Stratégies de génération de tests à partir de modèles UML/OCL interprétés en logique du premier ordre et système de contraintes

Jérome Cantenot

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

Numéro d'ordre : ???? **École doctorale SPIM**

Stratégies de génération de tests à partir de modèles UML/OCL interprétés en logique du premier ordre et système de contraintes

THÈSE

présentée et soutenue publiquement le

pour l'obtention du grade de

Docteur de l'université de Franche-Comté

(Spécialité Informatique)

par

Jérôme Cantenot

Composition du jury

Président :	Bruno Legeard, Professeur à l'université de Franche-Comté
Directeurs :	Fabrice Bouquet, Professeur à l'Université de Franche-Comté Fabrice Ambert, Maître de Conférences à l'Université de Franche-Comté
Rapporteurs :	Michel Rueher, Professeur à l'université de Nice Sophia Antipolis Ioannis Parissis, Professeur à l'université Pierre Mendes France - Grenoble 2
Examinateur :	Pierre-Alain Muller, Professeur à l'université de Haute Alsace

Institut FEMTO-ST - Département d'Informatique des Systmes Complexes - UMR CNRS 6174

Mis en page avec la classe thloria.

Remerciements

À mes encadrants, Fabrice Bouquet et Fabrice Ambert, pour m'avoir offert l'occasion de réaliser cette thèse.

À mes condisciples, notamment Pierre-Christophe et Betty, pour m'avoir toujours aidé en cas de difficulté.

À mes co-bureaux, Kalou et Seb, pour leur bonne humeur qui a soutenu mon moral.

À ma tante pour ses conseils et relectures.

À mes parents pour leur soutien et leur encouragement durant toutes ces années.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Sommaire

Ce manuscrit de thèse rapporte les travaux de recherche menés au sein du Département Informatique des Systèmes Complexes (DISC) de l'institut Femto-ST présent à l'université de Franche-Comté sous la direction du professeur Fabrice Bouquet et du maître de conférences Fabrice Ambert.

Cette thèse traite de problématiques relatives à la génération automatique de tests à partir de modèles utilisés dans la phase de validation fonctionnelle d'un développement logiciel.

Ce premier chapitre introduit le contexte scientifique puis développe la problématique en découlant. Ensuite les contributions apportées par ces travaux sont résumées. Finalement la structure du manuscrit est détaillée.

1.1 Contexte

Aujourd'hui notre société est construite sur l'usage de logiciels informatiques. Considérez simplement une journée de votre vie quotidienne : Vous utilisez une voiture récente, elle est gérée par un logiciel embarqué ; Vous retirez de l'argent à un distributeur, la transaction emploie un logiciel ; Vous vous reposez en jouant à Angry Birds sur votre téléphone, vous utilisez un logiciel ; Vous travaillez dans un bureau, vous utilisez un logiciel ...

Maintenant imaginons une journée en cas de défaillance des logiciels pour prendre conscience de leur importance au quotidien. De plus, cette problématique est amplifiée pour les logiciels critiques tel l'avionique où une erreur peut conduire à des pertes en vies humaines.

Une approche pour renforcer notre confiance dans un logiciel est de soumettre son fonctionnement à une série de tests. Dans ce cadre, le laboratoire DISC a notamment travaillé sur une approche où les tests sont générés automatiquement depuis un modèle encodant les différents comportements du système. Cette approche utilise un solveur de contraintes ensemblistes [LPU02] pour raisonner sur le système.

Cependant les modèles employaient le langage B et reposaient sur une notation Pré/Post qui était peu utilisée dans le monde industriel. C'est pourquoi les travaux de recherche suivants [BGL+07] ont étendu la modélisation à un langage dérivé de UML/OCL qui est devenu un standard industriel et dispose d'un écosystème large.

Durant ces dernières années, un outil de raisonnement sur un modèle mathématique, le prouveur SMT (Satisfiability Modulo Theory), a vu ses performances croître considérablement [SC]. Ce dernier est capable de déterminer si une formule logique du premier ordre a une solution. De plus, son expressivité est importante grâce à la combinaison de nombreuses théories telles les théories des entiers ou des tableaux.

Ainsi une question émerge naturellement : l'utilisation d'un prouveur SMT dans le cadre de génération de tests à partir de modèles a t-elle un intérêt ?

1.2 Problématique

L'objectif des travaux présentés dans ce manuscrit est de répondre à la question suivante : *Comment générer automatiquement des tests à partir d'un modèle comportemental d'un système écrit en UML4MBT/OCL4MBT, dérivé d'UML/OCL, à l'aide d'un prouveur SMT, utilisé seul ou conjointement à un solveur de contraintes ?*

Ainsi si notre démarche s'inscrit dans le cadre du test à partir de modèles, elle ne couvre pas l'ensemble de son spectre. Ainsi le choix et la génération des cibles de test répondant à des critères de sélections et visant à couvrir les fonctionnalités du système sont extérieurs à la démarche. Dans notre cadre, les cibles sont fournies par un processus externe. De même, notre démarche génère des cas de test abstraits sans présager de leur exécution ou de leur utilisation.

Pour employer un prouveur SMT afin de générer des tests, le système, ses comportements et les cibles de test doivent être modélisés sous la forme de formules compatibles avec le langage d'entrée des prouveurs. Ainsi un premier problème est la conversion d'un modèle UML4MBT en formules logiques.

Un deuxième problème est de minimiser le temps de génération des tests. Pour cela une analyse des caractéristiques des prouveurs et des formules soumises est nécessaire. Elle doit conduire à déterminer les paramètres influant sur le temps de résolution et la probabilité d'atteindre les cibles de test. Ces paramètres conduisent à la création de stratégies chargées de guider notre démarche de génération de tests.

La dernière question est liée à la collaboration entre un prouveur SMT et un solveur de contraintes. Quel intérêt à la collaboration et dans l'affirmative : comment échanger des informations ? Quelle forme prennent ces informations ? Quand intervient l'échange dans notre démarche ? Les réponses à ces questions viendront de l'étude de la complémentarité des avantages de ces deux outils.

1.3 Contribution

Pour apporter une réponse aux problématiques développées ci-dessus, les travaux de thèse se sont divisés en quatre phases.

Première phase : Elle consiste en la création d'une démarche de génération automatique de tests à partir de modèles en s'appuyant sur un prouveur SMT. La démarche est décomposable en deux étapes principales. En premier, une représentation formelle du système et de ses comportements, manipulable par le prouveur, est créée depuis la modélisation du système et des cibles de test. En second, cette représentation est soumise au prouveur et sa réponse est analysée pour générer des cas de test. La représentation formelle est assurée dans notre démarche par un méta-modèle dédié qui utilise un sous-ensemble du langage d'entrée du prouveur disposant d'une expressivité suffisante pour encoder l'ensemble des comportements du système. De plus, ce méta-modèle contient les informations relatives à la génération de tests tels les cibles de test et au processus de résolution du prouveur.

Deuxième phase : Elle capitalise sur la capacité de création d'un cas de test pour créer des stratégies de génération automatique de tests afin de répondre à des campagnes de tests. L'objectif est de réduire le temps de génération en considérant les relations entre les cibles de test, en adaptant et en hiérarchisant les problèmes soumis au prouveur et en employant la parallélisation.

Troisième phase : Elle élargit les possibilités de notre démarche en utilisant conjointement un solveur CSP et un prouveur SMT pour générer des tests. Ainsi un méta-modèle dédié à la génération de tests à l'aide d'un solveur est créé et des règles de collaboration entre ces outils sont énoncées. Ceci conduit à la création de nouvelles stratégies combinant les avantages des prouveurs et des solveurs.

Quatrième phase : Elle est l'évaluation de ces stratégies sur différents cas d'études. En particulier, les caractéristiques des outils fondant les stratégies sont vérifiées et le gain apporté par l'ajout de la parallélisation au sein de notre démarche est analysé.

1.4 Structure du document

Ce manuscrit de thèse s'articule autour de quatre parties.

Partie I : Elle introduit le domaine du test logiciel et présente les langages de modélisation employés dans notre démarche. Puis dans un second temps, une analyse des principaux outils de générations de tests à partir de modèles est conduite afin de déterminer comment raisonner sur les modèles. Finalement, les deux outils retenus, solveur CSP et prouveur SMT, dans ce but sont décrits.

Partie II : Elle présente notre démarche pour générer des tests à l'aide de stratégies multi-threads employant des prouveurs SMT et des solveurs CSP depuis une modélisation du système écrite en UML4MBT/OCL4MBT.

Partie III : Elle analyse les performances de l'implémentation des stratégies sur la plate-forme Hydra sur différents cas d'études.

Partie IV : Elle résume les contributions apportées par ces travaux et présente différentes pistes pour améliorer notre démarche.

Le **chapitre 1** est la présente introduction.

Partie I : Contexte et problématique

Le **chapitre 2** présente le domaine du test logiciel en se concentrant sur le test à partir de modèles. Puis il décrit les langages de modélisation.

Le **chapitre 3** décrit les principaux outils et méthodes de génération de tests à partir de modèles pour déterminer les approches et outils à employer dans notre démarche.

Le **chapitre 4** détaille les outils retenus pour raisonner sur les modèles ; les solveurs CSP et les prouveurs SMT.

Partie II : Contributions - Tests, prouveurs et solveurs

Le **chapitre 5** introduit l'exemple fil rouge, un robot, servant à illustrer notre démarche dans les chapitres suivants.

Le **chapitre 6** formalise notre processus de génération de tests à l'aide de prouveurs et de solveurs depuis un modèle écrit en UML4MBT/OCL4MBT.

Le **chapitre 7** donne différentes stratégies de génération de tests utilisant un prouveur SMT et la parallélisation.

Le **chapitre 8** décrit le méta-modèle SMT4MBT qui autorise l'écriture de la recherche d'un cas de test avec des formules logiques compréhensibles par un prouveur SMT. Il liste également les règles de conversion d'un modèle UML4MBT en un modèle SMT4MBT.

Le **chapitre 9** est consacré aux stratégies de génération de tests employant conjointement un prouveur SMT et un solveur CSP. Pour cela, il introduit notamment le méta-modèle CSP4MBT qui est le pendant de SMT4MBT adapté aux solveurs CSP.

Partie III : Étude de cas

Le **chapitre 10** présente l'implémentation des stratégies de générations de tests dans la plate-forme Hydra et l'influence de cette dernière sur les performances.

Le **chapitre 11** évalue les performances des stratégies sur différentes études de cas.

Partie IV : Conclusion et perspectives

Le **chapitre 12** donne une synthèse des travaux présentés dans ce document.
Le **chapitre 13** liste différentes pistes pour améliorer notre démarche.

Première partie

Contexte et problématique

Chapitre 2

Test à partir de modèles - MBT

Sommaire

L'objectif du test n'est pas de garantir un fonctionnement parfait dans toutes les situations ce qui est généralement impossible mais de garantir un degré de confiance dans le fonctionnement du logiciel. Ce degré est déterminé par le domaine où le logiciel est utilisé et la fonctionnalité considéré. Il est évident que les secteurs critiques mettant en jeu des sommes importantes ou des vies humaines tels le secteur des transports ou bancaires, nécessitent un degré supérieur à celui des loisirs par exemple où une défaillance aurait des conséquences moindres.

Le test est un domaine large qui répond à des problématiques différentes. Celui-ci est souvent qualifié pour permettre d'affiner son usage. Nous pouvons citer par exemple : le test de robustesse, le test fonctionnel ou encore le test de vulnérabilité. Ainsi la première section de ce chapitre présente les notions liées au test et ses catégories usuelles.

Le sous-domaine du test concerné par notre démarche est le test fonctionnel à partir de modèles. Cependant ce type de test est également divisible en plusieurs sous-domaines d'intérêt qui sont notamment définis par le choix d'un langage de

modélisation et de la méthode de sélection des tests. Ce sous-domaine est le sujet de la deuxième section. La section suivante présente les langages de modélisations employés par notre démarche pour représenter les comportements d'un système dans le cadre de la génération de test. Finalement une synthèse des informations apportées dans ce chapitre est écrite dans la dernière section.

2.1 Le test logiciel

Après avoir présenté les motivations conduisant à employer le test pour accroître le degré de confiance dans les logiciels, cette section détaille les principales catégories de tests en listant les critères de classification usuels. Dans la dernière section, les différents processus de génération de tests fonctionnels sont introduits.

2.1.1 Motivation

Tout utilisateur de logiciel informatique a déjà été aux prises avec un bug informatique. Si ce dernier peut simplement avoir comme conséquence un redémarrage du logiciel ou de l'ordinateur, il peut aussi en résulter des pertes financières ou humaines si des systèmes critiques sont touchés. Ainsi, l'année 2011 a compté des bugs importants [bilpcd]. Par exemple, l'armée américaine a déployé un cloud pour soutenir ses troupes en Irak et en Afghanistan mais des bugs récurrents ont conduit à l'abandon de ce projet. Le seul coût de conception est estimé à 2,7 milliards de dollars.

Un autre bug fameux est celui rencontré par la fusée Ariane 5. La fusée s'est auto-détruite une quarantaine de secondes après son décollage car sa trajectoire était incorrecte. L'origine de ce problème est un bug logiciel : la conversion d'un nombre stocké sur 64 bits en un nombre stocké sur 16 bits a produit des valeurs incorrectes. Une analyse plus détaillée est disponible dans le rapport [LL97].

Ainsi les bugs des logiciels informatiques ont un coût important. Ce coût a été évalué pour les États-Unis à un pour cent du produit intérieur brut ou en 2013 à 312 milliards d'euros par an par une étude de Cambridge [jbs]. Une étude [Tas02] du National Institute of Standards and Technology américain estime que le manque d'infrastructures de tests adéquates coûte entre 22,9 et 55,2 milliards de dollars.

En conséquence une des phases importantes de la réalisation d'un logiciel est de s'assurer que ses fonctionnalités soient conformes aux spécifications du cahier des charges. Il existe deux méthodes pour s'assurer de la conformité. La première est de vérifier le logiciel au moyen d'une méthode formelle à même de fournir une preuve. Cette vérification est définie de la manière suivante :

Définition 1 (Vérification-IEEE) *La vérification est le processus permettant d'obtenir une preuve formelle (mathématique) de la conformité du logiciel à ses spécifications*

Les définitions issues du glossaire standard de l'IEEE (Institute of Electrical and Electronics Engineers) pour la terminologie de l'ingénierie logicielle [Sep90] sont

annotées par l'acronyme IEEE.

La vérification a pour avantage d'obtenir une certitude vis-à-vis de la conformité du logiciel aux spécifications mais elle nécessite que ces dernières soient écrites dans un langage formel. Ceci facilite l'automatisation de la procédure mais nécessite des compétences plus importantes lors de la rédaction de ces dernières par rapport à un langage naturel. De plus, une méthode formelle utilise forcement une méthode ancrée sur la manipulation de formules mathématiques. La transformation et la manipulation de systèmes complexes sont délicates et résultent souvent en un problème de mise à l'échelle.

Une deuxième possibilité est de valider le logiciel en fonction d'un certain nombre de critères. L'idée n'est pas d'obtenir une preuve mathématique de sa conformité aux spécifications mais simplement d'obtenir un degré de confiance dans son fonctionnement. Une solution est de tester le logiciel pour évaluer son comportement.

Définition 2 (Test logiciel-IEEE) *Le test logiciel est l'analyse d'un logiciel ou une partie d'un logiciel pour détecter les différences entre le fonctionnement actuel et désiré du logiciel correspondant à des bugs et d'évaluer les fonctionnalités de la partie du logiciel analysée.*

2.1.2 Critères de classification

Les trois principaux critères de classification sont le type d'accès aux données, le niveau du système concerné et l'objectif.

Accès aux informations

Un premier critère pour classifier les tests est d'analyser quel types d'informations sont accessibles. Il existe trois type d'accès nommés : boite blanche, boite noire et boite grise. Ces catégories sont résumées dans la figure 2.1.

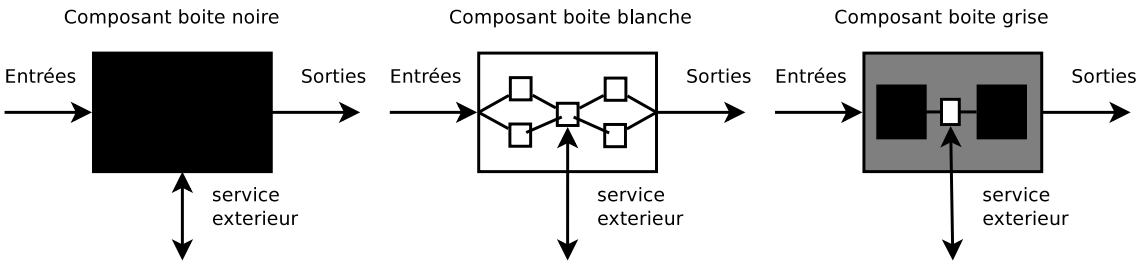


Figure 2.1 – Catégorie des tests en fonction de l'accès aux informations.

Boîte noire Le terme boite noire, dont la définition est donnée ci-dessous, appliqué au domaine du test signifie que l'interaction avec le composant à tester est limité à des stimulus d'entrées et à des variables de sorties pour observer le comportement

du système. Lors de ce type de test, le fonctionnement interne et donc l'implémentation des fonctionnalités sont inconnus. Ainsi le test est créé uniquement depuis les spécifications du système.

En conséquence, les tests en boite noire sont générés depuis une abstraction du système. Le choix de l'abstraction est donc crucial. Elle doit permettre d'établir les stimulus et les informations observables du système avec un niveau de détails suffisant pour obtenir des tests pertinents. Si lors de l'exécution de certaines fonctionnalités, le composant interagit avec d'autres, il peut être impossible d'observer les états intermédiaires du composant.

Définition 3 (Test boite noire-IEEE) *Les tests boites noires ignorent le fonctionnement interne du système à tester et utilisent uniquement les réponses du système à des stimulus.*

Boîte blanche Le test boite blanche, dont la définition est donnée ci-dessous, est l'opposé du test en boite noire. Ce type de test a accès à l'implémentation des fonctionnalités du système. Il permet ainsi de pouvoir identifier les différents traités dans le code ou les algorithmes utilisés. Par exemple, dans le cas d'une fonction de tri à bulle, le test en boite blanche peut être conduit depuis le code source alors que le test en boite noire est réalisé depuis une spécification précisant que le tri à bulle est capable de trier un tableau d'éléments avec une complexité de $\Theta(n^2)$.

Le test en boite blanche a deux inconvénients majeurs. Le premier est le problème de la disponibilité des informations. Ainsi l'obtention des sources du logiciel à valider peut être difficile en dehors de l'équipe de développement. Le deuxième problème est la sensibilité à l'évolution. Ainsi la validation d'une fonctionnalité en utilisant son implémentation interagit directement avec le code pour comparer avec les valeurs attendues et doit être aussi modifiée lors de l'évolution du code source.

Définition 4 (Test boite blanche-IEEE) *les tests boites blanches prennent en compte le fonctionnement interne du système à tester. Dans le cas d'un logiciel, cela signifie que l'on a accès au code source.*

Boîte grise Le dernier type de test, le test en boite grise, dont la définition est donnée ci-dessous, est une combinaison du test en boite noire et en boite blanche. Ainsi le test a accès à une partie du fonctionnement interne du système. L'objectif est de supprimer la limitation du test en boite noire en ajoutant des informations sur le fonctionnement lorsqu'elles sont requises pour un test. Cependant ce type de test continue de manipuler une abstraction. Une analyse de l'intérêt de ce type de test est donnée dans l'article [BW97].

Définition 5 (Test boite grise) *Les tests boites grises prennent en compte une partie du fonctionnement interne du système à tester.*

Niveau du système

Un deuxième critère de classification des tests est le niveau du système analysé. L'International Software Testing Qualifications Board (ISTQB) définit quatre niveaux [otISTQB12] :

Unitaire Un test unitaire prend place sur un unique composant d'un système. Le but est de vérifier que chaque composant fonctionne comme prévu.

Intégration Les tests d'intégrations sont exécutés pour exposer des erreurs au niveau des interfaces et des intégrations entre des composants ou des systèmes. Ces tests sont réalisables en boite blanche, grise ou noire.

Système Ces tests opèrent sur un système intégré pour vérifier la conformité du système aux spécifications. Ces tests sont le plus souvent des tests en boite noire.

Acceptation Le test d'acceptation est un test formel prenant en compte les besoins de l'utilisateur, les spécifications et les processus métiers et visant à déterminer si oui ou non un système satisfait aux critères d'acceptation.
Il permet à un utilisateur, un client ou à une entité habilité à décider s'il accepte le système. Ce type de test intervient généralement entre deux phases du processus de développement logiciel et appartient à la catégorie du test en boite noire.

Objectif

Un troisième critère de classification des tests est son objectif. L'objectif correspond aux types d'erreurs que le test vise à découvrir. Parmi les nombreux objectifs existants, nous pouvons citer les tests de :

Performance Les tests de performance étudient le comportement du système lors d'une utilisation intensive telle que rencontrée avec un nombre d'utilisateurs important ou des informations conséquentes.

Fonctionnel Les tests fonctionnels s'assurent de la conformité d'un système vis-à-vis des fonctionnalités décrites dans les spécifications.
Ces tests constituent l'immense majorité des tests exécutés. Ils sont réalisés en boite noire. De manière classique, un test fonctionnel suit la procédure suivante : identifier les fonctionnalités du système, créer le jeu de données de test en fonction des spécifications, déterminer le résultat attendu d'après les spécifications, exécuter le cas de test et comparer le résultat obtenu à celui attendu.

Robustesse Les tests de robustesse évaluent jusqu'à quel degré un système fonctionne correctement quand il subit des perturbations telles que des problèmes de réseaux ou de matériels.

Vulnérabilité Les tests de vulnérabilités visent à découvrir des failles dans le système qui pourraient conduire un intrus à avoir accès à des fonctionnalités ou des informations. Ces tests s'assurent généralement de la sécurité du réseau, des logiciels, des applications clients ou des applications serveurs.

Non-régression Les test de non-régression s'assurent qu'une modification n'introduit pas d'erreur dans le logiciel. Ce type de test peut être appliqué à n'importe quel niveau mais il est souvent identifié ainsi principalement au niveau du test du système.

Ergonomie Les tests d'ergonomie se concentrent sur les problèmes liés à l'utilisation du logiciel par un utilisateur tel qu'une interface non-intuitive.

2.1.3 Processus de test pour le test fonctionnel

Après avoir présenté brièvement les différents types de test, cette section se concentre sur le test fonctionnel qui est celui retenu dans le cadre de notre démarche.

La figure 2.2 représente les principaux processus, à l'exception du test à partir de modèles qui est traité dans la section suivante, pour tester un système d'un point de vue fonctionnel. La description de ces processus est principalement dérivée du livre [UL10]. Un cas de test est défini dans cette partie par une séquence d'instructions à exécuter sur le modèle et les résultats correspondants attendus. Le symbole $\otimes$ dans la figure représente une étape du processus. Cette dernière peut être exécutée par un des intervenants suivants, dénoté par un symbole sous $\otimes$:

Concepteur Le concepteur est responsable de la création des cas de test. Ces derniers doivent remplir les objectifs de couverture fixés dans le plan de tests. En conséquence, le concepteur doit connaître intimement le système sous test et les stratégies de création de tests.

Testeur Le testeur est chargé d'exécuter des cas de test manuellement. Cette tâche ne requiert pas de compétence particulière en programmation et en modélisation.

Programmeur Le programmeur est capable de comprendre le langage servant à implémenter le système et à créer des scripts employant directement des fonctions de l'implémentation. En règle générale, le programmeur a participé à la création de l'implémentation.

Ordinateur Si l'étape du processus ne requiert pas d'intervention humaine, un ordinateur précise qu'elle est automatisée.

Les quatre processus décrits sont :

Manuel Le premier processus, le plus basique, est le processus manuel. Dans ce dernier, la personne réalise les cas de test manuellement pour répondre aux besoins exprimés dans le plan de tests. Ces cas sont exprimés dans un langage que peut comprendre la personne tel le français et leur rédaction peut être simplifiée en prenant en compte que leur exécution sera manuelle. La création manuelle des cas de test entraîne le plus souvent une couverture partielle du modèle car le concepteur est rarement à même d'envisager l'ensemble des possibilités et surtout le temps d'exécution des tests peut être long. Quand le testeur exécute les tests, il rédige en même temps le rapport de tests présentant les résultats. Les interventions humaines importantes engendre un coût élevé mais permettent de s'adapter rapidement en cas de modification des spécifications ou du plan de tests.

Capture Ce second processus est une amélioration du précédent. L'idée est de mémoriser les actions effectuées par le testeur lors de l'exécution d'un cas de test dans un script grâce à un outil de capture. Ensuite quand ces tests doivent être ré-exécutés lors d'une autre campagne de tests, un outil d'exécution permet de jouer ces scripts sur le système sous test. Ainsi le coût engendré par la présence d'un testeur est supporté uniquement lors de la première exécution. Le framework Selenium [HK06] propose de tels outils. Cette approche est malheureusement très sensible aux modifications apportées au système sous test et est généralement réservée aux tests à travers des interfaces graphiques.

Script Le troisième processus utilise également des scripts de tests. Cependant ceux-ci sont écrits manuellement par un programmeur. Si ce procédé oblige à disposer d'une personne comprenant parfaitement le système sous test, le coût de la ré-exécution de ces scripts est négligeable. Il existe plusieurs langages pour écrire ces scripts tels que TTCN-3 (Testing and Test Control Notation) [GHR+03] où TSL (Test Specification Langage) [HVFR04].

Mot-clé Une autre approche est d'améliorer le processus par script en réduisant la sensibilité des scripts aux modifications intervenant sur le système sous test. Pour ce faire, une couche d'abstraction supplémentaire est introduite. Ainsi les scripts créés utilisent une notation abstraite sous forme de mot-clés pour gagner en généricité. Ensuite un programmeur est chargé d'établir un lien entre ces mot-clés et l'implémentation via un code d'adaptation. Ainsi en cas de modifications du modèle, seul ce code est amené à évoluer. Les frameworks Concordion [Cona] et FitNesse [MC05] se conforment à ce processus.

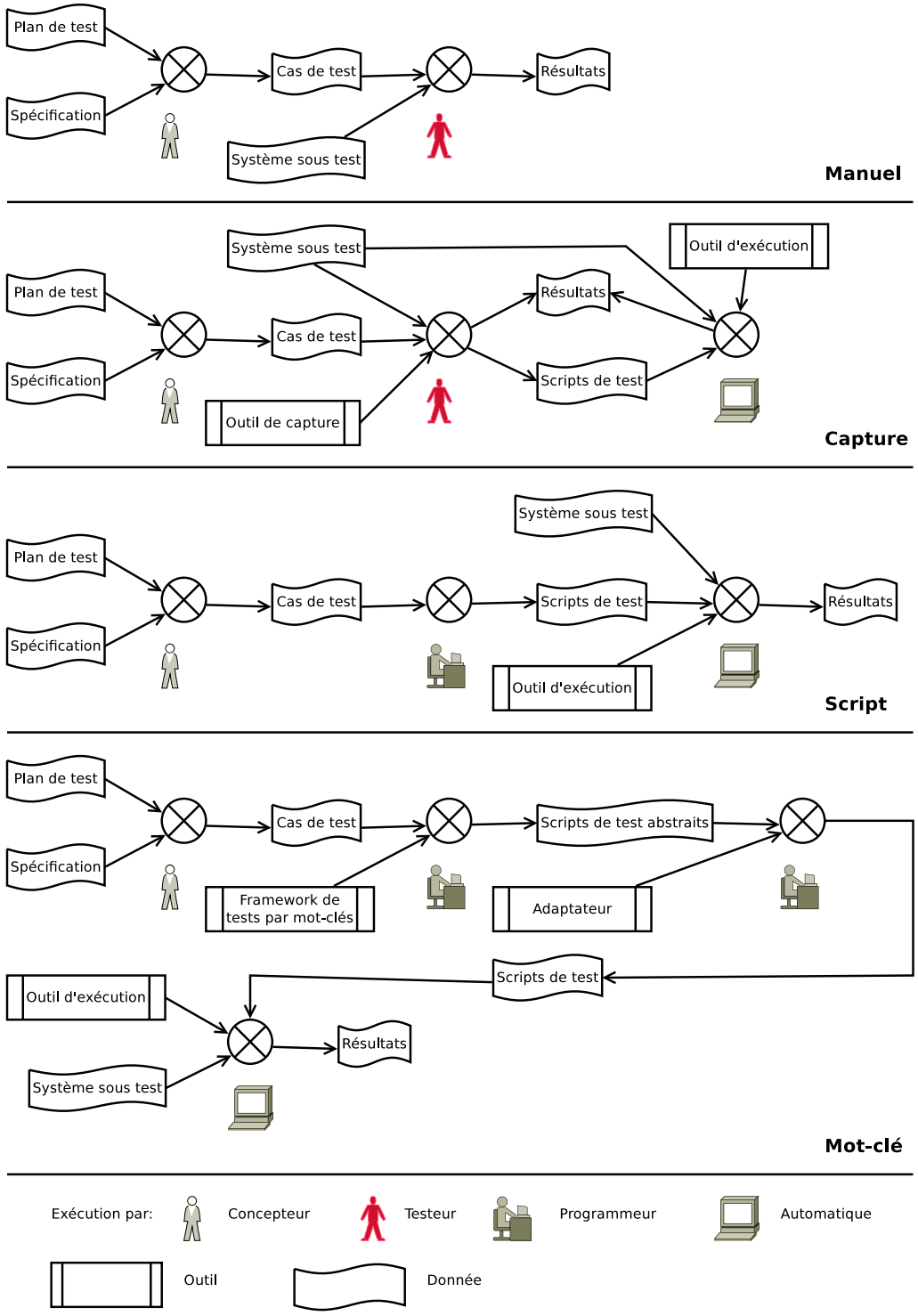


FIGURE 2.2 – Processus de test pour le test fonctionnel.

2.2 Le test à partir de modèle

Maintenant que le domaine du test logiciel a été abordé, cette section se concentre sur le test à partir de modèles, objet de notre démarche. Après avoir détaillé les différentes étapes du processus et mis en valeur les avantages et les inconvénients du test à partir de modèles, les différentes sous-catégories de ce processus sont détaillées en s'appuyant sur les choix de modélisation et de sélection des tests.

2.2.1 Description

La définition suivante du test à partir de modèles (Model-Based Testing) est extraite du livre [UL10].

Définition 6 (Test à partir de modèles (MBT)) *Le test à partir de modèles consiste en la génération de cas de test avec des oracles depuis un modèle comportemental.*

Ces tests appartiennent à la catégorie des tests fonctionnels automatisés en boite noire. L'oracle doit permettre de déterminer la réussite d'un test en comparant le résultat obtenu lors de son exécution avec une valeur attendue. Le modèle comportemental décrit les comportements en reliant les valeurs d'entrées et de sorties.

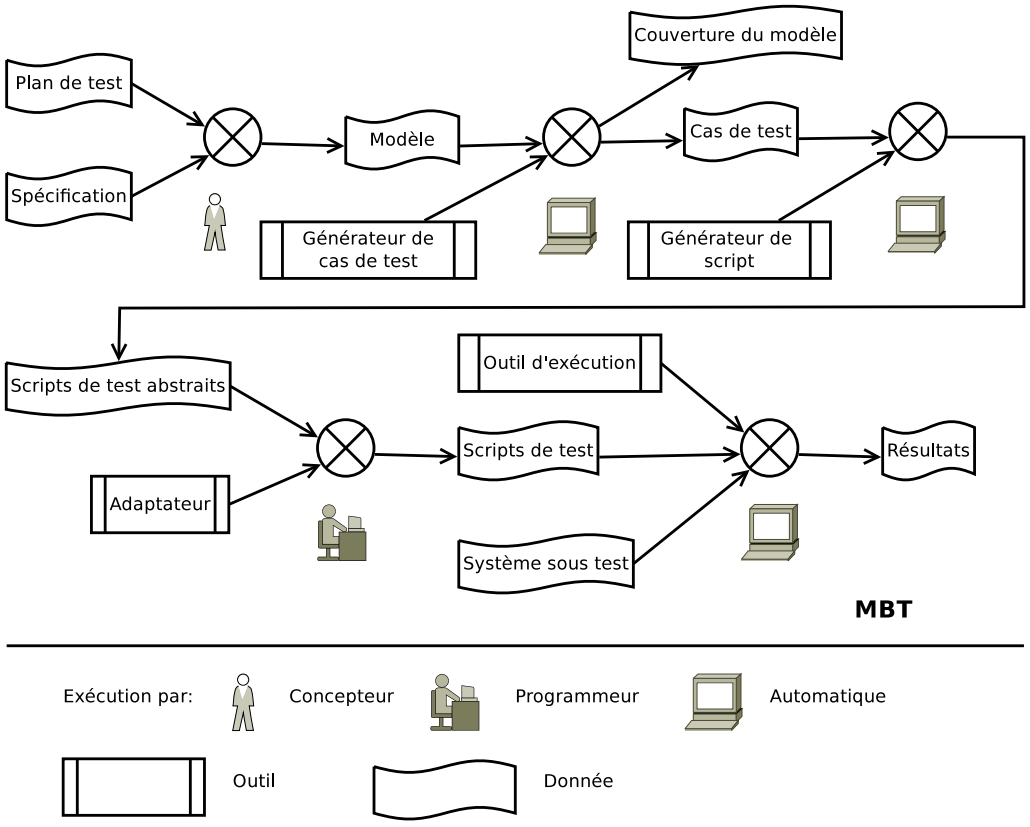


Figure 2.3 – Processus de test dans le cadre du test à partir de modèle

La figure 2.3 présente le processus réalisé dans le cadre du test à partir de modèles. Cette figure respectent les mêmes conventions de notations que la figure 2.2 présentée à la section 2.1. Ce processus est défini par les cinq étapes suivantes :

1. La première phase est la modélisation du système sous test par le concepteur. Ainsi le concepteur doit posséder des compétences avancées dans le domaine des langages de modélisation. Le modèle est une abstraction car il doit préciser uniquement les informations utiles pour obtenir des tests pertinents. Une présentation des différents choix de modélisation est donnée dans la sous-section suivante.
2. La seconde phase est la génération de cas de test de manière automatisée par un outil adéquat. À la différence des cas de test créés dans les autres processus de génération de tests fonctionnels, présentés à la section précédente, ceux créés dans le cadre du MBT sont abstraits. Cela signifie que les cas de test sont décrits dans le même niveau d'abstraction que le modèle et donc il peut être éloigné de l'implémentation. L'avantage c'est que nous pouvons les manipuler automatiquement dans la suite du processus. Le choix des cas de test est déterminé le plus souvent par un critère de couverture. Les critères les plus courants sont présentés à la sous-section 2.2.3.
3. La troisième phase est la génération de scripts de tests abstraits. Cette phase diffère de celles réalisées dans le cadre du processus de génération de tests par scripts ou guidé par mot-clés car elle est réalisée automatiquement grâce à la description formelle des cas de test. Elle permet de déplier les cas de test abstraits sur un ensemble de valeurs ou d'entités qui ont été abstraites.
4. La quatrième phase est l'adaptation des scripts de tests abstraits à l'implémentation du système sous test. Cette phase est réalisée par un programmeur connaissant l'implémentation du système. Cette réification, constituée d'une couche d'adaptation, permet de séparer clairement le code du modèle de test. La combinaison de cette étape avec la précédente correspond à la concrétisation des cas de test abstraits. Elle est souvent basée sur l'approche à partir de mot-clés [BL03].
5. La dernière étape est l'exécution des scripts de tests sur le système sous test par un outil d'exécution. Cette exécution permet d'obtenir les résultats des tests. Un autre avantage de la formalisation des cas de test est la possibilité d'associer automatiquement le résultat d'un script au cas de test correspondants.

Avantages Le premier avantage et l'objectif principal du test à partir de modèles est de détecter des erreurs dans le système sous test. L'automatisation de la majeure partie du processus permet d'obtenir des taux de détection élevés, le plus souvent supérieurs aux taux obtenus avec un processus manuel. Ce fait est soutenu par plusieurs études [PPW+05], [V+06].

Un autre avantage est de réduire le temps et le coût consacré pour tester le système. Si la création du modèle entraîne un surcoût par rapport aux autres processus, ce dernier est largement compensé par un degré d'automatisation important dans

la suite du processus [PPW+05]. Cet avantage croît si les tests sont ré-exécutés régulièrement. De plus, la correspondance établie automatiquement entre les résultats d'un test et la partie du modèle couverte permet de gagner du temps lors de la phase d'analyse des résultats.

L'automatisation de la génération des cas de test permet d'obtenir des tests de meilleures qualités car leurs résultats sont répétables et leur pertinence peut être évaluée objectivement par des critères de couverture. De plus, si des comportements du modèle ne sont pas testés, alors cela peut signifier un manque de précisions dans les spécifications du modèle.

Le test à partir de modèles est particulièrement adapté si le système subit des évolutions. Dans ce cas, seul le modèle doit être modifié pour répondre aux nouvelles spécifications du cahier des charges. Hors le modèle étant une abstraction, sa taille est inférieure à celle du système.

Inconvénients Le modèle utilisé pour générer les cas de test est construit en s'appuyant sur les spécifications du cahier des charges. En conséquence, le test à partir de modèles nécessite d'avoir des spécifications en concordance avec le système. En particulier, une évolution du modèle doit toujours résulter d'une évolution correspondante des spécifications.

Une autre difficulté est de représenter l'ensemble des spécifications au travers d'un modèle. Par exemple des tests s'assurant de l'ergonomie ou de l'esthétisme du système sont très difficilement modélisables. Dans ce cas, des tests manuels sont plus adaptés.

Un dernier problème est d'analyser l'origine d'une erreur. Un test peut échouer car le système contient une erreur, le code d'adaptation pour exécuter le test sur l'implémentation est incorrect ou le modèle est erroné. En particulier, l'abstraction représentée par le modèle peut introduire des erreurs. De plus, l'automatisation des cas de test créée des séquences d'actions qui ne sont pas intuitives d'un point de vue humain.

2.2.2 Modélisation

Cette section présente les différents modèles utilisés dans le cadre du test à partir de modèles. Ils sont catalogués selon la taxonomie proposée dans l'article [UPL12]. Les quatre critères de classification sont le sujet du modèle, sa redondance au niveau de son utilisation, ses caractéristiques et son paradigme.

Sujet Le sujet du modèle est soit l'environnement du système, soit le comportement du système. La modélisation de l'environnement a pour but de réduire la taille de l'espace constitué par les variables d'entrées. Par exemple, dans le cas du langage UML, cette modélisation est assurée par le diagramme d'objets qui permet d'assigner les entités initiales du système modélisé et des valeurs utiles pour le test. Le modèle est une abstraction et par conséquent conserve uniquement les informations utiles pour les tests. Cette simplification permet de gagner en performance et en

lisibilité. Le modèle peut ainsi abstraire les fonctions, les données ou les communications. Par exemple, une chaîne de caractères mémorisant le nom d'une personne qui prend une infinité de valeurs, peut être modélisée par une énumération d'un nombre fini de noms.

Redondance La redondance est analysée en terme d'utilisation du modèle. Elle répond à la question : le modèle employé dans le cadre du test à partir de modèles doit-il être créé spécifiquement dans ce but ? Dans le cas d'une réponse négative, le modèle est utilisé conjointement pour la génération de tests et du code. Par exemple, il est possible de générer du code java ou C# à partir de diagrammes UML et de contraintes OCL. Le problème de cette approche est de réduire l'intérêt de l'abstraction proposée par le modèle. En effet, la génération de code impose d'avoir un modèle extrêmement détaillé.

Dans le cas d'une réponse positive, une redondance existe car le modèle dédié aux tests et le modèle pour le développement ont naturellement des parties communes [BDLN06]. Le surcoût engendré par la création d'un modèle dédié est compensé par la possibilité d'adapter le modèle aux besoins de l'outil de génération de cas de test.

Caractéristiques Les caractéristiques du modèle se rapportent dans cette section au déterminisme, à la mesure du temps et à l'analyse de son évolution. Le déterminisme dans le modèle n'a pas de relation directe avec celui du système. Ainsi un modèle non-déterministe est utilisable pour tester un système déterministe. La mesure du temps est particulièrement importante pour tester des systèmes temps réels. Le dernier critère porte sur l'évolution du système. Elle peut être continue, discrète ou hybride. Ces caractéristiques ont un impact sur le choix du paradigme de modélisation et le choix de l'outil.

Paradigme Le choix du paradigme de modélisation est principalement guidé par l'expressivité nécessaire pour représenter les comportements du système et les résultats attendus. Il existe les paradigmes suivants :

Pre/Post Dans ce paradigme, un état du modèle est représenté par un ensemble de variables. Un état est modifié par l'exécution d'opérations. Ces dernières sont définies par une pré-condition qui précise quand leur exécution est permise et par une post-condition qui donne les valeurs des variables à l'issue de leur exécution. Ce paradigme est le plus utilisé. Les langages B [AAH05], Java Modeling Language (JML) [LBR98], Object Constraint Language [Spe06] ou Spec# [BLS05] emploient ce paradigme.

Transition Dans ce paradigme, les différents états du modèles sont représentés par des éléments reliés par des transitions. Ce paradigme est appliqué par exemple dans les machines à états finis (Finite State Machine) [Eda95], les diagrammes d'états-transitions UML [RJB04] ou les diagrammes de flux Simulink [DH01].

Historique Avec ce type de notation, l'évolution du modèle dépend de ces états passés. La représentation du temps peut être discrète ou continue. Cette ca-

tégorie contient également les séquences de messages comme le diagramme de séquence UML.

Fonctionnel Dans ce cas, le modèle est représenté par des fonctions mathématiques. Elles sont écrites dans une logique du premier ordre pour les prouveurs SMT [BST10] (Satisfiability Modulo Theories) ou dans une logique d'ordre supérieure pour HOL [GM93] par exemple.

Opérationnel Les notations opérationnelles décrivent un système comme une collection de processus exécutés en parallèle. Un exemple est la notation Minizinc [NSB+07] utilisée comme langage d'entré des solveurs CSP (Constraint Satisfaction Problem).

Stochastique Un modèle stochastique est construit à partir de probabilité d'apparition des valeurs des variables et des événements. Un état des lieux de ces notations est proposé dans l'article [Ros00].

Flux de données Un dernier type de notation est celle s'appuyant sur les flux de données. Des exemples sont le langage LUSTRE [HCRP91] ou les diagrammes de blocs dans Simulink.

2.2.3 Sélection des tests

Dans le cadre du test à partir de modèles, les cas de test sont générés automatiquement par un outil en s'appuyant sur un modèle. Le choix des cas de test est guidé par un critère de sélection. Dans ce cas, le critère est construit à partir de propriétés du modèle et non de l'implémentation du système sous test.

Le taux de couverture associé au critère de sélection permet de mesurer l'adéquation de la campagne de tests. Il est également employé comme critère d'arrêt pour prendre la décision que le nombre de tests est suffisant en regard du plan de tests. En effet, un taux de couverture de cent pour cent est rarement atteignable à cause de limitations technologiques ou d'états inatteignables dans le modèle.

Couverture structurelle du modèle Les critères exploitant la structure du modèle comme les états, les transitions ou les préconditions et postconditions des opérations pour sélectionner les cas de test établissent une couverture structurelle du modèle. Une première série de contraintes utilise les graphes de contrôles. Ces critères sont équivalents à la couverture usuelle du code dans le cadre du test en boite blanche. Les plus connus sont : le *critère des assertions* portant sur l'ensemble des assertions atteignables, le *critère des branches* couvrant les différents flux de contrôle et le *critère des chemins* vérifiant l'ensemble des chemins dans le graphe de contrôle.

Un autre de type de critères est construit sur les graphes de flux de données. Ces critères s'assurent que l'ensemble des couples valides formés par une définition (une assignation) et une utilisation (une lecture) des variables soit couvert par un test.

Il existe des critères spécialisés pour les modèles employant des systèmes de transitions. Nous pouvons citer les critères : *états* qui s'assurent que tous les états atteignables sont atteints, *transitions* qui vérifient que l'ensemble des transitions

sont couvertes et *chemins* qui testent l'ensemble des chemins. Il existe également de nombreuses variantes pour résoudre le problème de l'explosion combinatoire introduit par la présence de boucles.

Couverture des données Les critères de couverture portant sur les données du modèle ont principalement pour objectif de réduire le nombre de tests couvrant les combinaisons des valeurs des variables d'entrées. Par exemple, une fonction dépendant de 5 variables dont les valeurs sont comprises entre 0 et 9 inclus autorise 10^5 combinaisons. Le critère donne ainsi un nombre de valeurs à tester. Il reste cependant à décider quelles sont les valeurs à sélectionner dans le domaine des variables. Les trois principales méthodes sont de prendre des valeurs aux bornes du domaine, selon une distribution probabiliste ou au hasard.

Couverture des exigences L'idée est de formaliser les exigences pour permettre à l'outil de génération de cas de test de les employer comme critère de sélection. Il y a deux possibilités : soit le modèle contient les exigences, soit les exigences sont formalisées à part avec une expression logique.

Couverture explicite de cas de test Ce critère est dérivé de la formalisation des exigences. Un ingénieur de test peut guider de manière très précise les parties à couvrir en utilisant ce formalisme. Cependant ce critère diminue le degré d'automatisation de la procédure.

Couverture probabiliste L'idée de ce critère est d'utiliser des profils d'utilisations recensant les probabilités d'exécution des opérations pour sélectionner des tests couvrant les comportements les plus fréquents. Cependant ce critère a besoin d'un modèle comportemental pour fournir les résultats attendus des tests.

Couverture des catégories d'erreurs Ce critère utilise le fait qu'un certain nombre d'erreurs ont des causes connues. Ainsi le modèle introduit volontairement des erreurs, par exemple à l'aide de mutation, et les tests doivent alors constater que des erreurs sont apparues.

2.3 Les langages UML4MBT et OCL4MBT

Notre démarche s'inscrit dans le cadre de la génération de tests à partir de modèles présenté ci-dessus. Par conséquent, un langage de modélisation est nécessaire. Ainsi notre démarche utilise le langage UML4MBT. La première partie de la section présente ses caractéristiques et son expressivité. Cependant ce langage n'est pas suffisant pour décrire les comportements du système. En conséquence, la deuxième partie de la section introduit le langage d'action nommé OCL4MBT qui permet de décrire l'aspect dynamique du système dans le modèle.

2.3.1 Langage UML4MBT

Raison Pour modéliser un système, nous avons choisi d'utiliser un langage adapté aux tests basés sur des modèles. Ce langage, nommé UML4MBT pour "UML for Model Based Testing", est défini initialement dans l'article [BGL+07] comme un sous-ensemble de "Unified Modeling Language" 2.0 (UML) [PMV03]. Pour satisfaire nos besoins, le langage adopte un point de vue spécifique sur le système centré sur la notion de système sous test.

Le choix d'un langage de modélisation est généralement guidé par ses performances en regard des critères suivants :

Adéquation Le premier critère est l'adéquation du langage au domaine cible. Le langage ayant été conçu spécifiquement pour le test à partir de modèle, la validation de ce critère est acquise. Un exemple d'utilisation de ce langage dans ce cadre est donné par le projet SecureChange qui l'a utilisé pour modéliser un module de Smart Card depuis une spécification nommée Global Platform [FB10].

Expressivité Le deuxième critère est le niveau d'expressivité offert par le langage. Ainsi le langage permet de décrire la structure du système au travers de diagrammes structurels (diagrammes d'objets et de classes) et le comportement du système avec un diagramme comportemental tel le diagramme d'états-transitions.

Apprentissage L'apprentissage mesure la facilité d'utilisation du langage. UML est devenu un standard dans de nombreux domaines industriels tels les systèmes informatiques, les télécommunications, le transport... Par conséquent, le langage est déjà utilisé par de nombreuses personnes. De plus, sa composante graphique simplifie sa prise en main.

Écosystème Un dernier point à considérer est l'écosystème gravitant autour du langage. L'implémentation du langage est construite en s'appuyant sur un framework répandu : Eclipse Modeling Framework (EMF) [SBMP08]. Ainsi la modélisation est réalisable au travers d'outils industriels standards tels Rational Software Architect (RSA) [Arc] ou Topcased [Top]. De plus, l'écosystème est étendu par la possibilité de conversion du langage dans le format XML Metadata Interchange (XMI) [PMV03].

Le point adopté sur le système et l'objectif de génération de tests ont conduit à restreindre à trois le nombre de diagrammes autorisés parmi les 13 diagrammes présents en UML 2.0.

Diagramme de classes Le premier diagramme autorisé et obligatoire est le diagramme de classes. Ce diagramme fournit une représentation structurelle statique du système. Ce diagramme contient des classes, des énumérations et des associations.

Les classes sont des abstractions des objets constituant le système. Elles sont définies par un ensemble d'attributs et d'opérations. Dans UML4MBT, les attributs sont de type booléen, entier borné ou valeur littérale pour les énumérations. Les réels, les rationnels et les chaînes de caractères sont interdits. Pour les remplacer, le

modèle utilise une énumération contenant une liste de valeurs choisies. Les opérations sont définies au travers d'une précondition, d'une postcondition et d'une liste de paramètres. Ces paramètres peuvent prendre les mêmes types que les attributs et un type objet. Les associations présentent dans le diagramme de classes ont pour particularité d'être des ensembles. Par conséquent, les associations ne peuvent pas contenir des doublons.

La modélisation dans le cadre du MBT a tendance à singulariser une classe pour représenter le système sous test. Techniquement dans le cadre de notre démarche, cette disposition n'est pas obligatoire. Une restriction supplémentaire est l'interdiction d'employer l'héritage entre les classes. Par conséquent, le polymorphisme est supprimé.

Diagramme d'objets Le deuxième diagramme structurel utilisé est le diagramme d'objets. Ce dernier modélise une instanciation du diagramme de classes et représente une photographie à un moment donné du système. Ainsi un lien étroit existe avec le diagramme précédent. Les objets présents dans le diagramme sont des instanciations des classes et les liens sont des instanciations des associations.

L'état défini au travers de ce diagramme est couramment référé comme état initial. Malgré ce nom, cet état peut correspondre à n'importe quel état valide du système. Ainsi dans le cas d'un distributeur de billet de banques, cet état peut correspondre à l'attente qu'un utilisateur s'authentifie ou à l'état ou un utilisateur identifié choisit un montant à retirer.

