de transformer un modèle x, conforme au métamodèle A, vers un modèle y, conforme au métamodèle B. L'utilisateur propose un ensemble de règles de transformation qui établissent respectivement un lien entre un élément du métamodèle source et un élement du métamodèle cible. ATL effectue un parcours du modèle cible, et à chaque instance d'élément source rencontrée présent dans une règle, une instance d'élément cible est générée en respectant cette règle.

Dans le cadre du projet UML4TST, deux transformations sont utilisées (voir figure 4.2).

La première prend en entrée le métamodèle UML2 et un métamodèle PROBLEM en sortie. L'objectif ici est de déterminer si l'instance UML2 est exportable vers UML4TST.

En effet, ATL intègre OCL pour pouvoir contraindre une règle, il est ainsi possible de générer une instance d'élément cible selon certaines conditions. De ce fait, la transformation vers le métamodèle PROBLEM est constituée d'expression OCL représentant la négation des conditions nécessaires à respecter pour autoriser l'export vers UML4TST. Le métamodèle PROBLEM est composé d'une seule métaclasse *Problem*. Celle-ci possède un attribut *severity* qui définit la sévérité du problème caractérisé par l'instance courante. Cet attribut peut prendre la valeur "warning", signifiant que le problème n'est pas bloquant et que la transformation vers UML4TST est malgré tout possible, ou prendre la valeur "error", signifiant que le problème est bloquant et que la transformation vers UML4TST ne peut être complétée. En d'autres termes, si la métaclasse *Problem* est instanciée et si son attribut *severity* vaut "error", cela signifie que les conditions ne sont pas réunies pour la transformation UML2 vers UML4TST. Cette première transformation est donc un système de vérification de compatibilité.

La seconde transformation est la plus importante, elle permet d'exporter un modèle UML2 en UML4TST. Les règles définies ne concernent que les trois diagrammes vus en section 2.1. Tous les éléments UML2 qui ne sont pas présents dans l'ensemble de règles seront simplement ignorés.

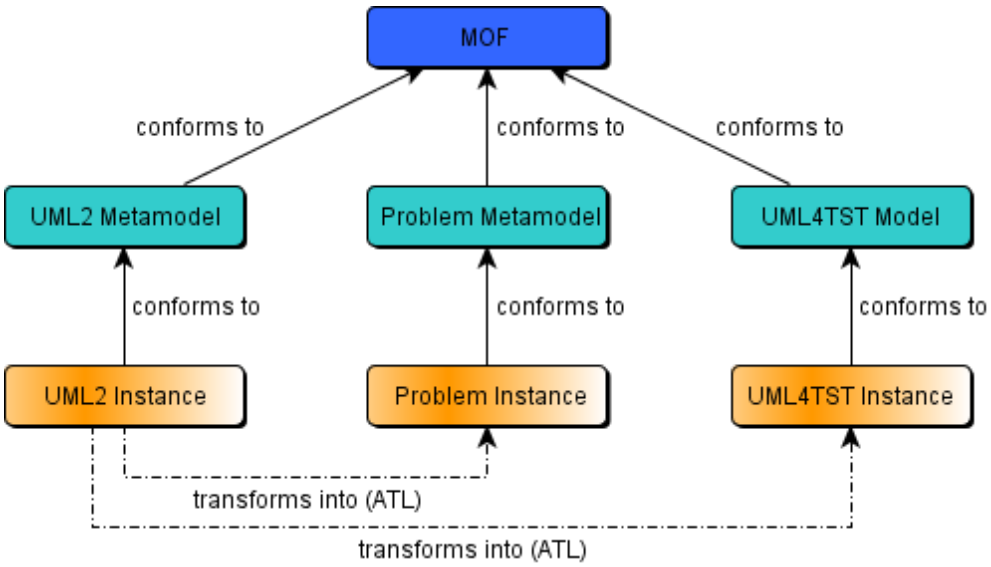

FIGURE 4.2 – Vue d'ensemble de la chaine de transformation

4.2.3 Transformation de la structure dynamique

Transformer la structure dynamique d'un modèle UML/OCL/Alf consiste principalement à parser les actions Alf selon une grammaire, pour extraire leur arbre de syntaxe abstraite respectif. Aussi, pour la définition, l'analyse et l'interprétation de bloc Alf, nous avons décidé d'utiliser le framework ANother Tool for Language Recognition (ANTLR [2]). En effet, c'est un outil relativement récent, qui dispose d'une communauté particulièrement active. De plus, il s'intègre parfaitement à l'environnement de développement Eclipse, que nous utilisons pour la réalisation de l'ensemble de ce projet.

Présentation du framework ANTLR

ANTLR est un langage outillé proposant un framework pour la construction de compilateur, traducteur, et dispositif de reconnaissance à partir de descriptions grammaticales, supplantées d'actions écrites en différents langages (Java, C, C++,...). ANTLR propose des outils pour la définition, l'analyse et l'interprétation de langage. Cet outil a une utilisation relativement étendue. La suite Xtext [22], qui permet de réaliser un éditeur de langage pour Eclipse, et basé sur une grammaire, utilise ANTLR pour proposer une analyse lexicale et syntaxique à la volée.

Un des avantages de cette technologie est qu'elle est particulièrement bien outillée. Un IDE a été développé par les créateurs de ANTLR, ANTLRWorks, qui permet la création assistée de grammaire, en proposant un système de navigation ainsi qu'un interpréteur de grammaire, et en repérant les éventuelles règles non-déterministes. D'autre part, un greffon eclipse nommé ANTLRIDE et possédant la majorité des avantages de ANTLRWorks. Ce greffon a été utilisé pour le développement du composant d'animation UML/Alf.

Il est possible de définir une grammaire de type LL(*), et d'obtenir un CST (Concrete Syntax Tree ou Arbre de Syntaxe Concrète) suite au parsing d'un fragment de code conforme à la grammaire. Il est aussi envisageable de surcharger chaque règle de grammaire avec des blocs Java pour réaliser une interprétation à la volée, mais cette méthode est peu recommandée pour un langage complexe. Dans ce cas, il est plus intéressant d'utiliser la technique de "Tree rewriting", qui consiste à influencer la conversion d'un fragment de code vers son arbre syntaxique équivalent. Cette technique permet d'ignorer tous les éléments grammaticales qui n'ont pas d'intérêt pour l'interprétation. C'est de cette manière qu'ANTLR propose de générer un AST (Abstract Syntax Tree ou Arbre de Syntaxe Abstraite). Enfin, pour interpréter un AST, deux méthodes :

- Elaborer une "grammaire d'arbre" (tree grammar). Le fonctionnement est alors identique à une grammaire classique, sauf que l'on traite ici des arbres (AST). il s'agit d'effectuer un parcours d'arbre (tree walking) dicté par un ensemble de règles, et d'exécuter les blocs Java contenus dans ces règles pendant le parcours.

- Utiliser la technique de "Tree pattern matching". L'AST sera parcouru entièrement quoiqu'il arrive, et deux types de règle sont proposés. Des règles qui ne s'appliquent à un sous-arbre qu'à la descente, et d'autres qu'à la montée. Cette technique est particulièrement utile lorsque l'on ne souhaite travailler qu'avec un certain sous-ensemble d'un Arbre donné, ou que l'on souhaite extraire un type d'information précis. Dans ce projet, cette technique est utilisée pour inverser l'ordre de certains nœuds de l'AST, étape nécessaire à la transformation d'un Arbre vers un graphe de flot de contrôle (voir section 4.2.3).

Les différentes possibilités offertes par ANTLR ont été utilisées afin de mettre en place une chaine d'analyse et d'interprétation, visible en figure 4.3.

Processus d'analyse lexical et syntaxique

Tout d'abord, pour définir le langage Alf, une grammaire de type LL(*) a été développée, que nous appellerons *AlfGram*. Deux sources d'influence ont permis sa réalisation. La première est la grammaire JavaCC proposée par les développeurs du langage Alf. Celle-ci est très complète, puisqu'elle ne se limite pas qu'aux statements mais permet de définir des modèles entièrement écrits en Alf (voir la section "Conformance" à la page 2 de la

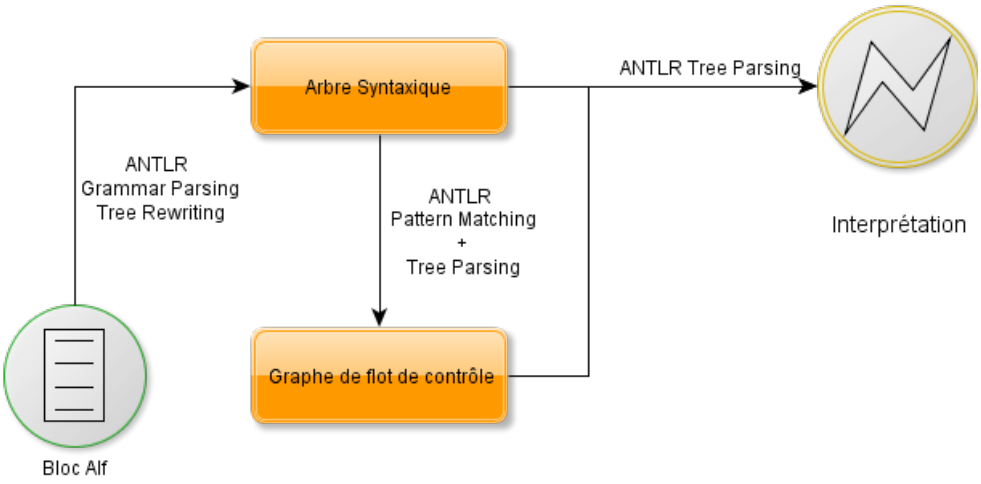

FIGURE 4.3 – Processus d'analyse et d'interprétation

spécification Alf [52]). Cependant, certaines constructions sont trop complexes, et des ambiguïtés sont apparues pendant la phase de diminution du pouvoir d'expression de la grammaire. Les sous-parties sources de problème ont été retirées, et c'est la grammaire XText / ANTLR issue de l'éditeur Alf du CEA qui a permis de compléter *AlfGram*. Cette grammaire, tout comme l'éditeur qui l'utilise, n'est pas finalisée, mais certaines constructions sont plus simples et plus facilement réutilisables. *AlfGram* est ainsi un savant mélange de deux grammaires, une de type JavaCC, et l'autre de type XText / ANTLR.

Lorsqu'un bloc Alf est parsé selon la grammaire *AlfGram*, le résultat obtenu est un AST, et il s'agit ensuite de transformer cet AST en un graphe de flot de contrôle. On utilise une grammaire d'arbre définir avec ANTLR pour parcourir l'AST et instancier une structure de graphe de flot de contrôle.

Extension du métamodèle UML4TST : Intégration d'une structure de graphe de flot de contrôle

Pour faciliter la mise en place du composant d'animation, nous avons décidé d'intégrer la structure de graphe de flot de contrôle développée pour notre solution au métamodèle UML4TST. Cette structure est utilisée pour l'interprétation du contenu Alf d'une méthode, et peut permettre, par exemple, de mesurer le respect d'un critère de couverture de type flot de contrôle. L'intérêt d'étendre UML4TST est de s'affranchir du processus de transformation d'un bloc Alf vers son AST avant chaque interprétation d'opération : des sous-parties de l'AST sont stockées dans les nœuds du graphe, qui peuvent être directement interprétées. Davantage d'informations à ce sujet sont proposées dans la sous-section 4.2.4.

La figure 4.4 ci-dessus propose une représentation graphique de l'extension apportée au métamodèle UML4TST, représentant un graphe de flot de contrôle. L'entité *CFGraph* représente le graphe lui-même. Il possède une tête et une liste de nœuds, tous deux de type *Node*. *Node* est en réalité une classe abstraite, et il existe quatre type de nœuds :

- *EndNode* : Le nœud final. Il ne contient rien (son attribut *code* est vide), et permet simplement de signaler la fin du graphe à l'animateur.

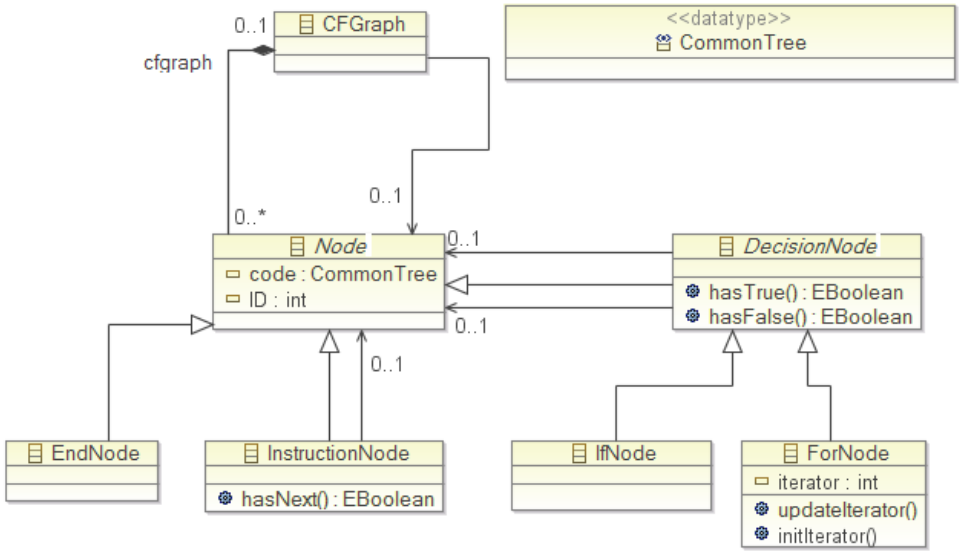

FIGURE 4.4 – Vue d'ensemble de la structure de graphe de flot de contrôle

- *InstructionNode* : Le nœud correspondant à un bloc d'instruction. Son attribut *code* contient le sous-arbre correspondant au bloc Alf.
- *ForNode* : Le nœud de décision issu d'une boucle for, son code contient la condition d'arrêt, exprimée par un sous-arbre dont la racine est de type *expression*.
- *IfNode* : Le nœud de décision issu d'un statement if, son attribut *code* contient la condition de branchement, exprimée par un sous-arbre dont la racine est de type *expression*.

Un lien a été établi entre l'entité *CFGraph* et l'entité *AlfActionBehabiour*. Il devient possible, pour une méthode donnée, de connaître son corps écrit en Alf, et le graphe de flot de contrôle correspondant. Une remarque cependant, les tranformations vues en section 4.2.2 ne permettent pas d'instancier le graphe de flot de contrôle d'une méthode, la manière de procéder est décrite dans la section suivante.

4.2.4 Processus d'interprétation

L'interprétation d'un bloc Alf nécessite le parcours de deux graphes : le graphe de flot de contrôle, et le sous-arbre de syntaxe abstraite contenu dans l'attribut "code" de chaque nœud du précédent graphe.

Le parcours d'un sous-arbre de syntaxe abstraite Alf, et donc son interprétation, est assuré par une seconde grammaire d'arbre de type ANTLR. Puique son rôle ne se limite qu'à traiter des sous-arbres, les sous-arbres qui sont contenus dans l'attribut *code* des nœuds du graphe, cette grammaire possède trois règles de point d'entrée :

- **EvaluateStatements** : cette règle attend comme racine un nœud de type "STATEMENTS", et son rôle est d'interpréter chaque fils.
- **EvaluateIfExpression** : prévue pour déterminer si la condition d'un statement If est vraie ou non. La racine attendue est de type "expression", et la valeur de sortie est le résultat booléen de l'évaluation de l'expression.