Une propriété importante de ce diagramme est l'obligation de contenir l'ensemble des objets utiles pour décrire l'état mais également employés lors des différents comportements du système. Ainsi dans le cas d'un distributeur de billet de banques, si les différents comportements nécessitent trois utilisateurs alors les objets représentant ces utilisateurs doivent être présents dans le diagramme. Par conséquent, les opérations des classes sont incapables de créer ou de supprimer des objets.

Diagramme d'états-transitions Le langage permet d'employer un diagramme comportemental, le diagramme d'états-transitions. Ce diagramme est optionnel dans le cadre de notre démarche. Les états du diagramme modélisent des états partiaux du système. Les transitions encodent une évolution du modèle. Une transition est définie par un événement, une garde et une action. Les événements sont dans ce langage des appels d'opérations.

Le langage UML4MBT impose des contraintes afin d'interdire les états parallèles. De plus, les gardes des transitions sont modifiées pour les rendre mutuellement exclusives. Cette modification rend les comportements encodés déterministes dans ce diagramme.

Ainsi les comportements du systèmes sont encodés par les transitions du diagramme d'états-transitions, les opérations du diagramme de classes ou plus généralement une combinaison des deux.

Propriétés Une première propriété est que le langage UML4MBT modélise uniquement des comportements déterministes. La deuxième propriété qui découle des restrictions sur les types et sur la création/suppression des objets est que le nombre d'états du modèle est fini.

2.3.2 Langage OCL4MBT

Le langage UML4MBT a permis de définir la structure et le comportement du modèle. Cependant pour pouvoir raisonner mathématiquement sur le modèle et supprimer les ambiguïtés, il est nécessaire d'employer un langage formel. Dans le cas d'UML, c'est le plus souvent le langage OCL (Objet Constraint Language) [RG98] qui tient ce rôle. De façon similaire, le langage OCL4MBT [BGL+07] permet de formaliser les comportements du modèle.

Le langage OCL4MBT n'est pas simplement un sous-ensemble d'OCL car la sémantique de ce langage n'est pas systématiquement respectée. OCL4MBT est employé dans deux situations : décrire une condition d'activation d'une action ou présenter une série d'actions à réaliser. Si OCL est adapté à la description de conditions, l'écriture des actions est plus simple dans un langage impératif. OCL4MBT résout cette contradiction en utilisant deux contextes : **assignation** pour les actions et **évaluation** pour les contraintes.

Contexte d'assignation Dans le contexte d'assignation, OCL4MBT est un langage impératif. Ainsi une formule, écrite en OCL4MBT dans ce contexte, décrit une liste d'instructions. Ce contexte contredit la sémantique usuelle d'OCL. Il est employé lors de la définition des postconditions des opérations du diagramme de classes et des actions des transitions du diagramme d'états-transitions. De part la nature du contexte, les opérateurs renvoient obligatoirement la valeur booléenne vraie. L'exemple $a = 1 \wedge a = 2$ se lit *la variable a prend la valeur 1 puis cette variable prend la valeur 2*. Dans cet exemple, les affectations renvoient la valeur vraie car une affectation ne peut pas échouer. Ce contexte autorise un nombre plus restreint d'opérateurs par rapport aux opérateurs autorisés dans le contexte d'évaluation. Par exemple, l'utilisation de l'opérateur *or* est interdite.

Contexte d'évaluation Dans le contexte d'évaluation, OCL4MBT est un langage déclaratif. Dans ce contexte, OCL4MBT respecte la sémantique usuelle du langage OCL. Ce contexte est utilisé lors de la spécification des préconditions des opérations du diagramme de classes, des gardes des transitions du diagramme d'états-transitions et des conditions présentes dans certains opérateurs tels l'opérateur *if-then-else*. L'exemple précédent $a = 1 \wedge a = 2$ se lit *la variable a vaut-elle 1 et la variable a vaut-elle 2*. Ainsi dans ce contexte, cette formule retournera systématiquement la valeur fausse lors de son évaluation.

Expressivité Les opérateurs du langage OCL4MBT sont listés dans le tableau 2.1. La colonne signature de la méthode précise sur quels types d'expressions l'opérateur

est applicable. Le dernier type de la signature donne le type de retour de l'opérateur. Les types employés sont les types booléen (*Bool*), entier (*Int*), objet (*Object*), classe (*class*) et ensemble d'objets (*Set*). La notation *A* peut être remplacée par n'importe quel type sauf le type classe. Tous les opérateurs sont disponibles dans le contexte d'évaluation. Seuls les opérateurs ayant le symbole √ dans la dernière colonne sont utilisables dans le contexte d'assignation.

TABLE 2.1 – Opérateurs du langage OCL4MBT

Opérateur	Symbole	Signature	Assignation
Égalité	=	A A Bool	√
Inégalité	<>	A A Bool	
Disjonction	*or*	Bool Bool Bool	
Conjonction	*and*	Bool Bool Bool	√
Disjonction exclusive	*xor*	Bool Bool Bool	
Négation	*not*	Bool Bool	
Inférieure stricte	<	Int Int Bool	
Inférieure	<=	Int Int Bool	
Supérieur stricte	>	Int Int Bool	
Supérieure	>=	Int Int Bool	
Addition	+	Int Int Int	
Soustraction	−	Int Int Int	
Opposé	−	Int Int	
Multiplication	*	Int Int Int	
Division entière	*div*()	Int Int Int	
Valeur absolue	*abs*()	Int Int	
Modulo	*mod*()	Int Int Int	
Maximum	*max*()	Int Int Int	
Minimum	*min*()	Int Int Int	
Existence d'objet	*oclIsUndefined*()	Object Bool	√
Collection d'objets	*allInstances*()	Class Set	
Cardinalité	*size*()	Set Int	
Inclusion	*includes*()	Set Object Bool	√
Inclusion multiple	*includesAll*()	Set Set Bool	√
Exclusion	*excludes*()	Set Object Bool	√
Exclusion multiple	*excludesAll*()	Set Set Bool	√
Ensemble vide	*isEmpty*()	Set Bool	√
Ensemble non-vide	*notEmpty*()	Set Bool	
Quantificateur existentiel	*exists*()	Set Bool Bool	
Quantificateur universel	*forAll*()	Set Bool Bool	√
Extraction d'objet	*any*()	Set Bool Object	

2.4 Synthèse

Une des méthodes pour s'assurer qu'un logiciel respecte les exigences spécifiées dans le cahier des charges est d'exécuter une série de tests. Si les tests sont incapables de vérifier une absence complète d'erreur, ils permettent de garantir un degré de confiance dans le fonctionnement du logiciel.

Le test est un terme regroupant des procédures répondant à des objectifs variés (montée en charge, critère de sécurité, conformité des fonctionnalités au cahier des charges, ...) intervenant dans les différentes phases du cycle de développement et s'appuyant sur une connaissance précise du logiciel (totale, limité au code, limité au cahier des charges, ...).

Parmi ces procédures, le test à partir de modèle est distingué car cette procédure est celle suivie dans le cadre de notre démarche. Cette procédure appartient à la catégorie des tests fonctionnels en boîte noire. Elle a de nombreux avantages tels un degré important d'automatisation ou un surcoût faible dans le cadre de campagnes de tests répétées.

Pour employer cette procédure, il est nécessaire de disposer d'un langage de modélisation, d'une couverture du modèle par des cibles de test, d'un générateur de tests capable d'interpréter le modèle et d'explorer l'espace de recherche constitué par les états du modèle et d'un exécuteur des cas de test sur l'implémentation.

Parmi ces besoins, le cas du langage de modélisation est traitée. Notre démarche emploie deux langages, UML4MBT et OCL4MBT dérivées de UML et OCL et créés spécifiquement pour modéliser les comportement d'un système en vue de la génération de tests. De plus, l'exécuteur n'est pas nécessaire dans notre démarche car le processus d'exécution des tests sur l'implémentation n'est pas géré. Notre démarche s'interrompt avec la création des informations suffisantes pour cette exécution.

Chapitre 3

Outils et méthodes pour le test à partir de modèles

Sommaire

Pour guider les choix de notre démarche de génération de tests à partir de modèles, ce chapitre fournit une analyse de l'existant en terme d'outils et de méthodes. En particulier, les possibilités offertes pour couvrir le modèle, générer des tests grâce à l'interprétation du modèle et à l'exploration de l'espace de recherche constitué des états du modèle et la représentation des cas de test sont décrites.

Dans un premier temps, une présentation des principaux outils s'inscrivant dans le cadre du MBT est donnée afin d'explorer les différentes méthodes existantes. L'objectif est en particulier de déterminer comment générer des tests depuis un modèle comportementale écrit en UML4MBT / OCL4MBT et d'étudier les méthodes de couvertures existantes.

Dans un second temps, les procédés de vérification et de génération de tests pour les modèles UML et OCL qui ne sont pas intégrés dans des outils MBT. L'objectif de cette étude est d'analyser comment un raisonnement formel sur les modèles peut être construit et quelles sont les limites associées.

Finalement, une synthèse résumant les enseignements tirés de ces analyses est présentée.

3.1 Outils pour le test à partir de modèles

Pour donner un état de l'art des méthodes de générations de tests à partir de modèles, les principaux outils de ce domaine sont décrits dans cette section. Ces descriptions permettent dans un second temps d'établir une synthèse de ces méthodes et d'étudier leur influence sur notre démarche.

3.1.1 Description des outils

Cette sous-section décrit brièvement les outils les plus répandus dans le test à partir de modèles. La liste est établie en utilisant notamment les études [UPL12], [SL10] et [BFS05]. En particulier, la description de chaque outil précise la méthode de modélisation et de génération de tests.

AETG [CDFP97]

L'outil Automatic Efficient Test Generator (AETG) appartient à la famille du test combinatoire et du test en boite noire. Un test combinatoire regroupe les valeurs des paramètres d'entrées du système dans des couples, des triplets ou des n-uplets. L'idée est de réduire les combinaisons car leur nombre s'accroît rapidement avec le nombre de paramètres et de valeurs sous l'effet de l'explosion combinatoire. Ainsi un système ayant 4 paramètres pouvant chacun prendre 3 valeurs a $3^4 = 81$ combinaisons. Regrouper ces valeurs par couple permet de diminuer le nombre de combinaisons à 9.

Cet outil utilise un modèle dédié au test qui modélise l'environnement du système sous test. Le modèle est écrit dans un langage propre à l'outil. Le langage repose principalement sur deux entités : *field* et *relation*. Les *field* représentent les paramètres d'entrées du système. Les *relation* contiennent une liste de *field* et pour chacun un ensemble de valeurs valides et invalides. Ainsi un test valide est une combinaison de valeurs valides. Un test invalide est une combinaison de valeurs où au moins une est invalide.

AGEDIS [HN04]

Automated Generation and Execution of test suites for DIstributed componed-based Software (AGEDIS) est un projet, sponsorisé par l'union européenne, visant à établir une intégration d'outils au sein d'un même framework pour établir des modèles comportementaux de systèmes distribués, générer des tests, exécuter des tests et analyser des tests. Un avantage important de cet outil est son architecture modulaire. Ainsi il est possible d'intégrer des outils commerciaux comme AsmL Test tool [BGN+03] de Microsoft en son sein.

Il utilise un langage de modélisation nommé AML pour Agedis Modeling Tool. Ce langage est un profil pour UML 1.4 pour créer des modèles comportementaux. Le diagramme de classes précise la structure du modèle. Le diagramme d'objets modélise l'état initial du système. Les diagrammes d'états-transitions encodent la

dynamique du système. Dans ce dernier, les comportements sont exprimés dans le langage Intermediate Format (IF) [BGM02].

La génération des cas de test utilise les algorithmes de l'outil TGV [JJ05]. La concrétisation de ces cas est supportée par des fichiers XML reliant les interfaces de l'implémentation du système avec les opérations du modèle. L'outil dispose également d'un outil d'exécution nommé Spider. Après l'exécution, les tests sont analysés et catalogués selon les catégories des erreurs mises en évidence.

AsmL Test Tool [BGN+03]

Cet outil, développé par Microsoft, s'appuie sur une formalisation du modèle sous la forme de machine à états abstraits pour générer des tests. Une machine à états abstraits est une généralisation des machines à états finis, automates reconnaissant exactement des langages rationnels, à des structures de données arbitraires. Cet outil est capable de générer des valeurs d'entrées pour les paramètres des opérations, des machines à états finis depuis une machine à états abstraits, des séquences d'opérations pour le test depuis les machines à états finis et d'exécuter des tests sur l'implémentation du système.

Le langage de modélisation est AsmL (Abstract State Machine Language). Ce langage combine le paradigme des machines à états abstraits avec le langage .NET. Ainsi le langage contient les entités classiques de .NET telles que les classes, les interfaces, les méthodes ou les propriétés. Les machines calculent les valeurs des variables après l'exécution d'un pas. L'affectation de ces valeurs a lieu simultanément.

Le processus le plus complexe est la conversion de la machine à états abstraits (ASM) en des machines à états finis (FSM). Dans le cas général, une ASM a une infinité d'états atteignables. L'idée est de les grouper avec une relation d'équivalence dans des super-états. Ces relations ne sont pas toujours suffisantes et c'est pourquoi une condition précisant si un état est pertinent dans le cadre de la génération de tests peut être ajoutée. Cette étape a besoin d'être guidée par une intervention humaine pour obtenir un résultat intéressant.

CASE Tool Autofocus [LP00]

Autofocus est un outil graphique qui se concentre sur les systèmes distribués. Pour simuler les comportements du modèle, un modèle exécutable est construit grâce à la conversion du modèle de test dans un langage de programmation logique par contraintes. L'outil pour fonctionner a besoin des spécifications des cas de test, d'un modèle de test et d'une implémentation.

Un des avantages de l'outil est d'autoriser différents types de spécifications pour les cas de test. Le premier type concerne les spécifications fonctionnelles. Les séquences d'opérations sont encodées à l'aide de machine à états non déterministes. Le deuxième type de spécification s'appuie sur la structure du modèle pour générer des tests applicables à plusieurs composants. Le troisième et dernier type appartient aux tests statistiques. Dans ce cas, des séquences d'opérations aléatoires sont générées et un critère d'évaluation sélectionne les séquences à conserver.

AutoLink [SEG+96]

Autolink est un outil de génération de tests développé à l'institut pour les télécommunications à Lübeck et à Telelogic AB à Malmö. Il est principalement utilisé pour établir des tests de conformité pour des protocoles de communication. Il permet d'obtenir des suites de tests au format TTCN-2 [GHR+03] depuis un modèle écrit en SDL [SSD97] (Specification and Description Language) et des objectifs de tests représentés au travers de diagrammes MSC [SSD97] (Message Sequence Charts).

Les objectifs de tests peuvent être construits automatiquement par exploration de l'espace de recherche formé par les états du modèle, être créés par observation des comportements du système lors de simulations, ou être écrits explicitement par l'ingénieur de test. Ensuite ces objectifs sont combinés avec le modèle pour obtenir des suites de tests au format TTCN-2.

BZ-Testing-Tools [LPU02] et Certify-It

BZ-Testing-Tools (BZ-TT) est un environnement pour le test à partir de modèles développé notamment par le laboratoire d'informatique de l'université de Franche-Comté. Ces caractéristiques principales sont le support de modèles écrit dans le langage B ou Z, la production des cas de test aux limites pouvant être positifs ou négatifs, une exécution des tests et une intégration poussée des outils.

La méthode de génération repose sur la construction d'une séquence d'opérations reliant l'état initial du modèle à un état encodant une cible de test. La construction s'effectue en testant la validité de l'exécution de chaque opération. Pour réduire la taille des états, les valeurs des variables sont limitées aux bornes de leurs domaines. Cette méthode s'appuie sur un solveur de contraintes ensemblistes, nommé CLPS-B [BLP02], qui a été enrichi pour gérer également les spécifications Z en plus de celles écrites en B.

La poursuite des travaux de recherche ont conduit à industrialiser cette approche dans un outil commercial, nommé Certify-It, au sein de la société Smartesting. Ce nouvel outil utilise des modèles écrits dans des standards industriels tels les langages UML/OCL [BGL+07] ou BPMN [WM08] (Business Process Modeling Notation).

Conformance Kit [KWKK91]

Conformance Kit est un outil commercial développé par KPN Research. Il génère des suites de tests au format TTCN-2 depuis une modélisation du système qui utilise des machines à états finis étendus (EFSM). Ces machines étendent les FSM en autorisant les transitions à être exprimées par une expression booléenne qui spécifie les conditions d'activation de la transition.

Les cas de test sont dérivés des EFSMs grâce à un des trois algorithmes suivants : visite des transitions, visite des partitions ou aléatoire. L'algorithme aléatoire fournit des séquences d'opérations aléatoires. La visite des transitions fournit des séquences couvrant les différentes transitions. La visite des partitions donne des séquences qui mettent le EFSM dans l'état initial puis exécutent une suite d'opérations dans le

but d'activer la transition à tester. L'état obtenu après l'exécution de la séquence est testé pour s'assurer de sa validité.

Par la suite cet outil a été étendu par PHillips Automated Conformance Tester (PHACT) [HFT00] en permettant l'autorisation des cas de test sur l'implémentation du système sous test.

GATeL [MA00]

GATeL est un outil pour tester des programmes réactifs. Le langage de modélisation employé est Lustre [HCRP91]. Ce dernier est un langage de spécification et de programmation pour les systèmes réactifs articulé autour des flots de données synchrones. L'entité de base du langage est le cycle qui est une unité de temps d'une horloge globale. Dans un cycle, des relations précisent les valeurs des variables de sorties en fonction des variables d'entrées.

Pour générer des cas de test, GATel précise l'état à atteindre avec un ensemble de contraintes. Ensuite des contraintes supplémentaires sont appliquées pour représenter des invariants ou des états à rencontrer avant d'arriver à l'état final. Finalement, GATeL utilise la programmation logique par contraintes pour obtenir une séquence reliant l'état à tester à l'état initial du modèle. La séquence est ainsi calculée à l'envers.

JTorX [Bel10]

JTorX est une ré-implémentation de l'outil TorX [TB03] contenant de nouvelles fonctionnalités. Cet outil est disponible librement et permet de générer des tests à la volée depuis un modèle écrit sous la forme d'un système de transition avec des labels (LTS). La théorie ioco implémentée dans l'outil est présentée dans l'article [Tre08]. Quand l'outil accède au modèle, il ne considère que la partie nécessaire pour le test recherché. Le choix des cas de test est guidé par des objectifs de tests. Ces derniers sont des automates ou des LTS.

Le modèle peut être écrit dans de nombreux langages tels GraphML [BELP10], GraphViz ou Aldebaran [Fer88]. Le nombre est accru par la possibilité d'utiliser les adaptateurs de TorX pour utiliser d'autres langages tels LOTOS [BB87] ou μCRL [GP95].

La génération des cas de test est réalisée par l'exploration de l'espace de recherche défini par le modèle. L'exploration commence à l'état initial du modèle. Ensuite l'outil choisit s'il veut observer le système ou envoyer un stimulus. En effet dans un système LTS, chaque transition est soit une observation soit un stimulus. Dans le cadre de ioco, le système est capable de recevoir tous les stimuli et le système sous test est capable de recevoir l'ensemble des observations. Le choix peut être guidé par les contraintes exprimées dans les objectifs de tests.

JTorX est capable de communiquer directement avec les programmes utilisant les entrées et sorties standards (stdout, stdin). Pour cela les noms des transitions correspondent aux méthodes des interfaces de l'implémentation. Sinon une couche d'adaptation doit être développée.

JUMBL [Pro03]

L'outil J Usage Model Builder Library (JUMBL) a pour objectif d'aider à la construction et l'analyse de modèles, la génération de cas de test, leur exécution et l'analyse des résultats dans le cadre du test statistique. Dans ce cadre, le modèle utilisé pour le test représente l'utilisation du système et non son comportement. L'idée est de construire des profils d'utilisations correspondant aux utilisations courantes du système. Par contre ce type de modèle ne peut pas fournir d'oracle pour les tests.

Les modèles sont écrits en The Modeling Language (TML) [Pro00]. Ce langage a été créé spécifiquement pour employer les chaînes de Markov comme base de la modélisation. Les chaînes de Markov permettent de préciser une probabilité sur l'usage des transitions. Ainsi les transitions les plus utilisées sont couvertes par des tests en priorité.

ParTeG [Wei09]

Le Partition Test Generator (ParTeG) est un plug-in pour Eclipse capable de générer des cas de test depuis un modèle UML 2.0. Le modèle peut utiliser les diagrammes de classes, d'objets et d'états-transitions. Ils sont créés depuis le modeleur open-source Topcased [FGC+06]. Le langage OCL est utilisé pour exprimer les contraintes.

Pour générer les cas de test, ParTeG transforme le modèle en un arbre de transition. Une branche de l'arbre est alors équivalent à un cas de test. Le nom partition vient du fait que les valeurs prises pour les tests sont les bornes des domaines constituées par les conditions d'activations des branches.

Les cas de test sont concrétisés sous la forme de test unitaire Java dans le format JUnit.

Conformiq [conb]

Conformiq propose un outil commercial de génération de tests à partir de modèles. Cet outil utilise le concept d'exécution symbolique pour la génération des cas de test. Le modèle utilise le diagramme d'états-transitions du langage UML et un langage propriétaire Qtronic Modeling Language (QML) pour ajouter des contraintes sous forme textuelle. Ce langage est une surcouche de Java.

L'outil fournit les tests dans les formats les plus courants tel que TTCN. La modélisation et la génération sont réalisées au travers de plug-in Eclipse. L'outil mesure également la couverture et la pertinence des tests générés grâce à différents métriques.

SpecExplorer [VCG+08]

SpecExplorer est un outil développé par Microsoft Research et intégré dans l'environnement de développement Visual Studio. Il est dédié aux tests sur des systèmes réactifs orientés objets en boite noire. La modélisation utilise le langage AsmL ou Spec# pour décrire le comportement du système au travers d'une machine à états

abstraits. Spec# [BLS05] est une extension du langage C# et est intégré dans le framework .NET. Ce langage modélise la spécification en décrivant les variables d'états et les règles de mises à jour de ces fonctions.

La première étape lors de l'utilisation de SpecExplorer est la création d'un graphe fini représentant les différents états du modèle depuis une machine à états abstraits pouvant prendre potentiellement une infinité d'états. Ce graphe est construit par exploration. Cette dernière essaie d'atteindre de nouveaux états depuis l'état initial en activant les méthodes dont les préconditions sont satisfaites. Ce processus se répète pour construire le graphe. Les paramètres des méthodes sont générés par des outils dédiés.

Pour limiter la taille de ce graphe, il est possible d'utiliser des filtres. L'outil peut filtrer les paramètres en réduisant leur valeur à des ensembles finis et pertinents. Il peut également limiter les méthodes autorisées en s'appuyant sur une connaissance du modèle fournie par l'utilisateur. Une autre possibilité est de conserver uniquement les états répondant à un prédicat donné. L'utilisateur peut aussi diriger l'exploration en indiquant une taille à la séquence d'états et en attribuant un ordre dans le choix des méthodes. Un dernier filtre est d'employer des super-états regroupant plusieurs états en utilisant des classes d'équivalences fournies par l'utilisateur.

Ensuite les cas de test sont générés en utilisant le graphe ainsi construit. Les tests sont concrétisés sous la forme de code C#. Cette phase de génération peut être en ligne ou hors ligne.

TGV [JJ05] et STG [CJRZ02]

TGV est un outil de génération de tests en boite noire développé par Verimag et IRISA Rennes. L'outil est construit autour de la modélisation Input Output Labeled Transition System (IOLTS) qui est une évolution du modèle LTS. Dans ce type de modèle, les transitions sont réparties dans trois catégories qui contiennent respectivement les actions sur le système exécutables par l'utilisateur, les méthodes pour observer l'état du système et les actions internes non déclenchables directement par l'utilisateur. Les relations de conformité sont de type ioco à l'instar de l'outil JTorX.

En entrée, l'outil prend une spécification du modèle encodant les comportements du système sous test, un critère de sélection et des options de configurations. Le critère de sélection peut être aléatoire, issue d'un critère de couverture, d'un objectif de tests décrit par un automate ou une combinaison de ces possibilités. Les options de configurations permettent de raffiner les résultats obtenus en posant des contraintes supplémentaires sur le modèle de spécification ou le critère de sélection.

TGV est intégré au sein de CAESAR/ALDEBARAN Development Package (CADP [FGK+96]), un framework pour simuler et tester des protocoles de communications. Un des avantages de TGV est la possibilité d'utiliser des langages variés pour la modélisation grâce à une conception modulaire. Ainsi TGV accepte le langage Lotos [BB87] au travers des APIs fournies par CADP, le langage SDL [SSD97] avec l'aide de l'outil ObjectGéode SDL tool [Fer88], IF [BGM02] avec l'outil AGEDIS [HN04] ou le langage UML restreint aux diagrammes de classes, d'objets, de déploiements et d'états-transitions grâce à l'outil UMLAUT [HJLGP99].

La méthode de génération des cas de test abstraits est constituée de cinq étapes.

1. La classification des comportements en fonction de leur acceptation vis-à-vis de l'objectif de tests et de la spécification. Cette étape revient à effectuer un produit synchrone.
2. Ensuite les comportements visibles sont extraits et les états n'étant pas observés par l'utilisateur sont marqués. Une détermination est réalisée par identification de méta-états ayant plusieurs transitions sortantes avec le même label.
3. Parmi les comportements visibles, il faut en sélectionner pour créer des cas de test.
4. Les comportements sélectionnés précédemment respectent l'ensemble des propriétés en regard d'un cas de test sauf la contrôlabilité. En effet il est nécessaire de gérer les inconsistances potentiellement introduites par les choix offerts par la présence de différentes entrées ou sorties.
5. Finalement, les cas de test sont synthétisés et formatés. TGV peut utiliser un format sous forme de graphe fourni par le framework CADP ou le format TTCN.

L'outil le plus proche de TGV est JTorx. Une comparaison sur une même étude de cas a conclu à des performances proches. Cependant TGV est à privilégier pour tester des erreurs précises et JTorx pour des tests intensifs sans but spécifique.

L'outil Symbolic Test Generation (STG) est construit sur les résultats obtenus par TGV. L'idée est de répondre aux problèmes créés par l'explosion de l'espace de recherche générée par l'énumération des valeurs des variables et l'obtention de tests difficilement compréhensibles par des humains. L'idée est de rester avec un niveau d'abstraction supplémentaire pour représenter les variables. Ainsi l'énumération de leurs valeurs n'est pas nécessaire pour obtenir des cas de test. Pour cela, STG utilise une modélisation de type Input Output Symbolic Transition System (IOSTS).

STG est ensuite capable de concrétiser les tests et de les exécuter sur le système sous test. Ainsi STG peut créer un programme C++ représentant les tests. La sélection des tests est issue d'un objectif de tests ou d'un critère de couverture [Jér09].

3.1.2 Comparaison et synthèse

Le premier critère de différentiation pour les outils présentés dans la sous section précédente est le choix du langage de modélisation.

- La première possibilité est d'utiliser un langage propre à l'outil comme dans le cas de l'outil AETG. Cette méthode permet d'adapter le langage à la méthode de génération mais réduit grandement la compatibilité et la ré-utilisation des modèles. De plus, elle impose de fournir des outils de modélisation propres et de former spécifiquement les modélisateurs. Ces contraintes sont incompatibles avec l'objectif de notre démarche qui est d'assurer une compatibilité avec des modèles existants et de limiter les besoins en formation des modeleurs.
- La deuxième possibilité est d'utiliser des langages prévus ou répandus dans des domaines scientifiques ou industriels particuliers. Ainsi le langage Lustre,

employé par GATeL, est dédié à la modélisation et programmation de systèmes réactifs. Le langage SDL, utilisé par TGV et AutoLink, permet de décrire formellement les protocoles de communications. De plus, il s'agit d'un standard de l'union internationale des télécommunications. Notre démarche ne s'inscrivant pas dans un domaine spécifique, ce type de langage n'est pas utilisable.

– La troisième possibilité est de choisir un langage de modélisation généraliste. Parmi les outils présentés, deux cas se détachent. Le premier cas sont les langages construits autour des machines à états abstraits ou finis tel le langage B, utilisé dans l'outil BZTT, ou le langage AsmL employé par SpecExplorer et l'AsmL Test Tool. Cependant ces différents langages n'ont pas exactement la même notation et leur utilisation n'est pas répandue dans le monde industriel. Le deuxième cas est le langage UML et ses différents profils. Ce choix a été fait par une majorité des outils présentés (AGEDIS, Certify-It, ParteG, Conformiq, TGV). Ce choix s'explique par la position du langage comme standard industriel, la disponibilité de nombreux outils de modélisations tels RSA, Topcased, BoumL, Modélio, ..., une capacité de personnalisation au travers de la création de profils et une représentation visuelle. Les diagrammes UML utilisés dans les outils sont restreints majoritairement aux diagrammes de classes pour décrire la partie statique du modèle, d'objets pour préciser les valeurs d'entrées des cas de test et d'états-transitions pour modéliser la partie dynamique. Cependant le langage UML n'est pas suffisant pour décrire sans ambiguïté le comportement du système. C'est pourquoi un langage de contrainte est systématiquement associé au langage UML. Malheureusement, il ne semble pas parmi les outils se dégager un consensus. Ainsi l'outil ParTeg utilise OCL, Certify-It une variante d'OCL, Conformiq son propre langage QML et AGEDIS ou TGV le langage IF. Ainsi pour les qualités présentées ci-dessus, le langage UML est le plus approprié pour notre démarche.

Le deuxième critère de différenciation est la méthode de génération des tests. Les outils appartiennent majoritairement à l'une des trois catégories suivantes :

– Le premier système est construit autour de la programmation logique par contrainte (CLP). Les outils BZTT et GaTeL appartiennent à cette catégorie.
– La deuxième possibilité est de s'appuyer sur les machines à états finis (FSM). Les outils SpecExplorer, Autolink, AsmL Test Tool et Conformance Kit ont fait ce choix. Cette méthode s'associe plus facilement à un langage de modélisation employant des machines à états abstraits. En effet, il est possible à l'aide d'une phase de concrétisation de dériver les FSMs des ASMs.
– La troisième catégorie est de se conformer à la théorie des systèmes de transitions avec des labels (LTS). Les outils TGV, STG, JtorX et AGEDIS utilisent cette théorie. Cette méthode est plus naturellement associée au langage de modélisation UML. Le choix du langage UML pour modéliser le système dans le cadre de notre démarche, nous entraîne naturellement vers une manipulation du modèle sous la forme de système de transitions.

Dans le cas de la seconde ou de la troisième possibilité, l'évolution du système peut être représentée sous la forme d'un graphe. Le problème devient alors de construire ce graphe totalement ou partiellement puis de l'exploiter pour générer des tests.

Dans le cadre de notre démarche, le caractère automatique interdit de compter sur l'utilisateur pour construire ce graphe. De plus, afin de limiter le problème d'explosion combinatoire, notre démarche construit et utilise dans le même temps la partie du graphe utile pour le cas de test recherché.

Pour le format des cas de test, la plupart des outils n'ayant pas un langage propre utilise une des versions du langage TTCN. C'est notamment le cas des outils Autolink, Conformance Kit, Conformiq et TGV. Cependant pour obtenir une sortie utilisable, il suffit qu'un cas de test contienne les informations suivantes : la séquence de méthodes exécutées par le système, l'objectif du test et la satisfiabilité.

Le dernier point sur lequel comparer les outils, le type de tests, n'a pas de conséquence directe sur notre démarche car nous considérons que les différents objectifs de tests nous sont fournis par un processus externe à notre démarche. Ainsi, que ces objectifs résultent d'une création manuelle ou d'une génération automatique visant à satisfaire un critère de couverture, notre démarche les considère indifféremment.

3.2 Méthodes de tests et de vérifications de modèle UML et OCL

La vérification ou la génération de tests d'un système modélisé à l'aide du langage UML sont des problèmes complexes. Ainsi la plupart des encodages s'appuyant sur UML emploient le diagramme de classes pour représenter la partie statique du système et l'article [BCD05] indique que le fait de raisonner sur ce type de diagramme a une complexité EXPTIME-complet. De plus, pour lever les ambiguïtés contenues dans UML, un langage de contrainte est également utilisé. Le langage de contrainte le plus utilisé dans ce cadre est OCL. L'expressivité gagnée par cet ajout rend dans le cas général le problème indécidable.

Le processus de vérification ou de validation se décompose en deux étapes principales. En premier, le modèle UML/OCL est converti sous la forme d'un modèle mathématique tel qu'une formule logique. En deuxième, un outil est utilisé pour raisonner sur ce modèle mathématique. Par conséquent, le choix de ce modèle découle naturellement du choix de l'outil.

3.2.1 Vérification des modèles

Une première série de techniques se concentre sur la validation du modèle et non de son implémentation. Cette validation permet de s'assurer par exemple que des opérations ne sont pas activables car leur précondition n'est jamais satisfiable ou qu'une instanciation de l'ensemble des classes est impossible. L'intérêt de ces travaux pour notre démarche est de proposer des méthodes de conversion de modèles en des représentations mathématiques et de donner une indication sur les limites de l'expressivité des modèles.

Un de ces outils est Kodkod [TJ07]. Kodkod est un moteur relationnel généraliste capable de trouver une solution à un problème exprimé dans un langage de contraintes qui combine la logique du premier ordre avec l'algèbre relationnel et les

fermetures transitives. Après avoir analysé et appliqué des transformations et simplifications, une représentation du problème sous forme normale conjonctive est soumise à un prouveur SAT. Cet outil est employé par le plugin OCL2Kodkod [KHG11] de l'environnement UML-based Specification Environment [GBR07] (USE). L'idée de ce plugin est de vérifier la conformité du modèle en fonction de propriétés définies dans la spécification. Pour cela, le plugin détermine si des instances du modèle conformes aux propriétés peuvent exister sans violer les contraintes définies dans les diagrammes.

Une autre approche connexe est le travail présenté dans l'article [SWK+10] où un modèle UML/OCL est converti dans une formule booléenne qui est ensuite soumise à un prouveur SAT. De manière similaire, l'idée est de s'assurer du respect de certaines propriétés telles la consistance qui s'assure de la génération d'états cohérents du système et l'indépendance qui vérifie que les contraintes OCL ne sont pas redondantes. Pour cela, des états du système vérifiant à la fois ces propriétés et les contraintes OCL doivent être créés. Pour s'assurer de la génération, le nombre d'états du système doit être fini. Ainsi le nombre d'objets du modèle est limité et les domaines des variables sont finis.

Une troisième approche se distingue par l'emploi de solveur CSP pour assurer une vérification formelle de modèles UML/OCL. Un outil respectant cette approche est UMLtoCSP [CCR07]. Cet outil permet de vérifier si un modèle respecte plusieurs propriétés telles que l'indépendance ou la consistance. Le processus a deux étapes. D'abord le modèle est converti en problème de type CSP puis ce problème est soumis à un solveur CSP. L'existence d'une solution signifie qu'une instanciation du modèle respectant les propriétés est possible. Le solveur CSP utilisé dans ce processus est ECLiPSe [AW06].

Une approche différente repose sur l'utilisation d'un prouveur de théorèmes. L'article [KFdB+05] présente un processus conforme à cette approche. L'utilisation de logique de plus haut niveau permet potentiellement de vérifier des modèles ayant un nombre infini d'états. Cependant le langage UML est restreint aux diagrammes de classes contenant uniquement des associations ayant une multiplicité maximum de 1 et ne contenant pas de classes génériques. Le prouveur employé dans ce processus est PVS [SORSC01].

Une dernière approche consiste à convertir le modèle dans un autre langage de modélisation disposant déjà d'un outil d'analyse. C'est l'approche retenue dans le cas de l'outil UML2Alloy [ABGR07] qui convertit un modèle UML/OCL en un modèle écrit avec Alloy. Ce langage [Jac02] est conçu pour encoder les propriétés structurelles et faciliter leur analyse. Il est ainsi capable de représenter la structure de manière textuelle et graphique et leur évolution via des formules logiques. Cependant il n'est pas prévu pour décrire l'interaction dynamique entre les objets ou leur implémentation. Une fois la conversion effectuée, l'analyse est possible grâce à l'Alloy Analyzer [Ana].

3.2.2 Tests à partir de modèles

Une deuxième série de techniques utilise les modèles pour générer des tests afin de s'assurer de la conformité de l'implémentation en regard de la spécification. Dans ce cadre, le modèle est toujours considéré comme valide. Les outils présentés dans la section 3.1 se conforment à cette technique et ils ne seront pas discutés à nouveau ici.

Une première approche emploie les prouveurs de théorèmes. L'article [BKLW11] décrit une méthode pour générer des tests à partir de spécifications encodées dans un modèle UML/OCL à l'aide du prouveur HOL/Isabelle. Ce prouveur a été enrichi pour traiter l'OCL avec un environnement dédié HOL-OCL [BW08]. Cette démarche est conçu pour s'intégrer dans le système interactif de génération de tests HOL-GENTEST [BW09]. Elle est décomposée en deux phases. Dans la première, le problème exprimé sous forme de formules logiques est partitionné en plusieurs cas par une transformation dans une forme normale. La deuxième est d'analyser ces cas de test pour déterminer une instance qui correspond aux données du cas de test recherché.

Une approche alternative est d'utiliser des solveurs CSP. Ainsi le travail décrit dans l'article [CCR09] enrichit le processus de vérification à l'aide de l'outil UML-toCSP au comportement dynamique du modèle. Le solveur est ainsi capable de tester des comportements décrits par le biais de préconditions et de postconditions. Pour l'instant, le processus est limité aux opérations du diagramme de classes pour modéliser le comportement du système. De plus, ce processus s'applique uniquement à une seule évolution du modèle via l'exécution d'une opération. Il est ainsi incapable de considérer des séquences d'opérations.

Une autre approche similaire à notre démarche est décrite dans l'article [SWD11]. Ainsi le langage UML supporte des restrictions proches de celles définies dans UML4MBT, seuls les diagrammes d'objets, de classes et d'états-transitions sont autorisés. L'idée est de convertir le modèle dans des formules compréhensibles par les prouveurs SMT. Ces formules utilisent la théorie des vecteurs de bits pour encoder les contraintes. Cet encodage oblige à limiter la vérification d'une propriété à des séquences d'opérations de longueurs fixes déterminées arbitrairement. La procédure ne considère pas la génération de multiple tests dans le cas d'une campagne de tests et les synergies possibles dans ce cas. Le prouveur SMT utilisé dans les expérimentations est Boolector [BB09].

3.3 Synthèse

L'analyse des outils et méthodes pour la génération de tests à partir de modèle nous a confortés dans le choix de nos langages de modélisation et nous a guidés dans nos choix pour notre démarche. Le langage UML4MBT utilisé conjointement avec le langage OCL4MBT permet de bénéficier d'un écosystème riche et de modéliser correctement les comportements du système. La majorité des outils ayant choisi des langages construits autour de UML/OCL emploie une représentation de l'évolution

du système sous la forme de système de transitions. C'est pourquoi notre démarche s'articulera autour de ce type de représentation.

De plus cette représentation combinée avec l'utilisation de trois diagrammes (objets, classes et états-transitions) issus de UML offre un accès à de nombreuses méthodes de couvertures du modèle. Il est ainsi envisageable de créer manuellement des cibles de test pour couvrir un comportement ou un scénario, c'est à dire une suite de comportements. Mais une génération automatique par l'application d'un critère de couverture est également possible. Par exemple, des cibles peuvent être générées pour couvrir l'ensemble des comportements élémentaires ou l'ensemble des transitons du système de transitions. Par conséquent, la création des cibles de test est exclue du périmètre de notre démarche.

Pour la génération de tests, quatre outils se détachent : le prouveur SAT, le prouveur SMT, le solveur CSP, et le prouveur de théorèmes. Le prouveur SAT est exclu de notre démarche car son expressivité est extrêmement limité et tout problème pouvant être résolu par ce dernier peut également être résolu par un prouveur SMT. Hors ce dernier propose une expressivité nettement supérieure grâce à la disponibilité de plusieurs théories. Le prouveur de théorème offre une excellente expressivité notamment grâce à des logiques d'ordres supérieures mais le processus de résolution n'est pas automatique. Par conséquent, cet outil n'est pas adapté à notre cas. Le dernier outil, le solveur CSP, s'insère sans difficulté dans notre démarche en terme de performance et d'expressivité. De plus les limitations du langage de modélisation UML4MBT qui conduisent à obtenir un nombre d'états finis du modèle sont appropriés pour une utilisation du solveur CSP. Pour conclure, la génération de tests dans notre démarche s'appuiera sur les solveurs CSP et les prouveurs SMT.

Les méthodes de génération de tests employant ces outils et reposant sur une représentation de type système de transitions s'articulent sur la construction de séquences de transitions validant les objectifs de tests. Cette construction provient sur une exploration de l'espace de recherche constitué par les états du modèles. Parmi les méthodes présentées pour réduire l'explosion combinatoire résultant de cette exploration et considérant l'utilisation d'un prouveur SMT, la piste la plus prometteuse semble être de construire uniquement les états rencontrés lors de la création des séquences d'animations.

Chapitre 4

Outils retenus pour la génération de tests

Sommaire

L'analyse des outils et méthodes de génération de tests à partir de modèles a conduit à sélectionner deux outils, les solveurs CSP et les prouveurs SMT, pour raisonner formellement sur les modèles. Ces derniers permettent ainsi d'interpréter les évolutions du modèle par une exploration de l'espace de recherche constitué par les états possibles des modèles.

Ces outils ont pour objectif de résoudre un problème exprimé sous forme de formules logiques contraignant un ensemble de variables. Ainsi une résolution est la détermination de l'existence d'une solution qui est une valuation des variables respectant l'ensemble des contraintes. Pour employer correctement ces outils dans le cadre de notre démarche, une analyse des caractéristiques de ces derniers est nécessaire. En particulier, les limites du processus de résolution et de l'expressivité doivent être déterminées.

L'objectif de ce chapitre est donc de donner une description de ces outils afin de répondre aux questions suivantes :

- Quel est l'algorithme à la base du processus de résolution du problème décrit sous forme de formules logiques ?
- Quelles sont ses caractéristiques principales, notamment en terme d'expressivité ?

– Comment communiquer de manière standardisée et avec quelle expressivité ?
– Quels sont les outils existants ?

4.1 Les prouveurs SMT

Dans notre démarche visant à générer des tests à partir de modèles, un des outils utilisés est le prouveur SMT qui permet de résoudre des formules écrites dans la logique du premier ordre. Cette section commence ainsi par présenter cet outil et expliciter son fonctionnement. Dans un deuxième temps, le langage SMT-lib qui est un standard pour communiquer avec des prouveurs est introduit.

4.1.1 Définition et fonctionnement

Pour commencer, le fonctionnement du prouveur SAT sera expliqué. Ceci facilitera la compréhension du prouveur SMT. En effet ce dernier a évolué depuis un prouveur SAT.

Prouveur SAT

Définition 7 (Prouveur SAT) *Un prouveur SAT est un outil capable de déterminer s'il existe une solution satisfaisant une formule propositionnelle qui est une expression booléenne écrite uniquement avec des variables booléennes, des parenthèses et les opérateurs de disjonction, de conjonction et de négation.*

L'algorithme central autour duquel est construit les prouveurs SAT est nommé Davis-Putnam-Logemann-Loveland [DLL62] (DPLL). Il s'agit d'un algorithme complet pour résoudre des problèmes écrits à l'aide de formules logiques propositionnelles en forme normale conjonctive à l'aide d'un processus de backtracking.

Pour expliciter le fonctionnement d'un prouveur SAT, le détail de la résolution de la formule $(\overline{a} \vee b) \wedge (\overline{c} \vee d) \wedge (\overline{e} \vee \overline{f}) \wedge (\overline{b} \vee \overline{e} \vee f)$ est donné ci-dessous. Dans cet exemple, le terme modèle correspond à une solution valide pour le problème.

$$\texttt{Opération} : \texttt{Problème} \parallel \texttt{Modèle} \tag{4.1}$$

$$D : (\overline{a} \vee b) \wedge (\overline{c} \vee d) \wedge (\overline{e} \vee \overline{f}) \wedge (\overline{b} \vee \overline{e} \vee f) \parallel \varnothing \tag{4.2}$$

$$P : (\overline{a} \vee b) \wedge (\overline{c} \vee d) \wedge (\overline{e} \vee \overline{f}) \wedge (\overline{b} \vee \overline{e} \vee f) \parallel a \tag{4.3}$$

$$D : (\overline{a} \vee b) \wedge (\overline{c} \vee d) \wedge (\overline{e} \vee \overline{f}) \wedge (\overline{b} \vee \overline{e} \vee f) \parallel a \wedge b \tag{4.4}$$

$$P : (\overline{a} \vee b) \wedge (\overline{c} \vee d) \wedge (\overline{e} \vee \overline{f}) \wedge (\overline{b} \vee \overline{e} \vee f) \parallel a \wedge b \wedge c \tag{4.5}$$

$$D : (\overline{a} \vee b) \wedge (\overline{c} \vee d) \wedge (\overline{e} \vee \overline{f}) \wedge (\overline{b} \vee \overline{e} \vee f) \parallel a \wedge b \wedge c \wedge d \tag{4.6}$$

$$P : (\overline{a} \vee b) \wedge (\overline{c} \vee d) \wedge (\overline{e} \vee \overline{f}) \wedge (\overline{b} \vee \overline{e} \vee f) \parallel a \wedge b \wedge c \wedge d \wedge e \tag{4.7}$$

$$B : (\overline{a} \vee b) \wedge (\overline{c} \vee d) \wedge (\overline{e} \vee \overline{f}) \wedge (\overline{b} \vee \overline{e} \vee f) \parallel a \wedge b \wedge c \wedge d \wedge e \wedge \overline{f} \tag{4.8}$$

$$P : (\overline{a} \vee b) \wedge (\overline{c} \vee d) \wedge (\overline{e} \vee \overline{f}) \wedge (\overline{b} \vee \overline{e} \vee f) \parallel a \wedge b \wedge c \wedge d \wedge \overline{e} \tag{4.9}$$

$$S : (\overline{a} \vee b) \wedge (\overline{c} \vee d) \wedge (\overline{e} \vee \overline{f}) \wedge (\overline{b} \vee \overline{e} \vee f) \parallel a \wedge b \wedge c \wedge d \wedge \overline{e} \wedge f \tag{4.10}$$

La première étape de la résolution (ligne 4.2) est de prendre la décision, notée D, d'affecter la valeur vraie à la variable a. Ensuite les conséquences de cette décision sont propagées, opération notée P, à la ligne 4.3. Ainsi pour que la proposition $\overline{a} \vee b$ soit vérifiée avec a étant vraie, la variable b doit également prendre la valeur vraie. Ce processus de décision et de propagation est répété des lignes 4.4 à 4.7.

La ligne 4.8 introduit une nouvelle opération notée B qui est le backtrack. En effet, la propagation conduite sur la proposition $\overline{e} \vee \overline{f}$ résulte en une solution incompatible avec la proposition $\overline{b} \vee \overline{e} \vee f$. Dans ce cas, l'algorithme revient à la dernière décision prise (ligne 4.6) et inverse cette dernière. Ainsi la valeur de e passe de vraie à fausse.

Finalement le processus alternant décisions et propagations reprend jusqu'à l'obtention d'une solution ou la conclusion que le problème n'est pas satisfiable. Dans l'exemple, la solution trouvée est $a \wedge b \wedge c \wedge d \wedge \overline{e} \wedge f$.

Le processus décrit ci-dessus est le processus de base. Les prouveurs SATs récents intègrent des fonctionnalités avancées pour améliorer les performances tels le backjump, une généralisation du backtrack pour revenir à des points de choix antérieurs ou l'apprentissage qui mémorise les causes d'erreurs pour limiter les conflits similaires.

Prouveur SMT

Définition 8 (Prouveur Satisfiability Modulo Theory (SMT)) *Un problème SMT est exprimé au travers d'une formule logique du premier ordre combinant différentes théories telles la théorie des entiers ou des réels. Ainsi un problème SMT est une généralisation d'un problème SAT où les variables booléennes sont remplacées par des prédicats utilisant différentes théories. Un prouveur SMT tente de déterminer si un problème SMT a une solution et dans l'affirmative retourne cette dernière sous la forme d'une valuation des variables.*

Le fonctionnement d'un prouveur SMT repose sur la combinaison d'un prouveur SAT et d'un T-solveur. Le T-solveur permet de résoudre une formule logique du premier ordre en s'appuyant sur une théorie donnée. Il existe deux approches basiques. La première traduit le problème en un problème équivalent compréhensible par un prouveur SAT. Cependant cette approche nécessite des formules de conversion complexes pour chacune des théories traitées.

La deuxième approche, employée dans l'exemple ci-dessous, convertit uniquement la partie de la formule retournée comme solution du problème par le prouveur SAT. Ainsi dans l'exemple, le prouveur doit trouver une solution si elle existe à la formule :

$$(\underbrace{a < b}_{p_1} \vee \underbrace{a < c}_{p_2}) \wedge (\underbrace{f(a) = f(c)}_{p_3} \vee \underbrace{a + b = c}_{p_4}) \wedge \underbrace{a + 2 = b + 2}_{p_5}$$

La première étape est de représenter la formule dans la logique propositionnelle. Dans notre cas, les propositions p_1, p_2, p_3, p_4 et p_5 sont créées.

$$SAT : (p_1 \vee p_2) \wedge (p_3 \vee p_4) \wedge p_5 \rightarrow S_1 \tag{4.11}$$
$$TSolver : S_1 = (p_1 \wedge p_3 \wedge p_5) \rightarrow \varnothing \tag{4.12}$$
$$SAT : (p_1 \vee p_2) \wedge (p_3 \vee p_4) \wedge p_5 \wedge \overline{S_1} \rightarrow S_2 \tag{4.13}$$
$$TSolver : S_2 = (p_1 \wedge \overline{p_3} \wedge p_4 \wedge p_5) \rightarrow \varnothing \tag{4.14}$$
$$SAT : (p_1 \vee p_2) \wedge (p_3 \vee p_4) \wedge p_5 \wedge \overline{S_1} \wedge \overline{S_2} \rightarrow S_3 \tag{4.15}$$
$$TSolver : S_3 = (\overline{p_1} \wedge p_2 \wedge p_3 \wedge p_5) \rightarrow \varnothing \tag{4.16}$$
$$SAT : (p_1 \vee p_2) \wedge (p_3 \vee p_4) \wedge p_5 \wedge \overline{S_1} \wedge \overline{S_2} \wedge \overline{S_3} \rightarrow S_4 \tag{4.17}$$
$$TSolver : S_4 = (\overline{p_1} \wedge p_2 \wedge \overline{p_3} \wedge p_4 \wedge p_5) \rightarrow a = 1, b = 1, c = 2 \tag{4.18}$$

La formule propositionnelle ainsi créée est soumise au prouveur SAT à la ligne 4.11. Ce dernier utilise l'algorithme DPLL, présenté ci-dessus, pour obtenir une réponse nommée S_1. Cette réponse ignore les prépositions n'influant pas sur la satisfiabilité de la formule. Dans notre cas, il s'agit de p_2 et de $p4$.

Dans un deuxième temps, la formule logique correspondante à la solution retournée par le prouveur SAT est soumise au T-solveur. Ce dernier s'appuie sur une théorie spécifique pour tenter de la résoudre. Dans notre cas, il s'agit d'une théorie traitant les fonctions, les inéquations et l'arithmétique basique. En fonction de la théorie sélectionnée, le prouveur SMT peut être incapable d'aboutir à une réponse. Dans notre exemple, le T-solveur conclut à l'absence de solution.

L'échec du T-solveur à retourner une solution conduit à enrichir la formule propositionnelle par l'ajout de la négation de la solution trouvée par le prouveur SAT durant la première étape du processus. Cet ajout garantit que le prouveur SAT retournera une solution différente.

Ces étapes continuent jusqu'à l'obtention d'une solution de la part du T-solveur où la conclusion par le prouveur SAT que la formule est insatisfiable. Dans cet exemple, le prouveur trouve la solution : $a = 1, b = 1, c = 2$.

L'exemple montré ici suit une procédure basique. Les prouveurs SMT récents incluent des améliorations significatives telles que le backjump, la capacité d'extraire le sous ensemble responsable de l'insatisfiabilité ou la possibilité d'employer le T-solveur incrémentalement durant la constitution d'une solution par le prouveur SAT.

4.1.2 Langage SMT-lib

Pour décrire un problème SMT, un langage spécifique, nommé SMT-lib [BST10], a été créé. Ce langage permet d'introduire une couche d'abstraction par rapport à différents prouveurs et ainsi assure une compatibilité avec ces prouveurs. SMT-lib est également prévu pour définir les logiques utilisées pour raisonner sur les problèmes. Les logiques définissent les théories, opérateurs et axiomes utilisés pour déterminer une solution. Ainsi l'expressivité du prouveur dépend directement du choix de la logique.

Architecture d'un problème SMT-lib La description d'un problème SMT à l'aide du langage SMT-lib comporte quatre parties principales :

Méta La première partie regroupe les informations aidant à résoudre le problème ou servant à le décrire mais non utilisables par le prouveur. Les commentaires et le nom du modèle rentrent dans cette dernière catégorie. Pour aider à la résolution, le problème peut spécifier une logique ou l'existence d'une solution. Ces informations peuvent permettre à des prouveurs d'optimiser leur processus de résolution.

Type La deuxième partie contient la déclaration des types. Si les types prédéfinis dans les théories tels les entiers ou les booléens sont manipulables par défaut, les types non-interprétés doivent être déclarés avant d'être employés.

Fonction La troisième partie est la définition des fonctions. Une fonction SMT-lib peut être de différentes arités et ainsi encoder à la fois la représentation d'une variable ou d'une application.

Contrainte La quatrième et dernière partie liste les différentes contraintes modélisant le problème. Ainsi une solution est une valuation des fonctions respectant l'ensemble des contraintes.

Prouveurs et compétition La création d'un langage commun pour des prouveurs SMT a permis d'évaluer leur performance à l'aide de benchmarks standardisés en fonction d'une logique. Cette évaluation a lieu dans une compétition nommée SMT-Comp [BDOS08]. Les benchmarks utilisés et les résultats de la compétition conduite en 2012 sont disponibles sur le site [SC].

Parmi les concurrents généralistes, gérant un nombre important de logiques, de ces dernières années, nous pouvons citer :

Z3 [DMB08] est un prouveur SMT développé par Microsoft Research et intégré dans des outils professionnels comme Visual Studio via l'outil d'analyse de code en boite blanche Pex [TDH08]. Z3 a l'avantage de supporter l'ensemble des logiques, d'avoir un développement soutenu et une documentation conséquente. Une utilisation gratuite est possible dans un cadre non commercial.

CVC3, CVC4 [BT07], [BCD+11] Cooperating Validity Checker 3 et sa ré-écriture CVC4 est un prouveur académique développé conjointement par l'université de New-York et l'université de l'Iowa. Ces prouveurs supportent l'ensemble des logiques définies dans le langage SMT-lib. Ces outils sont librement utilisables dans le cadre académique et commercial.

MathSat5 [CGSS13] est la dernière version du prouveur MathSat qui a été développée conjointement par l'université de Trento et le FBK-IRST. Ce prouveur supporte la plupart des logiques. De plus, il est librement utilisable dans le cadre académique et industriel.

Logiques SMT-lib La figure 4.1 liste les logiques définies dans le langage SMT-lib sous la forme d'un graphe. Dans ce dernier, les théories sont représentées avec des carrés. Les théories définissent sur quels types la logique peut raisonner. Ces

types sont les entiers, les réels, les vecteurs de bits, les booléens et les tableaux. Dans le graphe, un fils est capable de résoudre le même de type de problème que le père. Ainsi plus une logique est proche des théories, plus elle est restrictive en terme d'expressivité.

Les noms des logiques ont été créés par composition. Les préfixes suivants, apparaissant dans leur ordre d'écriture, existent :

QF précise que la logique n'emploie pas de quantificateur.

A raisonne également sur des tableaux.

UF autorise l'emploi de fonctions non interprétées.

Les suffixes donnent une indication sur les opérateurs et les théories employées. Ils existent les suffixes suivants :

IDL,RDL logique différentielle sur les entiers (resp. les réels).

LIA,LRA arithmétique linéaire sur les entiers (resp. les réels).

NIA,NRA arithmétique sur les entiers (resp. les réels).

LIRA formule linéaire avec des tableaux à une ou deux dimensions à valeurs réelles et indices entiers.

NIRA formule linéaire avec des tableaux de tableaux à valeurs réelles et indices entiers.

BV logique sur les vecteurs de bits.

AX logique étendue sur les tableaux.

Ainsi la logique *QF_UFLIA* raisonne à partir de formules d'arithmétique linéaire sur les entiers comprenant des fonctions non interprétées et n'ayant pas de quantificateur.

Exemple de problème SMT-lib Pour illustrer le langage SMT-lib, un problème de coloriage de graphe est donné ci-dessous. L'objectif est de colorier une carte contenant la France, l'Allemagne, la Belgique, le Luxembourg et les Pays-Bas sans employer une même couleur pour deux pays adjacents à l'aide des couleurs rouge, jaune, verte et bleue.

La logique utilisée pour décrire le problème est la logique sans quantificateur avec des fonctions et des types non-interprétés. Après avoir déclaré la logique employée, un nouveau type est créé pour représenter les couleurs. Ensuite quatre variables sont créées pour encoder les différentes couleurs et une contrainte permet de s'assurer qu'elles prennent des valeurs différentes.

L'étape suivante est de créer une variable pour chacun des pays. La première série de contraintes s'assure que chaque pays utilise une des couleurs définies précédemment. La deuxième série de contraintes précise pour chaque pays ses voisins.

La dernière étape est de demander au prouveur SMT de vérifier si le problème a une solution et dans ce cas de la fournir. Un exemple de solution serait : la France et les Pays-Bas sont en jaune, la Belgique est en rouge, l'Allemagne est en bleu et le Luxembourg est en vert.

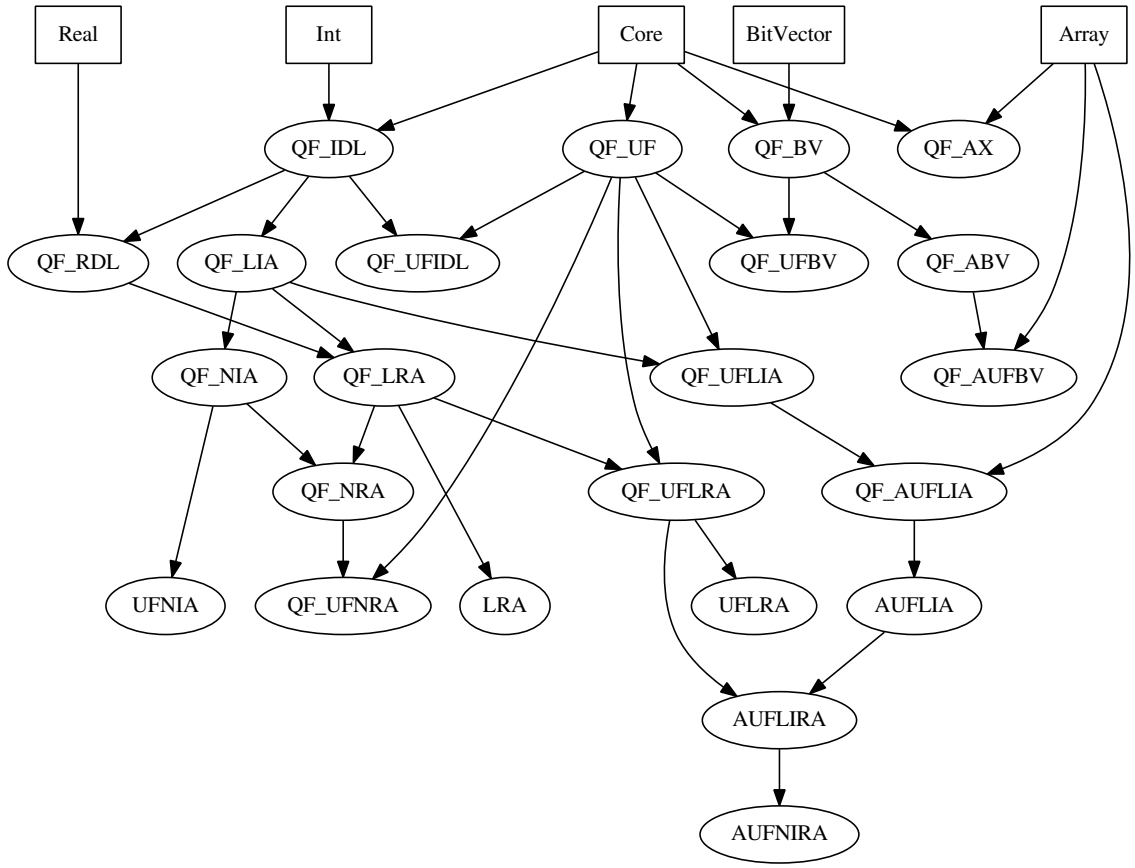


FIGURE 4.1 – Liste des logiques définies dans le langage SMT-lib

```
Problème de coloriage de graphe

(set-logic QF\_UF)

(declare-sort Couleur 0)

(declare-fun Rouge () Couleur)
(declare-fun Bleu () Couleur)
(declare-fun Vert () Couleur)
(declare-fun Jaune () Couleur)

(assert (distinct Rouge Vert Jaune Bleu))

(declare-fun Allemagne () Couleur)
(declare-fun France () Couleur)
(declare-fun Belgique () Couleur)
(declare-fun PaysBas () Couleur)
(declare-fun Luxembourg () Couleur)

(assert (or (= Allemagne Rouge) (= Allemagne Bleu) (= Allemagne Vert)
            (= Allemagne Jaune)))
```

```
(assert (or (= France Rouge) (= France Bleu) (= France Vert)
            (= France Jaune)))
(assert (or (= Belgique Rouge) (= Belgique Bleu) (= Belgique Vert)
            (= Belgique Jaune)))
(assert (or (= PaysBas Rouge) (= PaysBas Bleu) (= PaysBas Vert)
            (= PaysBas Jaune)))
(assert (or (= Luxembourg Rouge) (= Luxembourg Bleu) (= Luxembourg Vert)
            (= Luxembourg Jaune)))

(assert (and  (not (= Allemagne France)) (not (= Allemagne Belgique))
              (not (= Allemagne PaysBas)) (not (= Allemagne Luxembourg))))
(assert (and  (not (= France Belgique)) (not (= France Allemagne))
              (not (= France Luxembourg))))
(assert (and  (not (= Belgique France)) (not (= Belgique PaysBas))
              (not (= Belgique Luxembourg)) (not (= Belgique Allemagne))))
(assert (and  (not (= PaysBas Belgique)) (not (= PaysBas Allemagne))))
(assert (and  (not (= Luxembourg Belgique)) (not (= Luxembourg Allemagne))
              (not (= Luxembourg France))))

(check-sat)
(get-model)
(exit)
```

4.2 Les solveurs CSP

Un deuxième outil employé dans notre démarche est le solveur CSP en domaines finis. Cette section suit une architecture similaire à la précédente. Ainsi la section commence par présenter cet outil et expliciter son fonctionnement. Puis dans un deuxième temps, un langage commun de description de problème CSP, MiniZinc, est introduit.

4.2.1 Définition et fonctionnement

Définition 9 (Problème de satisfaction de contraintes) *Un problème de type CSP est défini par un triplet (X,D,C) où :*

X *est l'ensemble des n variables du problème notées* $x_0, x_1, \ldots, x_n$.

D *est l'ensemble des domaines associés aux variables. Le domaine d'une variable* x_i *est noté* D_{x_i}.

C *contient l'ensemble des contraintes portant sur les variables du problème. Une contrainte* C_i *est définie par le couple* (X_i, R_i) *où* X_i *regroupe les variables utilisées par la contrainte et* R_i *est la relation précisant les valeurs possibles pour ces variables en regard de la contrainte.*

Définition 10 (Solveur CSP) *un solveur CSP est un outil capable de retourner une solution à un problème de type CSP. Une solution est une affectation de valeur*

à l'ensemble des variables respectant les contraintes du problème ou la conclusion que le problème est insatisfiable. Dans le cas de problèmes en domaines finis, les solveurs sont toujours capables de conclure.

Un CSP peut dans tous les cas être représenté par un graphe de contraintes car les contraintes n-aires peuvent être ré-écrites sous la forme de contraintes binaires. La solution la plus basique consiste à énumérer les solutions et à tester leur validité. La construction de ces solutions peut être incrémentale en appliquant par exemple un algorithme de backtracking. Cependant pour réduire le nombre de solutions potentielles à tester, il est possible d'employer des algorithmes de consistances portant sur les noeuds, les arcs [BHL03], les chemins... Après la réduction des domaines, une phase d'énumération est nécessaire pour obtenir une solution si le problème est satisfiable.

Si un problème comporte plusieurs solutions, un solveur CSP n'est pas capable à priori de les hiérarchiser et retourne la première solution trouvée. Un problème prenant en compte cette contrainte est un problème d'optimisation.

Définition 11 (Problème d'optimisation de contraintes) *Il est défini par un quadruplet (X,D,C,F) où (X,D,C) décrit le problème CSP et F est une fonction objective qui associe un nombre à une solution. Ainsi la solution retournée est celle minimisant ou maximisant ce nombre.*

Ce type de problème entraîne un surcoût par rapport à un problème CSP car potentiellement le solveur doit découvrir l'ensemble des solutions pour pouvoir les comparer et donc explorer une partie plus conséquente de l'espace de recherche. En effet, une partie de l'espace de recherche peut être ignorée uniquement si le solveur a la certitude qu'elle ne contient pas de solution ou des solutions inférieures à celles déjà obtenues.

Exemple solveur CSP Dans cet exemple, le solveur doit résoudre le problème CSP suivant où :

$$
\begin{aligned}
&X = \{a, b, c\} \\
&D = \{D_a = [2, 7], D_b = [1, 8], D_c = [-1, 10]\} \\
&C = \{a \geq b, a \leq c, a + b = c\}
\end{aligned}
$$

Pour la résolution, l'exemple utilise un algorithme de consistance aux bornes. L'application incrémentale des contraintes permet de réduire les domaines des variables.

$$
\begin{array}{rl}
\texttt{contrainte} & \| \texttt{propagation} \rightarrow \texttt{résultat} \\
a \geq b & \| a \geq 1 \rightarrow D_a = [2,7] \\
& \| 7 \geq b \rightarrow D_b = [1,7] \\
a \leq c & \| a \leq 10 \rightarrow D_a = [2,7] \\
& \| 2 \leq c \rightarrow D_c = [2,10] \\
a + b = c & \| 2 - 7 \leq a \leq 10 - 1 \rightarrow D_a = [2,7] \\
& \| 2 - 7 \leq b \leq 10 - 2 \rightarrow D_b = [2,7] \\
& \| 2 + 1 \leq c \leq 7 + 7 \rightarrow D_c = [3,10]
\end{array}
$$

La deuxième phase de la résolution est la génération d'une solution par énumération des valeurs des domaines des variables. Dans l'exemple, les valeurs aux bornes sont privilégiées.

$$
\begin{array}{rl}
\texttt{choix} & \| \texttt{contrainte} \rightarrow \texttt{résultat} \\
a = 2 & \| a \geq b \rightarrow D_a = \{2\}, D_b = [1,2], D_c = [3,10] \\
& \| a \leq c \rightarrow D_a = \{2\}, D_b = [1,2], D_c = [3,10] \\
& \| a + b = c \rightarrow D_a = \{2\}, D_b = [1,2], D_c = [3,4] \\
b = 1 & \| a \geq b \rightarrow D_a = \{2\}, D_b = \{1\}, D_c = [3,4] \\
& \| a \leq c \rightarrow D_a = \{2\}, D_b = \{1\}, D_c = [3,4] \\
& \| a + b = c \rightarrow D_a = \{2\}, D_b = \{1\}, D_c = \{3\}
\end{array}
$$

Le résultat retourné par le solveur est donc $a = 2$, $b = 1$ et $c = 3$. D'après le processus de résolution suivi, l'importance de l'ordre de traitement des contraintes et d'énumération des valeurs apparaît. Ainsi cet ordre conditionne en partie la vitesse du processus et la solution retournée.

Propriétés En résumé, un solveur CSP en domaines finis possède les propriétés suivantes :

- Le solveur est toujours capable de conclure ; soit une solution est retournée, soit le problème est insatisfiable.
- Les performances décroient avec l'augmentation du nombre de points de choix.
- L'ordre de traitement des contraintes et variables influe sur la vitesse de résolution et la solution trouvée.
- La pertinence de la solution peut être augmentée par l'utilisation d'une fonction objective dans le cadre d'un problème d'optimisation ou les choix faits par les algorithmes telle l'utilisation prioritaire des valeurs des bornes des domaines des variables durant la construction de la solution.

4.2.2 Langage MiniZinc

Pour exprimer un problème CSP de manière compréhensible pour différents solveurs CSP, un langage de standardisation a été proposé dans l'article [NSB+07]. Ce langage MiniZinc standardise la déclaration des variables, des contraintes et permet également de diriger le processus de résolution. Il s'agit d'un langage de haut-niveau destiné à un utilisateur humain. Ce langage est ensuite converti dans un langage de bas-niveau à destination des solveurs.

Architecture d'un problème Minizinc La description d'un problème CSP dans ce langage utilise les éléments suivants :

- Le premier élément est une directive d'inclusion qui permet de fractionner le problème en plusieurs fichiers.
- Le deuxième élément est la déclaration de variables. Ces variables ont une portée globale. Elles sont classifiées dans deux catégories : paramètre et décision. Dans la définition du problème, les paramètres ont une valeur assignée explicitement et les variables de décision obtiennent une valeur uniquement par le processus de résolution. Les variables ont un des types suivants : types de base (réel, entier, chaîne de caractère, booléen, annotation), entier ou réel borné, un tableau ou un ensemble. Seuls les types réel, entier ou booléen sont admis pour les variables de décisions.
- Un autre élément est l'assignation d'une valeur à une variable. S'il s'agit d'une variable de décision, cette assignation est équivalente à une contrainte.
- Les contraintes forment le cœur du modèle. Une contrainte est une formule logique qui doit être vérifiée pour obtenir une solution. Pour exprimer les contraintes, le langage dispose des opérateurs suivants : $\leftrightarrow$, $\rightarrow$, $\leftarrow$, $\vee$, $\wedge$, $\oplus$, $=$, $!=$, *in*, *subset*, *union*, *diff*, *superset*, *symdiff*, .., *intersect*, $++$, $+$, $-$, $*$, $/$, *div* et *mod*.
- Pour guider le choix d'une solution quand plusieurs existent, une fonction objective est présente. Il y a trois possibilités : une solution quelconque, une solution maximisant une expression arithmétique, une solution minimisant une expression arithmétique.
- Le format d'une solution n'est pas prédéfini. L'utilisateur peut le spécifier via des fonctions d'affichage.
- MiniZinc offre la possibilité de définir des prédicats afin de factoriser des contraintes complexes ou longues. Un exemple classique est le prédicat *all different*.
- Finalement le langage autorise l'usage d'annotations pour guider les choix faits par le solveur durant l'exploration de l'espace de recherche. En pratique, le choix d'une stratégie adaptée a une influence importante en terme de temps de résolution.

Solveurs et compétition A l'instar des prouveurs SMT implémentant le langage SMT-lib, il existe également une compétition pour les solveurs CSP utilisant le langage MiniZinc, le MiniZinc Challenge [SBF10]. Les résultats des dernières éditions

sont disponibles sur le site [min]. Dans ce concours, les benchmarks écrits en MiniZinc sont convertis dans un langage de plus bas niveau, FlatZinc, compréhensible par les concurrents. Les benchmarks soumis aux solveurs appartiennent à l'une des catégories suivantes : *FD Search* où le processus de résolution est guidé par les informations présentes dans le problème sous forme d'annotation, *Free search* où le solveur n'a pas de contrainte sur le processus de résolution et *Multi-core* qui est une résolution sans contrainte sur un processeur multi-cœur.

En s'appuyant sur les résultats à cette compétition au cours des dernières années, les solveurs suivants se distinguent :

Gecode [SLT06] est un solveur open-source de contraintes en domaines et ensembles finis. Les différents processus de résolutions sont proposés sous la forme d'une bibliothèque C++ multi-plateforme. Ce solveur a gagné les différentes épreuves du concours ces dernières années. En plus de MiniZinc, de nombreux langages de programmations (Java, JavaScript, C++, Lisp, . . .) sont utilisables au travers d'interfaces spécifiques.

JaCoP [jac] est également un solveur open-source de contraintes en domaines finis et sur les ensembles d'entiers. Ce solveur est implémenté dans le langage Java. Il est particulièrement employé dans le domaine de la création automatique de circuit électronique.

fzn2smt [BSV10] est un outil qui résout le problème en le convertissant dans le formalisme SMT-lib v1.2 pour le soumettre à un prouveur SMT qui par défaut est Yices [DDM06]. Ce solveur ne peut pas concourir dans la catégorie de la recherche guidée car les annotations prévues pour un solveur CSP n'ont pas de sens pour un prouveur SMT. De plus, les contraintes d'optimisations n'ayant pas d'équivalent direct, le processus de résolution dans ce cas revient à effectuer des appels successifs au prouveur en partitionnement les domaines des variables concernées par l'optimisation.

Exemple de problème MiniZinc Le problème donné ci-dessous est le même que celui servant à illustrer le formalisme SMT-lib à savoir un problème de coloriage de graphe à quatre couleurs portant sur une carte représentant la France, la Belgique, le Luxembourg, l'Allemagne et les Pays-Bas.

La première étape est la déclaration des variables et constantes. Dans notre cas, un tableau permet d'associer le nom d'une couleur à un chiffre compris entre 0 et 3 inclus. Ensuite une variable est définie pour chaque pays avec comme domaine les indices du tableau de couleurs.

La deuxième étape est de préciser quels sont les pays adjacents. Pour cela, une liste de contraintes utilisant l'inégalité suffit.

La troisième étape précise quel est l'objectif de résolution et le format de la solution si elle existe. Dans notre cas, l'objectif est de vérifier si le problème a une solution sans en privilégier une en cas de solutions multiples. Après résolution le solveur retourne : *France : jaune, Allemagne : Bleu, Luxembourg : vert, Belgique : rouge, Pays-Bas : jaune.*

```
Problème de coloriage de graphe

couleur = ["rouge", "vert", "jaune", "bleu"];

var 0..3: France;
var 0..3: Belgique;
var 0..3: PaysBas;
var 0..3: Allemagne;
var 0..3: Luxembourg;

constraint France != Belgique;
constraint France != Allemagne;
constraint France != Luxembourg;
constraint Belgique != PaysBas;
constraint Belgique != Allemagne;
constraint Belgique != Luxembourg;
constraint PaysBas != Allemagne;
constraint Allemagne != Luxembourg;

solve satisfy;

output ["France:", show(couleur[France]),
        ", Allemagne:", show(couleur[Allemagne]),
        ", Luxembourg:", show(couleur[Luxembourg]),
        ", Belgique:", show(couleur[Belgique]),
        ", Pays-Bas:", show(couleur[PaysBas])]
```

4.3 Synthèse

A l'issue de ce chapitre et cette partie, tous les outils et le périmètre de notre démarche de génération de tests à partir de modèles ont été présentés. En effet une démarche s'appuyant sur le MBT nécessite un langage de modélisation, une couverture du modèle, un générateur de tests et un exécuteur des cas de test.

Pour assurer la modélisation deux langages ont été retenus. Ils ont été décrits à la section 2.3. Le premier langage, UML4MBT, est un sous ensemble d'UML créé spécifiquement pour le test à partir de modèle. Ce langage permet de décrire le système au travers de trois types de diagramme. Le diagramme de classes est responsable de la modélisation de la partie statique du système. Le diagramme d'objets autorise une représentation de l'état initial du système et des valeurs valides. Le diagramme d'états-transitions est capable de modéliser la partie dynamique du système. Pour formaliser et lever les ambiguïtés du système, ses différents comportements sont décrits dans un second, OCL4MBT, dérivé d'OCL. Ces comportements sont représentés via les méthodes des classes et les transitions du diagramme d'états-transitions.

A l'issue de l'analyse des outils existants pour le MBT présentée dans le chapitre 3, le choix a été fait de rendre notre démarche indépendante aux différentes

options disponibles pour assurer la couverture du modèle. Ainsi tous les types de couvertures pouvant être exprimés en s'appuyant sur des modèles comportementaux écrits dans les langages UML4MBT et OCL4MBT sont compatibles avec notre démarche.

Pour constituer le générateur de tests de notre démarche, deux outils, décrits dans ce chapitre, sont employés. Le premier outil utilisé dans notre démarche pour raisonner sur un modèle mathématique est le prouveur SMT. Ce type de prouveur a pour objectif de déterminer si une formule logique du premier ordre est satisfiable et dans ce cas de retourner une solution. La particularité de ces solveurs est de pouvoir combiner différentes théories telles la théorie des entiers ou des tableaux pour aboutir à une conclusion. Pour utiliser indifféremment des solveurs différents, il existe un langage commun, SMT-lib, pour décrire les problèmes à résoudre. Le second outil employé pour manipuler notre modèle sous une forme formelle est le solveur CSP (Constraint Satisfaction Problem). Ce solveur est capable de déterminer si il existe une solution à un problème exprimé sous la forme d'une liste de contraintes portant sur des variables associées à des domaines finis. Pour s'assurer une indépendance vis à vis d'un solveur particulier, il existe un langage commun pour décrire ces problèmes nommé MiniZinc.

Le dernier point d'un processus MBT, l'exécution des cas de test, est externe au périmètre de notre démarche. Cette dernière s'interrompt avec la création des cas de test qui contiennent les informations nécessaires et suffisantes pour leur exécution sur l'implémentation du système et l'analyse du résultat.

Deuxième partie

Contributions - Tests, prouveurs et solveurs

Chapitre 5

Robot - Exemple fil rouge

Sommaire

Ce chapitre présente l'exemple qui sera régulièrement utilisé par la suite pour illustrer les différents points de notre démarche. Cette dernière génère des tests à partir d'un modèle écrit en UML4MBT/OCL4MBT grâce à l'utilisation d'un prouveur SMT ou d'un solveur CSP.

Cet exemple fil rouge est un système de transbordement de pièces entre des tapis roulants grâce à un robot muni d'une pince. Sa modélisation repose sur un diagramme d'objets et un diagramme de classes. Ce système a d'abord été choisi pour sa taille restreinte et sa complexité limitée qui permettent de saisir rapidement son fonctionnement. De plus, ses spécifications contiennent un objectif d'efficacité en regard du temps de transbordement d'une pièce. Cet objectif entraîne la création de cibles de test et de scénarios d'exécution intéressants. Finalement la modélisation met en exergue des spécificités liées aux langages UML4MBT/OCL4MBT.

La première partie de ce chapitre présente le système, en particulier ses composants et son fonctionnement. Cette partie introduit également les contraintes posées pour améliorer l'efficacité en interdisant les mouvements inutiles dans le cadre du transfert des pièces. La seconde partie est dédiée à la modélisation du système. En particulier, les contraintes de modélisation découlant du langage sont mis en avant et le comportement du modèle est décrit au travers du code OCL4MBT des opérations.

5.1 Description du système

Notre exemple fil rouge est un robot industriel utilisé pour transborder des pièces entre différents tapis roulants. La figure 5.1 est une représentation schématique du

système. Ce dernier est constitué de :

- Un tapis roulant, nommé "quai de chargement", amenant les pièces. Ce dernier est situé en bas et à gauche dans l'atelier. Les pièces arrivantes ne sont pas toutes semblables. Ils existent trois types de pièces, nommées T1, T2 et T3.
- Deux tapis roulants évacuant les pièces et nommés "quai de déchargement". Ces derniers sont positionnés en haut et de chaque coté de l'atelier. Le quai de déchargement gauche évacue les pièces de type T1 et T3. L'autre quai de déchargement est dédié aux pièces de type T2 et T3.
- Un robot capable de transborder les pièces entre le quai de chargement et les quais de déchargement. Le robot possède deux degrés de liberté. Il peut changer de côtés par une rotation à droite ou à gauche et de niveau par une translation haute ou basse. Durant la translation, le robot se déplace de manière à surplomber les quais de déchargement en position haute et le quai de chargement en position basse. Il est également capable de saisir les différents types de pièces à l'aide d'une pince.

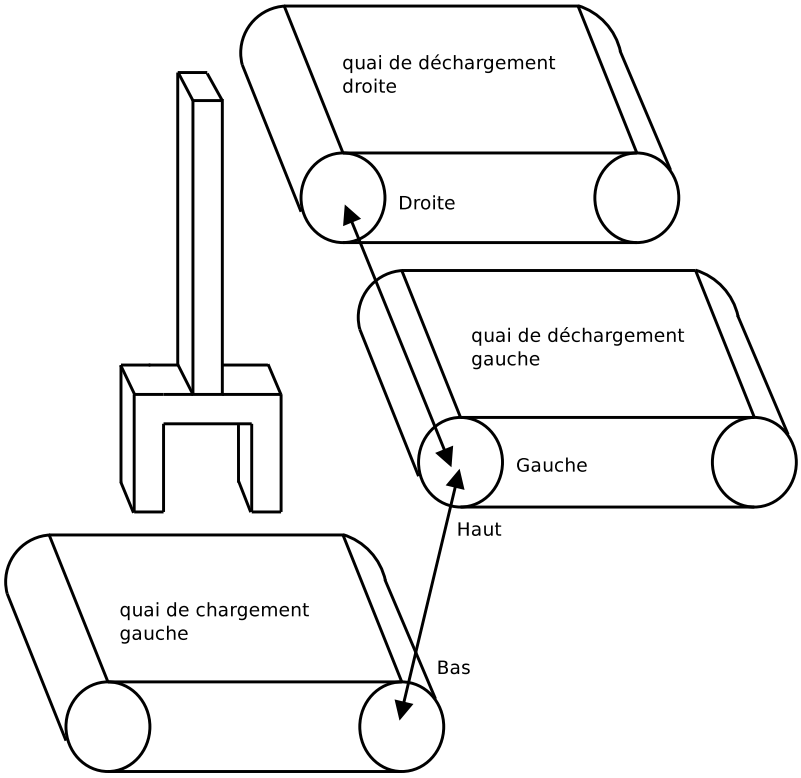


FIGURE 5.1 – Schéma du robot de transbordement

D'après les spécifications du système, dans le cas des pièces de type T3, le robot peut choisir indifféremment le quai de déchargement de gauche ou le quai de droite. Cependant une évacuation par le quai de droite nécessite un temps plus important que par le quai de gauche car le robot doit effectuer une rotation supplémentaire. En conséquence, afin de minimiser le temps de traitement des pièces, la contrainte suivante est ajoutée au système : *Les pièces de catégorie T3 seront évacuées par*

le quai de gauche si ce dernier est libre. Dans le cas contraire, les pièces seront évacuées par le quai de droite s'il est libre.

Une contrainte d'efficacité du système est également introduite. Pour cela, elle est intégrée dans les conditions d'activation des opérations, les éléments de contrôle de l'état de la pince et des quais. Ainsi une séquence constituée d'une suite de 10 rotations est interdite car le transfert des pièces n'a pas progressé durant cette séquence.

5.2 Modélisation

La modélisation du système est réalisée au travers les langages UML4MBT et OCL4MBT. Le modèle emploie un diagramme de classes et un diagramme d'objets pour décrire le système. Ces diagrammes sont présentés dans la figure 5.2. La modélisation s'articule autour des trois éléments principaux du système qui sont le robot, le quai de chargement et les quais de déchargement. Ces différents éléments sont modélisés au travers de classes. En particulier, la majorité de la logique fonctionnelle du système est décrite dans la classe modélisant le robot.

Une spécificité du langage UML4MBT est l'absence du type chaîne de caractères. En conséquence, les variables employant ce type lui substituent généralement des énumérations. Dans notre cas, la position du robot manipulateur est spécifiée au travers de deux énumérations, **position** pour la hauteur et **coté** pour la coordonnée horizontale. Les différents quais ont une position fixe ; le quai de chargement est en bas à gauche, les quais de déchargement sont en haut à droite et à gauche.

La modélisation des différentes catégories de pièces, nommées **T1**, **T2** et **T3**, nécessite un choix. La première solution, issue d'un point de vue objet sur le système, est d'utiliser une classe par catégorie de pièce héritant d'une classe abstraite commune. Cette solution rencontre deux difficultés liées au langage UML4MBT. En premier, le polymorphisme n'est pas supporté. Cependant il est possible de résoudre ce problème en se limitant à une seule classe contenant un attribut stockant la catégorie d'une pièce. La deuxième difficulté découle de la propriété du langage UML4MBT qui certifie que l'ensemble des instances utilisées doit être déclaré dans le diagramme d'objets. Ainsi puisque chaque élément du système peut contenir une pièce et qu'il existe trois catégories, tester l'ensemble des configurations nécessite la création de $5 * 3 = 15$ instances. En conséquence une autre solution a été retenue. Chaque classe d'un élément du système dispose d'un attribut spécifiant si une pièce est présente sur cet élément. Dans ce cas, l'attribut indique également sa catégorie. Cet attribut utilise une énumération **Piece** listant les catégories auxquelles une valeur **L** signifiant une absence de pièce a été ajoutée.

Le premier élément du système est le quai de chargement. Cet élément est représenté par une classe et une instance. Son comportement est retranscrit au travers d'une unique opération nommée **a_p**. Cette dernière fait apparaître une pièce du type choisi en paramètre sur le quai si ce dernier est inoccupé.

```
operation
  a_p(P : Piece)
```

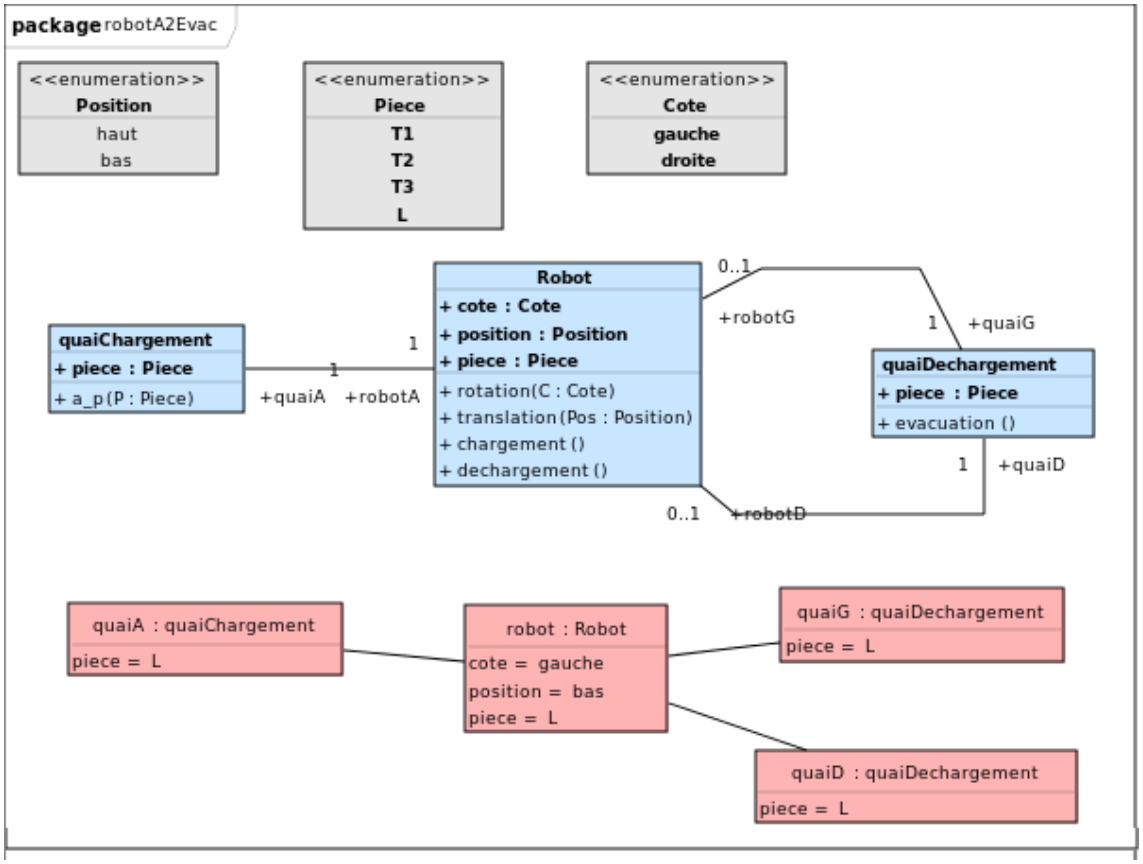


FIGURE 5.2 – Diagramme de classes et d'objets du modèle Robot

```
pre
  self.piece = Piece::L and P <> Piece::L
post
  self.piece = P
```

Le deuxième élément du système est le quai de déchargement. Les deux quais sont représentés au travers d'une classe et de deux instances. Le comportement de ces quais est retranscrit au travers d'une unique opération nommée **evacuation**. Cette dernière évacue une pièce présente sur le quai.

```
operation
  evacuation()
pre
  not (self.piece = Piece::L)
post
  self.piece = Piece::L
```

Le dernier élément du système est le robot manipulateur. Le robot est modélisé au travers d'une classe et d'une instance. Cette classe concentre la majorité de la logique du système. Les attributs de la classe mémorisent la position du robot et la présence ou absence d'une pièce. La première opération de cette classe, nommée **rotation**, est chargée de modifier la position horizontale du robot. Ainsi une rotation est possible si le robot tourne vers une position différente de celle actuellement occupée, si la

rotation est utile dans le cadre du transfert d'une pièce et si le robot est en position haute.

```
operation
  rotation(C : Cote)
pre
  (C = Cote::droite
  and self.position = Position::haut
  and self.cote = Cote::gauche
  and (self.piece = Piece::T2
    or ( self.piece = Piece::T3
      and self.quaiD.piece = Piece::L
      and not ( self.quaiG.piece = Piece::L ))))
or  (C = Cote::gauche
  and self.position = Position::haut
  and self.cote = Cote::droite
  and (self.piece = Piece::L
    or (self.piece = Piece::T3
      and self.quaiG.piece = Piece::L)))
post
  if
    C = Cote::droite
  then
    self.cote = Cote::droite
  else
    self.cote = Cote::gauche
  endif
```

La seconde opération, nommée **translation**, est chargée de modifier la position verticale du robot manipulateur. Cette translation est autorisée uniquement si la position est différente de celle actuellement occupée et si elle est utile. C'est à dire que le robot peut s'élever uniquement s'il tient une pièce et descendre si la pince est vide.

```
operation
  translation(Pos : Position)
pre
    (self.position = Position::bas
    and not (self.piece = Piece::L))
  or  (self.position = Position::haut
    and self.piece = Piece::L
    and self.cote = Cote::gauche)
post
  if
    Pos = Position::haut
  then
    self.position = Position::haut
  else
    self.position = Position::bas
  endif
```

La troisième opération, nommée **chargement**, est responsable de la saisie d'une pièce par le robot. Toujours dans un souci d'efficacité, le robot est autorisé à saisir

uniquement les pièces du quai de chargement. Par conséquent le robot doit être positionné en bas et à gauche.

```
operation
  chargement()
pre
  self.piece = Piece::L
  and not (self.quaiA.piece = Piece::L)
  and self.cote = Cote::gauche
  and self.position = Position::bas
post
  self.piece = self.quaiA.piece
  and self.quaiA.piece = Piece::L
```

La dernière opération, nommée **déchargement**, s'occupe d'ouvrir la pince du robot pour lâcher la pièce sur un des quais de déchargement. Par conséquent le robot doit être positionné en haut. De plus, dans le cas des pièces **T3** le quai gauche est privilégié s'il est inoccupé.

```
operation
  dechargement()
pre
  self.position = Position::haut
  and ((self.quaiG.piece = Piece::L
      and self.cote = Cote::gauche
      and (self.piece = Piece::T1
        or  self.piece = Piece::T3))
    or (self.quaiD.piece = Piece::L
      and self.cote = Cote::droite
      and (self.piece = Piece::T2
        or (self.piece = Piece::T3
          and not (self.quaiG.piece = Piece::L)))))
post
  if
    self.cote = Cote::gauche
  then
    self.quaiG.piece = self.piece
    and self.piece = Piece::L
  else
    self.quaiD.piece = self.piece
    and self.piece = Piece::L
  endif
```

Finalement, le modèle décrit un état initial au travers du diagramme d'objets. Dans cet état, aucune pièce n'est présente dans les différents éléments et le robot est positionné en bas et à gauche.

5.3 Synthèse

Dans ce chapitre, nous avons présenté l'exemple fil rouge qui consiste en un robot manipulateur chargé de transférer des pièces entre un tapis roulant d'arrivée et deux

tapis roulants de sortie. Le robot doit choisir le tapis d'évacuation en fonction du type de pièce et d'une contrainte qui consiste à minimiser le temps de transfert.

Ce système a mis en évidence les choix de modélisation découlant de spécificités du langage UML4MBT telles que l'absence de chaîne de caractères ou l'obligation de déclarer l'ensemble des instances utilisées par le modèle dans le diagramme d'objets.

Le modèle sera utilisé dans les chapitres suivants pour illustrer les différents points de notre démarche de génération de tests .

Chapitre 6

Processus de génération de tests à l'aide de SMT et de CSP (MBT)

Sommaire

Notre démarche s'inscrit dans le cadre du test à partir de modèle. Le but de ce chapitre est de présenter une vue d'ensemble des différentes phases de notre processus et de formaliser les notions employées. L'idée maîtresse reste classique dans le domaine du MBT [BFS05] ; il s'agit d'encoder dans un modèle mathématique compréhensible par un prouveur SMT ou un solveur CSP les comportements du système à tester. Ce travail se démarque par l'utilisation d'un prouveur pour générer des tests depuis un modèle UML dans le cadre de processus multi-thread et l'utilisation conjointe d'un solveur pour améliorer ces stratégies.

Pour ce faire, nous devons répondre aux questions suivantes : Comment animer un modèle UML4MBT dont le comportement est décrit au travers de contraintes OCL4MBT présentes dans les opérations du diagramme de classes et dans les transitions du diagramme d'états-transitions ? Comment guider la résolution d'une animation via un prouveur SMT pour en extraire des cas de test si l'animation réussit ? Quels types de cible de test sont possibles dans notre démarche ?

La première partie de ce chapitre décrit notre démarche pour générer des tests à partir d'un modèle grâce à son animation via un solveur CSP et/ou un prouveur SMT. Cette partie présente l'ensemble des phases du processus de manière succincte afin d'obtenir une vue d'ensemble de la démarche. Une explication détaillée de chaque phase sera fournie dans les chapitres suivants. La seconde partie est consacrée à une définition formelle du système de transitions et de son animation. La troisième partie présente les notions liées au test dans le cadre de notre système de transitions. Finalement la dernière partie développera une synthèse des informations présentées dans ce chapitre.

6.1 Démarche

Le but de notre démarche est la génération de tests depuis un modèle à l'aide d'un solveur CSP ou d'un prouveur SMT grâce à l'animation du système. L'idée majeure est de représenter l'animation et les cibles de test sous la forme d'une formule logique manipulable par des prouveurs SMT ou des solveurs CSP. Si cette formule est valide pour une valuation donnée des variables alors cette dernière permet la création d'un cas de test.

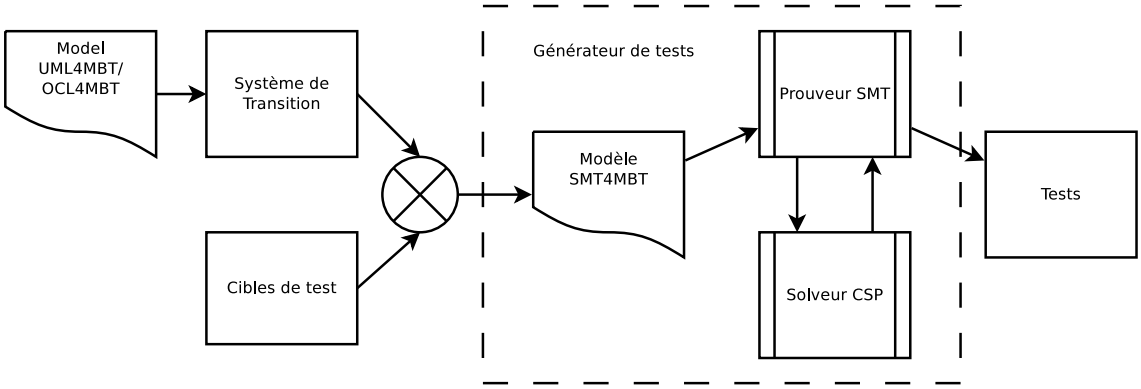


FIGURE 6.1 – Processus de génération de tests via l'animation d'un modèle vu au niveau de ses composants

Notre démarche est présentée au travers du schéma 6.1 et comporte les éléments suivants :

Modèle est la modélisation du système écrite dans des langages adaptés à la génération de tests, nommés UML4MBT/OCL4MBT présentés section 2.3.

Système de transitions est un point de vue spécifique sur le système choisi dans le but d'animer le modèle.

Cibles de test sont des prédicats spécifiant ce que doivent couvrir les tests sur le système. Elles sont fournies sous la forme de formules logiques utilisant les entités du modèle. Une présentation détaillée des cibles est donnée à la section 6.3.

Générateur de tests est le composant chargé de générer des cas de test correspondant aux cibles grâce à la résolution de formules logiques. Il utilise un processus d'animations du modèle guidé par des stratégies de parcours de l'espace de recherche. Il est présenté dans le chapitre 7.

Tests sont la résultante de notre démarche. Le test est constitué d'un cas de test et de la liste des cibles de test qu'il couvre.

Les différentes phases de notre démarche sont détaillées dans la suite de cette section.

6.1.1 Modèle et langage de modélisation

Comme expliqué à la section 2.3, nous avons retenu UML4MBT pour modéliser notre système. Il est utilisé en conjonction avec OCL4MBT pour formaliser le comportement du système. Ce dernier est exprimé au travers de contraintes présentes dans les opérations du diagramme de classes et les transitions du diagramme d'états-transitions. Les particularités de ce langage ont été détaillées précédemment. Pour rappel, ces langages ont les avantages suivants :

- Les notions et concepts décrits au travers de UML4MBT/OCL4MBT sont particulièrement adaptés à notre démarche puisque UML4MBT/OCL4MBT signifie UML/OCL pour le test à partir de modèle.
- Ce langage est dérivé de UML/OCL qui est devenu de-facto le standard industriel. Pour renforcer cet avantage, le méta-modèle du langage est construit en s'appuyant sur un framework répandu : EMF (Eclipse Modeling Framework). Ceci permet de s'insérer plus facilement dans un éco-système logiciel croissant.
- La modélisation du système peut être réalisée au travers d'outils industriels standards tels que RSA (Rational Software Architect) d'IBM ou Topcased, logiciel libre soutenu par différents partenaires industriels et académiques.
- Les propriétés du langage tel un nombre fini d'état du modèle, simplifient la génération de tests.

6.1.2 Système de transitions

Afin de générer des tests au travers de l'animation du modèle, un point de vue spécifique sur le système est adopté pour représenter les différents comportements au sein d'un système de transitions. Dans ce cas, une animation est une séquence de transitions. Chaque transition permet d'agir sur l'état du modèle en assignant de nouvelles valeurs aux variables d'états. La définition formelle de ce système de transitions est donnée dans la section 6.2. Ainsi ce point de vue apporte des informations supplémentaires pour la conversion en formules logiques des différents

comportements du système en catégorisant les éléments du modèle en fonction de leur utilité vis à vis de l'animation.

6.1.3 Scénario d'animations

Le scénario d'animations regroupe l'ensemble des informations nécessaire pour générer des cas de test à l'aide d'un solveur ou d'un prouveur. Il est défini formellement à la section 6.3. Avant cela, nous allons présenter les éléments qu'il utilise :

Objectif est la formule guidant la résolution pour vérifier si certaines cibles de test sont atteignables. Ainsi vérifier l'objectif permet de construire une animation valide. A contrario, ne pas réussir à atteindre l'objectif prouve son inatteignabilité en regard des contraintes portant sur l'animation exécutée.