- **EvaluateForLoop** : même type de règle que la précédente mais destinée à la condition d'une boucle For.

Le parcours du graphe de flot de contrôle associé à une méthode est réalisé par la classe *AlfInterpreter*. Elle est constituée d'une méthode générique *execute* qui prend en entrée un nœud quelconque de type *Node*, et d'un ensemble de méthodes spécifiques, chacune dédiée au traitement d'un type de nœud. Ainsi, pour parcourir le graphe, le nœud correspondant à la tête du graphe est donné en entrée à la méthode générique, qui redirige ce nœud vers la méthode spécifique correspondant à son type :

- S'il s'agît d'un nœud de type InstructionNode, la méthode *executeInstruction* est invoquée. Elle invoque à son tour la règle *EvaluateStatements*, pour interpréter le bloc d'instruction, et fait ensuite appel à la méthode générique, avec en entrée le nœud suivant.

- La méthode *executeIfDecision* est invoquée s'il s'agit d'un nœud de type IfNode. Cette méthode fait appel à la règle d'interprétation *EvaluateIfExpression*, qui retourne le résultat de l'évaluation de son expression booléenne. Si le résultat est positif, c'est le nœud fils True qui est donné en entrée à la méthode générique, sinon le fils false est donné.

- Un nœud de type ForNode est traité par la méthode *executeForDecision*. Elle fait appelle à la règle *EvaluateForLoop*, et selon l'évaluation de la condition, donne en entrée à la méthode générique le nœud contenu dans la boucle for, ou le nœud suivant.

- Si le nœud donné en entrée est de type *EndNode*, cela signifie que l'exécution est arrivée à son terme, et la méthode générique termine.

4.3 Cas pratique : modélisation et Animation d'un Cinéma

Nous nous proposons dans cette section de mettre en pratique l'environnement de travail développé pour notre solution, en modélisant puis en animant l'application ECinema, une application web d'achat de billets de cinéma vue en section 2.4.

4.3.1 Modélisation de l'application avec Papyrus MDT

Nous ne donnerons dans ce document qu'un rapide aperçu du procédé de modélisation de système avec Papyrus MDT. En effet, de nombreux tutoriels existent pour ce propos. Une présentation de type diaporama [14] est disponible sur la page d'accueil du modeleur, qui explique comment installer et prendre en main l'outil.

L'ergonomie générale de Papyrus MDT est représentée en figure 4.5. Nous avons extrait cinq vues du modeleur, chacune entourée par un rectangle coloré, et numérotée de 1 à 5. La vue n°1, en bleu, est le canevas de modélisation. Cette vue permet de représenter les entités de modélisation. Il est possible de les organiser comme on le souhaite, de leur définir une taille, et une couleur. La vue de propriétés (vue n°2 rouge) liste toutes les informations liées à un élément. Dans la figure ci-dessous, la classe ECinema est sélectionnée, et la vue n°2 nous indique la visibilité de la classe, si elle est abstraite, etc...

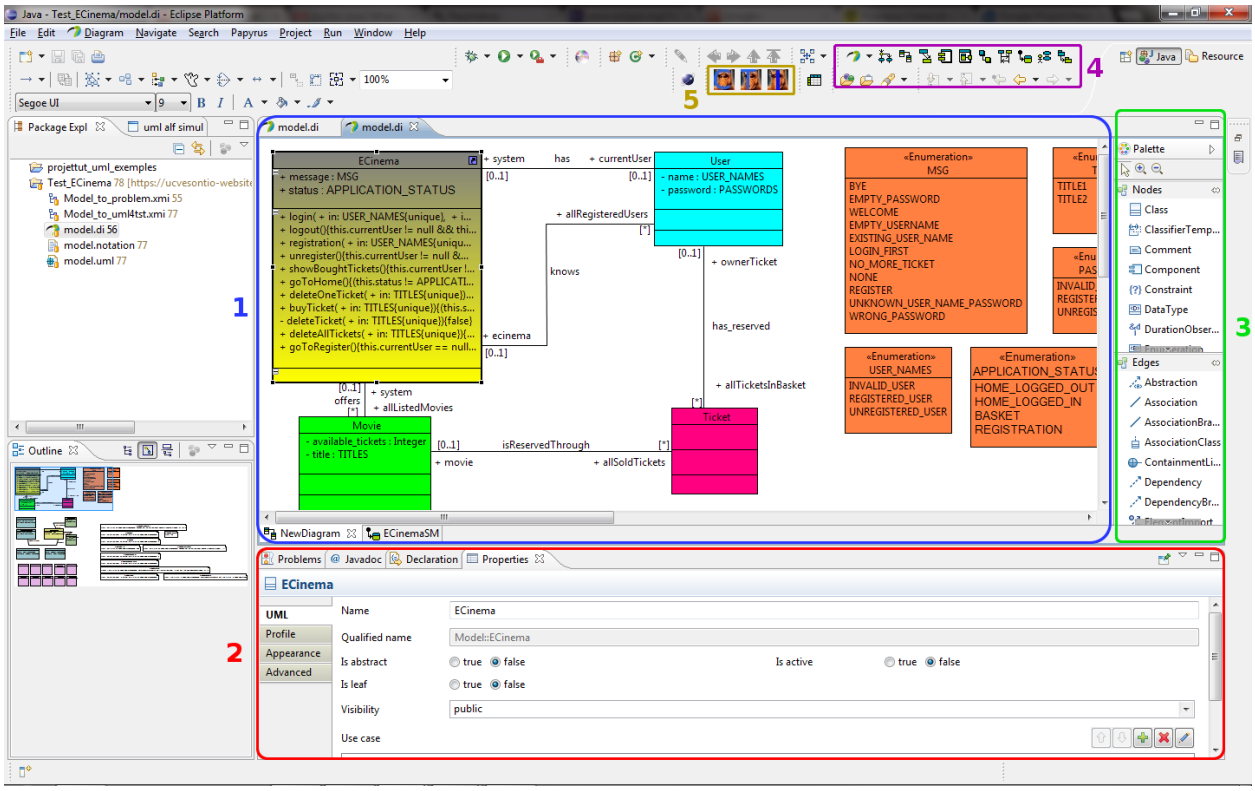

FIGURE 4.5 – Modélisation de ECinema avec Papyrus MDT

La majeur partie des données est modifiable. La vue n°3 (sertie d'un rectangle vert) consiste en une boite à outil, ou un panel. Toutes les entités est les liens utilisables sont représentés, et sélectionner l'un deux influe sur la vue n°1, puisqu'un clic gauche dans cette vue aura pour conséquence d'ajouter une instance de l'élément sélectionné en vue n°3 au modèle en cours d'élaboration. La quatrième vue, en violet sur la figure, propose un ensemble de boutons pour la création de diagrammes. Un modèle peut être représenté selon plusieurs diagrammes, c'est le cas pour ECinema qui dispose d'un diagramme de classes et d'un diagrammes d'objets. La cinquième vue, entourée de jaune, est composée de trois boutons, ajoutés à Papyrus MDT pour servir notre solution de génération de tests. Nous expliquerons leur utilité dans la section 4.3.2.

Lorsque Papyrus MDT est muni du greffon contenant l'éditeur Alf, il est alors possible de renseigner le corps d'une opération avec Alf. La marche est suivre est consultable en figure 4.6. Par exemple, nous souhaitons décrire le fonctionnement de l'opération *login*. Il faut effectuer un clic droit sur l'opération, et se rendre au menu *Edit Operation*, et au sous-menu *Using ALF Editor*. Une éditeur s'ouvre, et nous pouvons ajouter le code Alf correspondant au comportement souhaité de l'opération. On peut aussi constater sur la deuxième image de la figure 4.6 que l'analyse lexicale et syntaxique à la volée n'est pas terminée, puisqu'une grande partie du code est surligné.

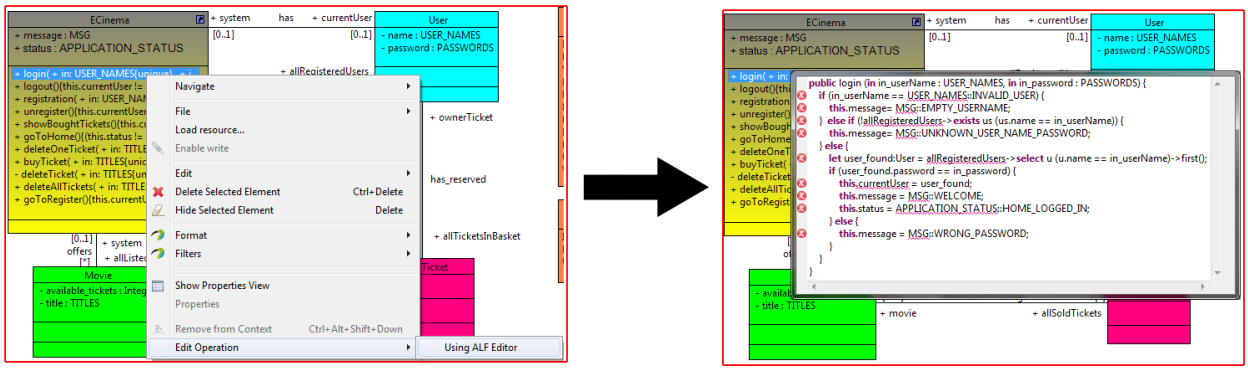

FIGURE 4.6 – Ajouter une action Alf avec Papyrus MDT

4.3.2 Animation manuelle du modèle issu de l'application

Lorsque l'étape de modélisation est accomplie, il est possible de procéder à une animation manuelle du modèle. La figure 4.7 illustre l'utilité des trois boutons ajoutés pour notre solution.

Le premier bouton a pour fonction d'exécuter la transformation UML vers Problem. Le résultat obtenu est un fichier de type xmi, visible dans l'encadré de gauche de la figure 4.7. Si la méta-classe *Problem* est instanciée, cela signifie que les conditions nécessaires à la transformation vers UML4TST ne sont pas réunies, ou qu'il y a une potentielle perte d'information. Selon si la sévérité des instances de Problem est de type "warning" ou "error", la seconde transformation peut-être interdite.

Lors de la transformation de vérification du modèle ECinema, seuls des problèmes à faible sévérité (de type "warning") ont été découverts : certains attributs de classe n'ont pas de valeur par défaut définie. Définir une valeur par défaut peut être utile, car lorsqu'un slot (instance d'attribut) d'une instance de classe n'a pas de valeur, c'est la valeur par défaut qui est affectée. Nous ignorons donc ces warning, puisque notre modélisation précise une valeur pour chaque slot de chaque instance de classe. La transformation vers UML4TST est donc réalisable.

Le second bouton permet de transformer les structures statique et dynamique du modèle. Concernant la partie statique, il faut d'abord s'assurer que la transformation de vérification a été effectuée avec la version actuelle du modèle. Si ce n'est pas le cas, la vérification est effectuée. Puis, la transformation ATL vers UML4TST est exécutée. Concernant la partie dynamique, la liste des opérations du SUT est parcourue et pour chacune d'elles, leur contenu Alf est parsé, l'AST correspondant extrait, et le graphe de flot de contrôle associé construit et intégré au modèle UML4TST.

Si les transformations de la structure statique et de la structure dynamique n'ont pas échoué, le modèle UML4TST est affiché à l'ingénieur validation. Il est visible à la figure 4.7 (entouré d'un rectangle bleu). Il a été choisi de représenter le modèle sous forme de liste, car ce format est moins complexe à élaborer qu'un format graphique, et la lisibilité est satisfaisante.

Pour exécuter une opération, l'ingénieur validation doit s'assurer que les transformations ont été correctement effectuées, et que le modèle UML4TST est visible. Si tel est le cas, il suffit de faire un clic-droit sur une des opérations du SUT, comme le montre la figure 4.8. Dans le menu qui s'affiche, deux actions : *Check Operation Precondition* et *Exe-*

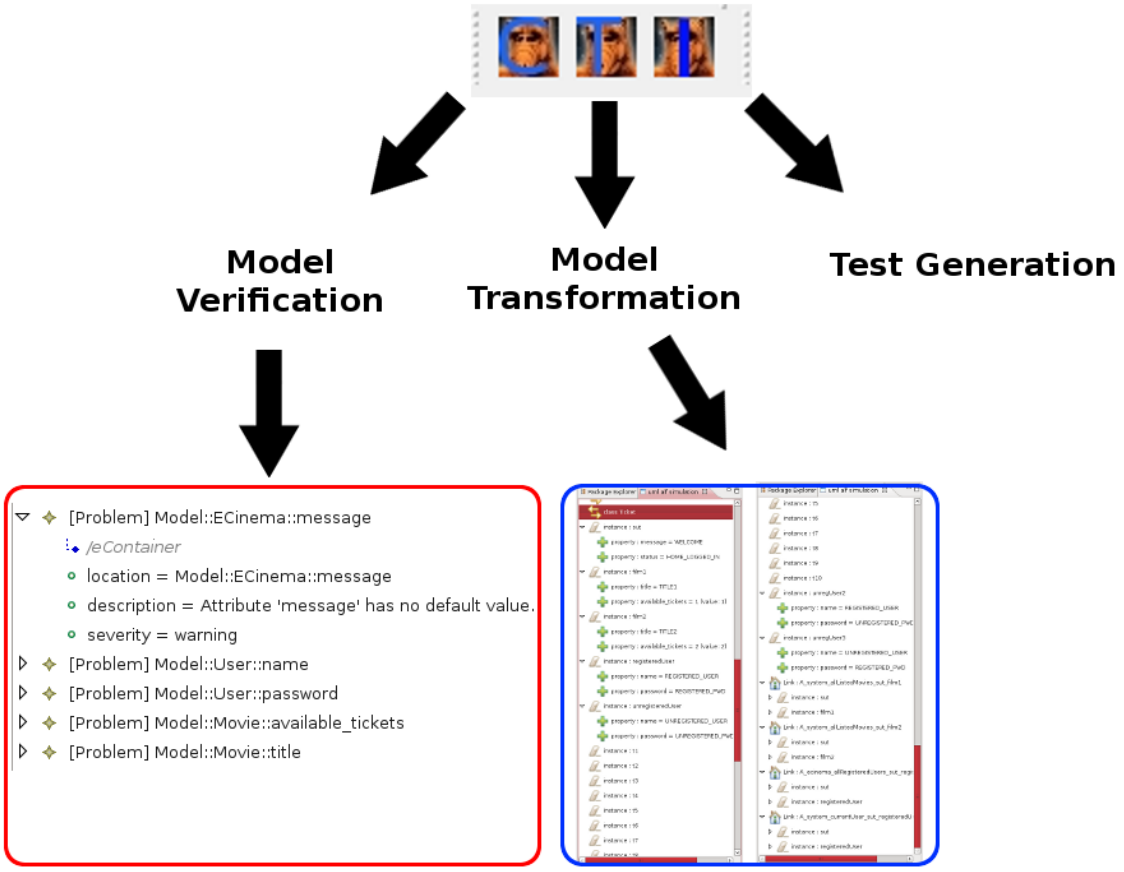

FIGURE 4.7 – Barre d'outils pour l'animation

cute Alf Body. La première action permet de vérifier si l'opération est exécutable compte tenu de l'état du modèle. L'ingénieur validation doit préciser les éventuels paramètres nécessaires, en choisissant parmi une liste déroulante (s'il s'agit de valeurs booléennes, de valeurs d'énumération, ou d'objets), ou en renseignant un champ (s'il s'agit de valeurs numériques entières). La seconde action effectue d'abord la même tâche que la première, à savoir vérifier la précondition de l'opération. Si la précondition est vérifiée, la classe *AlfInterpreter* est invoquée, et le graphe de flot de contrôle de l'opération choisie est exécuté (comme vu en section 4.2.4). Le modèle ainsi que son affichage sont alors mis à jour, et l'ingénieur validation peut observer le résultat de l'exécution de l'opération.