Initialisation correspond à l'état initial de notre système. Cet état contient l'ensemble des variables d'états. La possibilité de spécifier un état initial permet d'exécuter une animation sous une forme fractionnée. Ainsi, une animation de X pas peut être remplacée par N animations de X/N pas afin de réduire la taille de la formule et par conséquent l'empreinte mémoire et la vitesse de résolution.

Motif d'animations est une séquence spécifiant quelles sont les transitions autorisées durant l'animation. Par défaut, le motif autorise l'ensemble des comportements du système décrit par les opérations des classes présentes au sein du diagramme de classes et les transitions dans le diagramme d'états-transitions. L'intérêt de réduire le nombre de transitions autorisées est évidemment de gagner en performance grâce à la réduction du nombre de contraintes issues de ces transitions. Une description formelle du motif d'animations est écrite à la section 6.3.

Longueur est le nombre de transitions à effectuer durant l'animation. Dans la suite, l'exécution d'une transition est également appelée un pas. Ainsi, la longueur d'une animation peut être comptée à l'aide d'un nombre de pas. Évidemment, plus une animation comporte un nombre important de pas, plus la probabilité d'atteindre une cible de test augmente. Cependant, le temps de résolution nécessaire pour obtenir une réponse de la part du solveur ou du prouveur augmente également. En effet, le processus de conversion entraîne une relation linéaire entre le nombre de pas et le nombre de contraintes et de variables. Il faut cependant nuancer cette affirmation car en fonction des prouveurs, il peut exister un écart important dans les temps de résolutions selon la satisfiabilité du problème. Il est souvent plus rapide d'obtenir une réponse dans le cas de problème insatisfiable avec un prouveur SMT. Ceci s'explique de façon théorique car nous sommes dans le cadre de problème NP et donc le temps de résolution augmente exponentiellement en fonction du nombre de pas. Nous avons pu vérifier cela expérimentalement en observant que le temps de résolution d'une animation de X pas est toujours plus important que le temps de résolution de deux animations de $X/2$ pas. De plus, l'écart entre ces deux temps grandit rapidement avec le nombre de pas.

6.1.4 Encodage en formules logiques

La phase suivante de notre démarche est la conversion du modèle sous la forme d'un système de transitions et des scénarios d'animations en des modèles adaptés aux prouveurs SMT et aux solveurs CSP utilisant des formules logiques du premier ordre. Quelque soit le modèle choisi, il est décomposable en 3 parties :

- **La première partie** du modèle est consacrée à l'encodage des transitions. Une transition est ici une formule spécifiant la valeur des variables d'états après son exécution en fonction de leur valeur avant l'exécution. Cette partie du modèle ne subit pas de modification durant les différentes animations ayant pour objectif de générer des cas de test différents car les différent comportements exprimés au travers du modèle ne changent pas. La seule opération envisageable sur cette partie est l'application d'un filtre afin de restreindre le nombre de comportements autorisé pour augmenter les performances. Cette partie est de loin la plus conséquente, au point que les autres parties sont négligeables en terme de nombre de formules.
- **La seconde partie** correspond à l'initialisation du système sous la forme d'une valuation des variables d'états. Dans le cadre de la génération de tests, seuls les états valides, c'est à dire définis par le modèle ou atteignables par une animation commençant par un état valide, sont considérés. Par conséquent, l'initialisation décrit un état modélisé dans le diagramme d'objets UML4MBT où un état rencontré lors d'une animation précédente. Le choix de cet état initial a des conséquences importantes sur la longueur des animations et l'atteignabilité des cibles de test. Par exemple, si l'on considère notre exemple fil rouge et que l'on souhaite tester si une pièce de type T3 peut être présente sur le quai de déchargement gauche, l'état initial influe sur la longueur de l'animation nécessaire pour générer le test correspondant. En effet entre le cas où une pièce de ce type est déjà saisie par le robot et le cas où aucune pièce n'est présente dans le système, il est évident que le test ne nécessitera pas les mêmes séquences de transitions.
- **La troisième partie** concerne les cibles de test. Elle se subdivise en deux. Elle doit encoder à la fois des formules permettant de décider si une cible de test est atteinte par l'exécution d'une séquence de transition et une formule permettant de piloter le processus de résolution en lui assignant un objectif. Cet objectif précise les cibles à atteindre obligatoirement lors d'une animation. Par définition, cette dernière formule diffère lors de chaque animation.

6.1.5 Test

La dernière phase de la démarche est la construction de tests grâce aux animations valides. En effet, si une animation est valide alors par construction au moins une cible de test a été atteinte et le cas de test correspondant peut être construit en utilisant la valuation des variables fournie par le prouveur SMT ou le solveur CSP. Une définition formelle du cas de test et de la cible de test sera donnée dans la section 6.3.

Cependant, le cas de test est pour l'instant exprimé au travers des variables soumises au solveur ou au prouveur. Ainsi, une étape de conversion est nécessaire pour ré-écrire le cas de test sous la forme d'une séquence d'opérations du diagramme UML4MBT de classe avec leurs paramètres valués et/ou de transitions du diagramme UML4MBT d'états-transitions.

6.2 Système de transition

Cette section a pour objectif de définir formellement le concept d'animation utilisé dans le reste de ce document. Ainsi l'animation du modèle s'appuie sur la mise en place d'un système de transitions. Par conséquent, cette section va définir dans l'ordre : les variables d'états et les états, le système de transitions et finalement l'animation du modèle.

6.2.1 Définitions préliminaires

Avant de discuter d'un système de transitions, la définition d'une variable d'état et par conséquent d'un état doit être explicitée.

Définition 12 (variable d'états) *Nous appelons variables d'états :*
- *les attributs des instances des classes*
- *les relations entre les classes du diagramme de classes associées aux instances présentes dans le diagramme d'objets UML4MBT.*

Notre exemple fil rouge contient les variables d'états suivantes : *quaiA.piece, quaiA.robotA, quaiG.robotG, quaiG.piece, quaiD.piece, quaiD.robotD, robot.cote, robot.position, robot.piece, robot.quaiA, robot.quaiD, robot.quaiG.*

Une des propriétés intéressante du langage UML4MBT est d'interdire la création dynamique d'instance. Par conséquent, toute instance utilisée au cours d'une animation est déclarée dans le diagramme d'objets et donc toutes les variables d'états sont obligatoirement déclarées dans ce diagramme. Cependant, les liens sont créés dynamiquement mais cela n'influe pas sur les variables les représentants mais seulement sur leurs valeurs.

Définition 13 (état) *Un état du modèle est déterminé par la valuation de n variables d'états* $\{x_1, x_2, \ldots, x_n\}$*. Si n est inférieur au nombre total des variables d'états, alors il s'agit d'un état partiel. Si n est égal au nombre total des variables d'états, alors il s'agit d'un état complet.*

Par définition, un état complet peut être compatible avec différent états partiels dans le sens ou les valuations des variables d'états ne se contredisent pas.

6.2.2 Définition du système de transitions

Pour animer un modèle en vue de générer des tests, nous allons exécuter les comportements du modèle encodés sous la forme d'un système de transitions. La

notation adoptée pour ce système est dérivée de celle présentée dans [FG09].

Le système de transitions est noté (D, Δ, ρ) où :

D est défini par $D = D_1 \times D_2 \times \ldots \times D_n$ où D_i est le domaine associé à une variable d'état x_i. Par conséquent, l'espace de recherche atteignable du système est défini par un sous-ensemble de D.

Δ est une relation de transition qui est la conjonction de l'ensemble des transitions δ_i. Les transitions valides entre deux états sont représentées par une relation de transition qui est un sous ensemble de $D \times D$. Dans notre cas, une transition correspond à une opération du diagramme de classes ou une transition du diagramme d'états-transitions. Une transition δ_i est une relation entre deux états.

ρ est une formule logique définissant l'état initial du système de transitions via une valuation des variables d'états. Cet état peut correspondre à l'état initial du système défini par le diagramme d'objets du modèle UML4MBT ou à un état rencontré lors d'une animation précédente valide.

Définition 14 (transition) *Une transition δ_i est définie par la relation $\delta_i = \alpha_i \wedge \beta_i \wedge \gamma_i$ où γ_i est la garde de la transition, β_i est l'action qui précise les nouvelles valeurs des variables d'états affectées par l'exécution de la transition après l'exécution de la transition et α_i défini les valeurs des autres variables d'états.*

Définition 15 (variable d'entrée) *Dans le cas d'une transition δ_i définie par $\delta_i = \alpha_i \wedge \beta_i \wedge \gamma_i$, les variables d'entrées sont les variables présentes dans la garde de la transition γ_i qui ne sont pas des variables d'états.*

Pour lever l'ambiguïté portant sur le terme *transition* qui peut se référer aux transitions δ du système de transitions ou aux transitions décrites dans le diagramme d'états-transitions, la convention suivante est adoptée :

- **Transition** se réfère au système de transition.
- **Transition UML4MBT** se réfère aux transitions décrites dans le diagramme d'états-transitions.

6.2.3 Définition de l'animation

Grâce à la définition formelle du système de transitions, il est maintenant possible d'introduire une définition de l'animation.

Définition 16 (animation) *Une animation est l'exécution d'une séquence finie de transitions δ_i par le solveur CSP ou le prouveur SMT. Une animation est dite valide si son exécution est conforme au modèle, autrement dit une animation valide $A = \langle s_0, \ldots, s_l \rangle$ d'un système de transitions $M = (D, \Delta, \rho)$ est une séquence finie telle que $\forall\, 0 \leq i < l : \Delta \models (s_i, s_{i+1})$ et $s_0 \in D$. Dans le cas contraire, une animation est dite invalide.*

DÉFINITION 17 (PAS) *La longueur d'une animation qui est une séquence finie de transitions est donnée en nombre de pas. Par conséquent, un pas d'une animation correspond à l'exécution d'une transition δ et donc à l'activation d'une opération ou transition présente dans le modèle UML4MBT.*

6.3 Notions liées au test

Cette section introduit les différentes notions liées au test et adaptées pour être utilisées en conjonction du système de transitions défini dans la section précédente. Pour générer des tests, il est nécessaire d'animer le modèle encodé sous la forme du système de transitions. Pour remplir ce but, un objectif de tests est assigné à chaque animation. Ce dernier précise quels tests seront créés si l'animation est valide. L'objectif de tests est une formule logique construite à partir de cibles de test, c'est à dire des prédicats de test.

La suite de cette section va donc répondre à ces trois questions : Comment suivre l'évolution de l'animation ? Comment construire une cible de test compatible avec le système de transitions mis en place depuis le modèle ? Comment créer un objectif de tests pour une animation ?

Finalement la définition de ces notions permet d'introduire une présentation formelle du scénario d'animations et du motif d'animations.

6.3.1 Statut de l'animation

Pour pouvoir écrire des prédicats de tests portant sur le système de transitions, il est nécessaire de disposer d'un moyen d'observer les états du système rencontrés pendant l'animation. Pour ce faire, les variables suivantes sont disponibles :

- Les variables d'états sont évidemment la première possibilité de connaître l'état du modèle durant une animation. Ainsi la valuation de l'ensemble de ces variables définit un état unique du modèle alors que la valuation d'une partie de ces variables caractérise un état partiel. Ainsi, un état partiel peut être un sous-état d'un autre état partiel ou d'un état complet.
- Pour observer l'activation d'une transition δ_i du système de transitions, une variable booléenne dédiée est créée. Cette variable est nommée variable de transition et notée cpt_i et correspond donc au niveau du modèle UML4MBT à l'exécution d'une opération du diagramme de classes ou d'une transition du diagramme d'états-transitions. Sur l'exemple fil rouge, les variables de transition suivantes existent : cpt_{a_p}, $cpt_{evacuation}$, $cpt_{rotation}$, $cpt_{translation}$, $cpt_{chargement}$, $cpt_{dechargement}$.
- Pour pouvoir tester des comportements précis, il est nécessaire de décomposer les transitions en un ensemble de comportements élémentaires. Le livre [Bei90] cite différentes méthodes pour ré-écrire les conditions des transitions afin d'expliciter les comportements élémentaires. En employant ces méthodes, pour un comportement i décomposable en x comportements élémentaires, x variables booléennes cpt_{i-elem} sont créées. Ces variables sont reliées aux variables de

transition par la relation : $cpt_i = \bigwedge_{elem=1}^{x} cpt_{i-elem}$. Ces variables sont appelées variables de transition élémentaire. Sur l'exemple fil rouge, les relations suivantes existent :

$$cpt_{rotation} = cpt_{rotation-gauche} \wedge cpt_{rotation-droite}$$

$$cpt_{translation} = cpt_{translation-haut} \wedge cpt_{translation-bas}$$

$$cpt_{dechargement} = cpt_{dechargement-gauche} \wedge cpt_{dechargement-droite}$$

Les opérations *a_p, evacuation* et *chargement* décrivent directement un comportement élémentaire. Ainsi, leurs variables de transition sont confondues avec leurs variables de transition élémentaire.

6.3.2 Cibles de test

En regard de la notation définie dans la section précédente, nous établissons les définitions suivantes :

Définition 18 (cas de test) *Un cas de test $t = \langle s_0, \ldots, s_l \rangle$ pour un système de transition $M = (D, \Delta, \rho)$ est une séquence finie telle que $\forall\, 0 \leqslant i < l : \Delta \models (s_i, s_{i+1})$ et $s_0 \in D$.*

Définition 19 (cible de test) *Une cible de test est un prédicat noté ϕ. L'ensemble des cibles de test est noté Φ. Un sous-chemin t_i du cas de test t est un sous-chemin commençant dont l'état initial est s_i. S'il existe un sous-chemin t_i d'un case de test $t = \langle s_i, \ldots s_l \rangle$ tel que $t_i \models \phi$ avec $0 \leqslant i \leqslant l$ alors la cible ϕ est satisfaite. Sinon la cible ϕ n'est pas satisfaite et elle est dite inatteignable pour une longueur l.*

Une cible de test ϕ est donc une formule logique construite en s'appuyant sur les variables d'états, de transition et de transition élémentaire. Grâce à ces variables, il est possible de construire des cibles permettant de valider des comportements divers. Ainsi la possibilité de vérifier si tous les comportements élémentaires sont atteignables découle directement de l'utilisation des variables de transition élémentaire. A l'aide des variables d'états, une cible de test correspondant à l'atteignabilité d'un état peut être construite. De plus, si ces différentes variables sont dupliquées pour chaque pas de l'animation, alors les cibles de test peuvent correspondre à des comportements plus complexes.

Par exemple, dans le cas du robot manipulateur, le comportement suivant doit être testé : *Une pièce arrive sur le quai de chargement, puis après diverses opérations, le robot effectue un déchargement. Après cette dernière opération, le système doit contenir une pièce de type T3 sur le quai de déchargement situé à droite.* Une formalisation de ce dernier est "$op_{a\_p}, op^*, op_{dechargement}, S_1$" où op_{a_p} et $op_{dechargement}$ représentent l'exécution des opérations *a_p* et *dechargement*, op^* indique l'exécution

d'un nombre indéterminé d'opérations successives et S_1 est un état partiel du système compatible avec la présence d'une pièce de type T3 sur le quai de déchargement situé à droite. Techniquement S_1 est défini par la valuation d'une partie des variables d'états. Ainsi S_1 est défini $quaiD.piece = Piece :: T3$.

Afin de transcrire ce comportement sous la forme d'une cible de test, la première étape est de définir un nombre fixe de pas pour le scénario d'animations correspondant. En effet, dans notre démarche, tous les scénarios soumis à un solveur ou à un prouveur ont une longueur fixe et connue. Cette contrainte est due aux choix fait durant le processus de conversion. Ainsi chaque étape du comportement qui autorise un nombre potentiellement infini d'actions doit être bornée. Le choix de la borne est un compromis entre le temps de résolution, favorisé par une borne faible, et la probabilité de trouver une séquence de transitions atteignant la cible de test, avantagée par une borne élevée.

Ce choix peut être guidé par la connaissance du modèle apportée par l'ingénieur de test et la complexité du modèle. L'idée est qu'un modèle complexe comportant un nombre de transitions élevées, a une probabilité plus élevée d'avoir besoin d'animations plus longues pour valider une cible de test. Si l'on revient au comportement donné en exemple et que l'on décide d'utiliser un scénario comportant N pas et donc exécutant N transitions alors l'étape op^* est remplacé par op^{N-2}, puisque deux pas sont utilisés par les opérations op_{a_p} et $op_{dechargement}$. Il est important de noter que par conséquent la résolution du scénario ne prouvera pas s'il existe dans l'absolu une séquence de transitions conforme à la cible mais s'il existe une séquence de N transitions conforme à la cible.

Rappelons que l'activation d'une transition est observable au travers des variables booléennes de transitions. Cependant, pour respecter la cible, il s'agit de tester non seulement si la transition choisie est activée mais que son activation intervient au bon moment. La notation x_i correspond à la variable ou la transition x lors du pas i de l'animation. Ainsi la cible de test associée au comportement est $cpt_{a\_p_1} \wedge op^{N-2} \wedge cpt_{dechargement_N} \wedge quaiD.piece_N = Piece :: T3$.

6.3.3 Objectif de tests

Nous définissons également un objectif de tests. L'objectif de tests indique quelles cibles doivent être atteintes durant l'exécution du scénario d'animations du système. L'objectif de tests guide la résolution du prouveur ou du solveur et est seul responsable de la validité de l'animation correspondante. Par conséquent, les trois propriétés suivantes existent :

- Toute animation n'ayant pas d'objectif de tests est forcément valide.
- Toute animation valide ayant un objectif de tests permet de construire au moins un test.
- Le problème encodant l'animation comporte au maximum un seul objectif de tests.

Dans notre démarche, la construction d'un objectif de tests est rendu obligatoire par l'impossibilité d'exprimer une contrainte de maximisation compatible avec un prouveur SMT. En effet, l'objectif de tests le plus efficace serait : *durant une*

animation du système, le nombre de cibles atteintes doit être maximisé.

Définition 20 (Objectif de tests) *Un objectif de tests est un prédicat* Θ *qui est construit uniquement en s'appuyant sur les cibles de test et les connecteurs logiques.*

Bien que les objectifs de tests puissent prendre des formes variées, en pratique, la majorité des objectifs de tests appartient à l'une de ces catégories :

- **Unique :** L'objectif porte sur une unique cible de test ϕ. On a donc $\Theta = \phi$
- **Liste :** L'objectif est d'atteindre un ensemble de n cibles de test et se note $\Theta = \bigwedge_{i=0}^{n} \phi_i$.
- **Minimum :** L'objectif est d'atteindre au moins une cible appartenant à une liste de n cibles de test et se note $\Theta = \bigvee_{i=0}^{n} \phi_i$.

6.3.4 Scénario et motif d'animations

Définition 21 (Motif d'animations) *Si* $M = (D, \Delta, \rho)$ *est un système de transition, un motif d'animations* M_A *est une séquence de transitions* $\langle S_0, \ldots, S_l \rangle$ *telle que* $\forall\, 0 \leq i \leq l$, $S_i \subset \Delta$.

D'après la définition, un motif d'animations précise pour chaque pas d'une animation les transitions autorisées sous la forme d'un ensemble de transitions. Ainsi l'animation réalisée par un prouveur ou par un solveur est équivalent à sélectionner une transition parmi celles présentes dans le motif.

Si plusieurs pas de la séquence autorisent les mêmes transitions, nous simplifions l'écriture en indiquant le nombre de pas identique par un exposant. Ainsi S_0^4 signifie que la séquence autorise quatre fois de suite les transitions comprises dans l'ensemble S_0.

Un motif peut également être défini en autorisant le même ensemble de transitions un nombre indéterminé de fois. Ceci est noté par le symbole $*$ en exposant. Par défaut, le motif d'animations autorise l'ensemble des transitions pour chacun des pas du scénario d'animations et se note donc $\langle \Delta^* \rangle$. Dans ce cas le motif est dit illimité.

L'intérêt de contraindre les motifs apparaît quand des informations supplémentaires sur le système sont connues. Par exemple, si l'objectif est d'atteindre une cible de test vérifiant le comportement du robot :Op_{a_p}, $Op_{chargement}$, $Op_{translation}$, $Op_{rotation}$, $Op_{dechargement}$, $Op_{evacuation}$ et que l'ingénieur de test dispose de connaissances approfondies sur le modèle qui dans ce cas est que tous les comportements du robot commencent par l'arrivée d'une pièce, alors la combinaison de ces informations aboutit à la création du motif d'animations $Op_{a\_p}, Op*$. Ce dernier permettra d'obtenir de meilleure performance grâce à la réduction de l'espace de recherche.

Définition 22 (Scénario d'animations) *Un scénario d'animations est un quadruplet* $(\Theta, M_A, L, \rho_{init})$ *où* Θ *est un objectif de tests,* M_A *est un motif d'animations,* L *est le nombre de pas du scénario et* ρ_{init} *est l'état initial de l'animation.*

Le scénario d'animations contient l'ensemble des informations utiles pour que le processus de résolution effectué par le prouveur ou le solveur résulte en une animation qui permette si elle est valide de créer un test.

Le scénario contient un nombre de pas car ce dernier ne peut pas obligatoirement être déduit du motif d'animations si ce dernier n'a pas de longueur fixe. Cependant le nombre de pas est toujours inférieur ou égal à la longueur du motif. Ce nombre est également utilisé pour convertir un motif illimité en motif limité, c'est à dire un motif ayant une longueur fixe. Cette conversion suit des règles identiques à celles employées pour convertir des cibles de test illimitées en cibles limités.

L'état initial utilisé dans un scénario est soit l'état initial du système de transition ρ qui est modélisé dans le diagramme d'objets, soit un état pris par le système lors d'une animation antérieure valide.

Il est important de noter que la soumission d'un scénario d'animations après encodage à un solveur ou à un prouveur peut résulter en des animations différentes. Par exemple, le scénario où l'objectif de tests est d'atteindre la cible $cpt_{dechargement}$, le motif d'animations n'engendre pas de restriction, le nombre de pas vaut 5 et l'état initial est défini par ρ (état décrit dans le diagramme d'objets) peut aboutir à différentes animations comme Op_{a_p}, $Op_{chargement}$, $Op_{translation}$, $Op_{rotation}$, $Op_{dechargement}$ ou Op_{a_p}, $Op_{chargement}$, $Op_{translation}$, Op_{a_p}, $Op_{dechargement}$.

6.4 Synthèse

Ce chapitre a présenté une vue d'ensemble de notre démarche visant à générer des tests via l'animation d'un modèle à l'aide d'un prouveur SMT et/ou d'un solveur CSP. La première étape de notre démarche est de considérer notre modèle sous la forme d'un système de transitions et d'obtenir une définition des cibles de test conforme à ce point de vue.

Ensuite un scénario d'animations est créé pour regrouper un ensemble de liste de cibles de test à atteindre et les comportements du système autorisés dans ce but. La combinaison du scénario avec le modèle permet d'écrire un problème à l'aide de formules logiques compréhensibles par un solveur ou un prouveur.

Le processus de résolution effectué par ces outils conduit à la création d'une animation,c-à-d une séquence de transitions. Si l'outil détermine l'existence d'une solution alors l'animation est valide et il est possible d'en déduire un test.

Les chapitres suivants vont détailler les phases centrales de notre démarche qui sont l'encodage d'un scénario d'animations ainsi que les stratégies mises en place pour optimiser le temps de génération des tests via différentes heuristiques portant sur la création de scénarios et leurs ordonnancements.

Chapitre 7

Stratégies de génération de test

Sommaire

Dans le chapitre précédent, une vue globale de notre démarche visant à générer des tests à partir d'un modèle a été présentée. La partie centrale de ce processus est la mise en place de stratégies de génération ayant pour objectif de diminuer le temps nécessaire pour obtenir les tests. Une stratégie est une heuristique décidant quels sont les scénarios d'animations à soumettre au prouveur SMT. Les stratégies combinant prouveurs et solveurs seront présentées à la section 9.2.

Dans ce chapitre, les questions suivantes seront abordées ; Comment et par qui est conduit une stratégie de génération de tests ? En particulier, quels sont les critères manipulables par une stratégie pour définir une heuristique ? A l'aide de ces critères, quelles stratégies peuvent être envisagées ? Dans le cadre de notre démarche qui est automatisée, comment choisir une stratégie et la configurer en fonction des caractéristiques du modèle et des cibles de test ?

La première partie de ce chapitre est dédiée aux caractéristiques communes aux différentes stratégies. En effet, chacune des stratégies est conduite au travers des mêmes composants et spécifie les problèmes soumis au prouveur avec une liste restreinte de paramètres. Afin d'augmenter les performances, une première solution de parallélisation est également abordée. La seconde partie décrit les différentes stratégies au travers de leur idée directrice, de leurs paramètres, de leur fonctionnement et de leurs propriétés. La dernière partie compare les différentes stratégies pour donner des pistes afin d'utiliser au mieux les choix offerts en fonction du modèle et des cibles de test.

7.1 Création et conduite d'une stratégie

Dans la première section de ce chapitre, une vue globale des stratégies de génération de tests est présentée. En particulier, les propriétés communes à l'ensemble des stratégies sont mises en avant. L'objectif est de répondre aux questions suivantes : quels sont les composants concernés par les stratégies ? Quels sont les facteurs manipulables par une stratégie ? Comment améliorer les performances avec une parallélisation de l'exécution ?

7.1.1 Composants

La figure 7.1 décrit le processus de génération de tests. Celui-ci est piloté par une stratégie. Le but de la stratégie est de générer des tests afin de répondre à un critère de couverture en un minimum de temps. Pour répondre à cet objectif, le choix des scénarios d'animations à soumettre au prouveur SMT est déterminant. Les différents composants nécessaires à ce processus sont détaillés dans la suite de cette section.

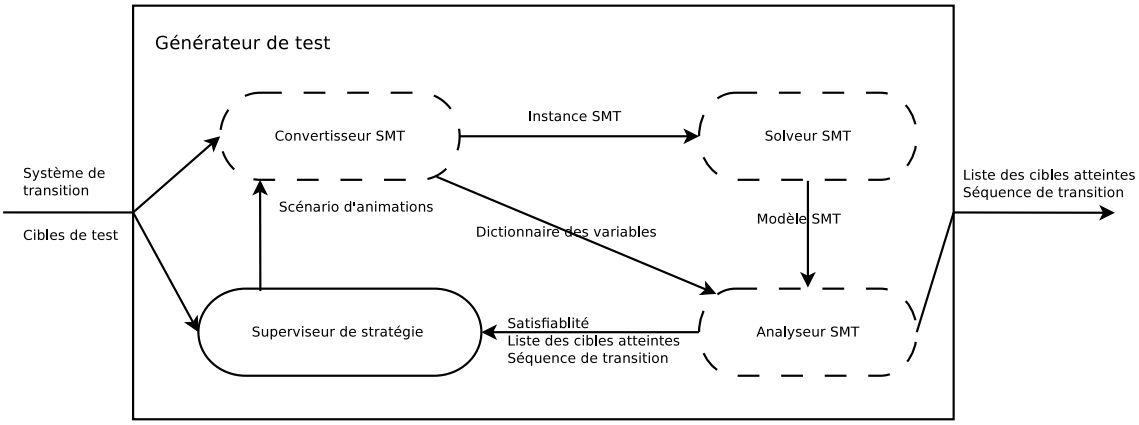


FIGURE 7.1 – Processus de génération de tests sous la conduite d'une stratégie

Convertisseur SMT Le premier composant est le convertisseur SMT qui est chargé de créer une instance SMT à partir d'un système de transitions et d'un scénario d'animations. Une instance SMT est un problème écrit en SMT-lib et donc

pouvant être soumis à un prouveur SMT. Une instance SMT contient quatre types d'informations : la déclaration des variables, une liste d'hypothèses qui sont des formules logiques supposées vraies dans le cadre de la résolution du problème, une formule logique à résoudre et des informations pour guider la résolution telle que la présence d'une solution ou le type de logique employé. Ce composant est également responsable de la création d'un *dictionnaire de variables* associant une variable issue du système de transitions aux variables SMT correspondantes.

Prouveur SMT Ce composant est chargé de résoudre un problème écrit sous la forme d'une instance SMT. Grâce à l'encodage choisi pour notre démarche et décrit en détail dans la section 8.2, le prouveur est toujours capable de conclure. Soit le prouveur retourne une solution, nommée *modèle SMT*, sous la forme d'une valuation des variables, soit le prouveur conclut à l'absence de solution.

Analyseur SMT Le but de l'analyseur SMT est d'extraire les informations utiles à la création de tests ou à la poursuite de la stratégie du *modèle SMT*. Pour ce faire, le *dictionnaire de variables* est utilisé pour représenter ces informations sous une forme compatible avec le système de transitions. Les informations suivantes sont extraites :

- La satisfiabilité pour déterminer si l'animation exécutée par le prouveur est valide. Par définition, une animation valide permet de construire au moins un test ;
- Les différentes cibles atteintes au cours de l'animation sont précisées. Cette information est utile car l'objectif de tests qui est une formule logique, peut être valide pour différentes combinaisons de cibles de test. De plus, il est possible d'atteindre au cours d'une animation des cibles non mentionnées dans l'objectif de tests défini dans le scénario ;
- La séquence de transitions définissant l'animation. Cette séquence correspond, après conversion, à une séquence d'opérations UML4MBT ayant leurs paramètres valués et/ou de transitions UML4MBT. De plus, cette séquence définit une suite d'états valides du modèle. Par conséquent, ces différents états sont utilisables en tant qu'état initial d'un scénario ultérieur.

Superviseur de stratégie Le dernier composant pilote la génération de tests au travers du choix et de l'ordre des scénarios d'animations à exécuter. Un scénario est défini au travers de quatre paramètres : sa longueur, l'objectif de tests, le motif d'animations et un état initial. Le choix des scénarios dépend de la stratégie à appliquer et des cibles de test restant à atteindre. En conséquence, chaque stratégie nécessite la création d'un superviseur de stratégie spécifique.

D'un point de vue du temps d'exécution, le temps pris par l'analyseur SMT et le superviseur de stratégie est négligeable devant le temps pris par le prouveur SMT et le convertisseur SMT. Le ratio du temps passé à générer des tests entre la résolution des instances SMT et la création de ces instances dépend de la complexité et de la taille du modèle ainsi que de la longueur des animations. Le choix a été fait de créer

les instances SMT sous la forme de fichiers afin de privilégier l'inter-compatibilité entre les différents prouveurs mais en contrepartie les temps d'écritures sur le disque peuvent être conséquents. Le temps de création des instances suit une relation linéaire fonction de la longueur de l'animation alors que le temps de résolution du prouveur a une complexité supérieure d'après l'étude empirique menée sur les différents cas d'étude (voir le chapitre 11). Par conséquent, avec l'augmentation de la longueur des animations, le temps de création des instances devient négligeable devant le temps de résolution.

7.1.2 Paramètres des animations

Le choix des paramètres des scénarios ont une influence directe sur les performances en terme de vitesse de résolution du problème par le prouveur et de probabilité d'atteindre les cibles de test.

Longueur Le premier paramètre définissant une animation est sa longueur, mesurée en nombre de pas. D'après notre démarche, un pas est équivalent à l'exécution d'une transition et donc à l'exécution d'une opération ou une transition UML4MBT. Ce paramètre est le plus influent sur le temps d'animation et la probabilité d'atteindre une cible de test. Une conséquence des choix faits durant l'encodage de l'animation en instance SMT est qu'une cible atteignable avec une séquence de X transitions est toujours atteignable avec une sequence de longueur Y si $Y > X$. Cette propriété découle de la présence d'une transition fictive n'influant pas sur l'état du modèle. Sans cette transition, le système de transitions peut rencontrer un état terminal qui est défini par une absence de transitions reliant cet état à un autre. Par exemple, dans le cas d'une carte bancaire, il existe une séquence définie par les opérations $op_{authentification}$, $op_{désactivation}$ mesurant deux pas, n'autorisant pas l'exécution d'une autre transition entre ces opérations et interdisant d'effectuer une transition quelconque après une désactivation. Cependant la présence d'une transition fictive assure la validité d'une animation de X pas comportant cette séquence et s'écrivant $op^{a}_{fictive}$, $op_{authentification}$, $op^{b}_{fictive}$, $op_{désactivation}$, $op^{c}_{fictive}$ avec $a+b+c = X-2$.

Objectif de tests L'utilisation d'objectifs de type *unique* ou *minimum* (cf. section 6.3) oblige le prouveur à atteindre au moins une cible de test. Ces objectifs ont toutefois un avantage en cas d'animations invalides : il permet de conclure sur le fait que les cibles non atteintes durant une animation de X pas ne sont jamais atteignables pour cette longueur à partir de l'état initial considéré et pour les transitions encodées. A contrario, les objectifs de types *liste* ont l'avantage d'obliger le prouveur à atteindre de multiples cibles, mais en cas d'animation invalide, aucune conclusion sur l'atteignabilité des cibles ne peut être formulée. L'intérêt de ce type d'objectif augmente si des informations supplémentaires sur le modèle sont disponibles pour grouper des cibles en fonction de leur interdépendance.

Motif d'animations Avec ce paramètre, le *superviseur de stratégies* précise les transitions autorisées durant les différents pas de l'animation.D'un point de vue technique, les transitions autorisées peuvent être spécifiées via une formule logique ou via un encodage spécifique du scénario ne comportant pas les formules logiques encodant les transitions interdites. La première possibilité permet de conserver intacte la majorité du fichier décrivant le problème mais ne garantit pas de gain sur le processus de résolution. La seconde possibilité entraîne une ré-écriture du fichier mais garantit un gain de performance lors de la résolution. Dans une instance SMT, les formules encodant l'état initial du modèle et l'objectif de tests sont négligeables en nombre devant les formules encodant les transitions modélisant le comportement du modèle. Ainsi pour un modèle ayant N transitions et en considérant que chaque transition est encodée par un nombre équivalent de contraintes, l'exécution d'une séquence de transitions précises réduit le nombre de contraintes et de formules d'un facteur $1 - 1/N$. Par exemple, un modèle avec 4 transitions encodées chacune avec 2 contraintes encode une scénario avec $4 * 3 = 12$ contraintes. Dans le cas d'un scénario où le motif d'animations autorise une unique transition à chaque pas, il suffit de $12 - 12 * (1 - 1/4) = 3$ contraintes.

État initial Le choix d'un état initial adapté permet de réduire le nombre de pas nécessaire pour atteindre une cible de test spécifique. Par exemple, la cible $Cpt_{chargement}$ du modèle robot nécessite au minimum une animation de 2 pas (Op_{a_p}, $Op_{chargement}$) si le scénario emploi l'état initial ρ du système de transition (état décrit par le diagramme d'objets) mais un seul ($Op_{chargement}$) si l'état initial comporte une pièce sur le tapis roulant inférieur. Cependant pour être sûr d'obtenir des animations valides, les états utilisés doivent avoir été rencontrés lors de l'exécution d'animations valides.

7.1.3 Parallélisation

Les stratégies peuvent diminuer le temps de génération des tests en instaurant une parallélisation des taches composant notre démarche. Ainsi dans la figure 7.1 présentant la démarche, les composants représentés par une ligne pointillée sont parallélisables. Le convertisseur SMT est capable de créer en même temps un ensemble d'instances SMT et l'analyseur SMT de traiter différents modèles SMT. Dans le cas du prouveur SMT, plusieurs instances SMT peuvent être résolues simultanément mais certains prouveurs sont également capables d'utiliser un processus de parallélisation pour accélérer la résolution du problème. Ainsi, pour ces prouveurs, un équilibre entre les types de parallélisation est nécessaire.

Dans le cas d'une stratégie parallélisée, le superviseur de stratégie est également chargé de la gestion des différents processus. Il choisit le nombre de processus et leur affectation. Le superviseur implémente deux méthodes. La première est d'associer les processus aux scénarios d'animations. Ainsi, le processus utilise successivement trois composants (convertisseur, prouveur et analyseur SMT) pour réaliser une animation. La deuxième, orthogonale, consiste à associer les processus aux composants. Chaque composant de la chaîne d'animation est exécuté par un nombre fixe de processus et

une animation met en œuvre une succession de processus. Le superviseur maintient également une liste synchronisée des cibles restant à atteindre.

Une instance SMT peut potentiellement être satisfaite avec différentes solutions. Par conséquent, plusieurs séquences de transitions sont capables de remplir un même objectif de tests. Ainsi la re-soumission d'une instance SMT à un prouveur peut retourner une nouvelle séquence de transitions qui atteint de nouvelles cibles de test. Par conséquent, l'exécution en parallèle de plusieurs résolutions du même scénario d'animations peuvent apporter des informations différentes. Ce comportement est du à l'impossibilité de guider le prouveur durant la résolution en privilégiant certaines contraintes. Il est amplifié par le fait d'essayer de maximiser le nombre de cibles de test atteintes.

Ainsi la probabilité d'obtenir une séquence de transitions différente pour chacun des scénarios, leurs différents objectifs de tests ne doivent pas être satisfaits par l'obtention d'une même combinaison de cibles de test. Les catégories d'objectif *unique* et *liste* possèdent naturellement cette propriété. Par contre, les objectifs de type *minimum* doivent être modifiés en cas de parallélisation. Ainsi, la liste des cibles constituant l'objectif est fractionnée en un ensemble de sous-listes disjointes.

Plus le temps de résolution d'une instance par un prouveur est conséquent, plus le choix du traitement à effectuer quand une résolution menée en parallèle fournie une séquence de transitions satisfaisant l'objectif de tests recherché, est décisif. Deux méthodes sont envisageables. La première est d'interrompre le processus et de sacrifier le temps de résolution déjà effectué en vue de résoudre une nouvelle instance SMT. La seconde est de laisser terminer le prouveur en tablant sur la probabilité que la séquence de transitions trouvée aura atteint une cible non précisée dans l'objectif de tests. Ainsi les deux critères principaux pour décider de la méthode à appliquer sont le temps de résolution moyen et la probabilité d'atteindre une cible annexe à l'objectif.

7.2 Stratégies de génération de test

Dans cette section, les différentes stratégies centrées autour d'un prouveur SMT sont détaillées. Pour décrire une stratégie, son principe, ses paramètres, son fonctionnement, ses propriétés et son application sur l'exemple fil rouge sont successivement présentés.

La figure 7.2 illustre le fonctionnement des différentes stratégies présentées dans cette section. Pour ces différents graphes, les vertex représentent une animation et donc la soumission d'une instance SMT à un prouveur. Le type de trait employé, continu ou pointillé, indique le statut de l'animation correspondante, respectivement valide et invalide. Les nœuds du graphe correspondent à un état du système. De façon similaire aux vertex, le type de trait précise la validité de l'état. Le contenu du nœud précise l'objectif de tests et l'ordre de traitement du scénario correspondant. Ainsi *2-A* signifie que le scénario voulant atteindre la cible A est le deuxième à être soumis au prouveur après conversion en instance SMT. La racine du graphe correspond à l'état initial du système tel que défini dans le diagramme d'objets UML4MBT du

modèle et encodé par ρ dans le système de transitions.

Les différentes stratégies sont appliquées sur l'exemple suivant. L'objectif est de générer les tests associés aux cibles A, B, C et D pour un modèle M. Le modèle M possède 3 opérations nommées OpA, OpB et OpC. Ces différentes cibles sont atteignables à l'aide de séquences de transitions constituées au minimum de 3, 5, 7 et 11 pas respectivement. De plus les cibles B et C sont mutuellement exclusives, c'est à dire qu'il n'existe aucune sequence de transitions permettant d'atteindre à la fois les cibles B et C. Finalement pour simplifier la représentation graphique, les objectifs de tests sont limités à la recherche d'une cible unique par scénario.

Les explications propres à chaque stratégie seront données dans les sections suivantes.

7.2.1 Basic

Principe Cette stratégie représente une approche basique de la génération de test. Après avoir décidé arbitrairement d'une longueur maximum pour les animations, des scénarios visant à atteindre au moins une cible de test sont soumises jusqu'à l'apparition d'une animation invalide.

Paramètres Cette stratégie ne nécessite qu'un seul paramètre nommé **MaxPas**. Ce dernier donne la longueur des animations exécutées au cours de la génération des tests.

Fonctionnement Le superviseur de stratégie génère un scénario d'animations de longueur *MaxPas* pas. *MaxPas* est choisi arbitrairement si l'utilisateur n'a pas de connaissance sur le modèle. Sinon l'utilisateur choisit une valeur pour *MaxPas* permet d'atteindre l'ensemble des cibles de test. L'objectif de tests est de type *minimum*, c'est à dire qu'au moins une cible de test parmi une liste est atteinte. Si l'animation est valide alors la séquence de transitions et les cibles atteintes sont transmises au superviseur de stratégie en vue de la création de tests et d'une nouvelle animation pour laquelle l'objectif de tests est actualisé en retirant les cibles précédemment atteintes. Si l'animation est invalide, alors la stratégie se termine et les cibles non trouvées sont inatteignables en un nombre *MaxPas* de pas. Ainsi dans l'exemple de la figure 7.2, le superviseur a besoin d'exécuter 4 animations de 12 pas pour atteindre l'ensemble des cibles. La longueur des animations a été choisie dans cet exemple pour avoir l'assurance d'atteindre l'ensemble des cibles.

Application Les performances en terme de vitesse de cette stratégie et des suivantes seront évaluées sur des modèles de tailles plus importantes que notre modèle fil rouge. Ces résultats sont disponibles dans le chapitre 11. Par contre, la taille restreinte de notre exemple, permet de documenter l'application de la stratégie en listant l'ensemble des scénarios d'animations soumis au prouveur. Ces scénarios sont détaillés dans la table 7.1 en indiquant dans l'ordre leur rang d'exécution, leur taille en nombre de pas, leur état initial et les cibles de test composant l'objectif de tests

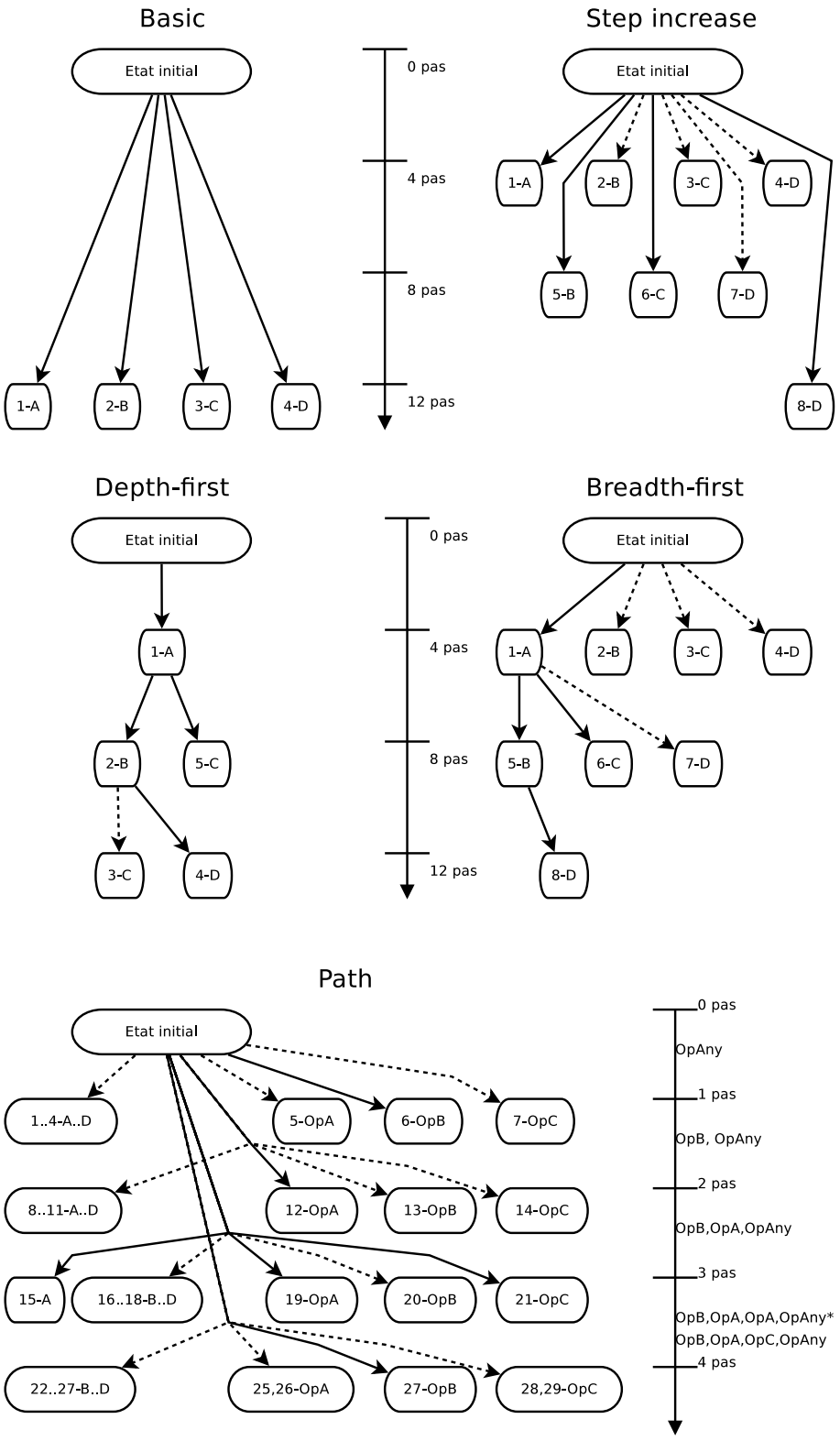


FIGURE 7.2 – Exécution des différents stratégies sur un exemple

de type *minimum*. Les cibles de test couvrent les comportements élémentaires. Elles ont été introduites à la section 6.3. Dans notre cas, nous avons choisi une longueur de 6 pas. De plus, la stratégie utilise l'état initial décrit par le diagramme d'objets et encodé par ρ dans le système de transitions. Le second tableau 7.2 décrit les tests créés grâce aux animations valides. Le numéro dans la première colonne précise l'animation à la base du test. Si une animation est absente de ce tableau, c'est qu'elle est invalide. Dans la séquence du test résultant de l'animation 1, l'opération *nop* est l'opération fictive introduite durant la phase d'encodage.

TABLE 7.1 – Scénarios d'animations soumis au prouveur lors de la stratégie *basic* sur le modèle Robot

Num.	Pas	État Initial	Cibles de test
1	6	ρ	Cpt_{a_p}, $Cpt_{evac.}$, $Cpt_{charg.}$, $Cpt_{decharg.-D}$, $Cpt_{decharg.-G}$ $Cpt_{rot.-D}$, $Cpt_{rot.-G}$, $Cpt_{translation-H}$, $Cpt_{translation-B}$
2	6	ρ	$Cpt_{decharg.-D}$, $Cpt_{rot.-D}$, $Cpt_{rot.-G}$, $Cpt_{translation-B}$
3	6	ρ	$Cpt_{translation-B}$

TABLE 7.2 – Tests créés lors de la stratégie *basic* sur le modèle Robot

Anim.	Test
1	cibles : Cpt_{a_p}, $Cpt_{evac.}$, $Cpt_{charg.}$, $Cpt_{decharg.-G}$, $Cpt_{translation-H}$
	séquence : ap(T1), charg., translation(haut), décharg., evac., nop
2	cibles : $Cpt_{decharg.-D}$, $Cpt_{rot.-D}$, $Cpt_{rot.-G}$
	séquence : ap(T2), charg., trans.(haut), rot.(droite), décharg., rot.(gauche)

7.2.2 Step-increase

Principe Cette stratégie repose sur le fait que le temps de résolution d'un scénario d'animations augmente avec sa longueur. Pour diminuer le temps de génération des tests, il faut donc privilégier les séquences de transitions les plus courtes possibles pour remplir l'objectif de tests associé au scénario.

Paramètres Cette stratégie est configurable au travers de trois paramètres :

InitLength est la longueur initiale en pas des scénarios d'animations qui ont pour état initial ρ (modélisé par le diagramme d'objets).

IncLength précise l'incrément en pas à ajouter à la longueur des scénarios si la longueur précédente était insuffisante pour atteindre l'ensemble des cibles de test.

MaxPas est la longueur maximale des scénarios soumis au prouveur en nombre de pas. Si l'utilisateur a une connaissance du modèle, alors ce nombre doit correspondre à la taille de la séquence de transitions la plus longue utilisée pour atteindre les cibles de test.

Fonctionnement Le fonctionnement de cette stratégie peut être vu comme des applications successives de la stratégie *basic* où la longueur des scénarios augmente progressivement. Ainsi dans la figure 7.2, l'exécution de la stratégie est équivalente à trois exécutions successives de la stratégie *basic* avec respectivement des scénarios de longueur 4, 8 et 12 pas et ayant pour objectifs de tests les ensembles de cibles de test $\{A, B, C, D\}$, $\{B, C, D\}$ et $\{D\}$. Si on note n la n-ième exécution de la stratégie *basic*, alors la longueur des animations exécutée est $n * IncSize + InitSize$. La stratégie s'interrompt quand toutes les cibles sont atteintes ou quand la longueur d'un scénario d'animations dépasse *MaxPas*.

L'intérêt d'utiliser deux paramètres pour calculer la longueur des animations est de pouvoir générer une première série de scénarios d'animations avec une longueur plus importante afin de trouver un équilibre entre la probabilité d'atteindre une cible et le temps de résolution. En effet, la plupart des modèles impose une séquence de transition constante depuis l'état initial. Par exemple, dans le cas d'un distributeur de billet, toutes les séquences commencent par les opérations permettant de s'authentifier. Supposons que l'authentification utilise 3 opérations, la probabilité d'atteindre une cible de test augmente fortement pour les scénarios d'animations ayant plus de 4 pas.

Un avantage de cette stratégie est la capacité de prouver qu'une cible est inatteignable à l'aide d'une séquence de transitions d'une taille donnée et d'un état initial spécifique. Dans ce cas, l'état initial est celui précisé dans le diagramme d'objets du modèle UML4MBT.

Application La stratégie *step-increase* est appliquée au modèle Robot avec la configuration suivante : $InitLength = 3$, $IncLength = 2$ et $MaxPas = 9$. Les tableaux 7.3 et 7.4 listent respectivement les scénarios d'animations exécutées durant la stratégie et les tests créés. Les cibles de test couvrent les comportements élémentaires. Elles ont été introduites à la section 6.3.