Dans notre cas, l'ingénieur validation a exécuté l'opération *login*. Le statut du système avant l'exécution de l'opération est "*HOME_LOGGED_OUT*", qui signifie que l'utilisateur fictif n'est pas connecté. L'opération *login* est exécutée, et le résultat est caractérisé par un changement de statut du système : il vaut désormais "*HOME_LOGGED_IN*", signifiant que l'utilisateur fictif est connecté à l'application. De plus, un lien entre l'utilisateur et le SUT a été créé, pour matérialiser la connexion.

4.4 Synthèse

Une partie importante du développement de la solution UML-Alf-TST a été consacrée aux modules qui sont utilisés pour ou pendant la génération automatique de tests à partir de modèles.

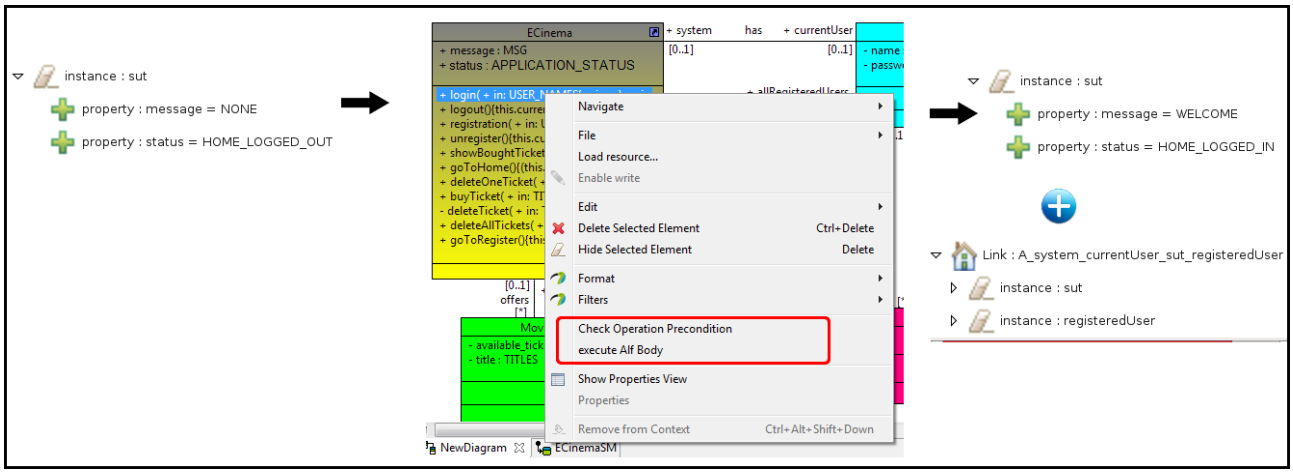

FIGURE 4.8 – Exécution d'une opération

Le premier outil permet la réalisation de modèles selon la notation UML/OCL/Alf. Le module a trois fonctions : modéliser un système avec UML de manière graphique, contraindre des opérations de classe avec une expression OCL, et renseigner le comportement de ces opérations avec Alf. Suffisamment de logiciels ont été conçus par le passé pour pouvoir réutiliser une ou plusieurs de ces technologies. Aussi, le modeleur choisi pour la modélisation UML et l'ajout de contraintes OCL est Papyrus MDT. Résultat de la fusions des modeleurs Papyrus UML et Top-Cased, Papyrus MDT est une suite de greffons qui dépendent de l'environnement de développement Eclipse. Il est ainsi facilement intégrable dans une chaîne outillée. L'ajout d'action Alf aux opérations de classe est permis avec le greffon de Papyrus MDT, nommé Alf Editor. Le développement de l'éditeur est en cours, mais la version actuellement disponible dans les dépôts de Papyrus MDT est suffisamment aboutie pour qu'Alf Editor soit intégré à la solution UML-Alf-TST.

Le second outil a pour fonction d'animer un modèle UML/OCL/Alf. Aucun travaux à ce jour ne portent sur le sujet, aussi il a été nécessaire de développer un composant d'animation dédié à notre solution. D'abord, animer un modèle implque de transformer celui-ci vers un format spécifique afin d'être plus facilement manipulable. Nous distinguons deux transformations : la transformation de la partie statique du modèle (diagrammes UML), et celle de la partie dynamique (contraintes OCL et actions Alf). La transformation de la partie statique est réalisée avec ATL, un langage outillé spécialisé pour cette tâche. Ainsi, le contenu UML est transformé vers son équivalent au format UML4TST. UML4TST est un métamodèle développé par le DISC pour représenter le sous-ensemble UML défini pour le test, qui a l'avantage d'avoir un pouvoir d'expression beaucoup plus restreint que celui d'UML. De plus, il est sans ambiguïtés, réduisant de ce fait la complexité du processus de génération de test. La transformation de la partie dynamique est assurée notamment par ANTLR, un outil pour la définition, l'analyse et l'interprétation de langages textuels. L'AST correspondant à une action Alf est d'abord extrait, puis un graphe de flot de contrôle est obtenu à partir de celui-ci. L'exécution d'une opération est réalisée grâce au parcours de son graphe de flot de contrôle, et à l'interprétation du contenu Alf de chaque nœud du graphe. L'animation de modèle consiste alors à exécuter une opération, et à mettre à jour le modèle en conséquence.

La chaîne outillée que nous avons développé rend possible la modélisation et l'anima-

tion de modèles décrits avec les langages UML/OCL/Alf. Il s'agit désormais de valider l'utilisation de cet outil dans le cadre d'une solution MBT. Pour cela, nous avons développé une stratégie de génération de test, qui permet de produire des cas de test de façon aléatoire. Sur l'interface de Papyrus MDT augmentée par notre ensemble de greffons, la génération de test est sollicitée en cliquant sur le troisième bouton "Génération de Tests", comme vu en figure 4.7.

Chapitre 5

Techniques de génération et expérimentations

Contents

Dans le chapitre précédent, nous avons présenté les modules de modélisation et d'animation développés pour la solution UML-Alf-TST. Ceux-ci permettent de représenter un système à l'aide des notations UML/OCL/Alf, de simuler l'exécution d'opérations du SUT, et de mettre à jour le système en conséquence. Même si le module d'animation rend possible une animation manuelle de modèles (voir section 4.3.2), son but premier est de fonctionner de paire avec un algorithme de génération. Ce dernier définit la suite de méthodes à simuler et le composant d'animation retourne le résultat, c'est-à-dire l'état du système.

Ce chapitre décrit plus en détail les stratégies de génération de test mises en place dans l'objectif de valider les outils d'animation et de modélisation pour une utilisation MBT. Deux types d'algorithme ont été développés, chacun d'eux a pour objectif de générer une suite de test qui satisfasse le critère tous-les-arcs pour chaque opération du SUT. Un premier algorithme génère des séquences de test selon une heuristique aléatoire, dont l'exécutabilité du contenu est évaluée avec l'animateur. Un second algorithme évolutionnaire de type génétique dédié à la génération de tests a été développé. Ici, une population

initiale de séquences de test est générées avec l'algorithme de génération aléatoire. A ces séquences sont ensuite appliqués des opérateurs propres à la génétique : mutation, croisement, etc... Dans l'optique de produire la population de séquences qui couvre le plus d'arcs possibles.

La première partie de ce chapitre est consacrée à l'algorithme de génération aléatoire et ses différentes optimisations. L'algorithme évolutionnaire et ses opérateurs génétiques sont abordés dans la seconde partie. Pour chaque technique de génération, une sous-partie est dédiée aux résultats de son expérimentation avec le modèle ECinema. La dernière partie propose une synthèse du travail accompli.

5.1 Génération aléatoire

La première stratégie mise en place consiste à générer des séquences de test dont le contenu, composé de comportements. Le choix des méthodes est déterminé de manière aléatoire. L'utilisation de ce genre de technique permet généralement d'obtenir un résultat rapidement, et a l'avantage d'être suffisamment peu complexe pour pouvoir être fonctionnelle rapidement.

L'objectif de cet algorithme est d'obtenir des séquences de test valides, i.e. dont chaque opération est exécutable compte tenu de l'état courant du modèle, en sachant que l'exécution d'une opération est à-même de modifier cet état. Pour cela, l'algorithme s'appuie sur l'état initial du modèle, et sur le module d'animation. Ce dernier réalise deux tâches. D'une part, il évalue la précondition des appels d'opération choisis aléatoirement, de manière à déterminer si ceux-ci sont intégrable à la séquence de test en cours de génération. D'autre part, il est utilisé pour simuler l'exécution du-dit comportement pour obtenir le nouvel état du modèle, et ainsi permettre le tirage de nouveaux comportements.

L'algorithme 5.1 décrit le mécanisme général de génération aléatoire de séquences de test.

5.1.1 Paramétrabilité et données d'entrée

Certaines données de l'algorithme sont mises sous forme de paramètres, de manière à pouvoir adapter son comportement aux différents modèles qu'il est susceptible de manipuler. Pour définir ces données, l'algorithme attend les valeurs en entrée. Pour chaque paramètre, on expliquera son rôle, et la valeur par défaut si elle n'est pas explicitement définie :

- *maxTime* : ce paramètre a pour fonction de définir le temps maximal d'exécution. Il borne ainsi l'algorithme en temps. Valeur par défaut : 30.000 millisecondes.
- *maxNbSeq* : ce paramètre permet aussi de borner l'algorithme. Il limite le nombre de séquences générées. Valeur par défaut : 2000 séquences
- *maxSeqSize* : il a été choisi de générer des séquences de taille égale. Ce paramètre fixe cette taille. Valeur par défaut : 20 opérations.
- *maxOpTries* : Lorsque une opération est choisie aléatoirement, l'algorithme tente de valider sa précondition en tirant une combinaison de paramètres. *maxOpTries* permet de fixer un nombre maximal de tentatives, avant de tirer une nouvelle opération. Valeur par défaut : 20 tentatives.

- *objCov* : il est possible que l'ingénieur validation ne souhaite pas couvrir tous les arcs de chaque opération d'un système, mais seulement un pourcentage précis. Ce paramètre défini le pourcentage souhaité. Valeur par défaut : 100%.

5.1.2 Présentation de l'algorithme

Le but de l'algorithme est de générer un ensemble de séquences de test, jusqu'à ce que l'un des objectifs de l'algorithme soit atteint. Pour cela, nous itilisons trois boucles "TantQue" imbriquées. La première boucle "TantQue" (lignes 3-22) est liée à la création de la suite de test. Chaque itération de cette boucle correspond à la création d'une séquence d'appels d'opération. Pour générer le contenu d'une séquence, la taille et le contenu de la séquence en cours *seqSize* sont initialisés puis on entre dans la seconde boucle "TantQue" (lignes 6-18). Ici, l'objectif est de générer le contenu d'une séquence, constitué d'une suite d'appels d'opération. Le tirage d'une opération est réalisé avec la méthode de tirage *tirageOp*, et le tirage de ses paramètres et la validation de sa précondition sont assurés par la 3ième boucle "TantQue" (lignes 8-17). Les éventuels paramètres sont tirés aléatoirement avec la méthode *tirageParams*, et la vérification de la précondition est effectuée à l'aide d'une close "Si" (lignes 10-16), en utilisant l'animateur. Si la précondition est évaluée positivement, l'opération est exécutée, toujours en utilisant l'animateur, le modèle est alors mis à jour, puis l'opération est ajoutée à la séquence. Sinon, les processus de tirage de paramètres et d'évaluation de la précondition sont sollicités à nouveau, jusqu'à ce que le nombre maximal d'essais pour une opération donnée soit atteint, ou que l'opération soit exécutée. Lorsque la séquence a atteint la taille maximale, la condition de la seconde boucle "TantQue" est vérifiée.

Enfin, la séquence est ajoutée à la suite et le nombre de séquences est incrémenté, pour ensuite évaluer le nouveau taux de couverture en prenant en compte la nouvelle séquence. Si le taux de couverture est suffisant, on sort de la première boucle "TantQue", et la suite de séquences, résultat de l'algorithme, est retournée.

5.1.3 Optimisations

Afin de réduire le taux d'échec dans le tirage d'opérations et de paramètres, des optimisations ont été élaborées. En effet, obtenir une opération valide, dont la précondition est satisfaite compte-tenu des paramètres tirés et de l'état du modèle, peut s'avérer très fastidieux. Rien n'empêche qu'une opération soit choisie indéfiniment alors que l'évaluation de sa précondition ne cesse d'échouer, parce que cette opération n'est pas exécutable dans l'état ou ce trouve le modèle, et ceci quelle que soit la combinaison de paramètres. Il devient nécessaire d'optimiser l'algorithme pour éviter la redondance dans le tirage d'opérations et de paramètres, pour réduire le nombre d'échecs et accroître la vitesse d'exécution de l'algorithme.

La première optimisation consiste à garder en mémoire, pour une opération en cours d'évaluation, la liste des ensembles de paramètres qui n'ont pas réussi à satisfaire la précondition. Cette liste, considérée comme une "liste noire", est prise en compte lors du tirage d'une nouvelle combinaison de paramètres, pour s'assurer de ne pas dupliquer les tentatives d'exécution. De cette manière, en plus de fixer arbitrairement un nombre d'essais maximal pour le tirage d'une opération, l'algorithme tente de trouver un ensemble de paramètres valides, et s'arrête s'il n'y a plus de combinaison possible. Cette optimisation permet ainsi d'empêcher une combinaison de paramètres d'être choisie une seconde

Algorithme 5.1 Algorithme de génération aléatoire

Entrées:

entier *objCov* ⇒ Taux de couverture à atteindre
entier *maxTime* ⇒ Temps maximal d'éxecution
entier *maxNbSeq* ⇒ Nombre maximal de séquences
entier *maxSeqSize* ⇒ Taille maximale d'une séquence
entier *maxOpTries* ⇒ Nombre de tentatives maximal pour le tirage d'une opération donnée

Locales:

Liste d'opérations *availableOps* ⇒ Opérations du SUT
Animateur *anim* ⇒ Animateur de modèles
entier *tCov* ⇒ taux de couverture courant
entier *elapsedTime* ⇒ Temps écoulé
entier *nbSeq* ⇒ Nombre de Séquences générées
entier *seqSize* ⇒ Taille de la séquence en cours
entier *opTries* ⇒ Nombre de tentatives pour le tirage d'une opération donnée
Séquence *currentSeq* ⇒ Séquence courante
Opération *currentOp* ⇒ Opération courante
TestSuite *testSuite* ⇒ La suite de séquences générée

1. **Début**
2. $tCov \leftarrow 0$, $nbSeq \leftarrow 0$
3. **TantQue** $tCov < objCov$ **et** $elapsedTime < maxTime$ **et** $nbSeq < maxNbSeq$ **Faire**
4. $currentSeq.init()$, $seqSize \leftarrow 0$
5. **TantQue** $seqSize < maxSeqSize$ **Faire**
6. $opTries = 0$, $currentOp \leftarrow tirageOp(availableOps)$
7. **TantQue** $opTries < maxOpTries$ **Faire**
8. $currentOp \leftarrow tirageParams(currentOp)$
9. **Si** $anim.evaluerPrecondition(currentOp)$ **Alors**
10. $anim.exec(currentOp)$
11. $currentSeq = currentSeq.[currentOp]$
12. $seqSize++$, $opTries \leftarrow maxOpTries$
13. **Sinon**
14. $opTries++$
15. **FinSi**
16. **FinTantQue**
17. **FinTantQue**
18. $testSuite = testSuite \cup \{currentSeq\}$
19. $nbSeq++$, $tCov \leftarrow$ evaluerCouverture()
20. **FinTantQue**
21. **Retourne** *testSuite*
22. **Fin**

fois, réduisant ainsi le nombre de tentatives d'exécution échouées, de même que le temps d'exécution.

Les modifications apportées à l'algorithme concernent la troisième boucle "TantQue", lignes 7-16. L'algorithme 5.2 permet de rendre compte de ces changements. Nous

Algorithme 5.2 Optimisation no.1 pour l'algorithme de génération aléatoire

```
6.  currentOp ← tirageOp(availableOps)
7.  paramFlag ← true
8.  blacklistedParams ← []
9.  TantQue opTries < maxOpTries et paramFlag Faire
10.   currentOp ← tirageParams(currentOp, blacklistedParams)
11.   paramFlag ← hasUntestedParamCombination(currentOp, blacklistedParams)
12.   Si anim.evaluerPrecondition(currentOp) Alors
13.     anim.exec(currentOp)
14.     currentSeq ← currentSeq ∪ currentOp
15.     seqSize + +
16.     blacklistedParams ← []
17.     opTries ← maxOpTries
18.   Sinon
19.     blacklistedParams ←= blacklistedParams ∪ currentOp.params
20.     opTries + +
21.   FinSi
22. FinTantQue
```

introduisons une variable *blacklistedParams*, représentant la liste noire, et nous ajoutons une action à la close "Sinon". Cette action consiste à ajouter les paramètres choisis dans la liste. De cette façon, lorsque la précondition de l'opération est fausse, l'algorithme visite la clause sinon, et ajoute les paramètres à la liste noire. La méthode de sélection aléatoire des paramètres *tirageParams* se base sur cette liste pour veiller à ne pas tirer une même combinaison plusieurs fois. Le code de cette méthode n'est pas visible dans ce document.

La seconde optimisation établit la liste des opérations que l'algorithme a déjà choisi, mais dont la précondition ne fut jamais satisfaite, quel que soit l'ensemble de paramètres proposé. Le tirage d'une nouvelle opération se base sur cette liste d'échecs, pour s'assurer qu'une opération ayant échoué précédemment ne soit choisie une nouvelle fois. Dès lors qu'une opération est tirée avec succès, cette liste est réinitialisée, puisque l'exécution de l'opération va modifier l'état du système, et des opérations dont la précondition ne pouvait être vérifiée seront peut-être désormais exécutables. Cette optimisation permet de réduire le nombre d'échecs de tirage d'opération : désormais, une opération non exécutable, compte-tenu de l'état du système, ne peut être choisie plusieurs fois. Et puisque le nombre d'échecs dans le processus de tirage d'une opération est réduit, la proportion de tirages réussis par rapport aux tirage ratés est accrue.

Les changements nécessaires à la mise en place de cette optimisation, visibles via l'algorithme 5.3, opèrent à plusieurs niveaux. D'abord, nous introduisons une nouvelle variable locale *blacklistedOps* à l'algorithme, qui contient la liste des opérations qui ne peuvent être exécutées. Ensuite, à la ligne 6, nous soustrayons les opérations blacklistées des opérations potentiellement exécutables, avant de passer la liste amoindrie à la méthode de tirage. Enfin, aux lignes 21-23, nous ajoutons une nouvelle close "Si", qui vérifie à l'aide de la méthode *hasUntestedParamCombination* si toutes les combinaisons de paramètres d'une opération ont été envisagées. Si tel est le cas, alors cette opération est ajoutée à la liste noire, et n'est dès lors plus utilisable par la méthode de tirage.

Algorithme 5.3 Optimisation no.2 pour l'algorithme de génération aléatoire

```
6.  currentOp ← tirageOp(availableOps − blacklistedOps)
7.  paramFlag ← true
8.  blacklistedParams ← []
9.  blacklistedOps ← []
10. TantQue opTries < maxOpTries et paramFlag Faire
11.   currentOp ← tirageParams(currentOp, blacklistedParams)
12.   paramFlag ← hasUntestedParamCombination(currentOp, blacklistedParams)
13.   Si anim.evaluerPrecondition(currentOp) Alors
14.     anim.exec(currentOp)
15.     currentSeq ← currentSeq ∪ currentOp
16.     seqSize + +
17.     blacklistedParams ← []
18.     blacklistedOps ← []
19.     opTries ← maxOpTries
20.   Sinon
21.     blacklistedParams ← blacklistedParams ∪ currentOp.params
22.     opTries + +
23.     Si non(hasUntestedParamCombination(currentOp, blacklistedParams))
        Alors
24.       blackListedOps ← blackListedOps ∪ currentOp
25.     FinSi
26.   FinSi
27. FinTantQue
```

Cette optimisation est davantage un complément de la première, puisqu'elle permet, d'une part, de ne pas avoir à garder en mémoire une matrice des combinaisons de paramètres pour pouvoir connaître les opérations à ne plus tirer. L'emploi d'une telle matrice pour un modèle complexe est sujette à l'explosion combinatoire du nombre de combinaisons de paramètres possibles, et son stockage peut potentiellement saturer la mémoire de la machine exécutant l'algorithme. D'autre part, il n'est pas possible avec la première optimisation de savoir si une méthode sans paramètre a été visitée ou non, puisque son éventuel tirage est conservé via ses paramètres. Tenir la liste des méthodes non-exécutables permet alors de garder une trace du tirage de méthodes sans paramètre.

5.1.4 Expérimentations avec ECinema

Pour évaluer l'efficacité de l'algorithme de génération aléatoire, nous l'avons appliqué au modèle d'exemple ECinema, présenté en section 2.4, pour générer une suite de tests abstraits.

A chaque mesure, les valeurs obtenues sont la moyenne de 100 exécutions de l'algorithme. De plus, les paramètres d'entrée sont fixés comme suit :

- *objCov* (taux de couverture objectif) : 100%
- *maxNbSeq* (nombre maximal de séquences générées) : 1000 séquences
- *maxOpTries* (nombre maximal de tentatives d'exécution par opération) : 20 tentatives

- *seqSize* (taille d'une séquence) : varie selon les expériences.
- *maxTime* (temps maximal d'exécution) : 30.000 ms

Les valeurs fixe des paramètres ci-dessus ont été obtenues après une première phase d'expérimentation, non présentée dans ce document. Les valeurs définies sont directement liées au modèle utilisé pour les expérimentations, et peuvent varier d'un modèle à un autre.

La figure 5.1 présente les statistiques issues de l'exécution de l'algorithme dans sa version basique (sans optimisations), avec des séquences de 10 opérations ($seqSize = 10$). On constate que l'algorithme est très efficace pour couvrir jusqu'à 80-85% des arcs, puisque ce taux est obtenu dès les premières séquences générées. Cependant, lorsque la barre des 80-85% est atteinte, l'efficacité de l'algorithme décroit fortement. En effet, près de 1400 séquences sont en moyenne générées pour couvrir 100% des arcs.

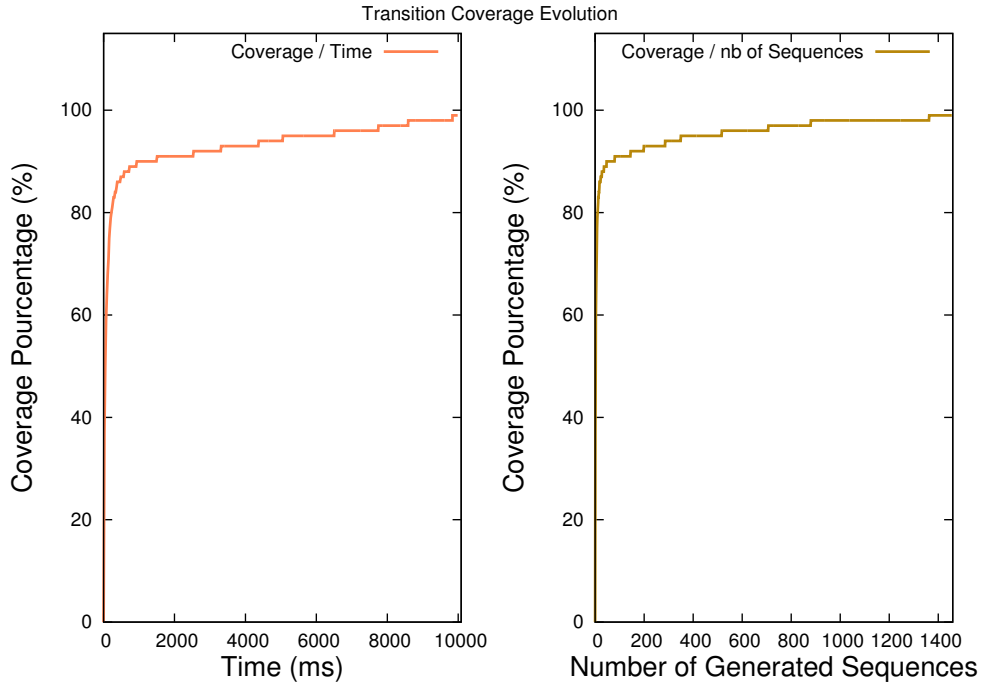

FIGURE 5.1 – Génération aléatoire, séquences de 10 pas, aucune optimisation

Les résultats issus de la figure 5.2 proviennent de l'exécution de l'algorithme, toujours dans sa version basique, mais avec des séquences de 20 opérations. Les constats sont les mêmes que précédemment : l'algorithme est très efficace pour couvrir 80-85% des arcs, puis nécessite de plus en plus de séquences pour couvrir les arcs restants. On note cependant une nette amélioration face à la précédente expérimentation. En effet seulement 7.000 millisecondes se sont écoulées (contre 10.000 millisecondes auparavant), mais surtout 400 séquences seulement ont été nécessaires pour couvrir la totalité des arcs (contre environ 1400 précédemment). On peut conclure que générer des séquences de 20 opérations permet d'obtenir de meilleurs résultats. Ceci peut s'expliquer par le fait qu'une séquence de 20 opérations représente un préambule plus important qu'une séquence de 10 opérations. En effet, certaines opérations ne sont exécutables que lorsque le système est dans un certain état, état obtenu par l'exécution d'une ou plusieurs autres opérations. D'où l'intérêt de disposer d'un préambule pour pouvoir exécuter certaines opérations. Avec une séquence

de taille 20, il y a plus de chances d'atteindre l'état requis qu'avec une séquence de taille 10.

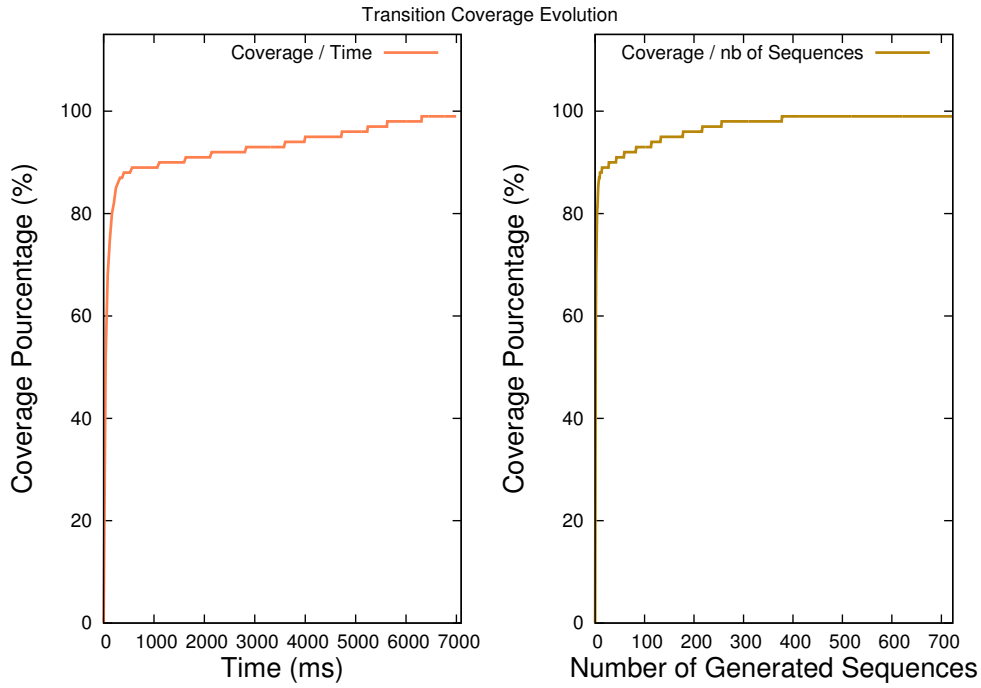

FIGURE 5.2 – Génération aléatoire, séquences de 20 pas, aucune optimisation

La dernière expérimentation de l'algorithme, dont les résultats sont présentés en figure 5.3, est effectuée une nouvelle fois avec des séquences de 20 opérations ($seqSize = 20$), mais en intégrant cette fois les optimisations présentées en section 5.1.3. Celles-ci permettent de garder en mémoire un historique des combinaisons de paramètres utilisés pour valider la précondition d'une opération, afin de ne pas tester une même combinaison plusieurs fois, et de garder en mémoire la liste des opérations qui ne se sont pas exécutables compte tenu de l'état du modèle, pour les retirer des opérations potentiellement sélectionnables par le tirage aléatoire. On constate la même baisse d'efficacité à 80%, bien que le temps d'exécution ait été divisé de moitié (3.000 millisecondes contre 7.000 et 10.000 précédemment). Le nombre de séquences nécessaires à la couverture totale des arcs a considérablement augmenté (environ 1100 séquences contre 400 sans optimisations). D'après ces résultats, on peut conclure que les optimisations mises en place permettent de réduire le nombre d'échecs de tentatives de validation des opérations, réduisant de ce fait le temps nécessaire pour construire une séquence, et donc le temps général d'exécution. Pourtant, ces optimisations semblent avoir pour effet de privilégier le tirages d'opérations faciles, i.e. des opérations peu contraintes, au détriment des opérations qui nécessitent un préambule précis pour pouvoir être exécutée.