TABLE 7.3 – Scénarios d'animations exécutés lors de la stratégie *step-increase* sur le Robot

Num.	Pas	État Initial	Cibles de test
1	3	ρ	Cpt_{a_p}, $Cpt_{evac.}$, $Cpt_{charg.}$, $Cpt_{decharg.-D}$, $Cpt_{decharg.-G}$ $Cpt_{rot.-D}$, $Cpt_{rot.-G}$, $Cpt_{translation-H}$, $Cpt_{translation-B}$
2	3	ρ	$Cpt_{evac.}$, $Cpt_{decharg.-D}$, $Cpt_{decharg.-G}$ $Cpt_{rot.-D}$, $Cpt_{rot.-G}$, $Cpt_{translation-B}$
3	5	ρ	$Cpt_{evac.}$, $Cpt_{decharg.-D}$, $Cpt_{decharg.-G}$ $Cpt_{rot.-D}$, $Cpt_{rot.-G}$, $Cpt_{translation-B}$
4	5	ρ	$Cpt_{decharg.-D}$, $Cpt_{rot.-G}$, $Cpt_{translation-B}$
5	5	ρ	$Cpt_{rot.-G}$, $Cpt_{translation-B}$
6	7	ρ	$Cpt_{rot.-G}$, $Cpt_{translation-B}$

TABLE 7.4 – Tests créés lors de la stratégie *step-increase* sur le modèle Robot

Anim.	Test
1	cibles : Cpt_{a_p}, $Cpt_{charg.}$, $Cpt_{translation-H}$
	séquence : ap(T1), charg., translation(haut)
3	cibles : $Cpt_{decharg.-G}$, $Cpt_{evac.}$
	séquence : ap(T1), charg., trans.(haut), décharg., évacuation
4	cibles : $Cpt_{rot.-D}$, $Cpt_{decharg.-D}$
	séquence : ap(T2), charg., trans.(haut), rot.(droite), déchargement
6	cibles : $Cpt_{rot.-G}$, $Cpt_{translation-B}$
	séquence : ap(T2), charg., trans.(haut), rot.(droite), décharg., rot.(gauche), rot.(droite)

7.2.3 Space-search

Principe Cette stratégie utilise la propriété que le temps de résolution d'un scénario d'animations de N pas est toujours supérieur ou égale à la somme des temps de résolutions de X scénarios d'animations de N/X pas. L'idée est de fractionner une animation en une séquence d'animations plus courtes. Pour cela, l'état initial des scénarios d'animations soumis au prouveur est modifié pour correspondre à un état atteint lors d'animations précédentes valides. Les différents états du système employés comme états initiaux des scénarios d'animations constituent un espace de recherche. Le gain pour le temps de résolution des scénarios d'animations par le prouveur est contrebalancé par une augmentation de la taille de l'espace de recherche. Cet espace de recherche est exploré à l'aide d'un algorithme de parcours en profondeur ou en largeur. Ces variantes sont nommées *depth-first* et *breadth-first*.

Paramètres Le comportement de cette stratégie est défini au travers de deux paramètres :

Length est la longueur des scénarios d'animations durant cette stratégie. Cette longueur est constante pour l'ensemble des scénarios.

MaxAnim est le nombre maximum de scénarios d'animations successifs autorisés durant la stratégie. Deux scénarios sont successifs si l'état atteint après l'exécution de la première animation correspond à l'état initial du second scénario. Par conséquent, cette stratégie peut atteindre des cibles nécessitant des animations d'une longueur égale à $Length * MaxAnim$ depuis l'état du modèle défini par le diagramme d'objets UML4MBT.

Fonctionnement Le fonctionnement de cette stratégie correspond à l'exploration d'un espace de recherche à l'aide d'un parcours en profondeur ou en largeur. L'espace de recherche est constitué d'un ensemble d'états du modèle atteints après l'exécution d'animations de longueur fixe définie par un paramètre de la stratégie. Les différents scénarios d'animations ont un objectif de tests de type *minimum* où au moins une cible de test parmi une liste doit être atteinte. La figure 7.2 montre l'application de cette stratégie sur un exemple simple. D'après le parcours en profondeur, le premier scénario d'animations a pour objectif de tests la cible A et utilise l'état défini par le diagramme d'objets comme état initial. Ensuite un scénario d'animations cherchant

la cible B est exécuté. Ce scénario utilise comme état initial l'état du modèle atteint après l'exécution de la première animation. Ce processus est réitéré tant que toutes les cibles ne sont pas atteintes. Si le nombre maximum de scénarios successifs est atteint, un processus de backtracking est enclenché. Dans l'exemple de la figure, l'animation invalide est due à l'incompatibilité des cibles de test B et C.

La caractéristique principale est la décomposition des animations en plusieurs sous-animations. La décomposition permet de réduire le temps de génération des tests mais elle augmente le nombre de résolutions réalisées par le prouveur et diminue la probabilité que ces animations soient valides. En effet l'existence d'une animation valide de N pas permettant d'atteindre l'état S_N depuis l'état S_0 grâce à l'exécution de N transitions ne garantit pas que les sous-animations de $N/2$ pas, crées par décomposition, reliant respectivement les états $\langle S_0, \ldots S_{N/2} \rangle$ et $\langle S_{N/2}, \ldots S_N \rangle$ soient toutes les deux valides. En effet, l'état $S_{N/2}$ peut conduire à ce que la deuxième sous animation soit invalide si cet état est incompatible avec les transitions de la sous-animation. Par exemple, cet état pourrait interdire l'exécution de transitions futures. Ce type d'état est nommé non-bloquant.

L'efficacité de cette stratégie dépend des caractéristiques du critère de couverture. Pour diminuer le nombre d'animations invalides, les états obtenus à l'issue des animations doivent autoriser l'exécution de futures transitions. Ces états dépendent en partie de la contrainte imposée au prouveur d'atteindre l'objectif de tests. Ainsi si les différentes cibles de test sont l'activation de comportements élémentaires alors la probabilité d'obtenir un état non-bloquant augmente.

Si l'exécution de la stratégie n'a pas permis d'atteindre l'intégralité des cibles et que le nombre maximum de scénarios d'animations successifs autorisés a été atteint alors il est possible de générer des états non-bloquants supplémentaires. Par exemple si dans le cas du robot, la stratégie est appliquée avec la cible de test $Cpt_{translation-H}$ et une longueur pour les scénarios de 2 pas, la cible ne sera pas atteinte car une animation de 3 pas est nécessaire. De plus l'absence d'autres cibles de test empêche d'exécuter des animations depuis un état initial différent de celui décrit dans le diagramme d'objets. Ainsi l'exécution de scénarios d'animations sans objectif de tests peut permettre d'obtenir un état depuis lequel la cible $Cpt_{translation-H}$ est atteignable en exécutant deux transitions. Pour ce faire, le prouveur peut exécuter des animations sans la présence d'un objectif de tests afin de s'assurer de leur validité. Dans ce cas, il est nécessaire de limiter leur nombre via un paramètre supplémentaire. Cette mesure a cependant un impact fort sur la performance et la présence importante de ce type de scénario est une indication que cette stratégie n'est pas adaptée au modèle et aux cibles de test.

Dans le cadre de cette stratégie, l'inatteignabilité des cibles de test ne peut pas être prouvée. En effet, l'espace de recherche exploré ne contient pas l'ensemble des états atteignables par l'exécution d'une transition. L'exploration complète n'est généralement pas envisageable à cause de la taille de l'espace. Cet espace correspond à D tel que défini dans le système de transition.

Application La stratégie *depth-first* est appliquée au modèle Robot avec la configuration suivante : $Length = 3$ et $MaxAnim = 3$. Les tableaux 7.5 et 7.6 listent respectivement les scénarios d'animations exécutés durant la stratégie et les tests créés. Les états initiaux indiqués par un numéro correspondent à l'état final de l'animation correspondante. Les cibles de test couvrent les comportements élémentaires. Elles ont été introduites à la section 6.3. Avec ces paramètres, les cibles $Cpt_{rot.-G}$ et $Cpt_{decharg.-D}$ n'ont pas été atteintes. Dans l'absolu, ces cibles sont atteignables. Les séquences générées comportent un élément de hasard car plusieurs séquences sont solution d'un même problème. En conséquence, l'exemple donné n'est pas systématiquement reproductible.

TABLE 7.5 – Scénarios d'animations exécutés lors de la stratégie *depth-first* sur le Robot

Num.	Pas	État Initial	Cibles de test
1	3	ρ	Cpt_{a_p}, $Cpt_{evac.}$, $Cpt_{charg.}$, $Cpt_{decharg.-D}$, $Cpt_{decharg.-G}$ $Cpt_{rot.-D}$, $Cpt_{rot.-G}$, $Cpt_{translation-H}$, $Cpt_{translation-B}$
2	3	1	$Cpt_{evac.}$, $Cpt_{decharg.-D}$, $Cpt_{decharg.-G}$ $Cpt_{rot.-D}$, $Cpt_{rot.-G}$, $Cpt_{translation-B}$
3	3	2	$Cpt_{evac.}$, $Cpt_{decharg.-D}$, $Cpt_{rot.-D}$, $Cpt_{rot.-G}$
4	3	2	$Cpt_{evac.}$, $Cpt_{rot.-G}$, $Cpt_{decharg.-D}$
5	3	2	$Cpt_{rot.-G}$, $Cpt_{decharg.-D}$
6	3	1	$Cpt_{rot.-G}$, $Cpt_{decharg.-D}$
7	3	ρ	$Cpt_{rot.-G}$, $Cpt_{decharg.-D}$

TABLE 7.6 – Tests créés lors de la stratégie *depth-first* sur le modèle Robot

Anim.	Test
1	cibles : Cpt_{a_p}, $Cpt_{charg.}$, $Cpt_{translation-H}$ séquence : ap(T1), charg., translation(haut)
2	cibles : $Cpt_{decharg.-G}$, $Cpt_{translation-B}$ séquence : déchargement, translation(bas), ap(T2)
3	cibles : $Cpt_{rot.-D}$ séquence : chargement, translation(haut), rotation(droite)
4	cibles : Cpt_{a_p}, $Cpt_{charg.}$, $Cpt_{translation-H}$ séquence : évacuation

7.2.4 Path

Principe Cette stratégie privilégie la réduction du nombre de contraintes et de variables présentes dans l'instance SMT encodant le scénario d'animations afin de réduire le temps de résolution de cette instance par le prouveur. L'idée est de restreindre les transitions du système de transitions à un sous-ensemble. Pour cela, le motif d'animations du scénario est utilisé.

Paramètres Le seul paramètre de cette stratégie, **MaxPas**, donne la longueur maximum des scénarios d'animations.

Fonctionnement Durant cette stratégie, les scénarios d'animations servent deux buts distincts. Le premier consiste de manière classique à atteindre une cible de test. Le deuxième est d'obtenir de nouveaux motifs d'animations. La figure 7.2 détaille l'application de cette stratégie avec des scénarios limités à 4 pas. Dans cette figure, les nœuds de gauche sont dédiés aux cibles de test et les nœuds de droite sont consacrés à la découverte des motifs d'animations. Pour simplifier la figure, la notation suivante est adoptée : $1..4 - A..D$. Cette dernière représente l'exécution de quatre scénarios ayant pour objectifs de tests respectivement les cibles A, B, C et D.

Le processus utilise deux phases en alternance. Dans la première, des scénarios ayant pour objectif des cibles de test non trouvées sont soumis au prouveur. Ces scénarios respectent les motifs d'animations définis précédemment. Dans la seconde, les motifs sont établis par construction. Pour cela, les motifs précédents sont agrandis par l'ajout d'une transition quelconque et l'avant dernier pas est restreint aux transitions valides. Ces dernières sont déterminées par l'exécution de scénarios dédiés avec des objectifs de tests du type *minimum*.

Dans la figure, le motif initial est *opAny* (une transition quelconque). Après la phase deux, ce motif est réduit à *opB* car seule la transition B a conduit à une animation valide. Par conséquent, un nouveau motif est construit par l'ajout d'une transition quelconque : opB, op^*. Ces deux phases se répètent tant que toutes les cibles de test ne sont pas atteintes ou que la longueur des animations est inférieure à la valeur du paramètre MaxDepth.

En conséquence, le nombre de scénarios exécutés durant cette stratégie est le plus élevé. Au pire, ce nombre vaut $NbTransition * MaxDepth * NbTarget$ où $NbTransition$ est le nombre de transitions appartenant à Δ, $MaxDepth$ est le paramètre précisant la longueur maximum des animations et $NbTarget$ est le nombre de cibles de test.

L'efficacité de cette stratégie dépend principalement du choix fait durant la phase de modélisation. Plus les opérations du diagramme de classes et les transitions du diagramme d'états-transitions sont proches d'un comportement élémentaire, plus l'efficacité augmente. En effet, cette stratégie réduit le nombre de contraintes encodant les comportements définis par les opérations et les transitions UML4MBT du modèle. Par conséquent, plus les contraintes sont minimalistes, plus le gain est important. Ainsi dans un modèle ayant X opérations , une séquence de transitions de N pas a un nombre de contraintes réduit par un facteur : $(N - 1 + X^2)/X$.

Un cas particulier est l'usage de cette stratégie quand le critère de couverture est l'activation des comportements élémentaires. Dans ce cas, les deux phases de la stratégie, la recherche des cibles de test et des scénarios d'exécution, fusionnent. En conséquence, le nombre d'animations exécutées durant la stratégie diminue et les performances augmentent.

De plus, cette stratégie a deux propriétés. La première est la capacité de prouver l'inatteignabilité d'une cible de test en un nombre de pas donné depuis l'état décrit par le diagramme d'objets. En effet, la stratégie explore l'ensemble des scénarios

d'exécution possibles de part leur processus de construction itératif. La seconde est que les séquences de transitions atteignant des cibles de test sont les séquences de longueur minimum. Cette propriété est due au fait que la longueur des scénarios durant la stratégie est incrémentée pas à pas avec une longueur initial de 1.

Application La stratégie *path* est appliquée au modèle Robot avec la configuration suivante : $MaxPas = 5$. Les tableaux 7.8 et 7.7 listent respectivement les scénarios d'animations exécutées durant la stratégie et les tests créés. Les cibles de test couvrent les comportements élémentaires. Elles ont été introduites à la section 6.3. Les séquences d'exécutions sont données dans le formalisme employé à la section 6.3. La cible $Cpt_{rot.-G}$ n'est pas atteinte avec ces contraintes.

TABLE 7.7 – Tests créés lors de la stratégie *path* sur le modèle Robot

Anim.	Test
1	cibles : Cpt_{a_p} séquence : ap(T1)
3	cibles : $Cpt_{charg.}$ séquence : ap(T1), charg.
5	cibles : $Cpt_{translation-H}$ séquence : ap(T1), charg., translation(haut)
7	cibles : $Cpt_{decharg.-L}$ séquence : ap(T1), charg., translation(haut), décharg.
8	cibles : $Cpt_{rot.-D}$ séquence : ap(T2), charg., translation(haut), rotation(droite)
10	cibles : $Cpt_{translation-B}$ séquence : ap(T1), charg., translation(haut), décharg., translation(bas)
11	cibles : $Cpt_{evac.}$ séquence : ap(T1), charg., translation(haut), décharg., évacuation
13	cibles : $Cpt_{decharg.-R}$ séquence : ap(T2), charg., translation(haut), rotation(gauche), décharg.

TABLE 7.8 – Scénarios animations exécutés lors de la stratégie *path* sur le Robot

Num.	Pas	État Init.	Contraintes
1	1	ρ	Cibles : Cpt_{a_p}, $Cpt_{evac.}$, $Cpt_{charg.}$, $Cpt_{decharg.-D}$, $Cpt_{decharg.-G}$, $Cpt_{rot.-D}$, $Cpt_{rot.-G}$, $Cpt_{translation-H}$, $Cpt_{translation-B}$ Séquence : $opAny$
2	1	ρ	Cibles : $Cpt_{evac.}$, $Cpt_{charg.}$, $Cpt_{decharg.-D}$, $Cpt_{decharg.-G}$, $Cpt_{rot.-D}$, $Cpt_{rot.-G}$, $Cpt_{translation-H}$, $Cpt_{translation-B}$ Séquence : $opAny$
3	2	ρ	Cibles : $Cpt_{evac.}$, $Cpt_{charg.}$, $Cpt_{decharg.-D}$, $Cpt_{decharg.-G}$, $Cpt_{rot.-D}$, $Cpt_{rot.-G}$, $Cpt_{translation-H}$, $Cpt_{translation-B}$ Séquence : a_p, $opAny$
4	2	ρ	Cibles : $Cpt_{evac.}$, $Cpt_{decharg.-D}$, $Cpt_{decharg.-G}$, $Cpt_{rot.-D}$, $Cpt_{rot.-G}$, $Cpt_{translation-H}$, $Cpt_{translation-B}$ Séquence : a_p, $opAny$
5	3	ρ	Cibles : $Cpt_{evac.}$, $Cpt_{decharg.-D}$, $Cpt_{decharg.-G}$, $Cpt_{rot.-D}$, $Cpt_{rot.-G}$, $Cpt_{translation-H}$, $Cpt_{translation-B}$ Séquence : a_p, $charg.$, $opAny$
6	3	ρ	Cibles : $Cpt_{evac.}$, $Cpt_{decharg.-D}$, $Cpt_{decharg.-G}$, $Cpt_{rot.-D}$, $Cpt_{rot.-G}$, $Cpt_{translation-B}$ Séquence : a_p, $charg.$, $opAny$
7	4	ρ	Cibles : $Cpt_{evac.}$, $Cpt_{decharg.-D}$, $Cpt_{decharg.-G}$, $Cpt_{rot.-D}$, $Cpt_{rot.-G}$, $Cpt_{translation-B}$ Séquence : a_p, $charg.$, $translation$, $opAny$
8	4	ρ	Cibles : $Cpt_{evac.}$, $Cpt_{decharg.-D}$, $Cpt_{rot.-D}$, $Cpt_{rot.-G}$, $Cpt_{translation-B}$ Séquence : a_p, $charg.$, $translation$, $opAny$
9	4	ρ	Cibles : $Cpt_{evac.}$, $Cpt_{decharg.-D}$, $Cpt_{rot.-G}$, $Cpt_{translation-B}$ Séquence : a_p, $charg.$, $translation$, $opAny$
10	5	ρ	Cibles : $Cpt_{evac.}$, $Cpt_{decharg.-D}$, $Cpt_{rot.-G}$, $Cpt_{translation-B}$ Séquence : a_p, $charg.$, $translation$, $decharg.$, $opAny$
11	5	ρ	Cibles : $Cpt_{evac.}$, $Cpt_{decharg.-D}$, $Cpt_{rot.-G}$ Séquence : a_p, $charg.$, $translation$, $decharg.$, $opAny$
12	5	ρ	Cibles : $Cpt_{decharg.-D}$, $Cpt_{rot.-G}$ Séquence : a_p, $charg.$, $translation$, $decharg.$, $opAny$
13	5	ρ	Cibles : $Cpt_{decharg.-D}$, $Cpt_{rot.-G}$ Séquence : a_p, $charg.$, $translation$, $rotation$, $opAny$
14	5	ρ	Cibles : $Cpt_{rot.-G}$ Séquence : a_p, $charg.$, $translation$, $rotation$, $opAny$

7.3 Comparaison et analyse

La création de plusieurs stratégies au sein de notre démarche conduit à une nouvelle problématique. Comment choisir une stratégie et sa configuration adaptée au modèle et aux cibles de test ? Cette section commence par proposer une liste de critères pour guider ce choix. Ensuite de nouvelles stratégies créées par hybridation sont introduites pour enrichir les choix de l'ingénieur de tests et répondre plus efficacement à certains modèles et cibles de test. Finalement, un résumé des stratégies

est donné sous la forme d'une comparaison de leurs caractéristiques principales.

7.3.1 Caractéristique du modèle et des cibles de test

Si notre démarche est complètement automatisée au niveau de l'exécution d'une stratégie pour générer des tests, le choix de la stratégie et sa configuration via des paramètres est laissé à l'ingénieur de tests. Hors ces choix ont une influence considérable sur le temps de génération des tests. Ce dernier peut être guidé par les caractéristiques du modèle et des cibles de test.

Une première caractéristique est la taille et complexité du modèle. Plus un modèle est complexe car il comporte un nombre important d'instances, d'attributs et d'opérations, plus le coût associé à une augmentation de la longueur des scénarios est important. Dans ce cas, les stratégies employant des scénarios à longueur fixe, telles que *depth-first* et *breadth-first*, sont à privilégier.

Une deuxième caractéristique est la modélisation des comportements. Plus les opérations présentes dans le diagramme de classes et les transitions du diagramme d'états-transitions sont proches d'un comportement élémentaire, plus les performances de la stratégie *path* sont bonnes. Une première approche pour analyser ce facteur est d'obtenir la moyenne du nombre de comportements élémentaires par opération ou transition. En première analyse, ce nombre peut être déduit du nombre de structures conditionnelles.

Si le modèle possède un diagramme d'états-transitions, il peut être utilisé pour donner une indication sur deux points. Le premier est la longueur des scénarios nécessaires pour atteindre les différents états du diagramme d'états-transitions. Cette information est utile pour déterminer la valeur du paramètre assurant la terminaison de la stratégie. En effet, si un état nécessite une séquence de N transitions, en considérant que les gardes requièrent l'exécution d'une seule opération, les scénarios durant la stratégie auront besoin d'atteindre une longueur de $2N$ pas.

Une deuxième indication porte sur la probabilité d'obtenir un état bloquant à l'issue d'une animation. Cette probabilité va être influencée par le nombre d'états finaux du diagramme d'états-transitions et la moyenne du nombre de transitions sortantes de ce diagramme. Plus la probabilité est faible, plus les stratégies reposant sur la décomposition des animations, *depth-first* et *breadth-first*, sont intéressantes.

Une autre possibilité pour guider nos choix est d'analyser les caractéristiques des cibles de test. Une première notion est l'interdépendance des cibles de test. Une cible A a une interdépendance maximum en regard d'une cible B si la cible A est toujours vérifiée si la cible B est atteinte. Par exemple, dans le robot les opérations *déchargement* et *chargement* ont des cibles interdépendantes. Les cibles $Cpt_{dechargement-D}$ et $Cpt_{dechargement-G}$ ont une interdépendance maximum vis à vis de la cible $Cpt_{chargement}$. Dans le même ordre d'idée, une cible A a une interdépendance nulle vis à vis d'une cible B s'il est impossible de vérifier la cible A quand la cible B est atteinte. Un exemple typique est que deux cibles correspondant à l'activation d'états finaux d'un diagramme d'états-transitions ont une interdépendance nulle. Dans le cas d'une interdépendance moyenne élevée, les stratégies *depth-first* et *breadth-first* auront de meilleures performances car les cibles de test vont privilégier des séquences

de transitions telles que les états du modèle obtenus à l'issue de ces séquences ont une probabilité plus faible d'être bloquants.

Une seconde notion est l'espacement des cibles entre elles. Si l'ingénieur de tests a une idée du nombre de pas requis pour atteindre les cibles de test alors il est possible de représenter cette notion à l'aide de l'écart type portant sur ce nombre de pas. Si l'écart type est faible, alors la majorité des cibles sont atteignables avec des scénarios de longueurs proches. En conséquence, les valeurs des paramètres de la stratégie doivent être adaptées pour minimiser le nombre d'animations invalides.

La dernière notion pouvant nous guider dans le choix d'une stratégie est la notion d'inatteignabilité. Si l'ingénieur de tests sait qu'une majorité des cibles de test sont inatteignables alors il convient de privilégier les stratégies ayant la capacité de prouver cette propriété : *step-increase* et *path*.

7.3.2 Stratégies hybrides

Les différents types de stratégie définis dans la section précédente ne sont pas forcément adaptés à l'ensemble des catégories de modèles existants. Une solution pour augmenter les performances est de combiner plusieurs stratégies afin de répondre aux différentes caractéristiques du modèle et des cibles de test.

Une première combinaison utile est l'utilisation conjointe des stratégies *basic* et *space-search*. Le but de cette stratégie est de répartir les cibles de test en fonction de leur atteignabilité. Si cette information est disponible avant l'exécution, il est alors possible de conduire deux stratégies en parallèle. La stratégie *basic* s'occupe des cibles inatteignables pendant que la stratégie *space-search* construit les tests dédiés aux cibles atteignables. Si l'information est indisponible, alors les stratégies sont exécutées séquentiellement. La stratégie *basic* est alors employée pour vérifier l'inatteignabilité des cibles non couvertes lors de l'application de *space-search*. Afin d'obtenir un comportement cohérent, les valeurs des paramètres doivent être ajustées pour que la longueur maximum des scénarios soit identique dans les deux stratégies. Par conséquent, la relation $length * maxAnim = maxPas$ doit être respectée.

Une deuxième combinaison est l'hybridation des stratégies *step-increase* et *space-search*. L'objectif est de pallier la dégradation des performances lors de l'utilisation de la stratégie *step-increase* quand la longueur des scénarios a augmenté au point que le temps de résolution correspondant est jugé trop important pour avoir des performances acceptables. L'idée est de fixer une longueur limite au delà de laquelle un changement de stratégie est effectué pour employer un processus utilisant des scénarios de longueur constante, c-a-d *depth-first* ou *breadth-first*. L'intérêt de cette combinaison est renforcé par le fait que la relation entre le temps de résolution et le nombre de contraintes chez la majorité des prouveurs contient des paliers lors de nos expérimentations (se référer chapitre 11). En conséquence, une augmentation d'un pas d'un scénario peut conduire à une augmentation brutale du temps de résolution. Lors de l'application de cette stratégie, la relation $maxPas = length$ est vérifiée.

TABLE 7.9 – Comparaison des stratégies de générations de test

Stratégie	Cible		Scénario		
	Unsat	Best	Const.	Max	Nombre
Basic	√		√	$MaxPas$	NbT
Step-increase	√			$MaxPas$	$(\lfloor \frac{MaxPas - InitLength}{IncLength} \rfloor + 1) * NbT$
Depth-first			√	$Length * MaxAnim$	$NbT^{MaxAnim}$ ou D
Breadth-first			√	$Length * MaxAnim$	$NbT^{MaxAnim}$ ou D
Path	√	√		$MaxPas$	$MaxPas * NbT * \|\Delta\|$
Basic + Space	√			$MaxPas$	$NbT_{unsat} + NbT_{sat}^{MaxAnim}$
Step + Depth				$MaxPas * MaxAnim$	$(\lfloor \frac{MaxPas - InitLength}{IncLength} \rfloor + 1 + NbT^{MaxAnim-1}) * NbT$

7.3.3 Comparaison des stratégies

Le tableau 7.9 présente une comparaison des principales caractéristiques des différentes stratégies introduites dans ce chapitre. Les différentes colonnes ont la signification suivante :

Stratégie Cette colonne spécifie le nom de la stratégie. Les deux dernières lignes concernent les stratégies hybrides présentées en 7.3.2. En particulier le terme *Space* se réfère aux stratégies utilisant des algorithmes d'explorations d'espaces de recherche : *depth-first* et *breadth-first*.

Unsat La colonne précise si la stratégie a la capacité de prouver qu'une cible est inatteignable. Cette inatteignabilité est évidemment relative à des séquences de transitions d'une longueur donnée.

Best Une coche dans cette colonne signifie que la stratégie fournit une séquence de longueur minimum pour atteindre une cible de test.

Const. Cette colonne indique si la stratégie emploie des scénarios d'une longueur constante. Le nombre de transitions appartenant à Δ a une influence moindre sur les stratégies de ce type. De plus, l'écart type du temps d'exécution des animations valides d'une même longueur est très faible. En conséquence, le temps d'exécution d'une animation peut être utilisé pour extrapoler le temps nécessaire à la stratégie.

Max La colonne indique la longueur maximum des scénarios employées durant la stratégie. Cette information est fournie au travers des paramètres de la stratégie. Pour rappel, *MaxPas* est la longueur maximum des scénarios, *Length* est la longueur des scénarios et *MaxAnim* est le nombre de scénarios successifs.

Nombre La dernière colonne donne le nombre d'animations effectuées durant une stratégie dans le pire des cas. Dans ces formules, NbT est le nombre de cibles de test qui sont catégorisées selon leur atteignabilité (NbT_{sat} si atteignable et NbT_{unsat} sinon. D et Δ se réfère au système de transitions où D est le domaine et Δ la conjonction des différentes relations de transitions. En particulier, $|\Delta|$ donne le nombre de transitions présentes dans le système.

7.4 Synthèse

Une stratégie de génération de tests est une heuristique portant sur le choix des problèmes à soumettre au prouveur et leur ordonnancement. Un problème est l'encodage sous forme de formules logiques d'un scénario d'animations du modèle.

Une stratégie est donc une suite d'itérations d'un processus qui créé un problème, le résout avec un prouveur et analyse la réponse pour créer un test et décider du comportement à adopter lors de l'itération suivante. Afin d'améliorer les performances, ce processus peut être parallélisé.

Grâce à ces paramètres, 4 stratégies ont été établies. Elles sont nommées : *basic*, *step-increase*, *space-search* et *path*. Le temps de génération des tests dépendant principalement du temps de résolution des problèmes soumis au prouveur, les stratégies doivent minimiser le nombre et la taille des problèmes. Pour ce faire, les stratégies peuvent privilégier les animations courtes, décomposer une animation en sous-animations ou réduire le nombre de transitions autorisées durant un scénario.

Dans le cadre de notre démarche, il est nécessaire de choisir en fonction du modèle et des cibles de test une stratégie et sa configuration. Ce choix est guidé par les caractéristiques du modèle (taille, complexité, présence d'un diagramme d'états-transitions, ...) et des cibles de test (atteignabilité, critère de couverture, interdépendance, ...).

Chapitre 8

Encodage SMT

Sommaire

Dans les chapitres précédents, la démarche globale et les stratégies associées à la génération de tests dans le cadre du Model-Based Testing ont été présentées. Il manque ainsi le détail de la dernière partie du processus qui est la représentation d'un scénario d'animations du modèle par un modèle mathématique manipulable. La solution, donnée dans ce chapitre, est d'employer un ensemble de formules logiques du premier ordre compatibles avec un prouveur SMT.

Cependant ce choix soulève de nouvelles questions. Dans le cadre du SMT, quelle théorie apporte une expressivité suffisante et préserve les performances ? Comment représenter les notions liées au tests au travers d'une formule logique et dans un langage, SMT-lib, de bas niveau ? Quelles sont les règles de conversion entre le langage de modélisation et le langage servant pour l'animation ?

La première partie de ce chapitre introduit le méta-modèle SMT conjuguant une représentation d'un scénario d'animations d'un modèle et les éléments propres aux tests à partir de modèle comme les objectifs de tests au travers de formules logiques du premier ordre. Ensuite, les règles de conversion sont décrites. Ces règles permettent d'encoder au sein du méta-modèle le système de transitions décrivant

le système. Finalement, une synthèse reprenant les propriétés importantes qui découlent des choix décidés lors du processus d'encodage d'un système de transitions dans un méta-modèle issu du langage SMT-Lib, sera présentée.

8.1 Méta-modèle SMT4MBT

L'idée au cœur de la génération de tests grâce à un prouveur SMT est de représenter l'animation d'un système et les cibles de test avec des formules logiques du premier ordre. Cependant le langage employé par les prouveurs n'est pas adapté à cet usage. Pour conserver l'objectif ayant présidé à la création d'une formule ou d'une variable, il est nécessaire d'ajouter de l'information sur ces dernières à l'aide de méta-données. Ces méta-informations relient les formules du prouveur aux notions utilisées dans une démarche MBT.

Un prouveur SMT résout des formules logiques du premier ordre grâce à la combinaison de différentes théories telles que la théorie des tableaux ou des vecteurs de bits. Le choix des théories et leurs combinaisons sont définies au sein de logiques. En conséquence, la première étape dans la constitution d'un méta-modèle adapté aux tests à partir de modèle est le choix de la logique. Ensuite les principes gouvernant l'architecture du méta-modèle sont introduits. Ces principes précisent quelles informations doivent être stockées et avec quelles modalités. Finalement le méta-modèle découlant de ces choix est décrit.

8.1.1 Choix de la logique

Le tableau 8.1 compare les logiques proposées par le langage SMT-lib en mettant en avant leur expressivité. Une description succincte et une explication des acronymes des différentes logiques sont disponibles dans le chapitre 4. Une coche ($\surd$) dans une ligne signifie l'utilisation par la logique de la théorie, dont voici la liste :

ArraysEx est la théorie des tableaux. Cette théorie utilise des quantificateurs pour définir certains de ses axiomes.

BitVectors est la théorie pour les vecteurs de bits de tailles fixes. Cette théorie fournit les opérations de concaténation et d'extraction en plus des opérateurs logiques et arithmétiques usuels sur les vecteurs de bits.

Core est la théorie de base qui établit les opérateurs logiques habituels. Cette théorie est utilisée par l'ensemble des logiques de façon implicite.

Ints est la théorie des entiers qui fournit les opérateurs arithmétiques usuels sur les entiers.

Reals est la théorie des réels qui fournit les opérateurs arithmétiques usuels sur les réels.

Reals_Ints est une théorie mixte sur les entiers et les réels. Les opérateurs sont restreints à ceux nécessaires pour des termes réels clos.

Quant. représente l'utilisation des quantificateurs : ∃ ou ∀.

UF autorise l'ajout de fonctions et de types non-interprétés. Ces fonctions supportent uniquement l'opérateur d'égalité et différence globale et leur signature est définie par l'utilisateur.

TABLE 8.1 – Comparaison des logiques SMT-libs

Logique	Théories						Quant.	UF
	ArraysEx	BitVectors	Core	Ints	Reals	Reals_Ints		
AUFLIA	√			√				√
AUFLIRA	√					√		√
AUFNIRA	√					√		√
LRA					√			
QF_ABV		√					√	
QF_AUFBV	√	√					√	√
QF_AUFLIA	√			√			√	√
QF_AX	√						√	
QF_BV		√					√	
QF_IDL				√			√	
QF_LIA				√			√	
QF_LRA					√		√	
QF_NIA				√			√	
QF_NRA					√		√	
QF_RDL					√		√	
QF_UF			√				√	√
QF_UFBV		√					√	√
QF_UFIDL				√			√	√
QF_UFLIA				√			√	√
QF_UFLRA					√		√	√
QF_UFNRA					√		√	√
UFLRA					√			√
UFNIA				√				√

Le choix de la logique est dicté par les critères suivants :

- Le prouveur doit être capable de conclure. Dans l'absolu, un prouveur SMT conclut lors de la résolution d'un problème à un des états suivants : SAT (il existe une solution), UNSAT (il n'existe pas de solution), UNKNOW (l'existance d'une solution est indéterminée). L'indétermination est la conséquence de la présence de quantificateurs dans le problème soumis au prouveur. Cependant leur absence ne garantit pas en toute circonstance d'obtenir une logique décidable. En effet, il n'existe pas de procédure complète et cohérente pour résoudre des formules logiques du premier ordre utilisant l'arithmétique linéaire avec des fonctions non-interprétées [Hal91].
- Les types disponibles doivent, si possible, avoir des équivalents dans les langages de modélisations UML4MBT/OCL4MBT. Au minimum, les types entiers et booléens ainsi que les opérateurs usuels sur ces derniers doivent être disponibles.

- Les associations et leurs instanciations présentes dans le diagramme de classes du modèle UML4MBT doivent pouvoir être encodé dans la logique choisie. Les ensembles sont représentables aux travers de vecteurs de bits, de tableaux ou de fonctions non interprétées.
- Un typage fort, si possible, des variables correspondant aux entités du diagramme de classes doit être conservé.
- La logique doit être implémentée par un maximum de prouveurs.
- La logique doit comporter un minimum d'opérateurs inutiles dans le cadre de notre démarche.

L'application simultanée du critère de typage et d'encodage des associations empêche l'utilisation de vecteurs de bits pour représenter les ensembles. En effet, dans ce cas l'ensemble des instances est encodé avec un type commun. Pour départager la représentation par des fonctions non-interprétées ou des tableaux, le critère du nombre de prouveurs compatibles est employé. D'après les compétitions SMT-Comp [BDOS08], les logiques contenant des tableaux sont moins implémentées que celles implémentant des fonctions non-interprétées. En conséquence, les ensembles seront encodés à l'aide de ces dernières.

En résumé, la logique ne doit pas contenir de quantificateurs mais autoriser les fonctions non-interprétées. De plus, elle doit s'appuyer sur la théorie des entiers. D'après le tableau 8.1, les logiques QF_UFLIA et QF_UFIDL peuvent convenir. Cependant QF_UFIDL ne supporte pas tous les opérateurs arithmétiques usuels comme $*$ où /. En conséquence, la logique QF_UFLIA est choisie.

Définition 23 (QF_UFLIA) *La logique QF_UFLIA correspond à l'arithmétique linéaire sur les entiers sans quantificateur utilisant des symboles non-interprétés pour les types, fonctions et prédicats.*

La logique QF_UFLIA est définie dans le langage SMT-lib [BST10]. Cette logique permet de construire des formules sans quantificateur reposant sur une extension arbitraire de la théorie des entiers avec des symboles libres pour les types, fonctions et prédicats contenant seulement des atomes linéaires. Un atome linéaire ne contient pas l'opérateur $*$, /, *div*, *mod* et *abs*. Les atomes peuvent employer ces opérateurs avec des coefficients concrets.

Le tableau 8.2 liste les opérateurs disponibles dans cette logique. Dans ce tableau, la première partie regroupe les opérateurs issus de la théorie booléenne ("Core") La seconde partie présente les opérateurs issus de la théorie des entiers ("Ints"). Dans ce tableau, Bool représente le type booléen, Int le type entier et A un type quelconque. Dans une signature d'un opérateur, le dernier type est le type de retour, les types précédents sont les types des opérandes. La propriété "par couple" de l'opérateur *distinct* signifie que les opérandes ont des valeurs différentes deux à deux.

8.1.2 Objectifs

L'objectif principal de la création du méta-modèle SMT4MBT est la conjugaison au sein d'un modèle unique des notions liées aux tests à partir de modèle et présentées

TABLE 8.2 – Opérateurs utilisables dans la logique QF_UFLIA

Opérateur	Symbole	Signature	Propriété
vrai	true	Bool	
faux	false	Bool	
négation	not	Bool Bool	
implication	$\Rightarrow$	Bool Bool Bool	associativité à droite
et	and	Bool Bool Bool	associativité à gauche
ou	or	Bool Bool Bool	associativité à gauche
ou exclusif	xor	Bool Bool Bool	associativité à gauche
égalité	$=$	A A Bool	
différence globale	distinct	A A Bool	par couple
if then else	ite	Bool A A A	
négation	-	Int Int	
soustraction	-	Int Int Int	associativité à gauche
addition	$+$	Int Int Int	associativité à gauche
multiplication	$*$	Int Int Int	associativité à gauche
division	div	Int Int Int	
modulo	mod	Int Int Int	
absolu	abs	Int Int	
infériorité	$\leqslant$	Int Int Bool	
infériorité stricte	$<$	Int Int Bool	
supériorité	$\geqslant$	Int Int Bool	
supériorité stricte	$>$	Int Int Bool	

à la section 6.3, du système de transitions représentant le système et d'un langage compréhensible par un prouveur SMT.

Par conséquent, le méta-modèle doit permettre d'encoder trois types d'informations. Tout d'abord, le méta-modèle doit pouvoir contenir les informations nécessaires pour animer un modèle. Dans notre cas, il s'agit du système de transitions $M = (D, \Delta, \rho)$ avec D l'espace de recherche, Δ l'ensemble des transitions et ρ l'initialisation du système. Deuxièmement, le méta-modèle doit pouvoir représenter l'ensemble des cibles de test Φ ainsi qu'un objectif de tests. Enfin, le méta-modèle doit pouvoir stocker la réponse du prouveur sous une forme permettant la création de tests.

Un deuxième objectif est de simplifier si possible le processus de conversion. Ainsi lors de la conversion d'un modèle SMT4MBT en un problème compréhensible par un prouveur, les entités le composant doivent rester dans la mesure du possible le plus proche possible d'entités SMT.

Pour relier les entités SMT et les entités UML4MBT, des méta-informations sont ajoutées aux entités SMT4MBT. Ainsi la raison de la création d'une contrainte ou d'une variable est préservée si cela s'avère utile dans une méta-information. Par exemple, une variable SMT4MBT encodant un slot d'une instance contient une méta-information qui stocke cette entité UML4MBT.

Afin d'obtenir un méta-modèle de taille restreinte, seul les éléments indispensables à l'encodage du modèle UML4MBT/OCL4MBT sont conservés. Par exemple, le langage UML4MBT n'autorise que les ensembles contenant des instances car ils permettent de représenter les instanciations des associations entre classes et les opérateurs ensemblistes disponibles ne permettent pas de créer d'ensembles d'un autre type.

8.1.3 Architecture

L'architecture du méta-modèle SMT4MBT est présentée par le diagramme de classes 8.1. Ce diagramme pour des raisons de visibilité ne contient pas les opérations et les attributs des classes. Ces derniers seront explicités dans les explications des différentes entités si cela s'avère nécessaire. Le méta-modèle s'articule autour de quatre éléments principaux, mis en évidence dans le diagramme par une police de taille supérieure, qui sont les types, les fonctions, les contraintes et le benchmark.

Les types

Le méta-modèle défini quatre catégories de type :

Entier Ce type correspond aux entiers. Dans un modèle SMT4MBT, une seule instance de cette classe est créée. Ce type a un équivalent direct dans la logique qui est fournie par la théorie des entiers ("Ints").

Booléen Ce type correspond aux booléens. Dans un modèle SMT4MBT, une seule instance de cette classe est créée. Ce type a un équivalent direct dans la logique qui est fournie par la théorie basique ("Core").

Libre Cette classe représente les types non-interprétés présents dans la logique QF_UFLIA. Ainsi chaque instance créée par l'utilisateur permet de définir un nouveau type. Cette classe est à la base du typage fort fournit par le méta-modèle. En effet, les classes et énumérations contenues dans le diagramme de classes UML4MBT sont encodées à l'aide de types libres. De plus, l'entité UML4MBT à l'origine de la création du type libre est conservée sous forme de méta-information. Pour rappel, les seuls opérateurs employables sur des fonctions d'un type libre sont l'égalité et la différence globale.

Collection Les collections sont des ensembles et peuvent contenir uniquement des éléments de type libre. Cette restriction découle du fait que les deux langages UML4MBT et OCL4MBT emploient uniquement des collections d'instances et que les instances sont encodées avec des fonctions de type libre. Les ensembles n'ont pas d'équivalents directs dans la logique QF_UFLIA.

Les fonctions

Une fonction SMT4MBT est toujours associée à un seul type. L'arité de la fonction dépend de son type. Les fonctions de type entier, booléen ou libre ont une arité de 1 ; les fonctions de type collection sont des applications d'arité 2. De façon similaire aux types libres, si une fonction SMT4MBT a une correspondance directe

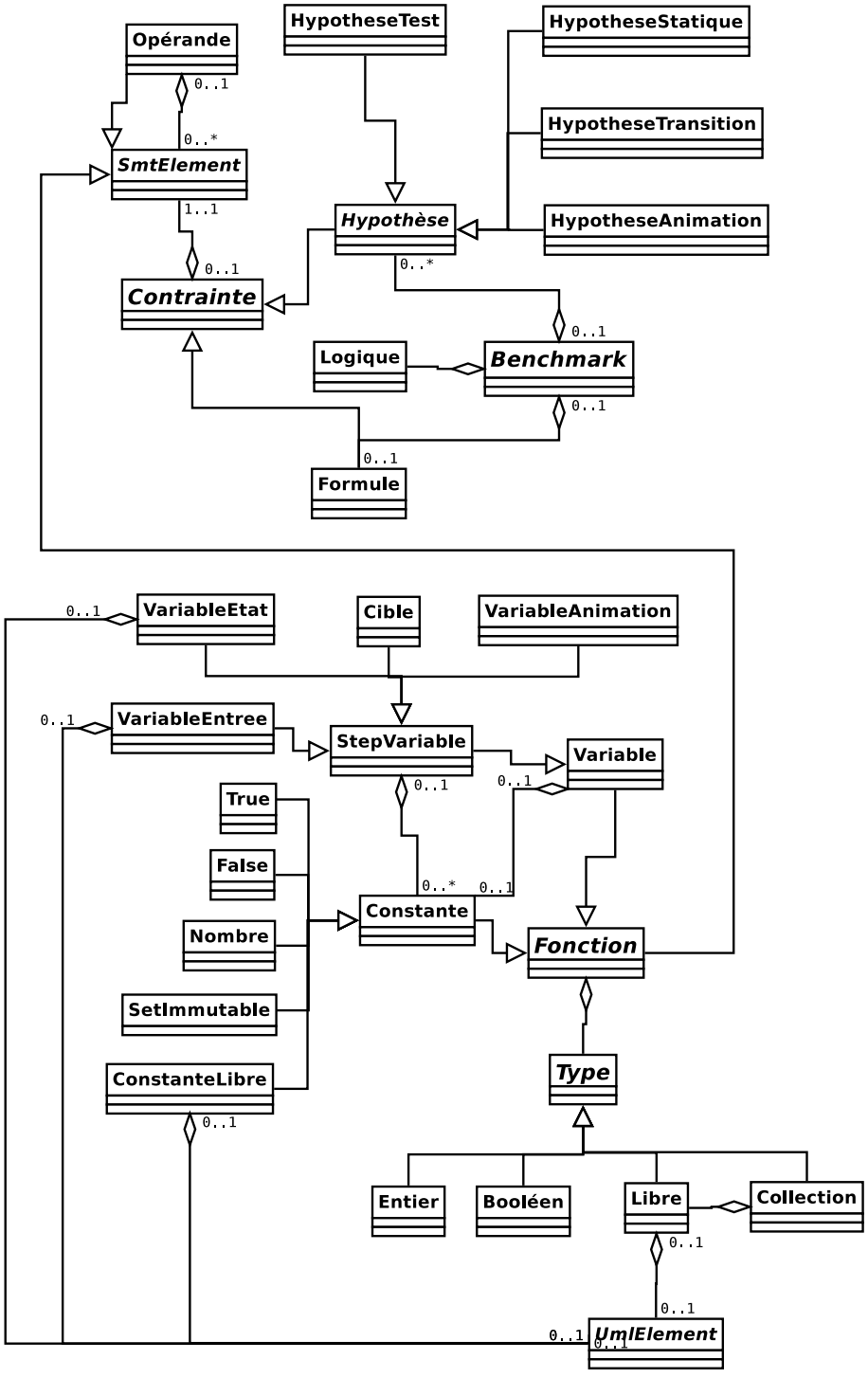


FIGURE 8.1 – Diagramme de classes du méta-modèle SMT4MBT

avec une entité UML4MBT, alors cette entité est précisée via une méta-information stockée sous la forme d'une association avec une instance de la classe UMLElement. Les fonctions du méta-modèle sont déclinées sous trois formes : constante, variable et stepVariable.

Constante Les constantes sont déclinées selon leur type. Une constante de type entier correspond à un nombre. Il existe deux constantes booléennes qui représentent les valeurs *vrai* et *faux*. Les instances de type collection sont employées pour encoder des ensembles dont le contenu n'est jamais modifié. Dans notre cas, ces constantes encodent des ensembles contenant l'ensemble des instances d'une classe. Ces ensembles sont utiles pour itérer sur les instances ou encoder l'opérateur OCL4MBT qui renvoie l'ensemble des instances d'une classe. La dernière catégorie de constantes regroupe les constantes d'un type libre. Lors de la conversion d'une constante ou d'une variable d'un type libre, représentant des entités UML4MBT d'un type identique, les fonctions SMT résultantes sont identiques. Par exemple, dans le cas du robot, la valeur *haute* de l'énumération *Position* et le slot *cote* de l'instance *robot* sont encodés respectivement avec une stepVariable et une constante d'un type libre correspondant à l'énumération *Position*. Après conversion de ces deux entités en des fonctions SMT, le prouveur ne traite pas ces fonctions de manières différenciées. Il n'y a pas de différence d'un point de vue syntaxique. Cependant, il est évident que d'un point de vue sémantique, ces fonctions ont des rôles distincts ; l'une représente une valeur constante, l'autre est une variable dont la valeur évolue au cours de l'animation. Ainsi la différentiation introduite par le méta-modèle SMT4MBT permet de préserver le sens attribué aux entités UML4MBT correspondantes.

Variable Les variables sont des fonctions SMT4MBT qui prennent une unique valeur durant l'animation exécutée par le prouveur. Ces variables sont donc utilisées pour stocker une information valide pour l'ensemble de l'animation. Un usage typique concerne les cibles de test. Ainsi le fait de savoir si une cible est atteinte durant une animation est encodée par une variable booléenne. La valeur d'une variable est représentée sous la forme d'une association avec une constante.

StepVariable les stepVariables sont des fonctions SMT4MBT qui peuvent potentiellement prendre une valeur différente à chaque pas de l'animation. Ces fonctions sont largement majoritaires par rapport aux variables car elles encodent la plupart des entités UML4MBT. Ainsi les variables d'états et d'entrées, définies à la section 6.2, sont encodées par des stepVariables. Les stepVariables sont également employées pour stocker la séquence de transitions exécutée durant l'animation. Dans ce cas, ces stepVariables sont des variables d'animation et leur statut est précisé par une méta-donnée encodée par une instance de variableAnimation. Ainsi l'analyse des variables d'animation et d'états est suffisante pour décrire de manière unique la séquence de transitions et les états correspondants exécutés durant l'animation. Les variables ou les stepVariables créées pour représenter des cibles de test sont systématiquement de type booléen.

Les contraintes

Une contrainte est une formule logique construite à l'aide des opérateurs du tableau 8.3. Dans ce tableau, A représente un type quelconque. Dans les signatures des opérateurs, le dernier type est le type de retour ; les autres sont les types des opérandes. L'opérateur d'appartenance renvoie 1 si l'élément est présent dans la collection et 0 sinon. Le choix de ne pas renvoyer un booléen permet une facilité d'écriture pour les opérateurs ensemblistes, notamment la cardinalité. L'écriture de ces opérateurs est donnée dans la section suivante. Il existe les deux types de contraintes suivantes :

Hypothèse Une hypothèse est une formule logique que l'on suppose vraie durant la résolution du problème par le prouveur. De même que pour les variables, les raisons ayant présidées à la création d'une hypothèse sont conservées sous la forme d'une méta-donnée qui est une instance d'une des classes suivantes : *hypotheseTest*, *hypotheseStatique*, *hypotheseTransition* et *hypotheseAnimation*. Ces méta-données permettent de classifier les contraintes. Ainsi les contraintes liées aux cibles de test appartiennent aux hypothèses de tests. Les contraintes définissant l'état initial du système, issues de ρ dans le système de transition ou d'un état rencontré lors d'une animation valide précédente, sont des hypothèses statiques. Les hypothèses de transitions sont l'encodage de Δ, ensemble des transitions du système de transitions. Finalement les hypothèses d'animation correspondent aux contraintes utilisées pour obliger le prouveur à exécuter une séquence de transitions. Un exemple de contrainte est l'impossibilité d'exécuter simultanément deux transitions.

Formule est une formule logique pour laquelle l'existence d'une solution est inconnue. D'un point de vue pratique, il est possible de remplacer cette formule par une hypothèse mais cette différentiation peut améliorer les performances de certains prouveurs et elle permet de respecter la structure d'un problème telle que spécifiée par SMT-lib. Dans notre cas, cette formule encode l'objectif de tests du scénario d'animations. Si cette formule est absente du problème, alors le prouveur doit systématiquement retourner une solution.

Le benchmark

Le Benchmark est la représentation d'un scénario d'animation qui peut être soumise à un prouveur après une conversion en un problème écrit en SMT-lib. Il est composé d'un ensemble d'hypothèses, d'une formule et d'une logique. Par défaut, la logique choisie est QF_UFLIA. Mais il est possible d'employer une logique plus restreinte en fonction des entités UML4MBT présentes dans le modèle. Par exemple, si dans un modèle il n'y a pas d'association et d'opérateurs ensemblistes, on peut utiliser une logique sans fonction et types non-interprétés comme QF_LIA.

Le méta-modèle est également capable de représenter une réponse du prouveur. Le statut, SAT si une solution existe, UNSAT sinon, est stocké par un attribut de la classe Benchmark. Les valeurs des variables et des stepVariables sont précisées au travers d'associations de ces dernières avec des fonctions constantes.

TABLE 8.3 – Opérateurs disponibles dans le méta-modèle SMT4MBT

Opérateur	Symbole	Signature	Propriété
négation	$\neg$	Booléen Booléen	
et	$\wedge$	Booléen Booléen Booléen	associativité à gauche
ou	$\vee$	Booléen Booléen Booléen	associativité à gauche
ou exclusif	$\otimes$	Booléen Booléen Booléen	
implication	$\Rightarrow$	Booléen Booléen Booléen	
égalité	$=$	A A Booléen	
différence globale	distinct	A A Booléen	par couple
if then else	ite	Booléen A A A	
négation	-	Entier Entier	
soustraction	-	Entier Entier Entier	associativité à gauche
addition	$+$	Entier Entier Entier	associativité à gauche
multiplication	$*$	Entier Entier Entier	associativité à gauche
division	div	Entier Entier Entier	
modulo	mod	Entier Entier Entier	
infériorité	$\leqslant$	Entier Entier Booléen	
infériorité stricte	$<$	Entier Entier Booléen	
supériorité	$\geqslant$	Entier Entier Booléen	
supériorité stricte	$>$	Entier Entier Booléen	
appartenance	has	Collection Libre Entier	

8.2 Règles de conversion vers SMT4MBT

La conversion entre le modèle UML4MBT et le modèle SMT4MBT est formalisée dans cette section à l'aide de règles. Pour écrire ces règles, cette section commence par introduire une notation pour représenter et interagir avec les entités des deux modèles. Ensuite, en s'appuyant sur cette notation, les trois parties suivantes présentent l'ensemble des règles de conversion en se focalisant successivement sur la traduction de la partie statique du modèle décrite dans les diagrammes de classes et d'objets, la transformation d'une contrainte OCL4MBT en une contrainte SMT4MBT et la création de règles assurant l'encodage d'un scénario d'animations.

8.2.1 Notations

Notation UML4MBT La définition formelle des règles de conversion entre un modèle UML4MBT et un modèle SMT4MBT nécessite d'introduire une notation précise pour manipuler les entités de ces modèles. Ainsi le modèle UML4MBT est vu sous la forme d'un 7-uplet $\Gamma_{uml} = \{Class, Slot, Enum, Link, State, Op, Tr\}$ où :

Class regroupe l'ensemble des classes du modèle issues du diagramme de classes.

Slot contient l'ensemble des attributs des instances des classes.

Enum est l'ensemble des énumérations.

Link représente toutes les instances des associations du modèle.

State contient les états du diagramme d'états-transitions. Cet ensemble est vide si le modèle UML4MBT n'utilise pas le diagramme d'états-transitions.

Op regroupe les opérations définies au sein des différentes classes du modèle.

Tr liste les différentes transitions UML4MBT entre les états du diagramme d'états-transitions. Cet ensemble est vide si le modèle UML4MBT n'utilise pas le diagramme d'états-transitions.

L'écriture des règles nécessite également la définition des opérateurs suivants :

inst(A) renvoie l'ensemble des instances de la classe A si $\Gamma_{uml} \models A \in Class$, c'est à dire si A est une classe présente dans le modèle UML.

var() renvoie l'ensemble des variables d'états.

enum(A) renvoie l'ensemble des valeurs de l'énumération A si $\Gamma_{uml} \models A \in Enum$.

pre(A) retourne la pré-condition associée à l'entité A si $\Gamma_{uml} \models A \in Op \vee A \in Tr$.

post(A) retourne la post-condition associée à l'entité A si $\Gamma_{uml} \models A \in Op \vee A \in Tr$.

param(A) renvoie l'ensemble des paramètres d'entrées et de sorties de l'opération A si $\Gamma_{uml} \models A \in Op$.

in(A) renvoie l'état de départ du diagramme d'états-transitions associé à la transition UML4MBT A si $\Gamma_{uml} \models A \in Tr$.

out(A) renvoie de manière similaire l'état d'arrivée de la transition UML4MBT A si $\Gamma_{uml} \models A \in Tr$.

Notation SMT4MBT A l'instar du modèle UML4MBT, le modèle SMT4MBT est défini par triplet $\Gamma_{smt} = \{T, F, C\}$ où :

T regroupe l'ensemble des types du modèle SMT4MBT. Par défaut, cet ensemble contient les types booléen et entier notés *Bool* et *Int*. Les types libres sont notés *nom* : *Named* où *nom* est le nom du type. Les types correspondant à des ensembles sont écrits *nom* : *eltNom* : *Set* où *nom* est le nom du type et *eltNom* est le type des éléments de l'ensemble.

F est l'ensemble des fonctions du modèle. Une fonction peut être une constante, une variable ou une stepVariable. Une fonction est écrite avec le formalisme suivant :

$$fun_{cat}(name : type)$$

fun prend une valeur spécifique pour chaque classe de fonction. Cette valeur est *var* pour une variable, *svar* pour une stepVariable et *con* pour une constante. Dans ce dernier cas, pour simplifier l'écriture, les valeurs booléennes sont notées *true* et *false* et les nombres entiers peuvent être écrits directement. Ainsi, $con(true : Bool)$ est équivalent à *true*.

cat précise la catégorie de la fonction si elle existe. Pour rappel, les catégories définies en 8.1 indiquent le rôle joué lors de l'exécution du scénario d'animations du modèle par cette fonction. Ce paramètre prend la valeur : *state* pour les fonctions d'états, *input* pour les fonctions d'entrées,

animation pour les fonctions liées à l'animation et *test* pour les fonctions reliées aux objectifs de tests.

name est le nom de la fonction.

type est le type de la fonction qui peut être booléen, entier ou libre.

Un exemple de fonction écrit avec ce formalisme est $svar_{state}(att1 : Bool)$ qui correspond à une stepVariable booléenne, nommée $att1$, et qui appartient à l'ensemble des variables d'états.

C contient l'ensemble des contraintes du modèle SMT4MBT.

Notation Règle La règle R-example présente la notation. Dans la règle, x_{uml} est l'entité UML4MBT ou OCL4MBT à convertir. $condition_{uml}$ est la condition portant sur des entités UML4MBT autorisant l'application de la règle. Le résultat de la conversion est représenté par x_{smt}. Finalement les entités *types*, *fonctions* et *contraintes* sont les entités ajoutées au modèle SMT4MBT durant la conversion.

$$\frac{\Gamma_{uml} \models x_{uml} \mid condition_{uml}}{\Gamma_{smt} \models x_{smt} \mid types \in T, fonctions \in F, contraintes \in C} \quad \text{(R-example)}$$

Si plusieurs règles s'appliquent car leurs conditions sont satisfaites simultanément, leur ordre d'application est arbitraire. Si la conversion concerne un opérateur ou une variable du langage OCL4MBT, l'ordre de précédence des opérateurs est celui défini de manière usuelle dans les formules logiques et arithmétiques.

8.2.2 Conversion pour le langage UML4MBT

Dans cette partie, les règles régissant la conversion des éléments de la partie statique du modèle UML4MBT issus du diagramme de classes et d'objets dans un modèle SMT4MBT sont présentées. L'idée est d'associer une fonction SMT à chaque élément du modèle UML4MBT puis de rajouter si nécessaire des formules pour respecter les contraintes portant sur ces éléments.

Les slots, instances des attributs des classes, sont converties en stepVariables SMT4MBT qui font parties des variables d'états. En UML4MBT, un slot est de l'un des types suivants : entier, entier borné, booléen, énumération. Chaque cas est traité par une règle particulière (respectivement R-Slot-int, R-Slot-range, R-Slot-bool et R-Slot-enum) chargée de créer la fonction et le cas échéant de contraindre son domaine aux valeurs autorisées.

Par exemple, l'application de la règle R-Slot-enum pour convertir le slot *position* de l'instance *robot* de la classe *robot* de notre exemple fil rouge entraîne la création de la stepVariable $svar_{state}(robot_{position} : Position)$ et de la contrainte $robot_{position} = haut \vee robot_{position} = bas$. Pour des raisons de facilité de lecture, les noms des variables sont simplifiés par l'omission d'un identifiant unique précédent le nom de l'entité UML4MBT. Ainsi un nom s'écrit en réalité $id1_x$ et non x avec $id1$ un identifiant unique. Cette simplification est conservée dans les exemples suivants.

$$\frac{\Gamma_{uml} \models x \mid x \in Slot \wedge x \in \mathbb{N}}{\Gamma_{smt} \models svar_{state}(x : Int) \mid x \in F} \quad \text{(R-Slot-int)}$$

$$\frac{\Gamma_{uml} \models x \mid x \in Slot \wedge x \in \mathbb{N} \wedge x_{min} \leq x \leq x_{max}}{\Gamma_{smt} \models svar_{state}(x : Int) \mid (x \leq x_{max} \wedge x \geq x_{min}) \in C} \quad \text{(R-Slot-range)}$$

$$\frac{\Gamma_{uml} \models x \mid x \in Slot \wedge x \in \{false, true\}}{\Gamma_{smt} \models svar_{state}(x : Bool) \mid x \in F} \quad \text{(R-Slot-bool)}$$

$$\frac{\Gamma_{uml} \models x \mid x \in Slot \wedge e \in Enum \wedge l \in enum(e) = \{l_0, l_1, \ldots, l_n\} \wedge x = l}{\Gamma_{smt} \models svar_{state}(x : e) \mid x \in F, (\bigvee_{j=0}^{n} x = l_j) \in C} \quad \text{(R-Slot-enum)}$$

Les paramètres des opérations des classes du modèle UML4MBT sont des paramètres d'entrées. Ils sont convertis en des stepVariables par les règles R-Param-int, R-Param-bool, R-Param-enum, R-Param-inst en fonction de leur type, respectivement entier, booléen, énumération et classe. Ensuite une contrainte est ajoutée si nécessaire pour restreindre leur domaine aux valeurs autorisées. Dans le cas où le paramètre est de type classe, son domaine est limité aux instances de la classe déclarées dans le diagramme d'objets. Une variable de type classe prend pour valeur les instances de cette classe.

Par exemple, la conversion du paramètre C de l'opération *rotation* de la classe *robot* par la règle R-Param-enum crée la stepVariable $svar_{input}(C : Cote)$ et la contrainte $C = gauche \vee C = droite$.

$$\frac{\Gamma_{uml} \models x \mid x \in Param \wedge x \in \mathbb{N}}{\Gamma_{smt} \models svar_{input}(x : Int) \mid x \in F} \quad \text{(R-Param-int)}$$

$$\frac{\Gamma_{uml} \models x \mid x \in Param \wedge x \in \{false, true\}}{\Gamma_{smt} \models svar_{input}(x : Bool) \mid x \in F} \quad \text{(R-Param-bool)}$$

$$\frac{\Gamma_{uml} \models x \mid x \in Param \wedge e \in Enum \wedge l \in enum(e) = \{l_0, l_1, \ldots, l_n\} \wedge x = l}{\Gamma_{smt} \models svar_{input}(x : e) \mid x \in F, (\bigvee_{j=0}^{n} x = l_j) \in C} \quad \text{(R-Param-enum)}$$

$$\frac{\Gamma_{uml} \models x \mid x \in Param \wedge c \in Class \wedge i \in inst(c) = \{i_0, i_1, \ldots, i_n\} \wedge x = i}{\Gamma_{smt} \models svar_{input}(x : c) \mid x \in F, (\bigvee_{j=0}^{n} x = i_j) \in C} \quad \text{(R-Param-inst)}$$

La règle R-Class convertit les classes du modèles UML4MBT en des types libres. Elle crée également une contrainte pour différencier les valeurs des fonctions représentant les instances de la classe. Cette règle permet de conserver un typage fort mais entraîne une difficulté sur l'encodage de la valeur *null*. Cette valeur est utilisée dans le modèle UML4MBT pour représenter une valeur non définie pour un lien de cardinalité 1. Il est impossible de l'encoder à l'aide d'une unique fonction à cause

du typage fort introduit par la création d'un type libre différent pour chaque classe. En conséquence, une valeur nulle spécifique à chaque classe est créée sous la forme d'une fonction constante. Ce choix est valide car le langage UML4MBT n'autorise pas le polymorphisme.

Par exemple la conversion de la classe *quaiDechargement* de l'exemple fil rouge avec cette règle crée trois entités : le type (*quaiDechargement* : *Named*), la constante $con(quaiDechargement_{none} : quaiDechargement)$ représentant la valeur nulle pour cette classe et la contrainte *distinct quaiG quaiD*.

$$\frac{\Gamma_{uml} \models x \mid x \in Class \wedge inst(x) = \{i_0, i_1, \ldots, i_n\}}{\Gamma_{smt} \models x : Named \mid x \in T, con(x_{none} : x) \in F, (distinct\ \ i_0, \ldots, i_n) \in C} \quad \text{(R-Class)}$$

Les instances sont converties en des fonctions constantes du type libre correspondant à leur classe avec la règle R-Instance.

Par exemple, l'application de cette règle pour convertir l'instance quaiG du modèle robot entraîne la création de la constante $con(quaiG : quaiDechargement)$.

$$\frac{\Gamma_{uml} \models x \mid x \in inst(class)}{\Gamma_{smt} \models con(x : class) \mid x \in F} \quad \text{(R-Instance)}$$

Les énumérations sont converties en des types libres à l'aide de la règle R-Enum. Dans cette règle, les variables l_i représentent les différentes valeurs de l'énumération. Cette règle est également responsable de la création d'une contrainte chargée de différencier ces valeurs pour le prouveur.

Par exemple, la conversion de l'énumération *Cote* de l'exemple fil rouge par cette règle crée le type (*Cote* : *Named*) et la contrainte *distinct gauche droite*.

$$\frac{\Gamma_{uml} \models x \mid x \in Enum \wedge l \in enum(x) = \{l_0, l_1, \ldots, l_n\}}{\Gamma_{smt} \models x : Named \mid x \in T, distinct\ \ l_0, \ldots, l_n \in C} \quad \text{(R-Enum)}$$

Après que le type libre correspondant à une énumération ait été créé, la règle R-EnumValue est responsable de la création de fonctions constantes pour chacune des valeurs des énumérations. Ces fonctions sont du type libre de l'énumération à laquelle appartient la valeur.

Par exemple, cette règle appliquée à la valeur *gauche* de l'énumération *Cote* crée la constante $con(gauche : Cote)$.

$$\frac{\Gamma_{uml} \models x \mid x \in enum(e)}{\Gamma_{smt} \models con(x : e) \mid x \in F} \quad \text{(R-EnumValue)}$$

La conversion des liens, instances des associations entre les classes, diffère selon la multiplicité. Si la multiplicité vaut 0..1 ou 1..1, alors le processus est traité de manière analogue à la conversion des slots de type instance. Ce cas est traité par la règle R-Link-single. Dans cette règle, $x_{a1 \to b}$ est un lien unidirectionnel de l'instance $a1$ de la classe a à une instance de la classe b et $|x|$ est la cardinalité de l'association correspondante. Ainsi un lien de ce type est représenté dans le modèle SMT4MBT par une stepVariable, variable d'état, du type libre correspondant à la

classe b. Une contrainte supplémentaire limite les valeurs de cette stepVariable aux constantes représentant les instances du modèle UML4MBT. L'intérêt de différencier la conversion des liens en fonction de leur multiplicité est d'obtenir de meilleures performances car les contraintes sur les ensembles sont plus complexes.

Par exemple, la conversion du lien reliant l'instance *quaiA* de la classe *quaiChargement* à une instance de la classe *robot* par cette règle crée la stepVariable $svar_{state}(robotA : robot)$ et la contrainte $robotA = robot$.

$$\frac{\Gamma_{uml} \models x \mid x = x_{a1\to b} \in Link \wedge max(|x|) = 1 \wedge inst(b) = \{i_0, i_1, \ldots, i_n\}}{\Gamma_{smt} \models svar_{state}(x_{a1b} : b) \mid x_{a1b} \in F, (\bigvee_{j=0}^{n} x_{a1b} = i_j) \in C} \quad \text{(R-Link-single)}$$

Dans le cas où la multiplicité du lien est supérieure ou égale à 2, la conversion est effectuée grâce à la règle R-Link-set. Dans cette règle, $x_{a1\to b}$ est un lien unidirectionnel de l'instance $a1$ de la classe a à des instances la classe b et $|x|$ est la cardinalité de l'association correspondante. Si la conversion conduit également à obtenir une stepVariable à l'instar de la règle précédente, son type change pour être un ensemble contenant des éléments du type libre représentant la classe b. La contrainte créée spécifie les fonctions pouvant appartenir à l'ensemble. Pour rappel, un ensemble est vu comme une application de b, le type de l'élément, vers $\{0, 1\}$ où 1 indique l'appartenance.

$$\frac{\Gamma_{uml} \models x \mid x = x_{a\to b} \in Link \wedge max(|x|) > 1 \wedge inst(b) = \{i_0, i_1, \ldots, i_n\}}{\begin{array}{r}\Gamma_{smt} \models svar_{state}(xab : set_x) \mid (set_x : x : Set) \in T, xab \in F, \\ (\bigwedge_{j=0}^{n} has(xab, i_j) = 0 \vee has(xab, i_j) = 1) \in C\end{array}} \quad \text{(R-Link-set)}$$

Si les liens sont bidirectionnels, il suffit d'ajouter une contrainte pour représenter cette réciprocité. Dans le cas du lien $x_{a1\leftrightarrow b1}$ bidirectionnel entre les instances $a1$ et $b1$ des classes a et b avec des multiplicités supérieure à 2, la contrainte est $\bigwedge_{j=0}^{nb} \bigwedge_{i=0}^{na} has(xaib, b_j) = has(xabj, a_i)$ avec $inst(b) = \{j_0, j_1, \ldots, j_{nb}\}$ et $inst(a) = \{i_0, i_1, \ldots, i_{na}\}$. Dans le cas d'un lien ayant une multiplicité de 1, les opérateurs *has* portant sur ce lien sont directement remplacés par la stepVariable encodant ce lien.

Le diagramme de classes peut contenir une valeur par défaut pour les attributs des classes et le diagramme d'objets peut également spécifier les valeurs des slots et des liens. Ces assignations ne sont pas traitées ici car elles sont équivalentes à des assignations écrites dans le langage OCL4MBT. Par conséquent, leur conversion emploie les règles explicitées dans la sous-section suivante.

8.2.3 Conversion pour le langage OCL4MBT

Cette partie détaille les règles pour convertir du code OCL4MBT présent dans les pré-conditions et les post-conditions des opérations ainsi que dans les gardes et actions des transitions UML4MBT en formules logiques compatibles avec le métamodèle SMT. Le langage OCL4MBT est contextuel. Par conséquent, un opérateur peut avoir un sens différent en fonction du contexte. Ainsi, les règles présentées

dans cette sous-section concernent dans l'ordre le code OCL4MBT indépendant du contexte, lié au contexte *évaluation* et soumis au contexte *assignation*. Ces contextes ont été présentés en détail dans la section 4.

Contexte neutre Les règles R-Access-int, R-Access-bool, R-Access-enum et R-Access-inst sont chargées de convertir les accès à un slot ou à un lien par une variable intermédiaire de type libre. Les accès ne sont pas dépendant du contexte OCL4MBT utilisé. Dans ces règles, nous avons $inst(c) = \{c_0, \ldots, c_n\}$. Ces règles traitent respectivement le cas ou le slot auquel on souhaite accéder est de type entier, booléen ou énumération. La dernière règle permet d'accéder à un slot de type classe ou à un lien de cardinalité 1.

Le principe de ces règles est de créer une nouvelle stepVariable qui représente l'accès puis d'ajouter une contrainte qui force cette fonction a prendre la valeur de l'élément auquel elle donne accès. La contrainte utilise le fait que le nombre d'instances dans le modèle UML4MBT est constant. En conséquence, il est possible d'itérer sur ces instances pour obtenir des contraintes telles que "si l'instance au travers de laquelle j'accède à ma variable est l'instance A alors la variable nouvellement créée est égale au slot ou lien appartenant à A".

Finalement le cas spécifique où l'accès est impossible suite à une valeur nulle est traité de manière préventive en rajoutant une condition sur le code OCL4MBT qui supprime ce cas de figure.

Par exemple, la traduction de la formule $self.quaiG.piece$ qui représente un accès au slot *piece* de l'instance stockée par le lien *quaiG* de l'instance *robot* par la règle R-Access-enum crée la fonction $svar(access : Piece)$ et la contrainte $quaiG_{link} = quaiG \Rightarrow access = quaiG\_piece \wedge quaiG_{link} = quaiD \Rightarrow access = quaiD\_piece$.

$$\frac{\Gamma_{uml} \models var.x \mid x \in \mathbb{N} \wedge x \in Slot \wedge var \in inst(c) \wedge c \in Class}{\Gamma_{smt} \models svar(access : Int) \mid access \in F, (\bigwedge_{i=0}^{n} var = c_i \Rightarrow access = c_i.x) \in C} \text{(R-Access-int)}$$

$$\frac{\Gamma_{uml} \models var.x \mid x \in \{true, false\} \wedge x \in Slot \wedge var \in inst(c) \wedge c \in Class}{\Gamma_{smt} \models svar(access : Bool) \mid access \in F, (\bigwedge_{i=0}^{n} var = c_i \Rightarrow access = c_i.x) \in C} \text{(R-Access-bool)}$$

$$\frac{\Gamma_{uml} \models var.x \mid x \in enum(e) \wedge x \in Slot \wedge var \in inst(c) \wedge c \in Class}{\Gamma_{smt} \models svar(access : e) \mid access \in F, (\bigwedge_{i=0}^{n} var = c_i \Rightarrow access = c_i.x) \in C} \text{(R-Access-enum)}$$

$$\frac{\Gamma_{uml} \models var.x \mid x \in inst(b) \wedge (x \in Slot \vee x \in Link) \wedge var \in inst(c) \wedge c, b \in Class}{\Gamma_{smt} \models svar(access : b) \mid access \in F, (\bigwedge_{i=0}^{n} var = c_i \Rightarrow access = c_i.x) \in C} \text{(R-Access-inst)}$$

Phi-fonction Sous le contexte d'assignation, OCL4MBT devient un langage impératif. Par conséquent, il est alors possible d'effectuer successivement une série d'assignations sur la même variable. Pour convertir ce comportement dans une formule logique compatible avec le méta-modèle SMT4MBT, la méthode retenue est de convertir la formule sous la forme *Single State Assignment* définie dans [CF89] pour assurer l'unicité de chaque fonction subissant une assignation. Par exemple, le code $a = 1 \wedge a = a + 1$ devient $a_0 = 1 \wedge a_1 = a_0 + 1$. De plus, une assignation est par définition toujours possible et valide. Par conséquent, tous les opérateurs employés dans ce contexte doivent renvoyer la valeur booléenne vrai.

Cependant, cette solution met en évidence qu'une entité UML4MBT peut être convertie en de multiples fonctions SMT4MBT. Ainsi dans l'exemple précédent, la variable a est associée à deux nouvelles variables a_0 et a_1. Pour relier les différentes fonctions représentant un même élément, une phi-fonction nommée φ est introduite. Le comportement de φ dépend du contexte. Dans le contexte d'assignation, φ renvoie une nouvelle fonction SMT4MBT créée par copie. Dans le contexte d'évaluation, φ renvoie la dernière fonction SMT4MBT correspondant à l'entité UML4MBT.

Contexte d'assignation La règle R-Assign-basic utilisée dans le contexte assignation du langage OCL4MBT convertit une assignation d'une valeur à une fonction de type entier, booléen, classe ou énumération. La fonction SMT4MBT v_1 subissant l'assignation ne peut pas être une constante. En effet dans le méta-modèle SMT4MBT, une constante est déjà la représentation d'une valeur.

$$\frac{\Gamma_{uml} \models v_1 = v_2 \mid v_1, v_2 \in \mathbb{N} \vee v_1, v_2 \in \mathit{Enum} \vee v_1, v_2 \in \mathit{inst}(c) \vee v_1, v_2 \in \mathit{Bool}}{\Gamma_{smt} \models \varphi(v_1) = v_2 \mid \varphi(v_1) \in F} \quad \text{(R-Assign-basic)}$$

La règle R-Assign-undefined convertit l'opérateur *oclIsUndefined* appartenant au langage OCL4MBT. Cet opérateur est utilisé pour indiquer qu'un lien de cardinalité 1 ne contient pas d'instance. Cette absence est représentée par la valeur nulle du type de l'élément contenu dans le lien.

$$\frac{\Gamma_{uml} \models x.\mathit{oclIsUndefined}() \mid x \in \mathit{inst}(c)}{\Gamma_{smt} \models \varphi(x) = c_{none} \mid \varphi(x) \in F} \quad \text{(R-Assign-undefined)}$$

L'assignation d'une valeur à un ensemble d'instances est convertie par la règle R-Assign-set. Cette assignation est équivalente à une itération sur les instances de la classe des éléments de l'ensemble durant laquelle une contrainte précise si chaque élément est présent ou non dans l'ensemble. Dans cette règle et les règles suivantes R-Assign-include, R-Assign-exclude, R-Assign-empty et R-Assign-forAll, la relation $\mathit{inst}(c) = \{i_0, i_1, \ldots, i_n\}$ est respectée.

$$\frac{\Gamma_{uml} \models x_1 = x_2 \mid x_1, x_2 \subseteq \mathit{inst}(c)}{\Gamma_{smt} \models \bigwedge_{i=0}^{n} \mathit{has}(\varphi(x_1), i_i) = \mathit{has}(x_2, i_i) \mid \varphi(x) \in F} \quad \text{(R-Assign-set)}$$

Les règles R-Assign-include et R-Assign-exclude convertissent les opérateurs d'inclusion et d'exclusion du langage OCL4MBT dans le contexte d'assignation. La

conversion commence par créer une stepVariable représentant un nouvel ensemble de même type que celui sur lequel s'applique l'opérateur puis ajoute les éléments présents dans l'ensemble d'origine à l'exception de l'élément à traiter qui est inclus ou exclus selon l'opérateur à convertir. Pour rappel, $has(x, i) = 1$ se lit i appartient à x et $has(x, i) = 0$ se lit i n'appartient pas à x. La notation $\bigwedge_{j=0,a+1}^{a-1,n} fonction(j)$ est équivalente à $\bigwedge_{j=0}^{a-1} fonction(j) \bigwedge_{a+1}^{n} fonction(j)$.

$$\frac{\Gamma_{uml} \models x.includes(i_a) \mid x \subseteq Inst(c) \wedge i_a \in Inst(c)}{\Gamma_{smt} \models has(\varphi(x), i_a) = 1 \bigwedge_{j=0,a+1}^{a-1,n} has(\varphi(x), i_j) = has(x, i_j) \mid \varphi(x) \in F} \quad \text{(R-Assign-include)}$$

$$\frac{\Gamma_{uml} \models x.excludes(i_a) \mid x \subseteq Inst(c) \wedge i_a \in Inst(c)}{\Gamma_{smt} \models has(\varphi(x), i_a) = 0 \bigwedge_{j=0,a+1}^{a-1,n} has(\varphi(x), i_j) = has(x, i_j) \mid \varphi(x) \in F} \quad \text{(R-Assign-exclude)}$$

La règle R-Assign-empty convertit l'opérateur qui assigne l'ensemble vide. Cette conversion est effectuée en itérant sur les instances de la classe des éléments de l'ensemble pour spécifier que ces instances n'appartiennent pas à l'ensemble.

$$\frac{\Gamma_{uml} \models x.empty() \mid x \subseteq Inst(c)}{\Gamma_{smt} \models \bigwedge_{j=0}^{n} has(\varphi(x), i_j) = 0 \mid \varphi(x) \in F} \quad \text{(R-Assign-empty)}$$

L'opérateur *forAll* dans le contexte d'assignation permet d'exécuter une formule booléenne, notée *expr*, pour chacun des éléments présents dans l'ensemble. Ainsi la règle R-Assign-forAll génère une contrainte qui vérifie si une instance est présente et dans ce cas applique l'expression.

$$\frac{\Gamma_{uml} \models x.forAll(expr(i_j)) \mid x \subseteq Inst(c)}{\Gamma_{smt} \models \bigwedge_{j=0}^{n} has(x, i_j) \Rightarrow expr(i_j)} \quad \text{(R-Assign-forAll)}$$

La conversion des appels d'opérations est plus complexe. Un appel d'opération est composé par trois parties : une opération définie à l'aide d'une pré-condition et d'une post-condition, une liste de valeurs associée aux paramètres d'entrées et éventuellement un paramètre de sortie. Dans le contexte d'assignation d'après le langage OCL4MBT, un appel ne doit pas avoir de paramètre de sortie et peut modifier l'état du système. Le processus de conversion d'un appel d'une opération op_1 comporte les quatre étapes suivantes :

1. Déterminer si l'appel est valide en contrôlant que la pré-condition de l'opération appelée est vérifiée avec les valeurs passées en paramètre. Par conséquent, cette condition est ajoutée dans le précédent contexte d'évaluation du code OCL4MBT. En pratique, ce contexte est rencontré dans la condition de l'opérateur *"if then else"*, dans la pré-condition de l'opération appelante ou dans la garde de la transition UML4MBT appelante.
2. Remplacer les paramètres d'entrées par leurs valeurs.

3. Substituer l'appel par une nouvelle stepVariable, $svar(op_{1,call} : Bool)$. Cette fonction est vraie quand l'appel de la fonction est exécuté au cours de l'animation.
4. Créer une contrainte qui implique la réalisation de la postcondition de l'opération si la stepVariable créée précédemment prend la valeur vraie : $op_{1,call} \Rightarrow post(op_1)$.

Contexte d'évaluation Sous le contexte d'évaluation, OCL4MBT respecte la sémantique standard du langage OCL. Pour rappel, dans ce contexte, φ renvoie la dernière fonction créée.

Les premières règles ont pour objectif de convertir les valeurs constantes et d'accéder aux fonctions créées par *varphi* dans le contexte d'assignation et aux fonctions résultant de la conversion des entités UML4MBT. Ainsi R-Eval-bool, R-Eval-number créent les constantes correspondant aux valeur vrai, faux et aux nombres et la règle R-Eval-var retourne la fonction SMT4MBT correspondant à l'entité UML4MBT.

$$\frac{\Gamma_{uml} \models x \mid x \in Bool \wedge x \notin Slot \wedge x \notin Param}{\Gamma_{smt} \models x} \quad \text{(R-Eval-bool)}$$

$$\frac{\Gamma_{uml} \models x \mid x \in \mathbb{N} \wedge x \notin Slot \wedge x \notin Param}{\Gamma_{smt} \models x} \quad \text{(R-Eval-number)}$$

$$\frac{\Gamma_{uml} \models x \mid x \in Slot \vee x \in Link \vee x \in Param \vee x \in Inst(e)}{\Gamma_{smt} \models \varphi(x)} \quad \text{(R-Eval-var)}$$

Les règles R-Eval-direct1, R-Eval-direct2 et R-Eval-different convertissent les opérateurs ayant un équivalent direct ou quasiment directe comme l'opérateur différent. Ainsi le symbole $\diamondsuit$ peut être remplacé par $\neg$ ou $-$ dans la règle R-Eval-direct1 et par $=, \neg, \vee, \wedge, \otimes, <, >, \leqslant, \geqslant, +, -, *$ ou *div* dans la règle R-Eval-direct2.

$$\frac{\Gamma_{uml} \models \diamondsuit x_1 \mid x_1 \in \mathbb{N} \vee x_1 \in Bool}{\Gamma_{smt} \models \diamondsuit x_1} \quad \text{(R-Eval-direct1)}$$

$$\frac{\Gamma_{uml} \models x_1 \diamondsuit x_2 \mid x_1, x_2 \in \mathbb{N} \vee x_1, x_2 \in Enum \vee x_1, x_2 \in inst(c) \vee x_1, x_2 \in Bool}{\Gamma_{smt} \models x_1 \diamondsuit x_2} \quad \text{(R-Eval-direct2)}$$

$$\frac{\Gamma_{uml} \models x_1 <> x_2 \mid x_1, x_2 \in \mathbb{N} \vee x_1, x_2 \in Enum \vee x_1, x_2 \in inst(c) \vee x_1, x_2 \in Bool}{\Gamma_{smt} \models \neg(x_1 = x_2)} \quad \text{(R-Eval-different)}$$

Un problème rencontré lors de la conversion de l'opérateur *if then else* est qu'après son exécution l'état du modèle peut diverger en fonction de la branche exécutée. Par exemple, la formule *if b then b = false else true* où b est prédicat

devient sous la forme *Single State Assignment if* b_0 *then* $b_1 = false$ *else true* où b_0 et b_1 sont des prédicats. Par conséquent, un appel à la fonction $\varphi(b)$ après l'opérateur "if then else" retourne une valeur différente pour chaque branche. Dans la branche "then", $\varphi(b)$ vaut b_1. Dans la branche "else", $\varphi(b)$ vaut b_0. En conséquence, il est nécessaire de disposer d'une méthode pour unifier les états du système. La méthode retenue est l'emploi de la Φ fonction décrite dans l'article [CF89]. L'idée est de créer une nouvelle fonction pour chacune des fonctions présentes dans une des branches de l'opérateur et d'assigner dans chaque branche à cette fonction la valeur retournée par φ. Ainsi la formule de l'exemple devient $ite\ \ b_0\ \ (b_1 = false)(true \wedge b_1 = b_0)$. La règle de conversion pour cet opérateur est R-Eval-ite.

$$\frac{\Gamma_{uml} \models if\ \ x_1\ \ then\ \ x_2\ \ x_3 \mid x_1, x_2, x_3 \in Bool}{\Gamma_{smt} \models ite\ \ b_1\ \ \Phi(b_2)\ \ \Phi(b_3} \qquad \text{(R-Eval-ite)}$$

Les règles R-Eval-abs, R-Eval-min et R-Eval-max permettent de convertir les opérateurs retournant la valeur absolue d'un nombre, le plus petit nombre parmi deux et le plus grand nombre parmi deux. Ces règles suivent un même schéma. Elles commencent par créer une nouvelle stepVariable de type entier. Ensuite elles utilisent une contrainte annexe pour attribuer une valeur à cette fonction.

$$\frac{\Gamma_{uml} \models x.abs() \mid x \in \mathbb{N}}{\Gamma_{smt} \models svar(x_{abs} : Int) \mid x_{abs} \in F, (ite\ \ x < 0\ \ x_{abs} = (-x)\ \ x_{abs} = x) \in C} \qquad \text{(R-Eval-abs)}$$

$$\frac{\Gamma_{uml} \models min(x_1, x_2) \mid x_1, x_2 \in \mathbb{N}^2}{\Gamma_{smt} \models svar(x_{min} : Int) \mid x_{min} \in F, (ite\ \ x_1 < x_2\ \ x_{min} = x_1\ \ x_{min} = x_2) \in C} \qquad \text{(R-Eval-min)}$$

$$\frac{\Gamma_{uml} \models max(x_1, x_2) \mid x_1, x_2 \in \mathbb{N}}{\Gamma_{smt} \models svar(x_{max} : Int) \mid x_{max} \in F, (ite\ \ x_1 > x_2\ \ x_{max} = x_1\ \ x_{max} = x_2) \in C} \qquad \text{(R-Eval-max)}$$

La règle R-Eval-undefined convertit l'opérateur *oclIsUndefined*. Comme expliqué pour la règle R-Assign-undefined, cet opérateur est équivalent à une égalité entre le lien et la valeur nulle.

$$\frac{\Gamma_{uml} \models x.oclIsUndefined() \mid x \in inst(c) \wedge c \in Class}{\Gamma_{smt} \models x = c_{none}} \qquad \text{(R-Eval-undefined)}$$

La règle R-Eval-allInstance convertit l'opérateur *allInstance* qui renvoie dans un ensemble toutes les instances de la classe sur laquelle il est appliqué. Ainsi la règle crée une fonction constante qui est un ensemble d'éléments du type de la classe correspondante puis à l'aide d'une contrainte ajoute l'ensemble des instances de la classe correspondante.

$$\frac{\Gamma_{uml} \models c.allInstance() \mid c \in Class \vee inst(c) = \{i_0, \ldots, i_n\}}{\Gamma_{smt} \models con(c_{all} : set_c) \mid c_{all} \in F, (\bigwedge_{j=0}^{n} has(c_{all}(i_j)) = 1) \in C} \text{ (R-Eval-allInstance)}$$

Les règles suivantes concernent la conversion des opérateurs ensemblistes. Les premières sont dédiées à l'égalité et à l'inégalité. Ces règles, R-Eval-setEqual et R-Eval-setNotEqual, utilisent le fait que deux ensembles sont égaux si chaque élément de l'un est présent chez l'autre.

$$\frac{\Gamma_{uml} \models x_1 = x_2 \mid x_1, x_2 \subseteq Inst(c) \wedge inst(c) = \{i_0, \ldots, i_n\}}{\Gamma_{smt} \models \bigwedge_{j=0}^{n} has(x_1, i_j) = has(x_2, i_j)} \text{ (R-Eval-setEqual)}$$

$$\frac{\Gamma_{uml} \models x_1 = x_2 \mid x_1, x_2 \subseteq Inst(c) \wedge inst(c) = \{i_0, \ldots, i_n\}}{\Gamma_{smt} \models \neg(\bigwedge_{j=0}^{n} has(x_1, i_j) = has(x_2, i_j))} \text{ (R-Eval-setNotEqual)}$$

Les règles R-Eval-include, R-Eval-exclude, R-Eval-includeAll, R-Eval-excludeAll convertissent les opérateurs d'inclusions et d'exclusions en utilisant l'opérateur d'appartenance *has*.

$$\frac{\Gamma_{uml} \models x.include(i) \mid x \subseteq inst(c) \wedge i \in inst(c)}{\Gamma_{smt} \models has(x, i) = 1} \text{ (R-Eval-include)}$$

$$\frac{\Gamma_{uml} \models x.include(i) \mid x \subseteq inst(c) \wedge i \in inst(c)}{\Gamma_{smt} \models has(x, i) = 0} \text{ (R-Eval-exclude)}$$

$$\frac{\Gamma_{uml} \models x_1.includeAll(x_2) \mid x_1, x_2 \subseteq Inst(c) \wedge inst(c) = \{i_0, \ldots, i_n\}}{\Gamma_{smt} \models \bigwedge_{j=0}^{n} has(x_2, i_j) = 1 \Rightarrow has(x_1, i_j) = 1} \text{ (R-Eval-includeAll)}$$

$$\frac{\Gamma_{uml} \models x_1.excludeAll(x_2) \mid x_1, x_2 \subseteq Inst(c) \wedge inst(c) = \{i_0, \ldots, i_n\}}{\Gamma_{smt} \models \bigwedge_{j=0}^{n} has(x_2, i_j) = 1 \Rightarrow has(x_1, i_j) = 0} \text{ (R-Eval-excludeAll)}$$

La règle R-Eval-size convertit l'opérateur de cardinalité. Cette règle justifie le choix d'avoir une fonction d'appartenance qui renvoie un entier compris entre 0 et 1 inclus et non un booléen. En effet, grâce à cette propriété, obtenir la cardinalité d'un ensemble est équivalant à calculer la somme des fonctions d'appartenance appliquées à tous les éléments potentiels de l'ensemble.

$$\frac{\Gamma_{uml} \models x.size() \mid x \subseteq Inst(c) \wedge inst(c) = \{i_0, \ldots, i_n\}}{\Gamma_{smt} \models \sum_{j=0}^{n} has(x, i_j)} \text{ (R-Eval-size)}$$

Pour convertir les opérateurs évaluant si un ensemble est vide ou non, les règles R-Eval-empty et R-Eval-notEmpty utilisent la cardinalité. En effet, un ensemble vide (resp. non vide) a une cardinalité égale (resp. non égale) à 0.

$$\frac{\Gamma_{uml} \models x.isEmpty() \mid x \subseteq Inst(c) \wedge inst(c) = \{i_0, \ldots, i_n\}}{\Gamma_{smt} \models (\sum_{j=0}^{n} has(x, i_j)) = 0} \quad \text{(R-Eval-empty)}$$

$$\frac{\Gamma_{uml} \models x.isEmpty() \mid x \subseteq Inst(c) \wedge inst(c) = \{i_0, \ldots, i_n\}}{\Gamma_{smt} \models (\sum_{j=0}^{n} has(x, i_j)) > 0} \quad \text{(R-Eval-notEmpty)}$$

Les règles R-Eval-union et R-Eval-inter chargées de la conversion des opérateurs d'intersection et d'union suivent un même schéma. Elles commencent par créer une nouvelle stepVariable représentant un ensemble du même type que ceux sur lesquels s'appliquent ces règles. Ensuite une contrainte itère sur l'ensemble des instances pouvant appartenir à ce nouvel ensemble et ajoute les instances à la stepVariable si elles appartiennent à l'union ou à l'intersection. Cette appartenance est testée au travers de $has(x_2, i_j) + has(x_1, i_j) = 0$ pour l'union et $has(x_2, i_j) + has(x_1, i_j) = 2$ pour l'intersection.

$$\frac{\Gamma_{uml} \models x_1.union(x_2) \mid x_1, x_2 \subseteq Inst(c) \wedge inst(c) = \{i_0, \ldots, i_n\}}{\begin{array}{l}\Gamma_{smt} \models svar(x_3 : set_c) \mid x_3 \in F, \\ \quad \bigwedge_{j=0}^{n} ite\ (has(x_2, i_j) + has(x_1, i_j) = 0)\ \ (has(x_3, i_j) = 0)\ \ (has(x_3, i_j) = 1) \in C\end{array}} \quad \text{(R-Eval-union)}$$

$$\frac{\Gamma_{uml} \models x_1.intersection(x_2) \mid x_1, x_2 \subseteq Inst(c) \wedge inst(c) = \{i_0, \ldots, i_n\}}{\begin{array}{l}\Gamma_{smt} \models svar(x_3 : set_c) \mid x_3 \in F, \\ \quad \bigwedge_{j=0}^{n} ite\ (has(x_2, i_j) + has(x_1, i_j) = 2)\ \ (has(x_3, i_j) = 1)\ \ (has(x_3, i_j) = 0) \in C\end{array}} \quad \text{(R-Eval-inter)}$$

Les règles R-Eval-exist et R-Eval-forAll sont chargées de la conversion des quantificateurs. Les quantificateurs appliquent une formule booléenne, notée *expr* sur chaque élément de l'ensemble. L'opérateur *exist* (reps. *forAll*) renvoie vrai si l'une (resp. toutes) des applications de cette formule est vérifiée.

$$\frac{\Gamma_{uml} \models x.exist(expr(i)) \mid x \subseteq Inst(c) \wedge i \in inst(c) = \{i_0, \ldots, i_n\}}{\Gamma_{smt} \models \bigvee_{j=0}^{n} ((has(x, i_j) = 1) \Rightarrow expr(i_j))} \quad \text{(R-Eval-exist)}$$

$$\frac{\Gamma_{uml} \models x.forAll(expr(i)) \mid x \subseteq Inst(c) \wedge i \in inst(c) = \{i_0, \ldots, i_n\}}{\Gamma_{smt} \models \bigwedge_{j=0}^{n} ((has(x, i_j) = 1) \Rightarrow expr(i_j))} \quad \text{(R-Eval-forAll)}$$

La dernière règle sur les opérateurs ensemblistes est R-Eval-any. Elle convertit l'opérateur *any* qui retourne une instance appartenant à l'ensemble qui vérifie une formule booléenne notée *expr*. Lors de la conversion, la contrainte créée utilise une structure de if then else imbriqués pour itérer sur les instances pouvant appartenir à l'ensemble.

$$\frac{\Gamma_{uml} \models x.any(expr(i)) \mid x \subseteq inst(c) \wedge i \in inst(c) = \{i_0, \ldots, i_n\}}{\Gamma_{smt} \models svar(i_{any} : c) \mid i_{any} \in F, \\ ite\ \ f(i_0)\ \ (i_{any} = i_0)\ \ (\ldots ite\ \ f(i_n)\ \ (i_{any} = i_n)\ \ (i_{any} = i_{none}) \ldots) \in C}$$

(R-Eval-any)

avec

$$f(i_j) := expr(i_j) \wedge has(x, i_j) = 1$$

La conversion d'un appel d'opération dans le contexte d'évaluation nécessite un processus plus complexe. Dans ce contexte, le langage OCL4MBT ajoute la restriction qu'un appel d'opération ne doit pas modifier l'état du système. Le processus de conversion d'un appel d'une opération op_1 comporte les quatre étapes suivantes.

1. Déterminer si l'appel est valide en contrôlant que la pré-condition de l'opération appelée est vérifiée avec les valeurs passées en paramètre. Par conséquent, cette condition est ajoutée dans le précédent contexte d'évaluation du code OCL4MBT. En pratique, ce contexte est rencontré dans la condition de l'opérateur *"if then else"*, dans la pré-condition de l'opération appelante ou dans la garde de la transition UML4MBT appelante.
2. Remplacer les paramètres d'entrées par leurs valeurs.
3. Créer une nouvelle stepVariable SMT4MBT, $svar(out : A)$, représentant le paramètre de retour *out* de type A et la substituer à l'appel d'opération. L'opération comporte obligatoirement un paramètre de retour car l'appel d'une opération sans effet sur le système et sans paramètre de retour n'a pas d'utilité. Dans le cas où ce paramètre serait absent, l'appel est ignoré.
4. Créer une contrainte affectant à cette fonction la valeur obtenue lors de l'exécution de la post-condition, $post(op_1)$, de l'opération.

8.2.4 Règles d'animation

Cette sous-section recense les règles de conversion visant spécifiquement à animer le modèle. Dans ces règles, si une stepVariable a le mot *next* en exposant, alors cette fonction fait référence au contenu de la stepVariable après l'exécution d'une transition. Finalement, l'ensemble de ces règles appartient à la catégorie *animation*.

$$\frac{\Gamma_{uml} \models x \mid x \in State}{\Gamma_{smt} \models svar_{anim}(x : State) \mid x \in F}$$

(R-Anim-state)

La règle R-Anim-state crée pour chaque état du diagramme d'états-transitions une stepVariable.

$$\frac{\Gamma_{uml} \models x \mid x \in Ops \wedge var() = \{v_0, \ldots, v_n\}}{\Gamma_{smt} \models svar_{anim}(x : Event) \mid x, svar_{anim}(trig_x : Bool), svar_{anim}(step_x : Bool) \in F, \\ ite\ \ a\ \ b\ \ c \in C}$$

(R-Anim-operation)

avec

$$a := (x = event \wedge pre(x))$$
$$b := (trig_x^{next} \wedge post(x) \bigwedge_{i=0}^{n} v_i^{next} = \varphi(v_i))$$
$$c := (notstep_x)$$

La règle R-Anim-operation associe trois fonctions à chaque opération. La stepVariable x représente l'opération à convertir. $trig_x$ est une stepVariable booléenne mémorisant si l'opération a été activée durant le dernier pas. La principale utilité de cette fonction est d'autoriser les transitons UML4MBT du diagramme d'états-transitions à conditionner leur activation à l'exécution d'opérations spécifiques. $step_x$ est une stepVariable booléenne qui prend la valeur vraie si l'opération correspondante est exécutée durant le pas actuel. La contrainte créée par cette règle peut se lire : "si la pré-condition de l'opération x est vérifiée et si cette opération est exécutée, alors cette activation est mémorisée pour le prochain pas, la post-condition est réalisée et les nouvelles valeurs des variables d'état sont assignées, sinon la non-activation de l'opération est mémorisée".