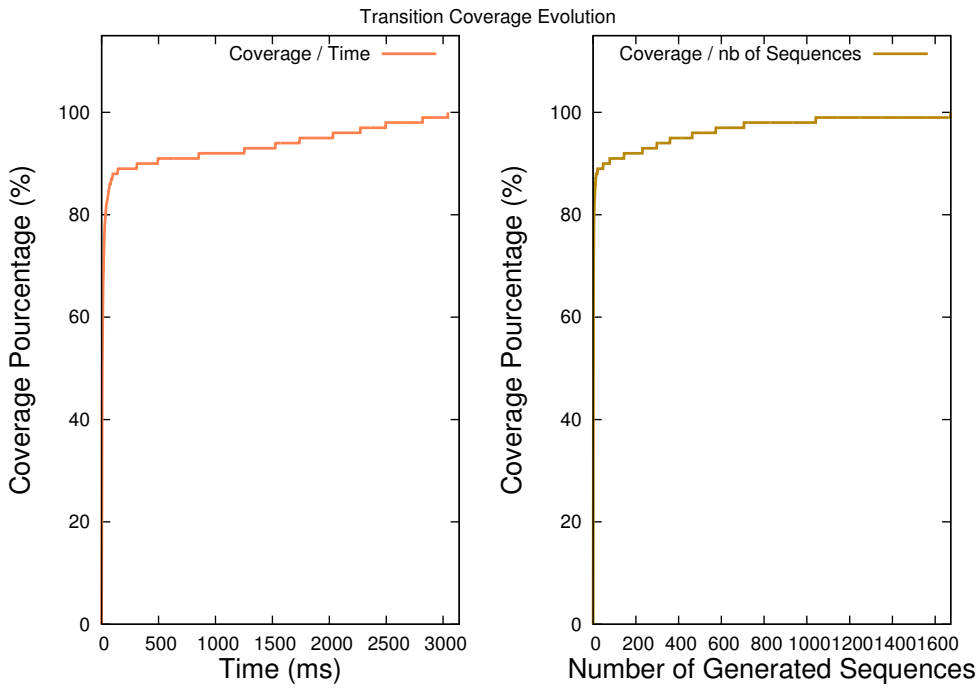

FIGURE 5.3 – Génération aléatoire optimisée, séquences de 20 pas

On le voit, même si cet algorithme est très efficace pour couvrir jusqu'à 80-85% des arcs, un grand nombre de séquences est nécessaire pour atteindre la couverture totale. Ceci s'explique par l'absence d'optimisation liée aux besoins de l'algorithme à un instant donné. Par exemple, lorsque tous les arcs restants à couvrir se trouvent dans la même opération, les efforts devraient être concentrés dans l'exécution de cette opération, c'est-a-dire trouver l'état du système et la combinaison de paramètres nécessaires à la validation de sa précondition.

Cependant, l'objectif de validation du composant d'animation est d'ores et déjà atteint. Nous sommes en mesure de produire une suite de test générée à partir des modules développées pour la création et l'animation de modèles. Dans la section suivante, nous proposons une première version de l'algorithme génétique dans le but d'améliorer la convergence de l'algorithme aléatoire.

5.2 Algorithme génétique

Le seconde stratégie mise en place consiste en un algorithme évolutionnaire de type génétique. L'objectif est d'être plus efficace que la génération aléatoire lorsqu'il ne reste que 15 à 20% des arcs non-couverts. Comme vu en section 5.1.4, l'algorithme aléatoire parvient à couvrir 80% des arcs avec relativement peu de séquences générées (1 à 10 séquences sont nécessaires) , mais atteindre la couverture totale est plus laborieuse (400 à 1400 séquences selon le paramétrage de l'algorithme). Ainsi, cet algorithme se base sur la génération aléatoire en définissant une population initiale de séquences de test couvrant la majeur partie des arcs, pour ensuite prendre la relève et exécuter les opérateurs génétiques (croisement, mutation, etc...), de manière a créer de nouvelles séquences et potentiellement accroître le taux de couverture.

A noter que dans une approche génétique classique, la solution à trouver est un individu, et c'est l'individu avec la meilleure évaluation de fitness qui est retourné. La version développée pour la solution UML-Alf-TST n'a pas pour but de sélectionner le meilleur individu, mais plutôt d'avoir une bonne population, à savoir une population dont l'ensemble des individus permettent d'atteindre le but fixé, à savoir la couverture de toutes les arcs. La section suivante présente plus en détails la représentation du problème.

5.2.1 Représentation génétique d'une suite de tests

Un algorithme d'ordre génétique, pour fonctionner, nécessite une population d'individus. Chaque individu peut être composé de gènes, et chaque gène peut être composé d'allèles (tout dépend la complexité du problème à résoudre). Les opérateurs génétiques interviennent ensuite au niveau des individus et de leurs gènes, afin d'altérer leur contenu ou de l'utiliser pour créer d'autres individus.

Dans un contexte de génération automatique de tests, la représentation génétique du problème a résoudre, une suite de tests qui couvre la totalité des arcs, est ainsi :

- **Population** : C'est une suite de test. Elle contient une liste de séquences de test.
- **Individus** : un individu est une séquence de test, composée d'appels d'opérations.
- **Gènes** : C'est un appel d'opération, potentiellement accompagné de paramètres.
- **Allèles** : Les allèles ne sont pas toujours présents. Ce sont les éventuels paramètres d'une opération.

Il faut aussi pouvoir évaluer les individus entre eux, en définissant ce que l'on appelle une fonction fitness. Lors d'une utilisation classique de l'algorithme génétique, la fonction fitness permet de savoir à quel point un individu s'approche de la solution attendue. Pour la solution UML-Alf-TST, la fonction fitness est associée au taux de couverture indépendant de chaque séquence. Le taux de couverture indépendant représente le pourcentage de transitions que couvre une séquence face à la somme des transitions de chaque opération du SUT. Dans la pratique, plus une séquence à un taux de couverture indépendant important, plus l'individu représentant cette séquence est considéré comme fort vis à vis de l'ensemble de la population.

Enfin, l'algorithme doit être capable de faire le tri entre les individus pour ne garder que les meilleurs, ceux dont la valeur fitness s'approche le plus de la valeur objectif, de manière à converger vers la solution optimale. C'est ce qu'on appelle le processus de sélection naturelle. La version développée pour ce mémoire procède en deux étapes : les séquences sont d'abord ordonnées selon leur valeur fitness (taux de couverture indépendant), puis les séquences les plus faibles sont supprimées, jusqu'à ce que la population atteigne la taille fixée par le paramètre .

5.2.2 Paramétrabilité et données d'entrée

Comme pour la génération aléatoire, certains aspects de l'algorithme génétique sont rendus paramétrables. Il est d'abord nécessaire de borner l'algorithme en temps d'exécution et en nombre de générations. L'algorithme attend ces valeurs en entrée pour fonctionner normalement.

Algorithme 5.4 Algorithme génétique pour la génération de Tests

Entrées:
entier *maxTime* ⇒ temps maximal d'éxecution
entier *maxGenNbr* ⇒ Nombre maximum de générations
entier *popSize* ⇒ Taille de la population
entier *initTCov* ⇒ Taux de couverture à atteindre par la population initiale
Locales:
algorithme *randomGen* ⇒ algorithme de génération
entier *tCov* ⇒ taux de couverture courant
entier *elapsedTime* ⇒ Temps écoulé
entier *nbGen* ⇒ Nombre de générations
TestSuite *initialPop* ⇒ Population issue de l'algorithme aléatoire
TestSuite *newPop* ⇒ Population ayant subi un cycle d'évolution
Début
2: $initialPop \leftarrow randomGen.generateTestSuite(popSize, initTCov)$
$tCov \leftarrow evaluer(initialPop)$
4: $nbGen \leftarrow 1$
TantQue $tCov < 100$ **et** $elapsedTime < maxTime$ **et** $nbGen < maxGenNbr$ **Faire**
6: "exécuter opérateurs génétiques"
$Cov \leftarrow evaluer(newPop)$
8: $newPop \leftarrow SelectionNaturelle(newPop)$
$nbSeq \leftarrow newPop.length$
10: $initialPop \leftarrow newPop$
$nbGen++$
12: **FinTantQue**
Retourne *initialPop*
14: **Fin**

De plus, un taux de couverture minimal que la population initiale doit atteindre, ainsi que sa taille, peuvent être renseignés. Ainsi, la génération aléatoire de la population initiale ne cesse que lorsque le taux et la taille de la population sont tous deux atteints.

Si aucune donnée n'est fournie en entrée, des valeurs par défaut sont affectées aux paramètres.

5.2.3 Déroulement de l'algorithme

Le mécanisme génétique est présenté via l'algorithme 5.4. D'abord, la population initiale est obtenue à la ligne 2 en utilisant l'algorithme de génération aléatoire. Le taux de couverture de cette première génération est évalué à la ligne 3, et à la ligne 4 le nombre de génération est initialisé à 1 (il n'y a qu'une génération).

Le cycle de générations est représenté à la ligne 5 par une boucle "TantQue", à chaque itération dans cette boucle est associée la création d'une nouvelle génération de la population. Pour sortir de cette boucle, il faut soit avoir couvert tous les arcs ($tCov = 100$) ou avoir dépassé le nombre de génération permis ($nbGen = MAXNBGEN$).

Si ces conditions ne sont pas réunies, on entre dans le corps de la boucle. Les opérateurs génétiques sont appliqués à la population à la ligne 6 (des informations quant à ces opérateurs sont données dans la section suivante). Le taux de couverture de la nouvelle

population est ensuite évalué (ligne 7), et à la ligne 8 la nouvelle population est soumise au phénomène de sélection naturelle. Les individus issus forment la nouvelle population, qui est affectée à la population courante à la ligne 9. Le nombre de génération ainsi que la taille de la population courante sont mis à jour aux lignes 10-11.

Lorsque l'on sort de la boucle, tous les arcs sont couvertes ou le nombre de génération maximal a été atteint. Dans tous les cas, la population obtenue est retournée (ligne 13).

5.2.4 Opérateurs génétiques

Le principe même de l'algorithme génétique est de faire évoluer la population initiale, dans le but de garder les individus dont le bagage génétique est le plus favorable, dont l'évaluation par la fonction fitness est la meilleure. Cette évolution est assurée par les opérateurs génétiques. Ce sont des procédés qui permettent de modifier les individus courants ou d'en créer de nouveau. Dans l'algorithme génétique basique, deux opérateurs font évoluer la population : le croisement et la mutation. Pour la génération automatique de tests, les opérateurs *Insertion*, *Ajout* et *Suppression* ont été intégrés, issus des travaux de Tonella [83]. Ils permettent entres autres d'ajouter du nouveau matériel génétique, dans le but d'introduire des appels d'opérations qui n'apparaissent dans aucune séquence, et couvrir des arcs jusqu'alors inactivables.

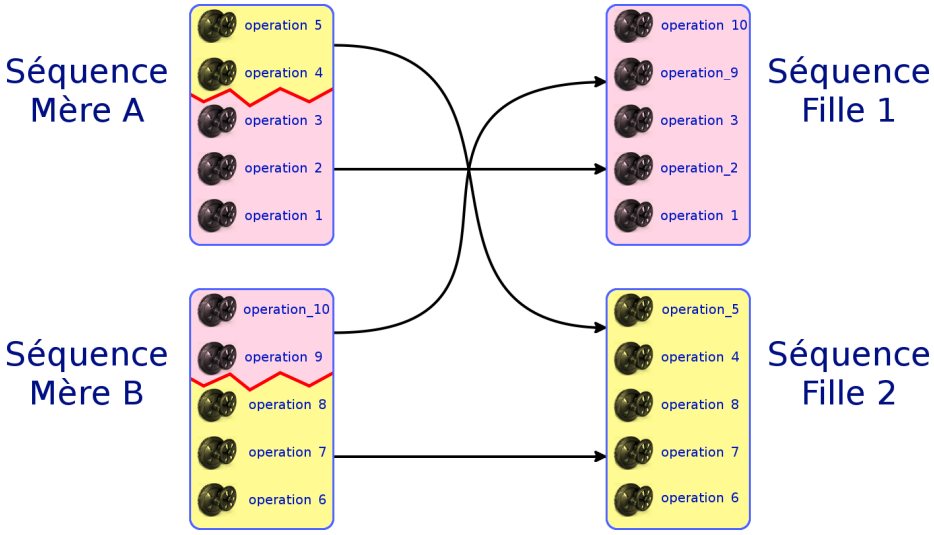

FIGURE 5.4 – Croisement de deux séquences de test

Croisement Comme vu en section (*pointer vers Etat de l'art*), il existe de nombreuses versions du mécanisme de croisement. Celle implémentée pour ce mémoire nécessite deux individus, deux séquences de test "mères" qu'on nommera M1 et M2. Un point de croisement est déterminé de manière aléatoire, qui correspond à un indice dans une séquence. De même, deux séquences de test "filles", qu'on appellera F1 et F2 , sont initialisées. Ensuite, le contenu des séquences mères est parcouru, et tant que l'indice des appels d'opération courants sont inférieurs au point de croisement, l'appel d'opération de la séquence mère M1 est transféré à la séquence fille F1, et l'appel d'opération de la séquence mère M2 est transféré à la séquence fille F2. Lorsque le point de croisement est atteint, les couples

mère-fille sont inversés, et le contenu de M2 est transféré à F1, et le contenu de M1 est transféré à F2.

Les deux nouvelles séquences sont alors exécutées avec l'animateur, et si la précondition de chaque opération est évaluée positivement, ces deux séquences sont intégrées à la nouvelle population.

Mutation Le procédé de mutation consiste à altérer le bagage génétique d'un individu, en sélectionnant un gène pour le faire muter. Pour une séquence de test, cela se traduit par la sélection aléatoire d'un appel d'opération, et par le remplacement de cet appel par un autre, tiré aléatoirement.

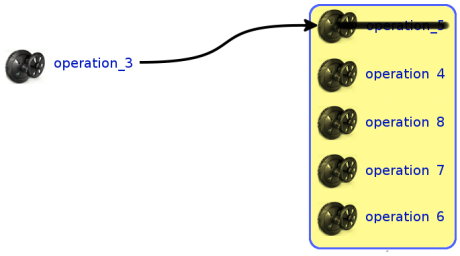

FIGURE 5.5 – Mutation d'une séquences de test

Généralement, la mutation ne créé pas d'individu, mais modifie celui sélectionné. Dans notre cas, il a été préféré de conserver la séquence de test intacte, et de créer un mutant de celle-ci. En effet, la modification d'une séquence entraine sa réévaluation, et si la précondition de l'appel d'opération mutant n'est pas validée, la séquence est alors supprimée. Le risque ici, en plus de réduire la taille de la population, est de détruire de bons individus selon leur valeur fitness, ayant pour effet de ralentir l'algorithme, voir de l'immobiliser, plutôt que de l'aider à converger vers la solution.

Les opérateurs génétiques représentés par les figures 5.6, 5.7 et 5.8 sont issus des travaux de Tonella [83]. Ils permettent d'accroître la capacité de l'algorithme à introduire du matériel génétique nouveau. En effet, notre objectif est de couvrir chaque transition de chaque opération. Pour pouvoir être appelée, une opération nécessite que l'état du système soit dans un état qui permette de vérifier sa précondition. Il est probable que la population de départ, obtenue aléatoirement, ne fasse jamais appel à cette opération, il faut donc pouvoir l'introduire par la suite.

Insertion L'opérateur d'insertion, comme son nom l'indique, permet d'augmenter le contenu d'une séquence en insérant un appel d'opération supplémentaire, quelque part dans cette séquence. Le schéma de la figure 5.6 donne une vue général du procédé. A l'instar de la mutation, un point d'insertion (un indice dans la liste d'appels) est déterminé aléatoirement. De même, l'appel d'opération à insérer est tiré de manière aléatoire, et est inséré au point d'insertion. Il n'est pas question ici de supprimer une partie du code génétique, seulement d'en ajouter.