Si cette règle et celle concernant les formules OCL4MBT sont appliquées à l'opération a_p de l'instance $quaiA$ de l'exemple fil rouge, les fonctions $svar(a\_p : Event)$, $svar(trig_{a\_p} : Bool)$ et $svar(step_{a\_p} : Bool)$ sont créées. La contrainte suivante est aussi créée :

$$ite\ \ (a\_p = event \wedge pre)\ \ (trig_{a\_p}^{next} \wedge post \wedge var)\ \ (\neg step_{a\_p})$$

avec

$$\begin{aligned}
pre &:= piece_{a_p} = L \wedge \neg(P_{a_p} = L)\\
post &:= piece_{a_p,1} = P_{a_p}\\
var &:= piece_{a_p}^{next} = piece_{a_p,1} \wedge cote_{robot}^{next} = cote_{robot} \wedge position_{robot}^{next} = position_{robot}\\
&\quad \wedge piece_{robot}^{next} = piece_{robot} \wedge piece_{quaiG}^{next} = piece_{quaiG} \wedge piece_{quaiD}^{next} = piece_{quaiD}\\
&\quad \wedge robotA_{quaiA}^{next} = robotA_{quaiA} \wedge quaiA_{robot}^{next} = quaiA_{robot} \wedge quaiD_{robot}^{next} = quaiD_{robot}\\
&\quad \wedge quaiG_{robot}^{next} = quaiG_{robot} \wedge robotG_{quaiG}^{next} = robotG_{quaiG}\\
&\quad \wedge robotG_{quaiD}^{next} = robotG_{quaiD} \wedge robotD_{quaiG}^{next} = robotD_{quaiG}\\
&\quad \wedge robotD_{quaiD}^{next} = robotD_{quaiD}
\end{aligned}$$

Dans cette contrainte, *pre* correspondant à la conversion de la pré-condition, *post* est la conversion de la post-condition, *var* affecte leur nouvelle valeur aux variables d'états du prochain pas et $piece_{a\_p,1}$ est la nouvelle fonction renvoyée par φ lors de l'assignation dans la post-condition de l'opération.

$$\frac{\Gamma_{uml} \models x \mid x \in Trs \wedge var() = \{v_0, \ldots, v_n\}}{\Gamma_{smt} \models svar_{anim}(x : Event) \mid x, svar_{anim}(step_x : Bool) \in F, ite\ \ a\ \ b\ \ c \in C}$$
(R-Anim-transition)

avec

$$a := (x = event \wedge pre(x) \wedge trig(x) \wedge in(x) = state)$$
$$b := (trig_{none}^{next} \wedge state^{next} = out(x) \bigwedge_{i=0}^{n} v_i^{next} = \varphi(v_i))$$
$$c := (\neq step_x)$$

La règle R-Anim-transition suit le même schéma que la règle précédente. Les différences principales sont la mémorisation de l'activation de la transition UML4MBT au travers de $trig_{none_{next}}$ dans l'hypothèse et la condition est modifiée pour vérifier que le modèle est dans un état valide au niveau du diagramme d'états-transitions et que durant le pas précédent l'opération correcte a été effectuée au travers de $trig(x) \wedge in(x) = state$.

$$\frac{\Gamma_{uml} \models model \mid \{tr_0, \ldots, tr_n, \} = Tr, \{op_0, \ldots, op_p\} = Op}{\begin{array}{c}\Gamma_{smt} \models model \mid (Event : Named) \in T, svar_{anim}(event : Event) \in F, \\ (distinct\ \ tr_0, \ldots, tr_n, op_0, \ldots, op_p), (\bigvee_{i=0}^{n} step_{tr_i} \bigvee_{i=0}^{p} step_{op_i} \vee step_{none}) \in C\end{array}} \text{(R-Anim-model)}$$

La règle R-Anim-model a pour objectif d'obliger le prouveur à exécuter une séquence de transitions. La règle crée un verrou avec la fonction *event* qui supprime la possibilité d'exécuter simultanément plusieurs transitions et une contrainte sur $step_x$ obligeant à exécuter au moins une transition. Cette règle s'applique sur le modèle UML4MBT.

$$\frac{\Gamma_{uml} \models model \mid \{tr_0, \ldots, tr_n, \} = Tr, \mid Tr \mid > 0}{\Gamma_{smt} \models model \mid (State : Named) \in T, svar_{anim}(state : State) \in F} \text{(R-Anim-state2)}$$

La règle R-Anim-state2 crée une stepVariable pour mémoriser quel est l'état issu du diagramme d'états-transitions pendant l'animation du modèle.

$$\frac{\Gamma_{uml} \models op_{none} \mid \wedge var() = \{v_0, \ldots, v_n\}}{\begin{array}{c}\Gamma_{smt} \models con_{anim}(op_{none} : Event) \mid \\ op_{none}, con_{anim}(trig_{none} : Bool), con_{anim}(step_{none} : Bool) \in F, \\ ite\ \ (x = op_{none})\ \ (trig_{none}^{next} \bigwedge_{i=0}^{n} a_i^{next} = a_i)\ \ (\neg step_{none}) \in C\end{array}} \text{(R-Anim-nop)}$$

La règle R-Anim-nop a pour but d'encoder une transition factice sans effet sur les variables d'état. Lors de l'animation par le solveur, la longueur de la séquence de transitions est fixée au préalable. Par conséquent, si durant l'exécution d'une séquence de transitions d'une longueur inférieure à celle fixée pour cette animation, le système rencontre un état final n'autorisant plus l'activation d'une transition, alors le solveur est incapable de retourner une solution. La transition factice permet de pallier ce problème. Cette règle revient à convertir une opération, op_x, sans paramètre avec $post(op_x) = true$ et $pre(op_x)$ avec la règle R-Anim-operation.

La validité des règles de conversion présentées ici a été établie par l'utilisation de tests unitaires lors de l'implémentation et de l'exécution des cas de test en résultant par l'animateur de l'outil industriel Certifi-It.

8.3 Conversion entre SMT4MBT et SMT-lib

La conversion d'un modèle SMT4MBT en un problème écrit dans le langage SMT-lib est directe et possède un coût faible. En effet, le méta-modèle a été construit autour d'une logique définie en SMT-lib. Par conséquent, la majorité des opérateurs et des types ont leur équivalent dans SMT-lib.

La conversion possède deux points notables. Le premier est la conversion des ensembles. Dans le méta-modèle, les ensembles peuvent contenir uniquement des fonctions de type libre représentant des instances du modèle UML4MBT. Ces ensembles sont convertis par des applications mathématiques qui associent au domaine du type libre le domaine $\{0, 1\}$. Par exemple, avec un ensemble *set* d'élément type A, un élément a de type a, l'application est $set : A \rightarrow \{0, 1\}$. En SMT-lib, les applications sont déclarées comme des variables. Le seul opérateur ensembliste, *has*, qui évalue si un élément est présent dans un ensemble, correspond ainsi à l'utilisation d'une application. Dans notre exemple, $has(set, a)$ est converti en $set(a)$.

Le second point traité lors de cette conversion est le problème posé par les stepVariables. En effet, ces fonctions peuvent prendre une valeur différente pour chacun des pas de l'animation. Durant la conversion, les stepVariables entraînent la création de fonctions du même type pour chacun des pas de l'animation. En conséquence, les contraintes comprenant des stepVariables doivent être également dupliquées pour chacun des pas.

Ainsi la conversion entraîne la propriété suivante : *Le nombre de contraintes et de variables du problème SMT-lib dépend linéairement du nombre de pas du scénario d'animation encodé.*

8.4 Synthèse

Le méta-modèle SMT4MBT est un langage intermédiaire reliant un langage de bas niveau à destination des prouveurs SMT, nommé SMT-lib, combinant différentes théories au sein de logiques employées pour décrire un ensemble de formules logiques du premier ordre et un langage de modélisation de haut-niveau supportant les notions liées au test à partir de modèle. Ce langage est dérivé de la logique QF_UFLIA choisie pour son expressivité et sa capacité à obtenir une réponse du prouveur en toute circonstance.

SMT4MBT dispose de la capacité d'encoder au sein d'un même modèle un scénario d'animations et la réponse du prouveur. Pour maintenir une correspondance avec la modélisation UML4MBT, une information additionnelle peut être attribuée aux différentes entités SMT4MBT sous la forme de méta-data. Ainsi une formule dispose à la fois d'un corps contenant sa description mathématique mais également d'un but renseignant sur l'utilité de la formule dans le cadre de l'animation du modèle.

Pour formaliser l'encodage du scénario d'animations d'un système modélisé au travers des diagrammes UML4MBT de classes, d'objets et d'états-transitions dans un modèle conforme au méta-modèle SMT4MBT, une liste de règles de conversion a été établie. Le processus de conversion est décomposable en trois parties.

En premier, la partie statique du modèle, décrite dans le diagramme de classes, est convertie. Dans un second temps, la partie dynamique, décrite dans les opérations du diagramme de classes et les transitions du diagramme d'états-transitions et écrite en OCL4MBT, est représentée sous forme de formules logiques compatibles avec SMT4MBT. Finalement, des contraintes représentant les comportements du système sous forme de système de transitions sont mises en place.

Cependant, un modèle SMT4MBT n'est pas compréhensible directement par un prouveur SMT. Une étape intermédiaire de traduction de ce modèle dans le langage SMT-lib est nécessaire. Grâce à l'influence de ce langage sur le méta-modèle, la traduction ne comporte pas de difficulté mais oblige à préciser la longueur de l'animation désirée. De par les choix pris durant le processus de conversion, une propriété importante apparaît : *les fonctions et les formules SMT croient linéairement avec la longueur de l'animation.*

Chapitre 9

Collaboration entre SMT et CSP

Sommaire

Dans les chapitres précédents, notre méthode pour générer des tests à partir de modèle à l'aide de stratégies usant d'un prouveur SMT a été présentée dans sa globalité. Cependant dans la section 6, la possibilité d'une collaboration entre un prouveur SMT et un solveur CSP a été envisagée pour améliorer les performances de notre méthode.

Pour avoir un intérêt à mettre en place cette collaboration, il y a nécessité de créer de nouvelles stratégies capitalisant sur les forces et les faiblesses des prouveurs SMT et des solveurs CSP. Ces stratégies soulèvent de nouvelles questions : Comment échanger de l'information entre ces outils ? Quels rôles sont adaptés à ces outils ? Comment représenter un scénario d'animations de manière compréhensible pour un solveur CSP ?

Afin d'apporter des éléments de réponse à ces questions, ce chapitre commence par établir un rôle spécifique à chaque outil en analysant leurs forces telles que la

rapidité d'exécution pour les prouveurs ou le contrôle plus fin sur le processus de résolution pour les solveurs. A l'aide de ces rôles, un processus de collaboration est établi. Il définit les informations à échanger ainsi que la méthode de partage. En particulier, le choix de communiquer directement entre les prouveurs et les solveurs est explicité. Dans une seconde partie, les stratégies collaboratives sont introduites. Ces dernières sont des améliorations des stratégies décrites à la section 7.2. Finalement l'usage d'un solveur implique la définition d'un nouveau méta-modèle nommé CSP4MBT, qui permet d'encoder un scénario d'animations du modèle UML4MBT. De plus ce méta-modèle doit être également compréhensible par un solveur CSP. Afin de créer des modèles CSP4MBT depuis des modèles SMT4MBT, les règles de conversions sont définies.

9.1 Principe de la collaboration

Pour établir une collaboration entre les solveurs CSP et les prouveurs SMT, il est nécessaire de comparer le fonctionnement de ces outils pour leur attribuer un rôle dans le cadre de notre démarche de génération de tests et pour établir des passerelles permettant des échanges d'information.

Les deux outils manipulent des modèles mathématiques fondés sur la logique du premier ordre. En conséquence, un échange d'information est possible via la transmission de formules entre les modèles. Cependant notre démarche doit établir un processus de collaboration pour communiquer des informations entre trois représentations : UML4MBT pour la modélisation du système, SMT4MBT pour le scénario d'animations soumis au prouveur et CSP4MBT pour le scénario d'animations transmis au solveur.

9.1.1 Comparaison entre les solveurs CSP et les prouveurs SMT

En comparant le fonctionnement d'un solveur à celui du prouveur, des avantages et inconvénients se dégagent. Le premier avantage est la possibilité d'intervenir sur la résolution au travers du choix et de l'ordre des contraintes. Ainsi dans le cas où l'on souhaite exécuter une séquence de transitions spécifiques, l'ordre d'apparition des contraintes représentant ce choix est déterminant pour la vitesse de résolution. Si ces contraintes sont écrites en premier, le nombre de points de choix diminue rapidement du fait de la construction incrémentale du graphe de contraintes.

Un second avantage est la capacité de classer les solutions et de choisir la meilleure selon un critère fourni par une fonction objective. Il est possible de demander la maximisation ou la minimisation d'une fonction durant la résolution. Ceci permet de traduire une notion absente en logique du premier ordre qui est "le plus possible" ou "le moins possible". Malheureusement l'efficacité de cette fonction diminue si le nombre de disjonctions présentes dans les contraintes augmentent.

Un troisième avantage est la possibilité d'obtenir des états du système plus intéressants dans le cadre de la génération de tests en choisissant certains valeurs telles

les valeurs aux limites, c'est à dire les bornes des domaines après leur réduction par la construction du graphe des contraintes.

Il n'y a malheureusement pas que des avantages, et un premier inconvénient est l'importante influence des choix effectuées durant la modélisation du système. La modélisation du système doit être menée en vue de l'utilisation de cet outil et nécessite une expertise dans ce domaine.

L'inconvénient majeur par rapport aux prouveurs SMT est une performance inférieure sur la résolution de scénarios d'animations avec des modèles de tailles conséquentes si ces modèles n'ont pas été spécifiquement créés dans ce but. Par conséquent, l'utilisation de solveurs CSP dans notre démarche sera limitée aux scénarios disposant d'informations supplémentaires, telles des motifs précis, afin de réduire au maximum le nombre de disjonctions et donc de points de choix pour obtenir de meilleures performances.

Si l'utilisateur possède des moyens d'influencer le processus de résolution conduit par un solveur CSP, il n'est pas possible d'intervenir sur la résolution conduite par un prouveur SMT. Dans le cas d'un prouveur, le problème se ramène à une formule logique du premier ordre. Cette formule peut être vue comme une formule booléenne dont les différentes variables sont des abstractions pour des formules écrites dans des théories variées, d'où le nom SMT (Satisfiability Modulo Theory). Le langage SMT-Lib, employé pour communiquer avec le prouveur, ne permet pas d'affecter une priorité de traitement à ces différentes propositions. En conséquence, le prouveur sera privilégié quand nous ne disposons pas d'information supplémentaire sur un scénario. Par exemple dans le cas ou le motif d'animations est celui par défaut. Pour une description plus détaillée du processus de résolution d'un prouveur, se reporter à la section 4.1.

9.1.2 Échange d'informations

L'échange d'informations entre le solveur CSP et le prouveur SMT appartient à l'une des deux catégories suivantes : en ligne et hors ligne. Dans le premier cas, les outils collaborent via un échange d'informations pendant leur processus de résolution. En interne, le prouveur stocke un ensemble de lemmes pour guider la résolution. Toute nouvelle information peut donc être transmise sous la forme d'un lemme. Pour le solveur, la construction incrémentale du graphe de contraintes permet une transmission d'informations sous la forme de nouvelles contraintes. Cependant ce premier type d'échange nécessite d'avoir un accès total aux processus de résolution. En pratique ceci nécessite de modifier le code source des applications ou d'avoir des APIs spécifiques aux outils. Cette contrainte a conduit à ne pas privilégier dans un premier temps ce type d'échange dans notre démarche.

Le deuxième type d'échange requiert d'ajouter les informations avant de lancer le processus de résolution. Ce type de collaboration a du sens uniquement lorsque les différents problèmes soumis aux solveurs et prouveurs ont des liens entre eux. Ceci s'inscrit parfaitement dans notre démarche puisque les différents problèmes correspondent à des scénarios portant sur un même modèle. Ainsi les cibles de test, les motifs et les états du modèle rencontrés pendant l'animation sont susceptibles

d'être échangés entre les scénarios. Par exemple, un état atteint lors d'une première animation peut être ré-employé comme état initial d'un second scénario.

Dans le cas des solveurs CSP, la transmission d'informations s'effectue simplement par l'ajout de contraintes. En fonction de la sémantique de cette contrainte, son ordre d'apparition varie. En règle générale, ces contraintes sont parmi les premières car elles ont généralement pour fonction de réduire le nombre de points de choix pour augmenter les performances.

Dans le cas des prouveurs SMT, la transmission d'informations s'avère plus complexe. En effet, l'ajout de formules logiques encodant ces informations n'ont pas toujours d'influence sur le temps de résolution à cause de l'absence de prise sur le processus de résolution par l'utilisateur. Ainsi l'ajout de formules spécifiant une séquence de transitions ne réduit pas le temps de résolution malgré des formules assignant directement des valeurs à des variables. En conséquence, il faut plutôt envisager de supprimer les formules devenues inutiles.

9.1.3 Procédure de collaboration

Pour autoriser la collaboration entre un prouveur SMT et un solveur CSP dans le cadre de la génération de tests à partir de modèle, un système d'échange d'informations nécessite une méthode pour traduire les représentations d'un même problème dans les langages propres à chaque outil.

Dans le cadre de notre démarche, trois représentations différentes existent :

UML4MBT est utilisé pour modéliser le système. Le comportement du système est encodé à l'aide d'un langage additionnel, OCL4MBT. Ces langages sont détaillés dans le chapitre 2.

SMT4MBT repose sur un méta-modèle et enrichit le langage commun des prouveurs SMT, nommée SMT-Lib, avec des concepts liées à la génération de tests. Ce langage est construit sur un ensemble de formules logiques du premier ordre annotées avec des méta-informations spécifiant leur objectif du point de vue du scénario dans le cadre de la génération de tests. Le détail du méta-modèle est donné dans la section 8.1.

CSP4MBT est l'analogue du méta-modèle SMT4MBT pour les solveurs CSP. Ce méta-modèle est construit autour d'un langage compatible avec un nombre important de solveurs, nommée MiniZinc. Cependant si ce langage permet de décrire un problème par le biais de contraintes portant sur un ensemble de variables dont la valeur est limitée à un domaine précis, l'objectif de ces contraintes dans le cadre de la génération de tests doit être stocké sous la forme de méta-informations. Ce méta-modèle sera présenté en détails dans la section 9.3.

Pour échanger des informations entre les différentes représentations, il existe deux possibilités. Ces dernières sont présentées au travers de la figure 9.1. Ces informations sont de deux natures : l'encodage d'un scénario dans un langage dédié aux solveurs ou aux prouveurs et l'analyse de la réponse de ces outils pour extraire les informations nécessaires à la conduite d'une stratégie.

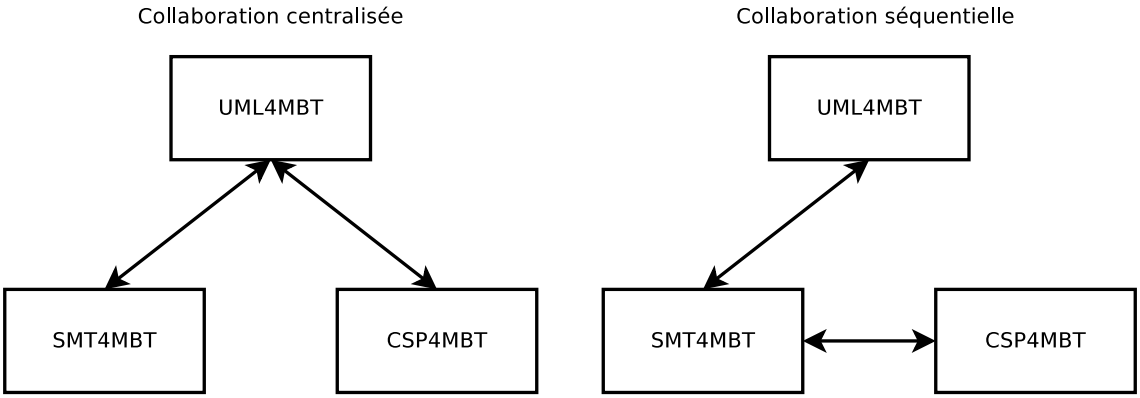


FIGURE 9.1 – Communication entre les différents modèles dans le cadre d'une collaboration entre des solveurs CSP et prouveurs SMT.

Collaboration centralisée La première méthode considère le modèle UML4MBT comme un pivot pour établir un échange d'informations entre les modèles SMT4MBT et CSP4MBT. Dans ce cas, les scénarios sont indépendants entre les deux outils et leurs objectifs ne sont pas a priori dissociés. Ces derniers sont encodés directement depuis le modèle initial écrit en UML4MBT/OCL4MBT. Cette collaboration n'a pas été retenue car notre démarche attribue des rôles spécifiques à chaque outil et l'encodage des informations n'est pas adapté à l'expressivité des langages UML4MBT/OCL4MBT ; le prouveur SMT intervient en premier pour découvrir des motifs d'animations valides. Le solveur CSP utilise ces motifs pour atteindre de nouvelles cibles de test.

Collaboration séquentielle La seconde méthode met en place une approche séquentielle. Le modèle CSP4MBT est créé depuis le modèle SMT4MBT qui lui même découle du modèle UML4MBT. Ce fonctionnement cantonne le solveur CSP à un rôle de soutien où il exploite les résultats du prouveur. Cependant ce rôle est adapté à notre démarche car les performances des prouveurs pour la résolution de problèmes de taille conséquente sont supérieures aux performances des solveurs. Cette approche entraîne un surcoût dû aux conversions successives. Mais en pratique cet inconvénient est contre-balancé par l'utilisation de stratégies multi-thread et la simplicité de la conversion entre les modèles SMT4MBT et CSP4MBT. En effet, la syntaxe du modèle SMT4MBT est plus restreinte que le modèle UML4MBT et la majorité des méta-informations stockées sont conservées dans le modèle CSP4MBT. Les règles présidant cette conversion sont explicitées à la section 9.3. Les stratégies collaboratives présentées dans la section suivante utilisent cette méthode.

9.2 Stratégies collaboratives

Après avoir présenté les architectures de collaboration entre un solveur CSP et un prouveur SMT, cette section présente leur application dans le cadre des stratégies

de génération de tests. L'architecture présentée dans la section 7.1 est enrichie pour inclure la possibilité d'animer le modèle à l'aide d'un solveur CSP.

Les propriétés d'un solveur CSP telles que la présence de fonctions objectives ou de génération de solutions multiples, sont à l'origine de nouvelles stratégies, dérivées des précédentes, et qui renforcent leur efficacité par le biais de la collaboration.

9.2.1 Architecture

La conduite de stratégies collaboratives menées à l'aide d'un solveur CSP et d'un prouveur SMT nécessite de mettre à jour l'architecture présentée dans la section 7.1 par la figure 7.1. La nouvelle version de l'architecture est présentée dans la figure 9.2. Les conventions précédentes sont respectées ; la ligne pointillée indique que l'exécution d'un composant peut être parallélisée.

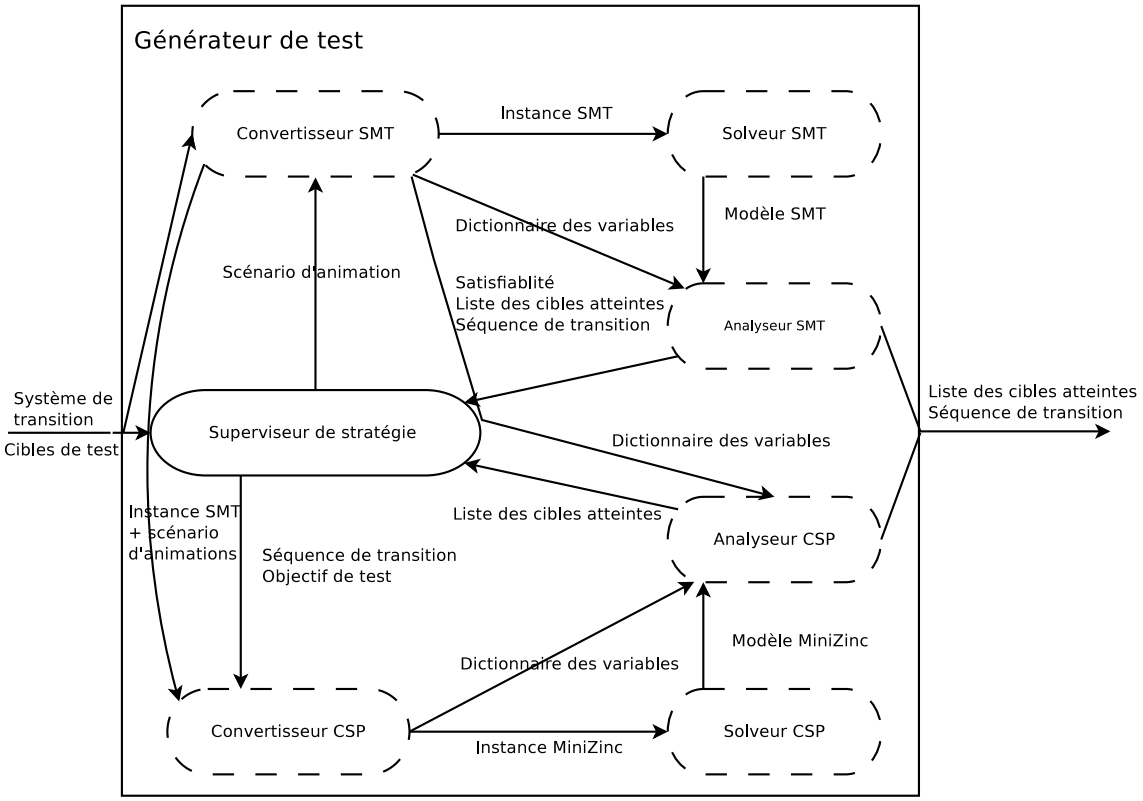


FIGURE 9.2 – Architecture des composants conduisant une stratégie collaborative

Cette architecture remplit trois fonctions. La première est d'animer le modèle à l'aide d'un prouveur SMT. L'animation est exécutée par quatre composants : le superviseur de stratégie, le convertisseur SMT, le prouveur SMT et l'analyseur SMT. Le comportement de ces composants n'a pas évolué et il est conforme à l'ancienne architecture.

La deuxième fonction est d'animer le modèle à l'aide d'un solveur CSP. Une caractéristique importante est que le solveur a uniquement une fonction de soutien. En

conséquence, le solveur est utilisé seulement pour rejouer des séquences de transitions découvertes par le prouveur SMT. Les quatre étapes de cette fonction : le choix du scénario, l'encodage du modèle, la résolution de cet encodage par un solveur CSP et l'analyse de la réponse du solveur, sont réalisées par les composants suivants :

Convertisseur CSP Ce composant a pour tâche d'encoder le modèle pour permettre son animation par un solveur CSP. Dans notre démarche, l'encodage ne s'applique pas directement sur le modèle écrit en UML4MBT/OCL4MBT mais sur l'encodage d'un scénario par le convertisseur SMT, une instance SMT4MBT. Ceci est possible car le solveur est uniquement utilisé en soutien du prouveur. Le scénario fournit le nombre de pas de l'animation, la définition de l'état initial et l'objectif de tests. Si les deux premiers composants sont conservés, le troisième doit être supprimé de l'encodage CSP. En effet, dans le cas où la stratégie emploie un solveur, l'objectif a déjà été rempli par le prouveur. En conséquence, l'objectif de tests est remplacé par une fonction objective chargée de maximiser le nombre de cibles de test atteintes. Afin de rejouer une séquence de transitions spécifique, un ensemble de contraintes est créé pour diriger le processus de résolution. Le résultat du travail de ce composant est un modèle écrit en MiniZinc et compréhensible par un solveur CSP adapté.

Solveur CSP Le solveur CSP est chargé de résoudre un problème spécifié sous la forme d'une instance MiniZinc. De part la modélisation initiale en UML4MBT, le nombre d'état du système est limité. Le domaine associé à chaque variable est borné. En conséquence, le solveur pourra toujours conclure et fournir une réponse. De plus, le temps de résolution sera réduit car l'obligation de rejouer une séquence de transitions, représenté par un motif d'animations, diminue grandement le nombre de points de choix lors de la résolution. La réponse du solveur est fournie sous la forme d'un modèle MiniZinc qui contient N solutions aux problèmes sous la forme d'une valuation des variables. Le nombre de solutions est déterminé par un paramètre contenu dans l'instance MiniZinc et déterminé par la stratégie.

Analyseur CSP L'analyseur CSP est responsable de l'analyse des réponses du solveur pour extraire les informations utiles pour la poursuite de la stratégie. Deux informations sont utiles : la liste des cibles atteintes et la séquence de transitions exécutée. Pour écrire ces informations au niveau du modèle UML4MBT, le composant dispose de deux dictionnaires de variables ; le premier associe les entités des modèles UML4MBT et SMT4MBT. Le second relie les éléments SMT4MBT avec ceux écrits en MiniZinc. La combinaison de ces dictionnaires permet donc de se ramener au niveau du langage UML4MBT. Dans le cas des séquences de transitions extraites, l'analyseur doit fournir le nom des transitions mais également les paramètres de ces transitions.

Superviseur de stratégie Ce composant conduit et dirige la stratégie par le choix des scénarios à effectuer et le choix des outils, solveur ou prouveur, à employer. Dans tous les cas, le but est d'atteindre l'ensemble des cibles de test afin de créer les tests correspondants. Dans le cadre de l'usage d'un solveur CSP, le

superviseur est habilité à ré-animer des séquences de transitions découvertes précédemment par le prouveur SMT. Évidemment si ces séquences spécifient le nom des transitions, elles ne précisent pas la valeur des paramètres de ces transitions.

La troisième et dernière fonction est de gérer la collaboration entre le prouveur SMT et le solveur CSP. Il existe trois points dans notre démarche où la collaboration s'effectue. Le premier est la conversion d'un modèle SMT4MBT en un modèle CSP4MBT au niveau du convertisseur CSP. Le second est l'extraction d'informations au niveau de l'analyseur CSP qui nécessite une conversion inverse entre le langage MiniZinc et SMT. Le dernier point est la gestion de l'exécution des scénarios en parallèle dans le cas de stratégies multi-threads. Le superviseur peut interrompre un scénario exécuté par un prouveur si en parallèle son objectif de tests est atteint par l'exécution d'un scénario par un solveur.

9.2.2 Step-increase collaboratif

Principe Cette stratégie est une amélioration de la stratégie *step-increase*. Le principe de fonctionnement est conservé : privilégier les animations courtes car le temps de résolution augmente quand la longueur de l'animation augmente. La collaboration avec un solveur CSP permet d'améliorer la rentabilité des séquences de transitions valides découvertes par un prouveur SMT en les rejouant avec un solveur et en utilisant une fonction de maximisation afin d'augmenter le nombre de cibles de test atteintes.

Paramètres Cette stratégie est configurable au travers de trois paramètres :

InitLength est la longueur initiale en pas des scénarios exécutées depuis l'état initial du système de transitions défini par ρ et correspondant à l'état décrit par le diagramme UML4MBT d'objets.

IncLength précise l'incrément en pas à ajouter à la longueur des scénarios si la longueur précédente était insuffisante pour atteindre l'ensemble des cibles de test.

MaxPas est la longueur maximale des scénarios soumis au solveur en nombre de pas. Si l'utilisateur a une connaissance du modèle, alors ce nombre doit correspondre à la taille de la séquence de transitions la plus longue pour atteindre toutes les cibles de test.

La figure 9.3 représente l'application de la stratégie *step-increase* collaborative sur un exemple simple. L'objectif est d'atteindre six cibles de test nommées A, B, C, D, E et F. Ces cibles sont atteignables avec une séquence comportant respectivement un minimum de 3, 11, 5 , 6, 1 et 11 pas. De plus, les cibles B et D sont mutuellement exclusives ; Si durant une exécution, la cible B est atteinte alors la cible D est inatteignable durant cette même exécution.

Dans cette figure, les arcs représentent des scénarios d'une longueur indiquée sur la droite en nombre de pas. Ils sont dessinés avec trois types de trait. Les lignes

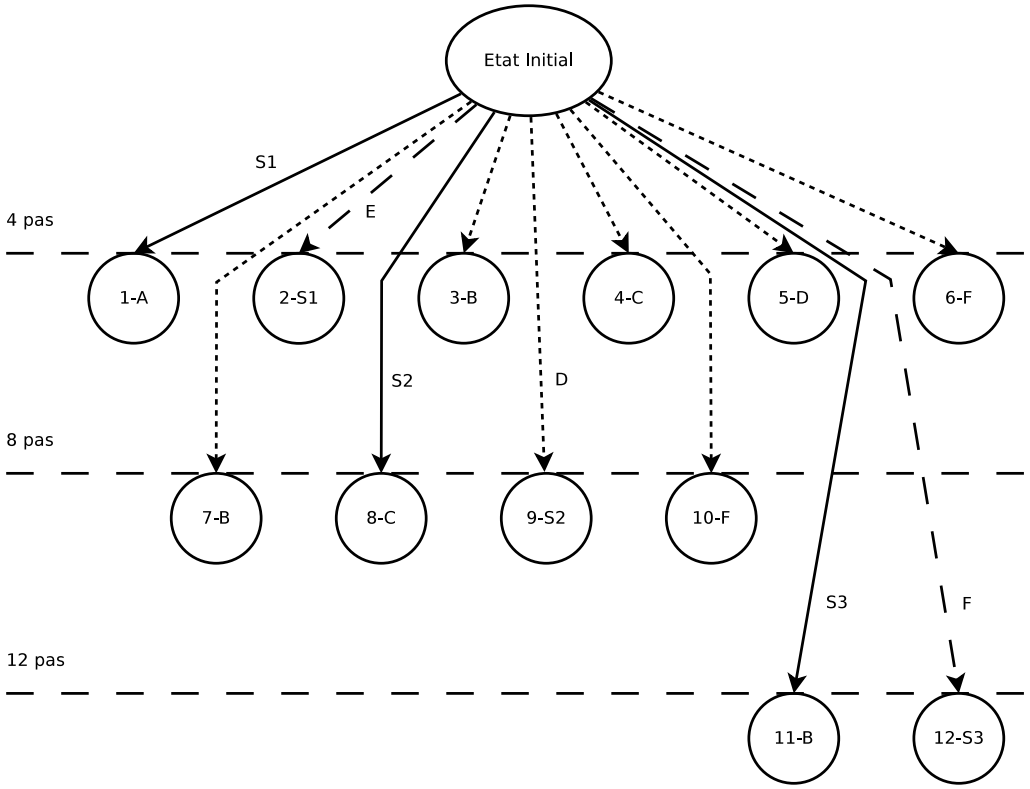


FIGURE 9.3 – Exécution de la stratégie *step-increase* collaborative sur un exemple

continues représentent des animations valides effectuées avec un prouveur SMT et la valeur associée à ces lignes est la séquence de transitions exécutée. Les lignes pointillées indiquent des animations invalides effectuées avec un prouveur SMT. Les lignes employant des tirets correspondent à l'exécution d'animations par un solveur CSP et les lettres associées à ces lignes précisent si des cibles ont été atteintes. Les nœuds de la figure correspondent à des états du modèle. Dans ces nœuds sont précisés un ordre de parcours du graphe à l'aide d'un nombre et un objectif à l'aide d'une chaîne de caractères. Dans le cas des scénarios utilisant un prouveur SMT, les objectifs spécifient les cibles de test à atteindre durant l'animation. Dans le cas des animations utilisant un solveur CSP, les objectifs sont les motifs d'animations à exécuter.

Fonctionnement Le fonctionnement de cette stratégie est illustré au travers de la figure 9.3. Durant cette exécution, la stratégie est configurée avec une longueur initiale (initLength) de 4 pas, une longueur incrémentale (incLength) de 4 pas et une longueur maximale (maxPas) de 12 pas.

La première étape de la stratégie consiste en l'exécution de scénarios de longueur initLength et ayant comme état initial celui décrit dans le diagramme d'objets du modèle UML4MBT. Les objectifs de tests de ces scénarios sont de type *minimum*, c'est à dire qu'au moins une cible parmi une liste de cibles non atteintes doit être trouvée. Dans la figure, pour des raisons de facilité de lecture, les objectifs sont

ramenés à une cible unique.

Dans le cas, où une cible est atteinte, la séquence de transitions ayant conduit à une animation valide est conservée. Cette séquence est ré-exécutée depuis le même état initial avec le solveur CSP. Durant cette exécution, la fonction de maximisation du solveur est employée pour augmenter les chances d'atteindre d'autres cibles de test.

Dans une deuxième étape, si certaines cibles de test n'ont pas été atteintes, la longueur des scénarios est incrémentée de la valeur du paramètre inclength. Puis le processus reprend depuis l'étape 1.

La stratégie se termine lorsque toutes les cibles de test sont atteintes ou que la longueur des scénarios excède la limite définie par le paramètre maxPas.

Un des avantages de cette stratégie est de conserver la propriété de la stratégie *step-increase* qui permet de prouver l'inatteignabilité d'une cible de test en regard d'une longueur donnée pour les scénarios. Un autre avantage est le surcoût quasi-nul en terme de temps de résolution si au moins un thread est dédié au solveur et un autre au prouveur. Dans ce cas, les animations exécutées par le solveur CSP peuvent être complètement parallélisées en regard des animations réalisées par le prouveur SMT.

9.2.3 Space-search collaboratif

Principe Cette stratégie combine une stratégie (*depth-first* ou *breadth-first*) explorant un espace de recherche, créé par le fractionnement d'animations en sous-animations, avec des animations réalisées à l'aide d'un solveur CSP. Le solveur CSP est utilisé pour ces deux propriétés principales : la possibilité d'employer une fonction objective pour maximiser le nombre de cibles de test atteintes durant une animation et la capacité à proposer plusieurs solutions en un temps raisonnable.

Paramètres Le comportement de cette stratégie est défini au travers de quatre paramètres :

Length est la longueur des scénarios durant cette stratégie. Cette longueur est constante pour l'ensemble des scénarios.

MaxAnim est le nombre maximum de scénarios d'animations successifs autorisés durant la stratégie. Deux scénarios sont successifs si l'état du modèle atteint après l'exécution de la première animation correspond à l'état initial du second scénario. Par conséquent, cette stratégie peut atteindre des cibles nécessitant des animations d'une longueur égale à $Length * MaxAnim$ depuis l'état du modèle défini par le diagramme d'objets UML4MBT.

Exploration est un paramètre précisant le type d'exploration du graphe construit depuis les animations exécutées. Deux types de parcours sont possibles : le parcours en profondeur ou le parcours en largeur.

NbSolution est le nombre maximum de solutions à générer à l'aide d'un solveur CSP pour augmenter l'espace de recherche et accroître la probabilité d'at-

teindre l'ensemble des cibles de test. En conséquence, ce paramètre a une influence majeure sur le temps de résolution.

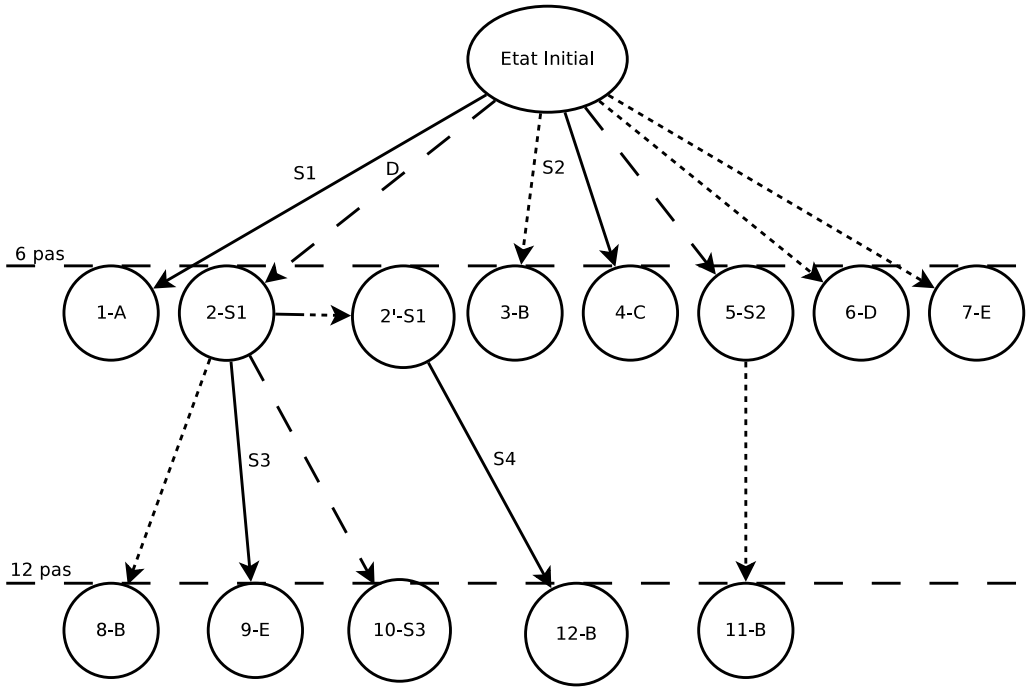


FIGURE 9.4 – Exécution de la stratégie breadth-first collaborative sur un exemple

Fonctionnement La figure 9.4 représente l'application de la stratégie Space-search collaboratif sur un exemple simple. L'objectif est d'atteindre six cibles de test nommées A, B, C, D, E et F. Ces cibles sont atteignables avec une séquence comportant respectivement un minimum de 3, 11, 5 , 6, 1 et 11 pas. De plus, les cibles B et D sont mutuellement exclusives ; Si durant une animation, la cible B est atteinte alors la cible D est inatteignable durant cette animation.

La figure respecte les conventions présentées dans la sous-section précédente concernant la figure 9.3. Le seul élément nouveau est une ligne mêlant pointillé et tiret et originaire d'un état atteint par une animation conduite à l'aide d'un solveur CSP. Cette ligne représente une demande présentée au solveur de fournir une nouvelle solution respectant des contraintes identiques à celle ayant permis d'aboutir à l'état précédent.

Dans l'application montrée au travers de la figure, la stratégie est exécutée avec des scénarios de longueur 6 pas, un maximum de 2 scénarios successifs, une exploration de type largeur et un nombre de solutions fournies par le solveur CSP de 1. Cette stratégie comporte 5 étapes.

La première étape concerne le prouveur SMT. Ce dernier essaie d'atteindre l'ensemble des cibles de test avec des scénarios de 6 pas ayant des objectifs de tests de type minimum, c'est à dire qu'au moins une cible doit être atteinte parmi une liste, et un état initial défini par le diagramme d'objets du modèle UML4MBT. Dans

l'exemple, pour faciliter la compréhension, l'objectif de tests se limite à une cible de test unique. Dans la figure, le prouveur est capable d'atteindre les cibles A et C. Les cibles B et E sont inatteignables à cause de la longueur actuelle des scénarios. Le scénario visant à atteindre la cible D a été interrompu car cette cible a été atteinte durant une autre animation.

La deuxième étape commence lorsque le prouveur SMT a exécuté une animation valide. La séquence de transitions correspondante est extraite au travers de la solution présentée par le prouveur SMT puis est transmise au solveur CSP pour être ré-exécutée. Cette exécution profite de la capacité du solveur à employer une fonction de maximisation pour augmenter nos chances d'atteindre de nouvelles cibles. Dans notre exemple, le solveur exécute deux séquences nommées OP1 et OP2 et durant la ré-exécution de OP1 la cible D est atteinte.

La troisième étape lance une nouvelle série de scénarios à l'aide d'un prouveur SMT. Cependant ces scénarios ont pour états initiaux ceux rencontrés lors des animations exécutées par le solveur CSP. Ces états sont donc extraits des réponses du solveur et convertis pour être compréhensible par un prouveur SMT. Dans notre exemple, cette étape est responsable de la découverte de la cible E.

La quatrième étape est de ré-itérer les étapes deux et trois jusqu'à ce que l'ensemble des cibles soient atteintes ou que le nombre maximum de scénarios successifs soit dépassé.

La cinquième étape est nécessaire si toutes les cibles n'ont pas été atteintes. Dans ce cas, l'espace de recherche est augmenté grâce à la création d'états supplémentaires qui sont atteignables par des séquences de transitions précédemment trouvées. Ces états sont fournis par le solveur CSP. Dans l'exemple, le solveur a fourni un nouvel état valide pour la séquence OP1. Finalement le processus est relancé depuis l'étape 3.

Cette stratégie pallie au désavantage majeur des stratégies *depth-first* et *breadth-first*. Dans un souci d'efficacité, les animations sont subdivisées en sous-animations. Cependant le gain en temps de résolution apporté par cette division est contrebalancé par l'impossibilité de prouver l'inatteignabilité de cibles de test. En effet, la division nécessite d'utiliser un état du système atteint lors d'une première animation comme état initial d'un second scénario. Mais cet état pivot peut être incompatible avec certaines cibles. Pour pallier cette difficulté, le solveur CSP peut être utilisé pour fournir d'autres états et ainsi accroître l'espace de recherche. Cette méthode ne permet pas de garantir que tous les états seront explorés mais la probabilité d'atteindre les cibles de test augmente. Cependant cette méthode est gourmande en temps car le nombre d'animations est multiplié par le paramètre NbSolution.

Les deux stratégies présentées ici permettent de combiner les avantages des prouveurs et solveurs. L'échange d'information entre ces outils est réalisé au niveau des méta-modèles associés. La section suivante présente le méta-modèle CSP4MBT et les règles de conversion depuis le méta-modèle SMT4MBT.

9.3 Méta-modèle CSP4MBT et encodage

Cette section commence par décrire le méta-modèle CSP4MBT construit spécifiquement pour le test à partir de modèle à l'aide de solveurs CSP dans le cadre de notre démarche. Le méta-modèle s'appuie fortement sur un langage directement compréhensible par un grand nombre de solveurs et nommé MiniZinc. Ce méta-modèle permet de faciliter les conversions depuis les autres méta-modèles et de capturer des notions propres à l'animation et aux tests.

La deuxième partie utilise ce méta-modèle pour définir des règles de conversion afin de convertir un scénario encodé pour un prouveur SMT en un problème compréhensible par un solveur CSP. La conversion dans l'autre sens est simplifiée car dans le cadre de notre démarche, seuls les états et cibles atteints durant une résolution menée par un solveur CSP sont intéressants du point de vue du prouveur.

9.3.1 Méta-modèle CSP4MBT

Le méta-modèle CSP4MBT définit un langage créé dans le but de combiner une représentation d'un problème compatible avec un solveur CSP et les notions liées à la génération de tests via l'animation du modèle. Notre choix pour la représentation du problème est guidé par les mêmes impératifs que le méta-modèle SMT4MBT : une expressivité suffisante pour encoder un scénario et une compatibilité avec un grand nombre de solveurs.

MiniZinc Le langage choisit pour communiquer avec les solveurs et qui remplit les conditions sus-nommées est MiniZinc. Ce langage s'inscrit dans une démarche de généralisation à l'instar de SMT-Lib. Une description des principales propriétés et de son expressivité est proposée à la section 4.2.

CSP4MBT La figure 9.5 donne le diagramme de classes du méta-modèle nommé CSP4MBT. Le méta-modèle est construit autour du langage MiniZinc restreint aux seuls éléments nécessaires pour encoder un scénario depuis un modèle SMT4MBT. Des éléments supplémentaires ont été ajoutés pour préciser le sens des différents éléments du point de vue de la génération de tests.

Dans cette figure, l'élément central est la classe **CSP** qui modélise un problème soumis à un solveur CSP. Ce problème est constitué d'un ensemble de contraintes et d'une fonction objective. Le but du solveur est de découvrir une solution satisfaisant toutes les contraintes et étant la meilleure au sens du classement établi par la fonction objective.

Les **contraintes** contiennent un unique CspElement de type booléen. Elles sont catégorisées selon leur utilité dans l'animation du modèle pour la génération de tests. Les quatre types sont :

ContrainteTest Ce type concerne les contraintes appliquées sur les cibles de test. Celles-ci peuvent correspondre à un objectif de tests ou des cibles de test complexes qui représentent des scénarios d'exécutions complexes.

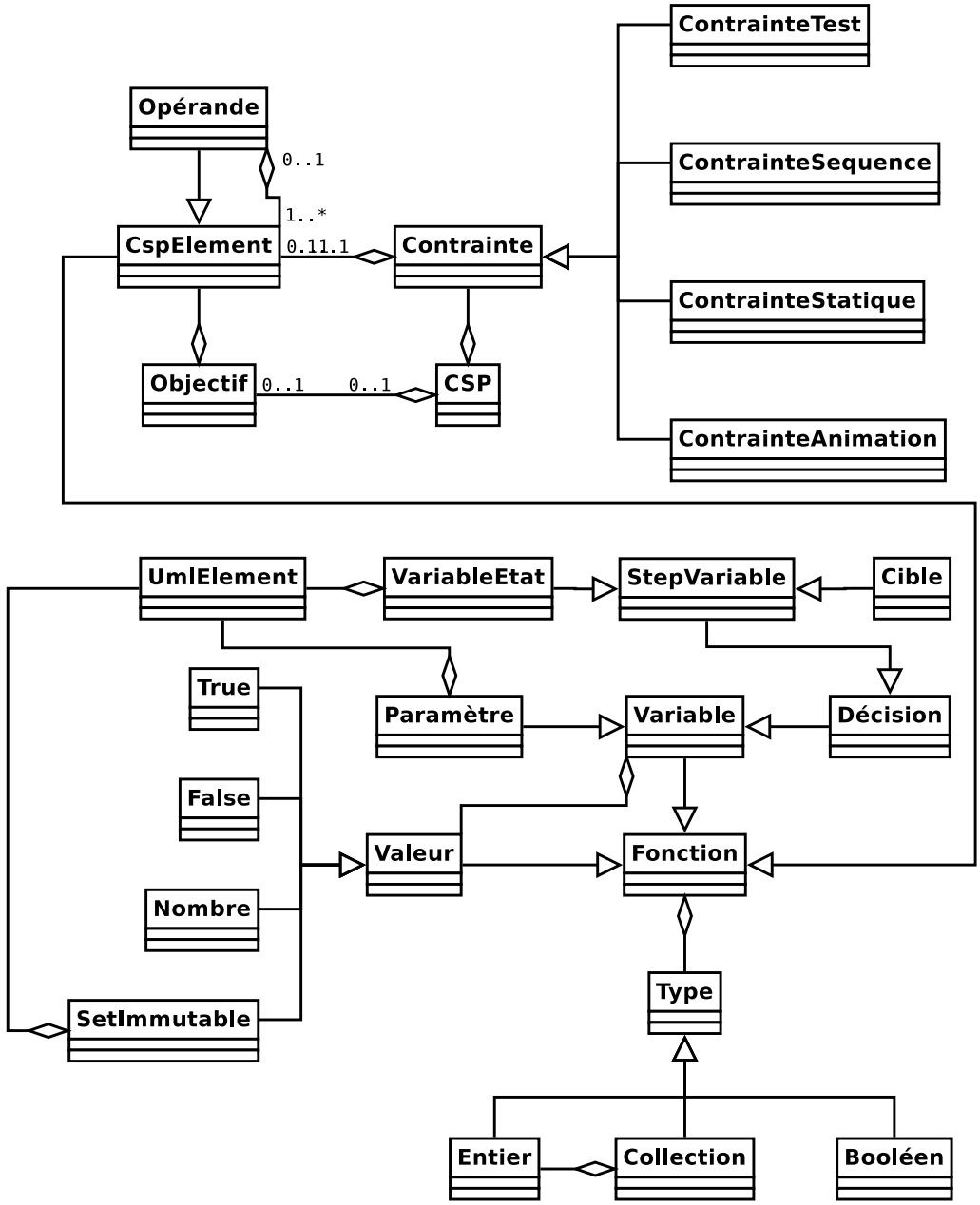


FIGURE 9.5 – Diagramme de classes du méta-modèle CSP4MBT

ContrainteSequence Dans le cadre de notre démarche, le solveur est employé uniquement pour ré-exécuter des séquences de transitions découvertes par des animations précédentes réalisées par un prouveur SMT. Ces séquences sont limitées au choix des transitions et ne précisent pas les paramètres de ces transitions. Les contraintes de type ContrainteSequence encodent les séquences à ré-exécuter.