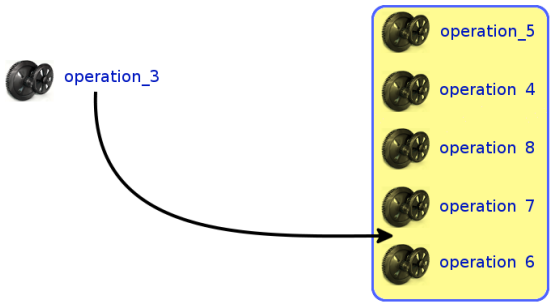

FIGURE 5.6 – Operateur génétique d'insertion

Ajout Le mécanisme d'ajout est en réalité un cas particulier de l'opérateur d'insertion. Il consiste à augmenter le contenu d'une séquence en insérant un appel d'opération. Cette action est consultable grâce au schéma de la figure 5.7. Seulement ici, l'appel d'opération qui est déterminé aléatoirement ne peut être ajouté qu'en fin de séquence. Il s'agit d'une concaténation. Il est important de distinguer l'ajout de l'insertion, leur but est différent : l'insertion cherche à appliquer un changement radical dans l'état du système en insérant un appel d'opération dans une séquence (tout en restant moins intrusif qu'une mutation). Chaque appel d'opération qui suit est alors altéré. L'ajout permet de faire évoluer l'état tel qu'il est à la fin de la séquence. Ce mécanisme peut s'avérer utile pour exécuter des opérations qui dépendent d'autres.

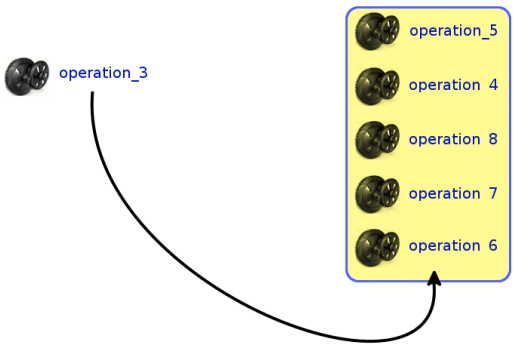

FIGURE 5.7 – Operateur génétique d'ajout

Suppression Un dernier opérateur vient compléter la liste, un processus de suppression visible via la figure 5.8, dont le rôle est, pour une séquence tirée, de supprimer un appel d'opération sélectionné aléatoirement. Le fait de simplement supprimer possède un avantage à la mutation, puisque les changements apportés à la séquence sont moins importants, il y a ainsi davantage de chance que la séquence soit toujours valide après suppression, tout en permettant un changement d'état.

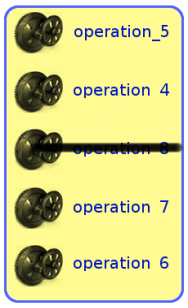

FIGURE 5.8 – Operateur génétique de suppression

Une optimisation des opérateurs de mutation, d'insertion et d'ajout consiste à orienter la sélection des opérations à introduire dans une séquence, en autorisant seulement les opérations dont il reste au moins une transition non-couverte. Il ne s'agit pas de remplacer les opérateurs par leur version optimisée, mais d'utiliser les variantes pour maximiser les résultats. Ainsi, la version basique des opérateurs est dédiée à l'introduction d'une opération quelconque, pour potentiellement changer l'état du système. Le but est d'atteindre un état permettant l'exécution d'une opération pas ou partiellement couverte. La version optimisée des opérateurs a pour rôle d'introduire une opération dont un ou plusieurs arcs ne sont pas couverts.

5.2.5 Expérimentations avec ECinema

Pour observer si cette seconde stratégie a permis de combler les faiblesses de la stratégie aléatoire, nous avons aussi appliqué l'algorithme génétique au modèle ECinema, présenté en section 2.4. Pour chacune des expérimentations réalisées, l'algorithme est paramétré comme suit :

- *initTCov* (taux de couverture objectif) : 80%
- *popSize* (taille de la population) : 30 individus
- *maxTime* (temps maximal d'exécution) : 30.000 ms
- *maxGenNbr* (Nombre total de générations) : 300 générations

A l'instar de l'algorithme de génération aléatoire, les valeurs fixe des paramètres ci-dessus ont été obtenues après une première phase d'expérimentation, non présentée dans ce document. Les valeurs définies sont directement liées au modèle utilisé pour les expérimentations, et peuvent varier d'un modèle à un autre.

La première expérimentation a pour objectif d'évaluer l'efficacité des opérateurs génétiques basiques, le croisement et la mutation. Les opérateurs que nous avons introduits ainsi que les optimisations ne sont donc pas utilisés. Les résultats présentés en figure 5.9 traduisent une inefficacité de l'algorithme génétique dans sa version basique. Dès lors que le processus génétique prend le pas sur la génération aléatoire, on ne constate qu'une infime progression du taux de couverture (environ 2%). Le nombre maximal de 300 générations est atteint avant même de pouvoir couvrir tous les arcs. En effet, les opérateurs génétiques basiques ne permettent pas d'introduire suffisamment de matériel génétique

nouveau, i.e. des opérations dont aucune séquence ne fait appel. Un croisement ne permet que de changer le préambule, puisque une portion d'une séquence est échangée avec une portion d'une autre. Une mutation peut potentiellement intégrer une nouvelle opération, mais la manière de procéder (remplacer une opération existante par une nouvelle) est peu efficace, parce qu'elle échoue trop fréquemment : près de 75% des opérations introduites par mutation ne sont pas exécutables, compte tenu du préambule que forment les opérations les précédant au sein d'une séquence.

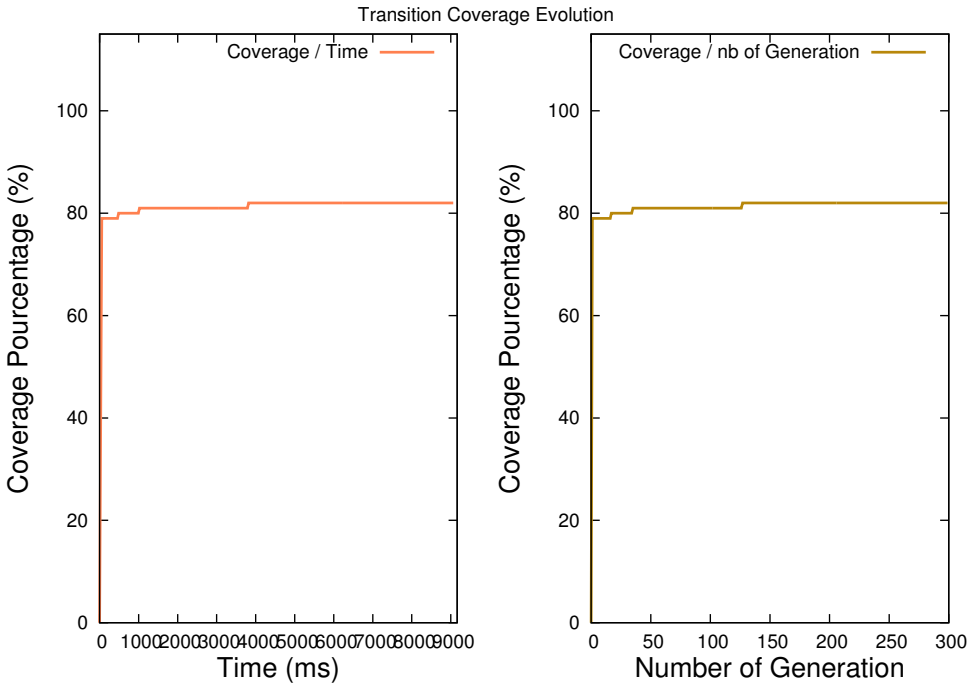

FIGURE 5.9 – Algorithme génétique, opérateurs basiques, aucune optimisation

La seconde expérimentation, dont les résultats sont exposés en figure 5.10, a consisté à exécuter l'algorithme génétique, cette fois avec tous les opérateurs à disposition. Nous avons ainsi introduit les opérateurs d'insertion, d'ajout, et de suppression. Ils ont deux objectifs : modifier davantage le préambule des séquences, pour potentiellement changer le chemin d'exécution des opérations appelées après le préambule, et intégrer des opérations dont la fréquence d'appel est insuffisante pour couvrir tous leurs arcs. On constate cependant que l'introduction des trois opérateurs n'a que très peu influencé les résultats. Comme pour la première expérience, lorsque le processus génétique prend le pas sur la génération aléatoire, pas ou très peu de progression est observée concernant le taux de couverture, et l'algorithme termine cette fois parce le temps maximal d'exécution est atteint. D'après ces résultats, on peut en déduire que le problème ne se situe pas au niveau de la modification de préambule, puisque de nombreuses séquences sont issues des cinq opérateurs (selon les opérateurs, entre 20% et 50% des modifications créé des séquences valides) , mais au niveau de l'introduction de matériel génétique nouveau. Malgré la mise en place des trois opérateurs supplémentaires censés remplir ce rôle, ce n'est toujours pas suffisant.

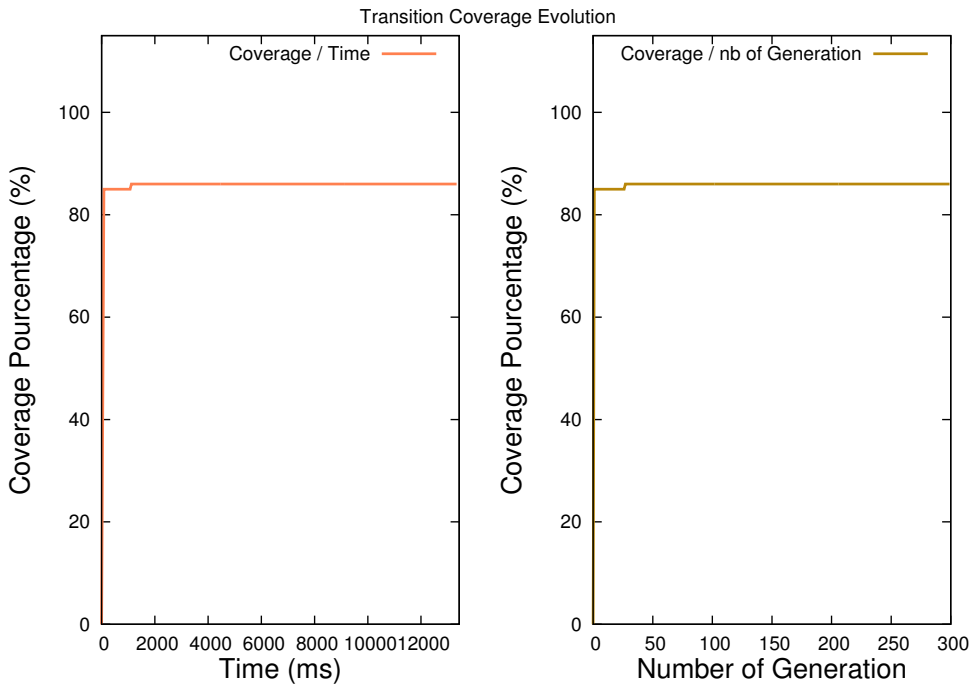

FIGURE 5.10 – Algorithme génétique, tous les opérateurs, aucune optimisation

Dans la dernière expérimentation, nous avons intégré les cinq opérateurs génétiques, ainsi que leur version optimisée. Ces optimisations ont pour objectif d'insister sur l'introduction d'opérations dont un ou plusieurs arcs ne sont pas couverts. Pour cela, seules ces opérations sont disponibles pour les opérateurs de mutation, d'insertion et d'ajout, pour accroître les chances d'augmentation du taux de couverture. Les résultats issus de cette expérimentation, exprimés en figure 5.11, traduisent une légère amélioration de l'efficacité de l'algorithme. Cette fois encore, c'est le temps maximal d'exécution qui a été atteint et non la couverture totale des arcs, mais on observe une légère évolution du taux de couverture pendant les 40 premières générations, jusqu'à atteindre en moyenne près de 90% de arcs couverts. On peut en conclure que les optimisations mises en place sont nécessaires pour rendre exécutables certaines opérations ou atteindre certains chemins d'exécution d'une opération. En revanche, ces optimisations ne sont toujours pas suffisantes pour couvrir 100% des arcs. Cela peut s'expliquer par le fait que certains comportements nécessitent un préambule très spécifique pour être exécutés. Actuellement, les efforts sont concentrés sur la construction d'une séquence, et devraient davantage être investis sur l'activation d'un comportement donné. Cela signifie qu'il serait nécessaire de guider la construction de préambule, qui est actuellement aléatoire, et de l'orienter dans l'objectif d'activer le-dit comportement.

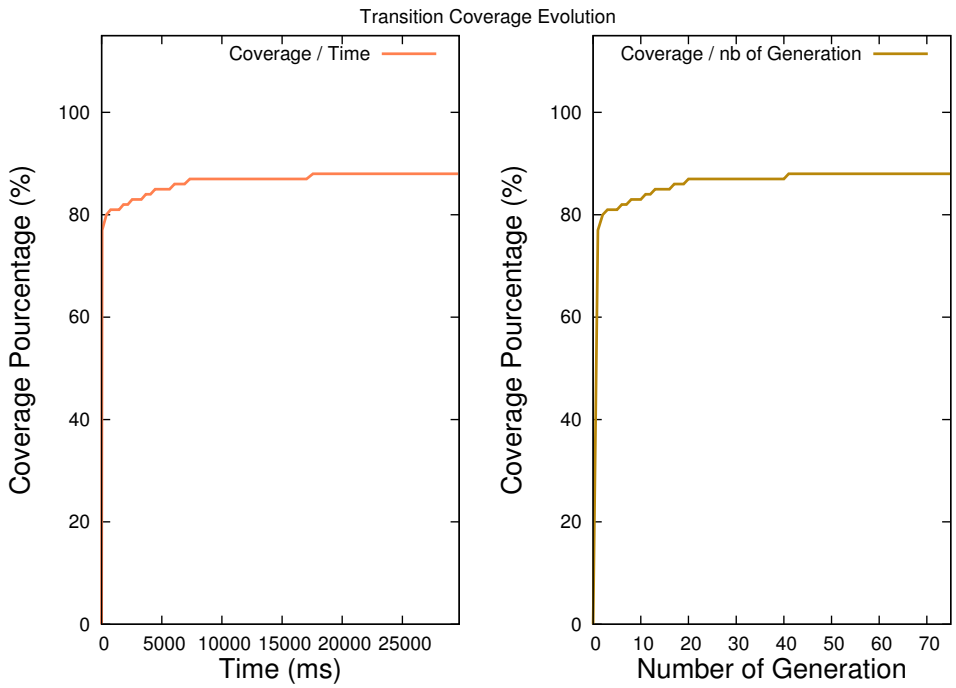

FIGURE 5.11 – Algorithme génétique, tous les opérateurs, avec optimisations

Les expérimentations le montrent, la version actuelle de l'algorithme génétique proposé dans ce document ne permet pas d'obtenir une suite de test qui couvre le critère de couverture tous-les-arcs. On peut en déduire que notre implémentation particulière de l'algorithme génétique, où l'objectif est de faire converger les individus (les séquences de test) vers un ensemble optimal (un ensemble de séquences qui permette de couvrir tous les arcs), est sans doute inadaptée. Nous avons pourtant décidé de procéder ainsi après avoir constaté qu'implémenter une version plus conventionnelle de l'algorithme ne nous permettrait pas d'améliorer les performances de la génération aléatoire. En effet, une version conventionnelle, où chaque individu est une potentielle solution, nécessite que les individus soient représentés non plus par une séquence de test, mais par une suite de test. Cela implique, pour l'initialisation de la population initiale, de générer non plus une mais un ensemble de suite de test. Cette action représente un coût en temps très important, à tel point qu'il est plus rapide d'obtenir un résultat avec l'algorithme de génération aléatoire que de générer une population initiale destinée à l'algorithme génétique. Cependant, même si notre implémentation semble meilleure, nous n'avons pas obtenu les résultats escomptés. Il pourrait être alors intéressant de modifier la représentation génétique établie en section 5.2.1, ou de trouver une fonction fitness plus adaptée. Cette approche naïve de l'algorithme génétique a établi un premier contact avec les algorithmes évolutionnaires. Des travaux futurs pourront viser à améliorer cette approche, en se basant davantage sur les fonctions de fitness utilisées dans la littérature. Nous n'avons pu utiliser ces techniques dans cette première approche principalement par manque de temps.

5.3 Synthèse

Pour la solution UML-Alf-TST, nous avons mis en place deux stratégies de génération automatique de tests.

La première stratégie est un algorithme de génération dite “aléatoire”, c'est-à-dire que le contenu des séquences est déterminé aléatoirement. En effet, lors de la création d'une séquence de test, constituée d'une suite d'appels d'opération, chaque opération à appeler est tirée dans la liste des opérations disponibles du SUT. Nous utilisons le module d'animation vu en section 4.2 pour exécuter l'opération et pour vérifier si sa précondition est satisfaite, compte tenu de l'état du système après exécution des opérations précédant celle-ci dans la séquence. La génération s'arrête lorsque le critère de couverture tous-les-arcs est atteint. De plus, l'algorithme est borné en temps d'exécution et en nombre de séquences. Pour accroître l'efficacité de l'algorithme et ajouter du contrôle sur le mécanisme de sélection aléatoire, des optimisations ont été mises en place. Elles consistent à garder en mémoire la liste des opérations sélectionnées consécutivement dont l'exécution a échoué, ainsi que la liste des combinaisons de paramètres avec lesquelles l'évaluation de l'opération concernée a été appelée. Le but de ces optimisations est d'empêcher l'algorithme de tirer plusieurs fois une même combinaison de paramètres ou une même opération, de manière à réduire le nombre d'échecs et accroître la convergence de l'algorithme. Les expérimentations menées pour mesurer l'efficacité de cette stratégie ont produit des résultats satisfaisant : l'algorithme parvient à couvrir le critère tous-les-arcs, et ce relativement rapidement. Pourtant, on note une baisse d'efficacité à mesure que le nombre d'arcs restants diminue, qui traduit une faiblesse connue de l'aléatoire lorsqu'il s'agit d'obtenir une configuration, un préambule d'appels d'opération précis, pour pouvoir activer un chemin d'exécution spécifique d'une opération.

La seconde stratégie a été élaborée dans l'objectif de pallier la baisse progressive d'efficacité de l'algorithme de génération aléatoire. Le principe consiste à implémenter une version de l'algorithme génétique vu en section 3.3.2 adaptée à la génération automatique de tests. La population est constituée d'individus représentés par des séquences de test, leur gènes sont caractérisés par des appels d'opération, et les allèles d'un gêne sont les éventuels paramètres d'une opération. La formation d'une population initiale est assurée par l'algorithme de génération aléatoire. Débute alors le processus d'évolution génétique : des opérateurs de mutation et de croisement permettent d'altérer les séquences obtenues aléatoirement, de façon à converger vers la solution optimale, une suite de test qui permette de couvrir le critère de couverture tous-les-arcs. Pour cela, nous avons défini une fonction fitness qui a pour rôle d'évaluer la qualité d'une séquence. Cette fonction est basée sur le taux de couverture indépendant, et plus ce taux est élevé pour une séquence, plus cette séquence est considérée comme un “bon individu”. D'une génération à l'autre, le mécanisme de sélection naturelle compare les séquences entre elles, et ne garde que celles qui ont la meilleure évaluation fitness. De plus, nous avons implémenté des opérateurs génétiques d'insertion, d'ajout et de suppression, issus des travaux de Tonella [83]. Ils permettent d'insister sur l'introduction de matériel génétique nouveau, nécessaire pour couvrir les arcs des opérations auxquelles l'ensemble des séquences ne fait pas ou peu appel. Toujours pour accroître la capacité d'introduction de matériel génétique nouveau, nous avons développé une version optimisée des opérateurs de mutation, ajout et suppression, qui consiste à n'introduire que des opérations dont un ou plusieurs arcs ne sont pas couverts. Malgré les techniques mises en place, les expérimentations effectuées ne montrent pas d'amélioration quant à la baisse d'efficacité de la génération aléatoire. Au contraire, l'algorithme génétique ne parvient que très difficilement à atteindre de nouveaux comportements. Aucune des expérimentations menées n'a permis de couvrir tous les arcs. Il est possible que l'inefficacité de l'algorithme soit issue de notre implémentation quelque peu particulière, qui consiste non pas à trouver la solution optimale parmi

une population, mais à considérer cette population comme la solution et à modifier cette solution pour la rendre optimale.

Même si la stratégie de génération aléatoire permet de fournir une suite de test couvrant le critère tous-les-arcs, nous ne sommes pas parvenu à développer les optimisations suffisantes pour pallier la baisse d'efficacité observée à partir de 80-85% d'arcs couverts. Cependant, ces résultats sont à nuancer : une seule étude de cas a été effectuée, et il est possible que les résultats constatés soient en réalité dus aux spécificités du modèle utilisé, ECinema. En effet, à chaque exécution, la même méthode de ECinema reste non-couverte. On peut supposer que ceci est du au fait que cette méthode est fortement contrainte. Elle est éventuellement couverte avec l'algorithme aléatoire qui réalise un très grand nombre d'essais en peu de temps, mais que l'algorithme génétique n'est pas assez rapide pour obtenir une solution dans des délais raisonnables.

Cependant, les objectifs fixés concernant la validation des modules d'animation et de modélisation ont été atteints. Nous avons permis la validation de la chaîne outillée par l'élaboration de stratégies de génération de tests, dont la première parvient à respecter le critère tous-les-arcs pour toutes les opérations du SUT. Les stratégies mises en place constituent un premier contact avec la nouvelle notation proposée dans ce mémoire, et l'établissement de stratégies de génération de tests plus efficaces fera l'objet de travaux futurs.

Chapitre 6

Conclusion et perspectives

Contents

Dans ce chapitre, nous présentons un bilan des travaux réalisés, et nous proposons des pistes ouvertes pour une éventuelle poursuite des recherches.

6.1 Conclusion

La solution que nous venons de présenter s'inscrit dans les travaux menés au sein du DISC, département informatique de FEMTO-ST, concernant la génération automatique de tests à partir de modèles. Nous proposons une combinaison de notations originale destinée à l'élaboration de modèles orientés pour le test, constituée d'UML, OCL, et Alf. Cette approche répond à des besoins liés à l'animation de modèles spécifiés avec le langage UML, grâce à l'introduction du langage Alf. D'une part, il offre une solution adaptée pour spécifier le comportement d'opérations de classe et l'effet de transitions d'un statechart. Ceci a pour objectif de représenter les effets du déclenchement des fonctionnalités du SUT sur son état, et par conséquent autorise l'animation de modèles. D'autre part sa syntaxe, très similaire à celle du langage Java, permet de décliner des tests de type boîte blanche. Cela a pour intérêt d'augmenter la qualité et la précision générale des tests générés avec ce type de techniques. Notre contribution se décompose en trois parties : établissement des notations, état de l'art des critères de couverture et des approches pour satisfaire ces critères dans un contexte MBT, et choix des technologies avec expérimentations.

D'abord, nous avons défini les langages qui composent notre choix de notation. Premièrement, le langage UML est utilisé pour décrire la structure et le comportement général du SUT. Parce qu'il est très expressif mais surtout parce qu'il contient certaines ambiguités, nous avons proposé un sous-ensemble du langage et une sémantique associée. Ce sous-ensemble est composé des diagrammes de classes et d'objets, dont le but est de représenter la partie statique du système (propriétés, méthodes) et l'état initial du système, et du diagramme d'états-transitions dont le but est de spécifier le comportement du système. Deuxièmement, le langage OCL apporte des précisions à la modélisation par

l'ajout de contraintes. Typiquement, ces contraintes seront exprimées sur les opérations et les transitions pour empêcher le modèle d'être dans un état interdit. Troisièmement, le langage Alf accroît encore le niveau de précision du modèle en spécifiant le comportement des opérations et transitions, et ainsi définir l'évolution de l'état du modèle.

Ensuite, nous avons établi un état de l'art des principaux critères de couverture de code source et de modèles à transitions. Pour la couverture de code Alf, nous nous sommes concentrés sur trois types de critères : critères basés sur les décisions, critères basés sur les chemins, critères basés sur les données. Leur objectif est de guider la génération de test et de pouvoir l'arrêt du test. Pour comprendre l'utilisation de ces critères, nous avons observé l'existant concernant la génération automatique des tests à partir de la stratégie que nous avons choisie : utiliser l'algorithme génétique. A ce jour, aucun travaux ne proposent d'utiliser l'algorithme génétique pour générer des tests à partir de modèles. Cependant, UML est un langage de modélisation graphique pour représenter des systèmes orientés objet, il est possible de s'inspirer des travaux de recherche sur le génération de tests de programmes orientés objet.

En outre, nous avons fait le point sur l'utilisation de technologies existantes et sur l'implémentation d'outils manquants pour mettre au point une chaîne outillée permettant la génération automatique de séquence de test à partir de modèles, représentés à l'aide de notre notation. Concernant la modélisation, notre choix s'est porté sur un modeleur existant, Papyrus MDT. Certaines fonctionnalités pourraient être développées pour améliorer son ergonomie, mais l'outil est d'ores et déjà opérationnel. De plus, un greffon contenant un éditeur pour l'ajout d'action Alf aux opérations de classe est actuellement en développement. En revanche, aucun logiciel actuellement ne permet l'animation de modèles UML dont le fonctionnement est spécifié avec Alf. Nous avons alors développé un composant d'animation dédié à notre solution. Pour cela, il a été nécessaire de transformer le modèle UML/Alf vers un format moins complexe donc plus approprié à l'animation et à la génération de tests. Ainsi, les éléments UML sont transformés avec la technologie ATL vers leur équivalent UML4TST, un métamodèle proche de la sous-partie UML que nous utilisons (voir section 2.1), et dédié au test. Chaque action Alf est transformée en un graphe de flot de contrôle, en utilisant le framework ANTLR. Les graphes obtenus sont ensuite rattachés à l'opération à laquelle ils appartiennent. L'animation est effectuée en parcourant le graphe de flot de contrôle de l'opération à exécuter, et en interprétant le contenu de chacun de ses nœuds.

Enfin, nous avons mis en place deux stratégies de génération de tests, dont l'objectif est de produire une suite de test qui puisse satisfaire le critère tous-les-arcs pour chacune des opérations du SUT. La première stratégie est un algorithme de génération dite "aléatoire", c'est-à-dire que le contenu des séquences est déterminé aléatoirement. A l'aide du composant d'animation, chaque opération d'une séquence est exécutée sur le modèle, et celui-ci est mis à jour en conséquence. Des optimisations ont été mises en place pour guider la génération vers l'objectif de test. Elles consistent à garder en mémoire, lors du tirage d'une opération, la liste des opérations et des combinaisons de paramètres sélectionnées qui n'ont pas pu être intégrées à la séquence en cours. Le but est de ne pas sélectionner ces valeurs une seconde fois, de manière à converger plus rapidement vers la bonne solution. La seconde stratégie a été élaborée dans l'objectif d'optimiser l'algorithme de génération aléatoire. Le principe consiste à implémenter une version de l'algorithme génétique adaptée à la génération automatique de tests. La population est constituée d'individus représentés par des séquences de test, leur gènes sont caractérisés par des appels d'opération, et les allèles d'un gêne sont les éventuels paramètres d'une

opération. La formation d'une population initiale de séquences de test est assurée par l'algorithme de génération aléatoire. Pour obtenir une suite de séquences de test qui puisse couvrir tous les arcs, des opérateurs génétique de mutation et de croisement sont appliqués aux séquences de manière à altérer leur contenu, pour éventuellement couvrir de nouveaux arcs. L'évaluation de la population est possible grâce à une fonction de fitness, qui compare le taux de couverture d'arcs indépendant de chaque séquence. Des opérateurs génétiques d'ajout, d'insertion et de suppression issus des travaux de Tonella [83] ont été intégrés, pour accroître les chances d'intégrer de nouvelles opérations dans les séquences.

Pour illustrer notre travail, nous l'avons appliqué à un modèle d'application web d'achat de billets de cinéma, nommé ECinema. Les résultats obtenus avec la stratégie de génération aléatoire sont satisfaisants : l'objectif de test, à savoir le respect du critère tous-les-arcs, est atteint et ce rapidement (entre trois et dix secondes). De plus, les optimisations développées permettent d'améliorer encore les résultats, en réduisant le temps de génération et le nombre de séquences nécessaires pour atteindre l'objectif de test. Cela dit, on peut observer une baisse d'efficacité à mesure que le nombre d'arcs à couvrir s'amenuit.

Nous avons développé la seconde stratégie pour pallier cette baisse. Cependant, cette stratégie nécessite d'être améliorée compte-tenu des résultats obtenus avec le modèle ECinema. En effet, nous ne sommes pas parvenus à couvrir le critère tous-les-arcs avec cette technique.

Les résultats obtenus sont à mettre en perspective. Nous n'avons pu illustrer les stratégies mises en place qu'avec un seul cas d'étude. Il est envisageable que des spécificités du modèle utilisé soient responsables du manque d'efficacité de l'algorithme génétique. Une étude pratique approfondie est nécessaire pour pouvoir donner un verdict quant à l'efficacité réelle de l'algorithme génétique implémenté dans ce mémoire.

Finalement, les travaux présentés dans ce mémoire ont permis de répondre partiellement aux problématiques liées au développement d'une solution de test originale. Une chaîne outillée pour la modélisation et l'animation de modèles UML/Alf a été développée, et nous avons validé l'utilisation de cette chaîne dans un contexte de test en implémentant une stratégie de génération de test de type aléatoire. Nous avons aussi cherché à optimiser cette stratégie en implémentant une première version de l'algorithme génétique. Cette première tentative ne s'est pas avérée efficace, aussi cette solution doit être améliorée.

6.2 Perspectives

Bien qu'opérationnelle, la solution présentée dans ce mémoire est encore aujourd'hui au stade de développement. De nombreuses pistes d'améliorations sont envisageables concernant notre solution. On peut regrouper ces pistes selon deux grands axes :

Le premier axe concerne l'extension des capacités fonctionnelles de notre solution.

Actuellement, les tests générés par les stratégies de génération mises en place sont de nature abstraite. Cela signifie qu'ils ne peuvent être exécutés sur le système réel. La phase de concrétisation des tests est pour l'instant manuelle. Pour compléter la chaîne outillée, il s'agirait de développer un module dont le rôle est la concrétisation automatique des tests abstraits. Les outils existants pour la génération de tests proposent différentes méthodes. Une méthode répandue consiste à concrétiser les tests abstraits générés avec

des tests concrets de type XUnit (JUnit [11], CUnit [**?**], etc...), jouables directement sur le système. Cette méthode est notamment proposée par l'outil CertifyIt [5] de Smartesting.

Du point de vue notation, il est nécessaire de réintégrer les éléments qui ont été supprimés. Il s'agirait donc de réintégrer le diagramme d'états-transitions du langage UML pour modéliser l'aspect dynamique du système, et le langage OCL pour exprimer les préconditions des opérations et les gardes des transitions. Pour cela, il faut développer des éditeurs dédiés (en utilisant XText [22], EMFText [8], etc...), et dans le cas d'OCL, implémenter un évaluateur de contraintes.

Le second axe concerne l'amélioration des résultats de la génération de tests.

Une piste d'amélioration consiste à retravailler l'algorithme génétique, utilisé pour optimiser la génération aléatoire de séquences de test. Notre implémentation actuelle de l'algorithme génétique est encore naïve. En effet, la fonction de fitness utilisée s'est avérée inadaptée au problème que l'on souhaite résoudre, à savoir améliorer la convergence de la génération aléatoire lorsqu'il ne reste que 20% des arcs à couvrir. Une solution possible consiste à s'inspirer davantage des techniques d'évaluation de fitness existantes, présentées dans les travaux existants (voir section 3.3.2).

De facto, certains travaux sur le test de classe basés sur l'algorithme génétique utilisent un graphe de dépendances pour représenter le corps des méthodes. Cette représentation permet de guider la génération de test avec des critères de couverture des données, comme le critère toutes les définitions ou toutes les utilisations. S'inspirer de cette technique pourrait permettre la génération automatique de test selon ces critères.

La réintégration du diagramme d'états-transitions permettrait de considérer l'utilisation de critères de couverture de modèles à transitions, comme le critères toutes-les-transitions, ou toutes les paires de transitions. Ainsi, il est concevable de combiner l'utilisation de critères pour la couverture de code Alf (voir section 3.1) et l'utilisation de critères pour la couverture de modèle (voir section 3.2.2), pour générer une suite de test intervenant à des niveaux différents (voir niveaux de test en section 1.1).

Bibliographie

[1] Acceleo - transforming models into code. http://www.eclipse.org/acceleo/, 2012.

[2] Another tool for language recognition (antlr) v3. http://www.antlr.org/, 2012.

[3] Argouml 0.34. http://argouml.tigris.org/, December 2012.

[4] Cameo simulation toolkit. https://www.magicdraw.com/simulation, 2012.

[5] Certifyit. http://www.smartesting.com/index.php/cms/fr/product/certify-it, 2012.

[6] Complete alf parser. http://lib.modeldriven.org/MDLibrary/trunk/Applications/Alf-Reference-Implementation/dist/, 2012.

[7] Eclipse project. http://www.eclipse.org/, 2012.

[8] Emftext. http://www.emftext.org/index.php/EMFText, 2012.

[9] Java modeling language (jml). http://www.cs.ucf.edu/~leavens/JML/, 2012.

[10] Jet. http://www.eclipse.org/modeling/m2t/?project=jet, 2012.

[11] Junit. http://www.junit.org/, December 2012.

[12] miuml. http://www.miuml.org/download/, 2012.

[13] Papyrus mdt 0.9. http://www.eclipse.org/modeling/mdt/papyrus, January 2012.

[14] Papyrus mdt tutorial presentation. http://www.eclipse.org/modeling/mdt/papyrus/usersTutorials/resources/TutorialOnPapyrusUSE_d20101001.pdf, 2012.

[15] Papyrus uml 1.12. http://www.papyrusuml.org, 2012.

[16] Rational softwar architect. http://www-142.ibm.com/software/products/fr/fr/ratisoftarch/, 2012.

[17] Staruml. http://staruml.sourceforge.net/en/, 2012.

[18] State chart xml. http://www.w3.org/TR/scxml/, 2012.

[19] Topcased 5.1.0. http://www.topcased.org/, November 2012.

[20] Use 3.0. http://www.db.informatik.uni-bremen.de/projects/USE/, September 2012.

[21] Xpand. `http://wiki.eclipse.org/Xpand`, 2012.

[22] Xtext. `http://www.eclipse.org/Xtext/`, 2012.

[23] RTCA (Firm). SC 167 and European Organisation for Civil Aviation Equipment. WG-12. *Software Considerations in Airborne Systems and Equipment Certification*. RTCA, Inc., 1992.

[24] J.R. Abrial. *The B-book : assigning programs to meanings*. Cambridge University Press, 2005.

[25] A.V. Aho and J.D. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., 1972.

[26] P.E. Ammann, P.E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Formal Engineering Methods, 1998. Proceedings. Second International Conference on*, pages 46–54. IEEE, 1998.

[27] J.C.M. Baeten and W.P. Weijland. Process algebra, volume 18 of cambridge tracts in theoretical computer science, 1990.

[28] E. Behrends. *Introduction to Markov chains*, volume 228. Vieweg, 2000.

[29] B. Beizer. *Software testing techniques*. Dreamtech Press, 2002.

[30] Fabrice Bouquet, Christophe Grandpierre, Bruno Legeard, Fabien Peureux, Mark Utting, and Nicolas Vacelet. A subset of precise UML for model-based testing. In *A-MOST'07, 3rd int. Workshop on Advances in Model Based Testing*, pages 95–104, London, United Kingdom, July 2007. ACM Press. A-MOST'O7 is colocated with ISSTA 2007, Int. Symposium on Software Testing and Analysis.

[31] S. Cook and J. Daniels. *Designing object systems*. Prentice-Hall, 1994.

[32] Arnaud Cuccuru. Papyrus support for alf. `http://www.omg.org/news/meetings/tc/agendas/va/xUML_pdf/Cucurru.pdf`, 2012.

[33] J.B. Dabney and T.L. Harman. *Mastering simulink*. Pearson/Prentice Hall, 2004.

[34] RA DeMillo, RJ Lipton, and FG Sayward. Program mutation : A new approach to program testing. *Infotech State of the Art Report, Software Testing*, 2 :107–126, 1979.

[35] M.C. Gaudel and P. Le Gall. Testing data types implementations from algebraic specifications. *Formal methods and testing*, pages 209–239, 2008.

[36] P. Godefroid, N. Klarlund, and K. Sen. Dart : directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[37] D.E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-wesley, 1989.

[38] D.E. Goldberg and C.H. Kuo. Genetic algorithms in pipeline optimization. In *PSIG Annual Meeting*, 1985.

[39] J.B. Goodenough and S.L. Gerhart. Toward a theory of test data selection. *ACM SIGPLAN Notices*, 10(6) :493–510, 1975.

[40] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *ACM SIGSOFT Software Engineering Notes*, volume 23, pages 53–62. ACM, 1998.

[41] A. Gotlieb, B. Botella, and M. Rueher. A clp framework for computing structural test data. *Computational Logic—CL 2000*, pages 399–413, 2000.

[42] A. Gotlieb, B. Botella, and M. Watel. Inka : Ten years after the first ideas. In *19th International Conference on Software, Systems Engineering and their Applications (ICSSEA'06), Paris, France*, 2006.

[43] C. Grandpierre. *Stratégies de génération automatique de tests à partir de modèles comportementaux UML/OCL.* PhD thesis, LIFC - University of Franche-Comté, 2008.

[44] Mentor Graphics. Object action language. `http://www.mentor.com/products/sm/techpubs/object-action-language-reference-manual-38098`, 2012.

[45] M. Grindal, J. Offutt, and S.F. Andler. Combination testing strategies : a survey. *Software Testing, Verification and Reliability*, 15(3) :167–199, 2005.

[46] D. Harel. Statecharts : A visual formalism for complex systems. *Science of computer programming*, 8(3) :231–274, 1987.

[47] D. Harel and P. Thiagarajan. Message sequence charts. *UML for Real*, pages 77–105, 2004.

[48] M.J. Harrold and G. Rothermel. Performing data flow testing on classes. *ACM SIGSOFT Software Engineering Notes*, 19(5) :154–163, 1994.

[49] R.L. Haupt, S.E. Haupt, and J. Wiley. *Practical genetic algorithms.* Wiley Online Library, 2004.

[50] J.H. Holland. *Adaptation in natural and artificial systems.* Number 53. University of Michigan press, 1975.

[51] W.E. Howden. Reliability of the path analysis testing strategy. *Software Engineering, IEEE Transactions on*, (3) :208–215, 1976.

[52] Object Management Group inc. Action language for foundational uml (alf) 1.0 beta 2. `http://www.omg.org/spec/ALF/`, December 2012.

[53] Object Management Group inc. Foundational subset for executable uml models (fuml) 1.0. `http://www.omg.org/spec/FUML/`, February 2012.

[54] Object Management Group inc. Object constraint language (ocl) 2.3.1. `http://www.omg.org/spec/OCL/`, January 2012.

[55] Object Management Group inc. Unified modeling language (uml) 2.4.1. `http://www.omg.org/spec/UML/`, August 2012.

[56] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. Atl : A model transformation tool. *Science of Computer Programming*, 72(1) :31–39, 2008.

[57] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. Atl : a qvt-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720. ACM, 2006.

[58] N. Kosmatov, B. Legeard, F. Peureux, and M. Utting. Boundary coverage criteria for test generation from formal models. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 139–150. IEEE, 2004.

[59] M.K. Kwan. Graphic programming using odd or even points. *Chinese Math*, 1(273-277) :110, 1962.

[60] A. Lakehal, I. Parissis, and L. Du-Bousquet. Critères de couverture structurelle des programmes lustre. *Approches Formelles dans l'Assistance au Développement de Logiciels*, page 185, 2004.

[61] CODRUT-LUCIAN LAZAR. Integrating alf editor with eclipse uml. *STUDIA UNIV. BABES-BOLYAI, INFORMATICA*, LVI(3) :2–32, 2011.

[62] Model Integration LLC. Starr's concise relational action language. `http://www.modelint.com/downloads/mint.scrall.tn.1.pdf`, 2004.

[63] A. Mahdian, A.A. Andrews, and O.J. Pilskalns. Regression testing with uml software designs : A survey. *Journal of Software Maintenance and Evolution : Research and Practice*, 21(4) :253–286, 2009.

[64] T.J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4) :308–320, 1976.

[65] S. Mellor, S. Tockey, R. Arthaud, and P. Leblanc. An action language for uml : proposal for a precise execution semantics. *The Unified Modeling Language.«UML»'98 : Beyond the Notation*, pages 514–514, 1999.

[66] S.J. Mellor, M. Balcer, and I. Foreword By-Jacoboson. *Executable UML : A foundation for model-driven architectures*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[67] G.J. Myers, C. Sandler, and T. Badgett. *The art of software testing*. Wiley, 2011.

[68] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann. Generating test data from state-based specifications. *Software Testing, Verification and Reliability*, 13(1) :25–53, 2003.

[69] T.J. Ostrand and M.J. Balcer. The category-partition method for specifying and generating fuctional tests. *Communications of the ACM*, 31(6) :676–686, 1988.

[70] A. Paradkar. Towards model-based generation of self-priming and self-checking conformance tests for interactive systems. *Information and Software Technology*, 46(5) :315–322, 2004.

[71] R.P. Pargas, M.J. Harrold, and R.R. Peck. Test-data generation using genetic algorithms. *Software Testing Verification and Reliability*, 9(4) :263–282, 1999.

[72] D. Pilaud, N. Halbwachs, and JA Plaice. Lustre : A declarative language for programming synchronous systems. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY*, volume 178, page 188, 1987.

[73] E. Planas, J. Cabot, and C. Gómez. Lightweight verification of executable models. *Conceptual Modeling–ER 2011*, pages 467–475, 2011.

[74] S. Prehn. *VDM'91 : Tutorials*, volume 2. Springer, 1991.

[75] A. Pretschner and J. Philipps. 10 methodological issues in model-based testing. *Model-Based Testing of Reactive Systems*, pages 11–18, 2005.

[76] P. Racloz. Introduction au réseaux de petri. *Presses polytechniques et universitaires romandes*, pages 107–239, 1996.

[77] C. Raistrick. *Model driven architecture with executable UML*, volume 1. Cambridge Univ Pr, 2004.

[78] S. Rapps and E.J. Weyuker. Selecting software test data using data flow information. *Software Engineering, IEEE Transactions on*, (4) :367–375, 1985.

[79] S. Schneider. *The B-Method : an introduction*, volume 200. Palgrave, 2001.

[80] S. Shlaer. The shlaer-mellor method. *Project Technology White paper*, 1996.

[81] Pathfinder Solutions. Platform independant action language. `http://www.ooatool.com/docs/PAL04.pdf`, 2004.

[82] J.M. Spivey. *The Z notation : a reference manual*. Prentice Hall International (UK) Ltd., 1992.

[83] P. Tonella. Evolutionary testing of classes. *ACM SIGSOFT Software Engineering Notes*, 29(4) :119–128, 2004.

[84] M. Utting and B. Legeard. *Practical model-based testing : a tools approach*. Morgan Kaufmann, 2007.

[85] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing. 2006.

[86] S. Wappler and F. Lammermann. Using evolutionary algorithms for the unit testing of object-oriented software. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1053–1060. ACM, 2005.

[87] E.J. Weyuker and B. Jeng. Analyzing partition testing strategies. *Software Engineering, IEEE Transactions on*, 17(7) :703–711, 1991.

[88] J.A. Whittaker. Stochastic software testing. *Annals of Software Engineering*, 4(1) :115–131, 1997.

[89] M.R. Woodward, D. Hedley, and M.A. Hennell. Experience with path analysis and testing of programs. *Software Engineering, IEEE Transactions on*, (3) :278–286, 1980.

[90] H. Zhu, P.A.V. Hall, and J.H.R. May. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, 29(4) :366–427, 1997.

Résumé

Ce document présente la combinaison de trois langages (UML, OCL, et Alf) comme une solution pour la validation de systèmes à l'aide d'une approche Model-Based Testing. D'abord, nous établissons une description complète des langages ainsi que leur rôle au sein d'une modélisation orientée tests. Ensuite, nous faisons l'état de l'art des principaux critères de couverture, et l'utilisation des algorithmes de recherche pour la génération automatique de test. En outre, nous présentons la chaîne outillée que nous avons développé, qui permet la modélisation et l'animation de modèles décrits avec la notation UML/OCL/Alf. Enfin, nous validons l'utilisation de cette chaîne outillée dans un contexte MBT par l'implémentation d'une stratégie de génération de test aléatoire qui respecte le critère tous-les-arcs pour chacune des opérations du SUT. Nous proposons aussi une piste d'optimisation de cette stratégie par un premier contact avec l'algorithme génétique adapté à la génération de séquences de test.

Mots-clés Model-Based Testing, MBT, UML, OCL, Alf, génération automatique de tests, génétique

Abstract

This paper presents a combination of three languages (UML, OCL, and Alf) as a solution for validating systems using model-based testing techniques. First, we establish a full description of the three languages and define their role in an MBT context. Then, we do a background check of the main coverage criterion, and the use of search-based algorithms to generate test data accordingly. Moreover, we present the tool chain we created with the purpose of modelling and animating models described using UML, OCL and Alf. Finally, we explain how we validated the use of such tool chain in an MBT context by implementing a random test generation strategy that meets all-branches coverage criterion for each operation of the SUT. We also provide a way to optimize this strategy that consists in a first version of the genetic algorithm adapted to test sequence generation.

Key-words Model-Based Testing, UML, OCL, Alf, automatic test generation, genetic