ContrainteStatique Ces contraintes représentent l'état du modèle avant l'animation. Ceci couvre principalement la définition de l'état initial du modèle, établi dans le diagramme d'objets ou atteint lors d'animations valides précédentes

et les contraintes statiques du diagramme de classes telles que la définition d'attributs bornés. D'un point de vue de la résolution par un solveur CSP, ces contraintes ont l'avantage d'avoir peu de disjonctions et donc d'avoir un temps de traitement rapide lors de la phase de résolution.

ContrainteAnimation Le dernier type de contraintes représente la majorité du genre. Ces contraintes regroupent l'ensemble des relations définies dans le système de transition par Δ.

Les **variables** présentes dans le méta-modèle conservent la distinction issue du langage MiniZinc entre paramètres et variables de décisions. Le **paramètre** a obligatoirement une valeur associée dans le modèle alors que les **variables de décisions** n'obtiennent une valeur qu'au travers de la solution fournie par le solveur. De plus, si ces dernières ont potentiellement une valeur différente pour chacun des pas d'une animation, alors une classe particulière nommée **StepVariable** est employée. Afin d'ajouter du sens aux variables, il est possible de les distinguer en utilisant la classe **cible** et **variableEtat**. Les cibles sont dédiées aux cibles de test et sont de type booléen. Les variables d'état sont limitées aux variables nécessaires pour définir un état du modèle.

La dernière partie de la figure regroupe les classes dédiées à la définition du **type** des fonctions et des **valeurs**. Ainsi deux types basiques sont supportés, les entiers et les booléens, et un type complexe, les ensembles d'entiers. Les valeurs possibles sont : vrai, faux, un nombre et un ensemble non modifiable de nombres. De ce point de vue, les variables de types paramètres peuvent être considérées comme des alias de valeurs constantes.

Finalement la classe **UmlElement** permet de conserver un lien vers le modèle initial écrit en UML4MBT/OCL4MBT. Cette information est ajoutée aux variables d'états, aux paramètres et aux ensembles immutables. Ces éléments, fournis dans la réponse du solveur, sont suffisants pour extraire les états rencontrés lors d'une animation.

9.3.2 Règles de conversion entre SMT4MBT et CSP4MBT

Notation La formalisation des règles de conversion entre un modèle SMT4MBT et un modèle CSP4MBT nécessite l'introduction d'une notation pour décrire les entités du modèle CSP4MBT. La description des entités du modèle SMT4MBT utilisera la notation présentée à la section 8.2. Ainsi un modèle CSP4MBT est vu sous la forme d'un triplet $\Gamma_{csp} = \{T_c, F_c, C_c\}$ où :

T regroupe l'ensemble des types du modèle. Par défaut, cet ensemble contient les types entier et booléen notés *Int* et *Bool*. Les collections contiennent uniquement des éléments de type entier. Par conséquent, le type *Set* est suffisant pour représenter l'ensemble des collections.

F contient l'ensemble des fonctions du modèle. Une fonction peut être une variable de décision ou un paramètre. Une fonction est écrite avec le formalisme suivant :

$$fun(name : type)$$

fun prend une valeur spécifique pour chacune des classes de fonction. Cette valeur est *param* pour un paramètre et *deci* pour une variable de décision. Pour simplifier l'écriture, les valeurs booléennes et numériques seront écrites directement.

name indique le nom de la fonction.

type précise le type de la fonction. Ce type appartient à l'ensemble T_c du modèle γ_{csp}.

C est l'ensemble des contraintes du modèle.

Les règles de conversion utilisent le formalisme présenté dans la règle R-csp-example ci-dessous. La partie haute de la règle concerne le modèle SMT4MBT. x_{smt} est l'entité à convertir et $condition_{smt}$ précise sous quelles conditions la règle peut s'appliquer. La partie basse concerne le modèle CSP4MBT. x_{csp} est le résultat principal de la conversion et *types, fonctions, contraintes* indiquent si un type, une fonction ou une contrainte appartient au modèle γ_{csp}.

$$\frac{\Gamma_{smt} \models x_{smt} \mid condition_{smt}}{\Gamma_{csp} \models x_{csp} \mid types \in T_c, fonctions \in F_c, contraintes \in C_c} \quad \text{(R-csp-example)}$$

Types Les règles de conversion pour les types booléens et entiers sont triviales car ces types existent dans les deux méta-modèles. La règle R-csp-named permet de convertir les types libres en type entier. Cette conversion perd le typage fort présent dans le méta-modèle SMT4MBT. De façon similaire, la conversion des types ensemblistes revient à créer un type unique représentant des ensembles d'entiers.

$$\frac{\Gamma_{smt} \models x \mid x \in T, x \in Named}{\Gamma_{csp} \models Int} \quad \text{(R-csp-named)}$$

$$\frac{\Gamma_{smt} \models x \mid x \in T, x \in Set}{\Gamma_{csp} \models Set} \quad \text{(R-csp-set)}$$

Fonctions La conversion des variables et stepVariables SMT4MBT crée une variable de décision avec le même nom à l'aide de la règle R-csp-var. Dans cette dernière, $type_c$ est la conversion du type à l'aide des règles présentées dans le paragraphe précédent.

$$\frac{\Gamma_{smt} \models x \mid x \in F, x = svar(n : type) \vee var(n : type)}{\Gamma_{csp} \models deci(n : type_c) \mid n \in F_c} \quad \text{(R-csp-var)}$$

La conversion des constantes est traitée selon leur type. Les constantes booléennes et les nombres sont trivialement convertis sans subir de modification. Les constantes de type libre nécessitent un traitement plus complexe présenté dans la règle R-csp-const. Ces constantes représentent des valeurs constantes telles les instances ou les valeurs des énumérations du modèle UML4MBT. Les types libres sont convertis en type entier. En conséquence, ces constantes doivent être converties en

des nombres différents. Ceci est assuré au travers de la fonction *inc*() qui renvoie un nombre unique par incrémentation.

$$\frac{\Gamma_{smt} \models con(n : libre) \mid n \in F}{\Gamma_{csp} \models param(n : Int) \mid n \in F_c, n = inc() \in C_c} \quad \text{(R-csp-const)}$$

Contraintes Les trois règles R-csp-unary, R-csp-binary et R-csp-distinct convertissent les opérateurs ayant un équivalent dans les deux méta-modèles. Dans ces deux premières règles, le symbole $\diamondsuit$ peut être substitué par respectivement les opérateurs $\neg, -$ ou $\wedge, \vee, \otimes, -, +, *, div, mod, <, >, \leqslant, \geqslant, has$.

$$\frac{\Gamma_{smt} \models \diamondsuit x \mid x \in Bool \vee x \in Int}{\Gamma_{csp} \models \diamondsuit x} \quad \text{(R-csp-unary)}$$

$$\frac{\Gamma_{smt} \models x_1 \diamondsuit x_2 \mid x_1, x_2 \in Bool \vee x_1, x_2 \in Int, x_1 \in Libre : Set \wedge x_2 \in Libre}{\Gamma_{csp} \models x_1 \diamondsuit x_2} \quad \text{(R-csp-binary)}$$

$$\frac{\Gamma_{smt} \models distinct\ x_1, \ldots, x_n \mid x_1, \ldots, x_n \in Libre^n}{\Gamma_{csp} \models allDifferent\ x_1, \ldots, x_n} \quad \text{(R-csp-distinct)}$$

Les deux opérateurs conditionnels *ite* et $\Rightarrow$ nécessitent une conversion plus complexe. En effet, $\Rightarrow$ n'a pas d'équivalent direct. La règle R-csp-impl effectue la conversion en remplaçant l'opérateur par une formule logique équivalente. L'opérateur *ite* existe dans le langage MiniZinc mais il comporte une condition importante sur l'expression booléenne guidant le choix de la branche. Cette expression ne peut pas contenir de variables de décisions. En conséquence, cet opérateur est également converti en une formule logique équivalente à l'aide de la règle R-csp-ite.

$$\frac{\Gamma_{smt} \models x_1 \Rightarrow x_2 \mid x_1, x_2 \in Bool}{\Gamma_{csp} \models (x_1 \wedge x_2) \vee (\neg x_1)} \quad \text{(R-csp-impl)}$$

$$\frac{\Gamma_{smt} \models ite\ x_1\ x_2\ x_3 \mid x_1, x_2, x_3 \in Bool}{\Gamma_{csp} \models (x_1 \wedge x_2) \vee (\neg x_1 \wedge x_3)} \quad \text{(R-csp-ite)}$$

La validité des règles de conversion présentées ici a été établie par l'utilisation de tests unitaires lors de l'implémentation et de l'exécution des cas de test en résultant par l'animateur de l'outil industriel Certifi-It.

9.3.3 Règles de conversion entre CSP4MBT et MiniZinc

Un des objectifs du méta-modèle CSP4MBT étant de rester le plus proche possible du langage MiniZinc, la conversion d'un modèle CSP4MBT dans ce langage est réalisée facilement et à faible coût en terme de temps et de complexité. Une des conversions plus importantes est l'écriture des ensembles. Les ensembles sont convertis en des tableaux d'entiers contenant l'ensemble des nombres associés aux constantes SMT4MBT du type correspondant et créés par la fonction *inc*().

De façon identique à l'écriture d'un modèle SMT4MBT en SMT-lib, il est nécessaire de dupliquer les variables de décisions et les contraintes les contenant pour chacun des pas du scénario encodé. Ainsi le nombre de variables et de contraintes grandit linéairement avec le nombre de pas.

Une des possibilités d'intervention sur le processus de résolution d'un solveur CSP est l'ordre des contraintes modélisant le problème. Notre but est réduire le temps de résolution en privilégiant les contraintes capables de réduire fortement les domaines des variables durant la construction du graphe des contraintes. En conséquence, les contraintes seront transmises dans l'ordre : *statique* → *sequence* → *test* → *animation* lors de l'écriture du problème en MiniZinc.

9.4 Synthèse

Pour établir une collaboration entre un prouveur SMT et un solveur CSP, une méthode de communication entre ces outils a été établie. Ces deux outils manipulent un problème encodé sous la forme de formules logiques du premier ordre. Par conséquent, l'échange d'information se fait par un ensemble de formules portant sur les variables d'états du modèle. Cette méthode établie ainsi un processus de communication entre ces deux outils.

Le rôle de chacun de ces outils est clairement défini pour profiter de leurs avantages respectifs. Le prouveur est chargé de trouver des motifs d'animations valides qui seront exploités par un solveur pour augmenter les chances d'atteindre les cibles de test. Grâce à ces rôles, la génération de tests est scindée en deux parties. La première concerne le modèle UML4MBT et le prouveur. La seconde concerne le modèle SMT4MBT utilisé pour le prouveur et le solveur. Dans ce cas, le modèle UML4MBT n'est pas directement utilisé par le solveur.

La collaboration a conduit à la création de deux stratégies de génération de tests : *step-increase collaboratif* et *space-search collaboratif*. Ces stratégies sont des améliorations de celles présentées à la section 7.2. Dans la première, la probabilité d'atteindre des cibles est renforcée par l'utilisation d'une fonction objective par le solveur qui vise à maximiser le nombre de cibles atteintes. Dans la deuxième stratégie, le solveur est principalement employé pour accroître la taille de l'espace de recherche en générant des états du modèle supplémentaires.

L'usage d'un solveur passe par un langage commun nommé MiniZinc. Cependant ce langage n'est pas adapté pour modéliser les notions propres aux tests à partir de modèle. En conséquence, un méta-modèle CSP4MBT est créé pour conjuguer ses notions avec une expressivité proche de celle fournie par MiniZinc. A l'instar du méta-modèle SMT4MBT, des méta-informations permettent de conserver la motivation à l'origine de la création d'une entité. Pour créer un modèle CSP4MBT depuis un modèle SMT4MBT, des règles de conversions ont été créées.

Troisième partie

Étude de cas

Chapitre 10

Implémentation

Sommaire

La partie précédente du manuscrit a présenté une méthode de génération automatique de tests à partir de modèles en utilisant un prouveur SMT et un solveur CSP. Cette méthode se décline en plusieurs variantes formalisées par des stratégies de génération de tests. Ces dernières reposent sur les performances des prouveurs et des solveurs en fonction des caractéristiques des problèmes soumis.

Ces différentes variantes ont été implémentées au sein d'une plate-forme de traitement de modèles nommée Hydra. Ce chapitre a pour objectif de caractériser notre implémentation en regard des propriétés nécessaires pour obtenir un fonctionnement correct des stratégies. Par exemple, la stratégie d'espace de recherche est efficace uniquement si la résolution d'une animation fractionnée est plus rapide qu'une animation globale.

Ce chapitre commence par décrire l'implémentation de la démarche de génération de tests au sein de la plate-forme Hydra. Ensuite cette implémentation est caractérisée au travers de l'évaluation des paramètres principaux des animations en terme de performance. Finalement une synthèse sera donnée dans la dernière section.

10.1 Implémentation

L'implémentation de notre méthode de génération automatique de cas de test à partir de modèles à l'aide d'un prouveur SMT et de solveur CSP est réalisée au sein d'une plate-forme modulaire de traitement sur des spécifications nommée Hydra.

Cette sous-section commence par présenter les caractéristiques principales de la plate-forme avant de détailler plus spécifiquement notre implémentation.

10.1.1 Plate-forme Hydra

Hydra est une plate-forme modulaire qui reprend les caractéristiques principales de ce type de projet (Eclipse plate-forme, Netbeans plate-forme). Ce logiciel a été développé au sein du Département d'Informatique des Systèmes Complexes. Il a les caractéristiques suivantes :

- Il est centré autour de la notion de spécification. Une spécification est une représentation d'un langage tel le langage B ou le langage UML4MBT. Ainsi, les fonctionnalités apportées par les différents plugins se concentrent sur la manipulation (conversion, modification, analyse...) d'une instance se conformant à une spécification.
- Les dépendances des plugins sont analysées et l'utilisation conjointe d'un gestionnaire de dépôt, Nexus, permet d'automatiser le téléchargement des plugins et de leurs dépendances.
- Le langage Java a conduit à obtenir une plate-forme multi-environnements. Cet outil est utilisé actuellement sous les systèmes d'exploitation Linux, Mac OS X et Windows.
- La plate-forme propose une interface graphique basique pouvant être enrichie de manière cohérente par les différents plugins. Les opérations basiques (parsing, affichage des instances) sont gérées directement par la plate-forme.
- Une interface en ligne de commande est également disponible. Cette interface est particulièrement utilisée lors de l'utilisation automatique de la plate-forme au sein de scripts ou de serveurs distants ne supportant pas l'affichage déporté.

Les plugins sont spécialisés en fonction de leur objectif. D'après l'architecture de la plate-forme, un plugin apporte une fonctionnalité ou groupe de fonctionnalités en fonction d'une spécification. Un plugin appartient à l'une des catégories suivantes :

Modèle décrit le méta-modèle d'une spécification et fournit les outils pour manipuler les instances du méta-modèle. Ce type est fondamental car Hydra est construit autour de la notion de spécification. Par conséquent, les plugins d'un autre type s'appuient nécessairement sur un plugin modèle.

Parser permet de lire un fichier encodant une instance d'un méta-modèle pour en obtenir une représentation partagée en mémoire. La création d'un type dédié pour les parsers permet d'obtenir un comportement cohérent, de factoriser le code et facilite l'intégration de cette fonctionnalité dans l'interface graphique ou en ligne de commande.

Traducteur regroupe les plugins chargés de convertir une instance d'un méta-modèle d'une première spécification vers une seconde spécification. De plus, si cela s'avère nécessaire, il est également responsable de la mémorisation d'informations servant à associer les éléments de l'instance convertie à l'originale. Néanmoins, la conversion ne doit pas être obligatoirement réversible.

Service propose une fonctionnalité non couverte par les types parsers et traducteurs directement accessible à l'utilisateur final au travers d'une interface graphique ou console. Ainsi la plate-forme propose des méthodes pour intégrer cette fonctionnalité de manière cohérente dans les interfaces. De plus, la fonctionnalité s'applique à une représentation présente en mémoire.

Bibliothèque est le pendant du type service. Il fournit des fonctionnalités employées uniquement par d'autres plugins. Ainsi ce type de plugin est transparent pour l'utilisateur. La plate-forme les récupère et les emploie automatiquement car ils sont définis comme dépendance par les autres plugins.

Outil autorise la communication avec un programme externe. La communication peut employer une interface de programmation, un fichier ou les sorties standards du système. Ce type de plugin est également responsable de la gestion des ressources consommées par le programme. L'utilisation de plugin de ce type peut conduire à des limitations en terme d'environnement pour correspondre aux cas d'utilisation du programme externe.

Actuellement, une nouvelle version est développée en s'appuyant sur l'architecture de la plate-forme Netbeans.

10.1.2 Implémentation

La description de notre implémentation s'articulera autour de la présentation des différents plugins la constituant. Ces derniers et leurs dépendances sont présentés dans la figure 10.1. Il existe les plugins suivants :

Hydra-Core est le plugin de base autorisant l'intégration au sein de la plate-forme. Il fournit les APIs pour stocker des instances d'une spécification et permettre à l'utilisateur de sélectionner et exécuter les différents plugins Service, Parser et Traducteur. Ainsi tous les autres plugins dépendent au final de ce dernier.

UML4MBT-model décrit le méta-modèle associé aux langages UML4MBT et OCL4MBT. Ce plugin permet également de naviguer au sein d'une instance. Le plugin regroupe deux langages car dans la pratique ils sont toujours employés simultanément. Ainsi, il est évident que OCL4MBT dépend de UML4MBT pour lui fournir un contexte. Par exemple, le symbole *this* est défini par le contexte. Dans une opération, *this* correspond à l'instance de la classe contenant l'opération. D'un autre coté UML4MBT pourrait employer un autre langage de contrainte à l'instar d'UML mais pour l'instant seul OCL4MBT existe.

UML4MBT-parser crée une instance du méta-modèle UML4MBT en mémoire en lisant un fichier au format UML4MBT.

UML4MBT-meta manipule une instance UML4MBT pour le modifier en lui ajoutant des informations. Ce plugin remplit des rôles externes et internes à notre démarche de génération automatique de tests. Dans notre démarche, la création des cibles de test n'est pas considérée car un procédé externe est chargé de les générer. Dans le cadre de nos expérimentations, ce plugin est responsable de cette étape. Il est capable de créer des cibles en fonction de deux critères : chaque transition du diagramme d'états-transitions et opération du diagramme de classes doit être exécutable ou chaque comportement élémentaire présent dans les opérations et les transitions doit être exécutable. Le rôle interne est limité à l'ajout d'une opération fictive sans condition d'activation et ne modifiant pas l'état du modèle. L'intérêt de cette opération est décrit à la section 7.1.

UML4MBT-SMT4MBT permet de convertir une instance UML4MBT en une instance SMT4MBT. Cette conversion nécessite de préciser la longueur des animations voulues depuis l'instance SMT4MBT. Le plugin conserve les informations utiles pour associer les fonctions SMT4MBT aux entités UML4MBT correspondantes. Les règles de conversion employées sont présentées dans le chapitre 8. Le temps d'exécution de cette phase de conversion impacte fortement les performances par rapport à la résolution d'un problème par un solveur ou un prouveur si le problème a une taille limitée.

SMT4MBT-model décrit le méta-modèle UML4MBT présenté à la section 8.1. Le plugin dispose de mécanismes pour s'assurer de la validité des contraintes stockées d'un point de vue syntaxique et pour optimiser les contraintes dont la résolution est triviale.

SMT4MBT-writer permet d'écrire dans un fichier un problème au format SMT-lib v1.2 ou v2.0. Ce problème est dérivé d'une instance SMT4MBT. La simplicité de la conversion a conduit à l'absence d'un plugin de traduction et de modèle SMTLib dédiés. La taille des fichiers pour des modèles industriels peut atteindre plusieurs centaines de méga-octets. Ainsi, le temps d'exécution lors de cette phase d'écriture est à considérer pour l'analyse des performances. Quand l'écriture de deux problèmes diffère seulement sur l'objectif de l'animation, il est possible de convertir le premier fichier pour qu'il représente le deuxième problème avec un coût quasi-nul.

SMT4MBT-prover autorise la soumission d'une instance SMT4MBT, une fois traduite sous la forme d'un problème conforme au langage SMT-lib, à un prouver SMT. Pour maximiser la compatibilité avec les différents prouveurs, la communication se fait par l'écriture et l'exécution d'un script. Le plugin est également responsable de l'analyse de la réponse du prouveur et, si elle est positive, de sa transcription en regard du modèle UML4MBT. Le processus d'analyse et de création de script oblige à adapter le code à chacun des prouveurs. Pour l'instant, les prouveurs CVC3, CVC4, Z3 et MathSAT5 sont supportés.

SMT4MBT-CSP4MBT est chargé de la conversion d'une instance SMT4MBT en une instance CSP4MBT. La conversion est effectuée à l'aide des règles

présentées à la section 9.3. Le processus conserve les informations nécessaires pour relier les fonctions CSP4MBT aux fonctions SMT4MBT correspondantes et par conséquent, aux entités UML4MBT avec les informations précédemment créées.

CSP4MBT-model décrit le méta-modèle CSP4MBT présenté à la section 9.3. Le plugin dispose de mécanismes pour s'assurer de la validité des contraintes stockées d'un point de vue syntaxique et pour optimiser les contraintes dont la résolution est triviale.

CSP4MBT-writer autorise l'écriture de fichiers au format Minizinc décrivant un problème CSP. Ce problème est dérivé d'une instance CSP4MBT. Le plugin a les mêmes caractéristiques que le plugin SMT4MBT ; l'analyse des performances doit considérer le temps d'écriture et le processus de conversion est suffisamment simple pour ne pas nécessiter un plugin traducteur.

CSP4MBT-solver résoud des problèmes CSP en les soumettant à des solveurs compatibles avec le format MiniZinc. De manière similaire à la résolution de formules par les prouveurs, la communication est réalisée au travers de scripts et la réponse est analysée. Pour l'instant, seul le solveur Gecode est supporté.

Test-strategies mène les stratégies présentées aux sections 9.2 et 7.2. Avec le plugin, un utilisateur peut générer des tests automatiquement depuis un modèle UML4MBT/OCL4MBT à l'aide des stratégies constituant notre démarche. Ce plugin est également responsable de la parallélisation. Ainsi, avant toute exécution, l'utilisateur sélectionne le nombre de processus utilisables. Les paramètres des stratégies sont renseignés par l'utilisateur au travers de l'interface graphique de la plate-forme.

Test-benchmark évalue les performances des stratégies. Il permet d'exécuter une suite de génération en variant les stratégies et leur configuration en ligne de commande.

10.2 Caractéristiques

Les différentes stratégies proposées dans notre démarche reposent sur toutes ou parties des hypothèses suivantes :

- Le temps de résolution d'un problème encodant une animation croit au moins d'un facteur $k > 1$ en fonction de la longueur d'une animation.
- Le temps de résolution d'une animation est supérieur au temps nécessaire pour résoudre un ensemble de sous animations constitutives de l'animation originale.
- Le temps de création et d'écriture de modèles SMT4MBT devient négligeable devant le temps de résolution pour un nombre de contraintes suffisamment grand.

L'objectif de cette sous-section est de vérifier la validité de ces hypothèses sur notre modèle fil rouge. L'ensemble des mesures ont été prises sur un ordinateur avec un processeur 64 bits de 2.20 Ghz et 6 Go de RAM. Les outils utilisés sont : Z3 en version 4.1, cvc3 en version 2.4.1, cvc4 en version 1.0 et MathSat en version 5.2.6.

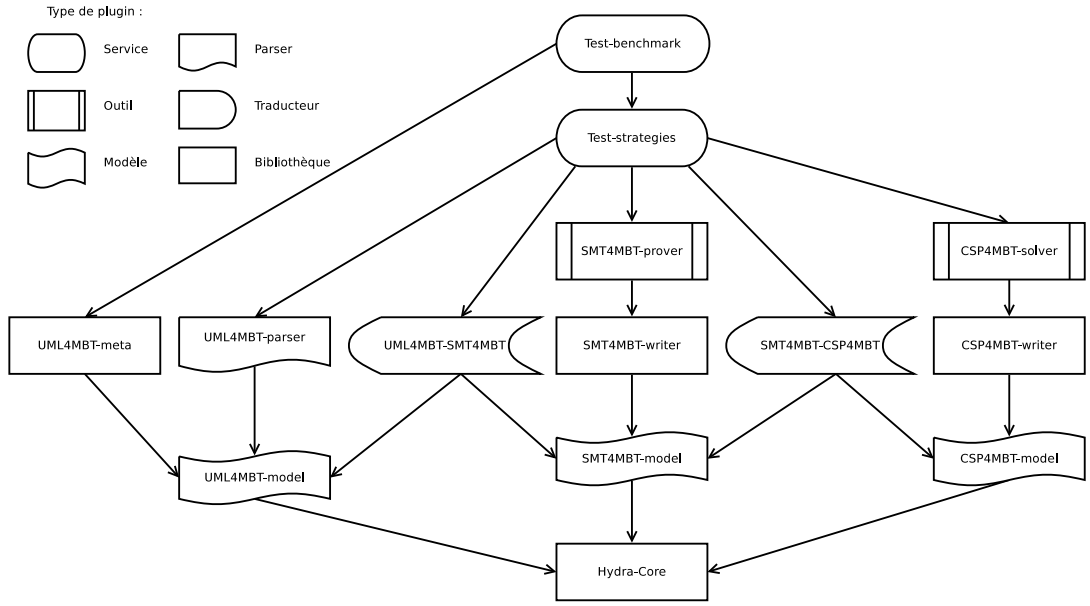


FIGURE 10.1 – Plugins utilisés pour la génération de tests et leurs dépendances

10.2.1 Longueur des animations

L'objectif de cette expérience est d'évaluer l'influence de la longueur d'une animation sur le temps de résolution. Il est important de se rappeler que nos règles de conversions conduisent à dupliquer toutes les fonctions SMT4MBT représentant des variables d'états ou d'entrées du modèle UML4MBT/OCL4MBT et toutes les contraintes contenant ces fonctions. Ainsi le nombre de fonctions et de contraintes croit linéairement avec la longueur des animations.

Dans le cadre de notre exemple fil rouge, les relations sont les suivantes :

$$NbFonctions = 48 * NbPas + 17$$
$$NbContraintes = 108 * NbPas + 44$$

Pour mesurer cette influence, des scénarios d'animations ayant une longueur variant de 1 à 2001 pas avec des incréments de 21 pas sont soumis aux différents prouveurs. Pour minimiser l'influence de l'ordre de traitement des contraintes, la mesure est répétée 10 fois pour chacun des scénarios en modifiant la graine du générateur de nombres aléatoires utilisés et seule la moyenne de ces mesures est considérée. Ces scénarios ont pour objectif de tests d'atteindre au moins une cible de test parmi toutes celles existantes. Puisque la cible Cpt_{a_p} est atteignable à l'aide d'une animation de 1 pas et que l'animation peut employée une transition fictive sans influence sur le système, tous les scénarios conduiront à la naissance d'animations valides.

Le graphique 10.2 donne le temps de résolution des scénarios d'animations en fonction de la longueur de ces derniers. D'après ce graphique, les faits suivants sont déductibles :

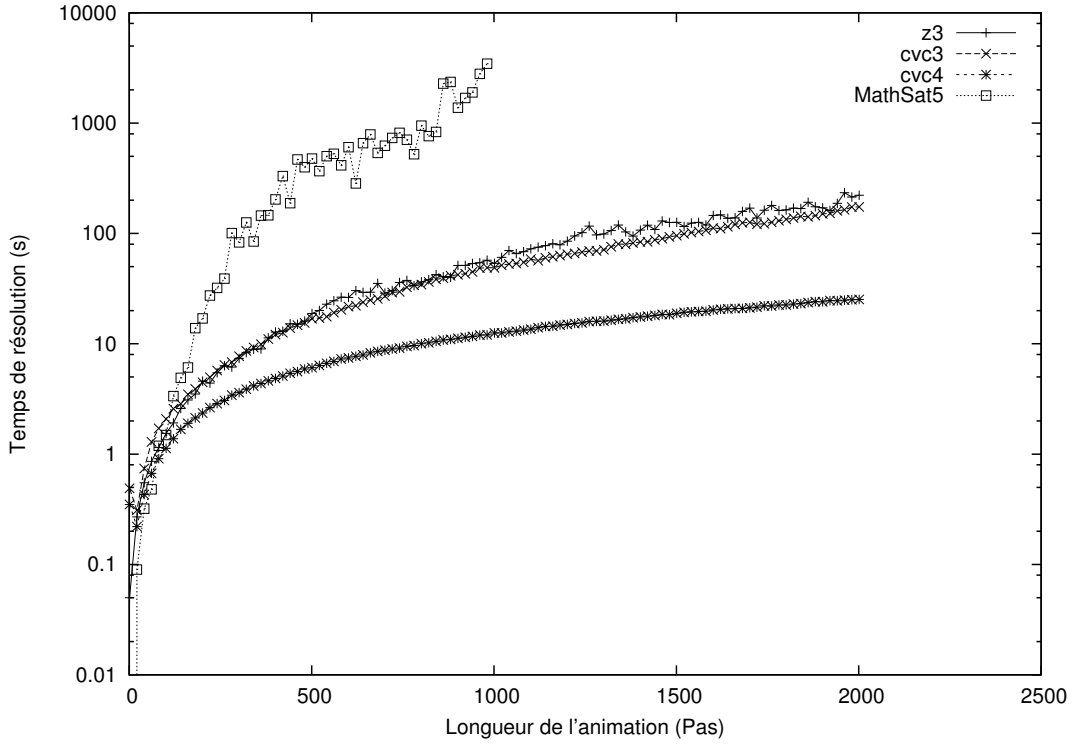


FIGURE 10.2 – Temps de résolution en seconde en fonction de la longueur des animations en nombre de pas.

- En moyenne, le temps de résolution s'accroît avec la longueur des animations. Ceci conforte l'intérêt d'utiliser la stratégie *Step-Increase*.
- En terme de vitesse de résolution, la hiérarchie des prouveurs du meilleur au moins bon est : CVC4, CVC3, Z3 et MathSat5. Les performances se divisent en trois groupes. CVC4 se détache clairement au niveau des performances. CVC3 et Z3 ont des résultats proches avec néanmoins un avantage croissant avec la longueur des animations pour CVC3. Les performances de MathSat5 sont inférieures de plus d'un ordre de grandeur. C'est pourquoi les mesures ont été arrêtées pour les animations de plus de 1002 pas.
- Les écarts en terme de performance grandissent avec la longueur des animations.
- Il est plus rapide d'animer successivement 2 sous-animations qu'une seule animation combinaison des deux précédentes. Ceci conforte l'intérêt d'utiliser la stratégie *Space-search*.

10.2.2 Taille du diagramme d'objets

L'objectif de cette expérience est d'évaluer l'influence de la taille du diagramme d'objets sur le temps de résolution. Cette expérience mesure ainsi la mise à l'échelle de notre démarche à des modèles de tailles industrielles. Elle évalue aussi l'intérêt de multiplier les instances afin de décrire simultanément plusieurs états du modèle afin

de diminuer la longueur des animations nécessaires pour atteindre les cibles de test. Par exemple, dans le cas du robot, la description simultanée d'un état où l'ensemble des quais sont vides et d'un état pour un deuxième système où une pièce est présente sur un des quais d'évacuation permet d'atteindre les cibles Cpt_{a_p} et $Cpt_{evacuation}$ à l'aide d'animation d'un pas.

Pour mesurer cette influence, des scénarios d'animations ayant une longueur de 30 pas sont soumis aux différents prouveurs. Ces scénarios ont pour objectif de tests d'atteindre au moins une cible de test parmi toutes celles existantes. De la même façon que précédemment, les résultats sont une moyenne de 10 mesures et tous les scénarios conduisent à des animations valides. Dans le cadre du robot, le diagramme d'objets est augmenté par des incréments de quatre instances qui sont des instanciations des quais et du robot. Ainsi le robot vérifie les relations suivantes :

$$NbFonctions = a * NbInstance + b$$
$$NbContraintes = a * NbInstance + b$$

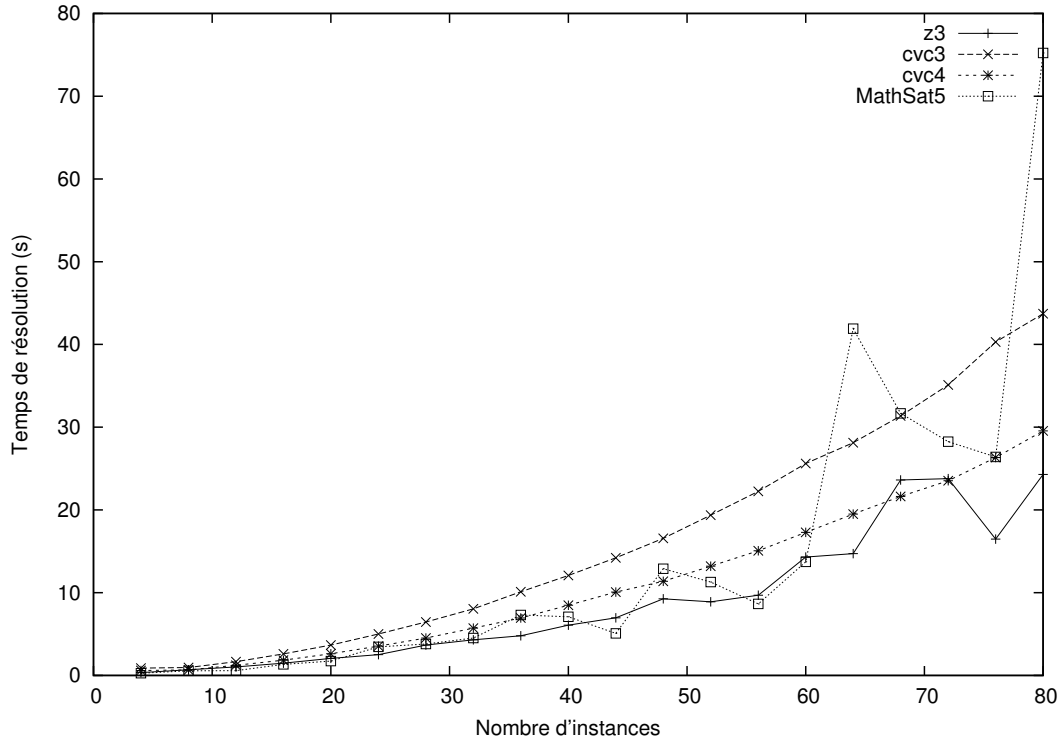


FIGURE 10.3 – Temps de résolution en seconde en fonction du nombre d'instances du diagrammes d'objets.

Le graphique 10.3 donne le temps de résolution des scénarios d'animations en fonction du nombre d'instances du diagramme d'objets. D'après ce graphique, les faits suivants sont déductibles :

– Les performances en termes de temps de résolution en fonction du nombre d'instances diffèrent par rapport à l'expérience précédente. Le classement est :

Z3, CVC4 et CVC3. Les performances de MathSat 5 sont trop erratiques pour permettre de le classer. De plus, les performances des différents prouveurs sont proches.
- Les écarts entre les prouveurs grandissent avec le nombre d'instances.
- Une méthode de modélisation mise en avant par cette expérimentation est de fournir plusieurs états du système en dupliquant des instances du diagramme d'objets. L'ajout d'instances accroît le nombre de fonctions et de contraintes dans le modèle SMT4MBT mais permet de réduire la longueur des animations nécessaires pour générer les cas de test.

10.2.3 Satisfiabilité des animations

L'objectif de cette expérience est d'évaluer l'influence de la satisfiabilité de l'animation sur le temps d'exécution. De manière intuitive, le temps de résolution semble être inférieur si l'animation est invalide car le prouveur n'a pas à générer de modèle et la découverte de contraintes non satisfiables peut terminer le processus de résolution. Par exemple, le traitement de la contrainte $2 = 3$ permet de conclure directement à l'absence de solution.

Pour mesurer l'impact de la satisfiabilité, les scénarios d'animations utilisés pour évaluer l'influence de la longueur des animations sont modifiés pour obtenir des animations invalides. L'objectif de tests de ces animations correspond à l'exécution successive de deux opérations *a_p*. Hors, l'opération "d'arrivée pièce" ne peut pas être exécutée successivement deux fois car le quai d'arrivée accepte une seule pièce à la fois. Le temps de résolution de ces scénarios est ensuite comparé au temps des scénarios originaux pour mettre en valeur le gain apporté par la non-satisfiabilité. De façon identique à l'expérience précédente, les mesures sont la moyenne de 10 exécutions.

Le diagramme 10.4 donne le gain en pourcent pour le temps de résolution lors de la soumission de scénarios d'animations non-satisfiables en fonction de la longueur de ces animations. Ainsi une valeur de 50% signifie que le prouveur est deux fois plus rapide pour retourner une réponse quand le scénario est non-satisfiable. D'après ce graphique, les faits suivants sont déductibles :
- La différence entre le temps de résolution d'un scénario d'animation satisfiable et un scénario non-satisfiable est la plus importante dans l'ordre pour : MathSat 5, Z3, CVC3 et CVC4. Cette différence augmente avec les prouveurs MathSat 5, Z3 et CVC3 . Elle est constante autour de 60% pour CVC4.
- La combinaison de ces résultats avec ceux de la première expérience montre que les performances des prouveurs sont beaucoup plus proches dans le cas de la résolution de scénarios d'animations non-satisfiables. En particulier, le prouveur MathSat 5 redevient compétitif dans ce cas de figure.
- D'après ces résultats, les stratégies engendrant plus de scénarios d'animations ont finalement un surcoût plus faible du fait de ces dernières. Ceci renforce l'intérêt de la stratégie *Space-search*. En effet, l'exploration de l'espace de recherche multiplie les scénarios non-satisfiable.

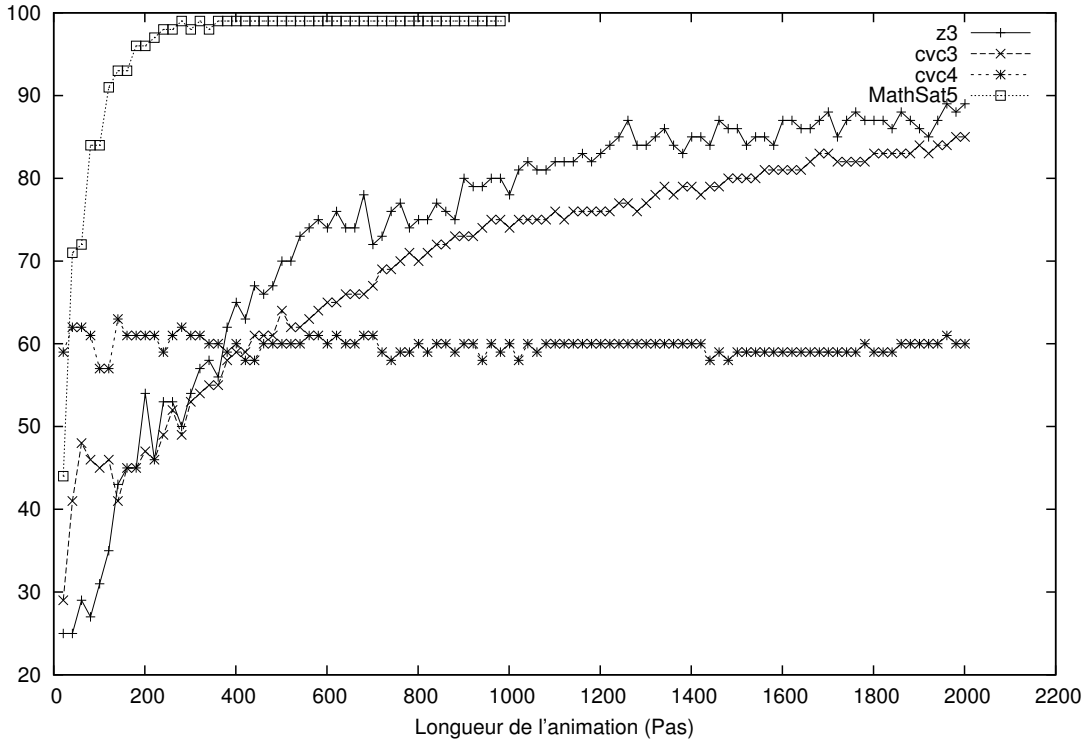


FIGURE 10.4 – Différence en pourcentage du temps de résolution de scénarios d'animations selon leur satisfiabilité en fonction de la longueur des scénarios en nombre de pas.

10.2.4 Répartition du temps de génération des tests

L'objectif de cette expérience est d'étudier la répartition du temps de génération des tests en fonction de la longueur des animations. D'après l'architecture de l'implémentation présentée dans la section précédente, le temps se divise principalement entre :

- Le temps de résolution d'un problème par un solveur ou un prouveur.
- Le temps d'écriture d'un problème dans un fichier et le temps nécessaire pour convertir le modèle UML4MBT/OCL4MBT dans un modèle SMT4MBT.

Pour connaître la répartition, le temps utilisé par chaque plugin est mesuré lors du processus permettant d'analyser la réponse d'un prouveur à la soumission d'un scénario d'animation créé depuis un modèle UML4MBT/OCL4MBT. Ce processus est répété pour des scénarios de longueurs différentes et conduisant toujours à l'obtention d'animations valides. De façon identique aux expériences précédentes, les mesures sont la moyenne de 10 exécutions.

Le diagramme 10.5 met en évidence la répartition du temps de génération de tests entre le temps d'écriture, le temps de conversion et le temps de résolution en fonction de la longueur des scénarios d'animations. D'après ce graphique, les faits suivants sont déductibles :

- Dans le cas du prouveur MathSat 5, le temps de génération d'un cas de test

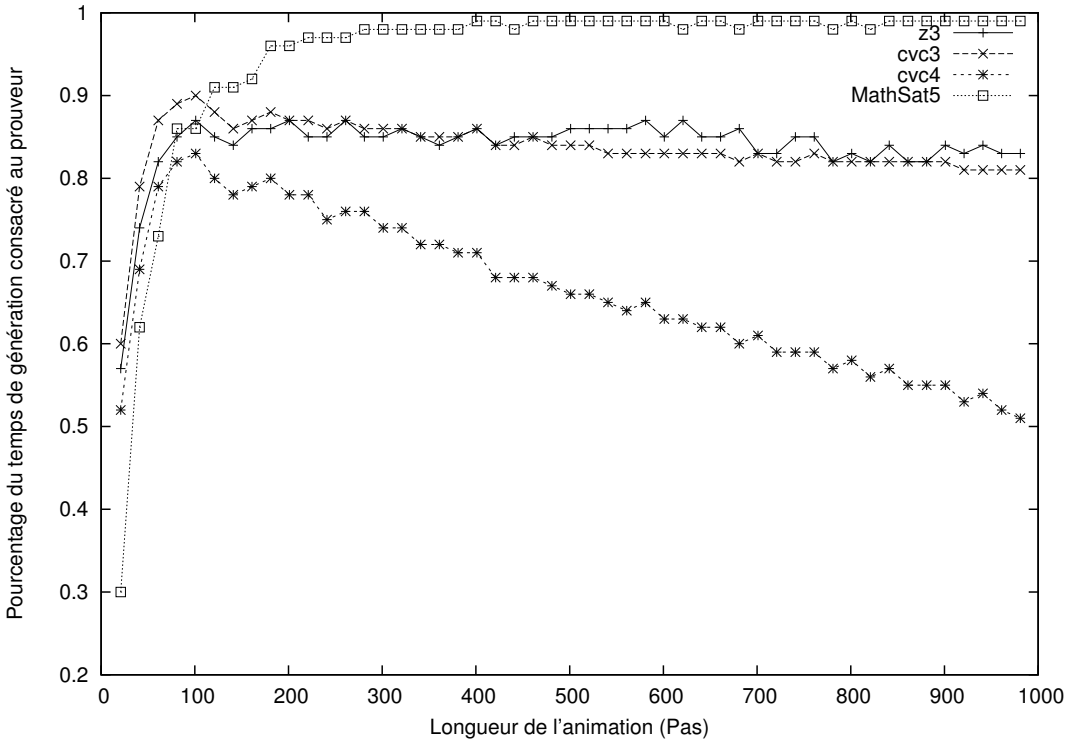


FIGURE 10.5 – Répartition du temps de générations des tests entre les différents plugins en fonction de la longueur des animations en nombre de pas.

est quasiment assimilable (a 2% près) au temps de résolution du scénario d'animations.
- Dans le cas des prouveurs CVC3 et Z3, le temps de résolution prend environ 85% du temps de génération.
- Dans le cas du prouveur CVC4, la répartition entre le temps de résolution et le temps de conversion et d'écriture s'oriente en faveur du temps de conversion quand la longueur augmente. Dans l'expérience, le temps de résolution passe de 85% à 50% du temps de génération quand la longueur évolue de 100 à 1000 pas.

D'après ces résultats, le temps de conversion et d'écriture des scénarios d'animations doit être pris en compte dans l'analyse des performances. Ainsi, les stratégies mutualisant le processus de conversion en exécutant plusieurs scénarios de longueurs identiques où seul l'objectif de tests où les contraintes modélisant l'état initial du système varie. Dans l'ordre, les stratégies privilégiant cette mutualisation sont : *Space-search*, *Basic*, *Step-Increase* et *Path*.

10.3 Synthèse

L'implémentation de notre démarche de génération automatique de tests à partir de modèles UML4MBT/OCL4MBT en utilisant un prouveur SMT ou un solveur

CSP a été réalisée au sein de la plate-forme modulaire Hydra. Cette plate-forme s'articule autour de la manipulation de spécifications, transcrites sous forme de méta-modèles. Elle offre les fonctionnalités classiques de ce type d'environnement telles une interface graphique cohérente ou une gestion des dépendances des modules.

Une première analyse des performances de cette implémentation confirme les hypothèses suivantes :

- Le temps de résolution d'un scénario d'animation par un prouveur est supérieure à la somme des temps de résolution de sous-animations constituant l'animation originale.
- Lors de la phase de modélisation, un modèle décrivant plusieurs états du système dans le diagramme d'objets en multipliant les instances des classes peut conduire à de meilleures performances si ces nouveaux états conduisent à la création de scénarios d'animations avec des longueurs inférieures.
- L'influence de la validité des animations sur le temps de génération diffère fortement selon les prouveurs. Plus un prouveur a un temps de résolution élevé pour les scénarios satisfiables, plus ce dernier diminue en cas de non-satisfiabilité.
- Le temps de génération de tests se répartit principalement entre le temps de conversion entre les modèles, le temps d'écriture dans un fichier et le temps de résolution des prouveurs et des solveurs. Cependant plus les scénarios ont une longueur importante, plus les temps d'écriture et de conversion deviennent négligeable devant le temps de résolution.
- En terme de performance, le classement des prouveurs est CVC4, CVC3, Z3 et MathSat 5. Les performances de MathSat 5 lors de la résolution de scénario satisfiable le déconsidère dans le cadre de notre démarche de génération de tests.

Ces premières expérimentations valident les hypothèses dont découlent les stratégies de générations de test. Ainsi la prochaine étape est d'évaluer les performances de ces dernières sur différents cas d'études.

Chapitre 11

Études de cas

Sommaire

Le chapitre précèdent a décrit l'implémentation des stratégies constituant notre démarche au sein de la plate-forme Hydra. Cette implémentation est utilisée dans ce chapitre pour évaluer les performances de notre méthode de génération de tests sur plusieurs systèmes.

Les questions suivantes sont abordées au travers d'expérimentations :

- Quelles stratégies ont les meilleures performances en fonction des caractéristiques des systèmes et de leur modélisation ?
- Quels gains apporte la parallélisation du processus de génération ?
- Quelle est l'influence du choix du solveur CSP et du prouveur SMT en terme de vitesse de génération ?
- Quels avantages apporte l'adaptation d'une stratégie à un modèle et une campagne de tests au travers de ces paramètres de configuration ?

Ce chapitre commencera par décrire les cas d'études et leur modélisation. En particulier, les choix de modélisation pris en fonction des caractéristiques des langages UML4MBT/OCL4MBT seront explicités. Ensuite, les résultats des stratégies appliquées aux cas d'études seront donnés pour être comparés et analysés dans un second temps. Finalement, une synthèse des connaissances apprises lors de ces études est écrite.

11.1 Modèles

Quatre modèles seront utilisés pour évaluer les performances des différentes stratégies de générations automatique de tests. L'un de ces modèles est le robot servant d'exemple fil rouge présenté à la section 5.2. Les trois autres modèles sont un gestionnaire de processus informatique, un distributeur de tickets de cinéma et un site de vente de polices d'assurances.

Après avoir décrit les nouveaux modèles, une comparaison de leurs caractéristiques principales sera présentée.

11.1.1 PID

Le modèle PID (Processus IDentification) représente un gestionnaire de processus informatique. Cette modélisation est réalisée au travers de diagrammes de classes, d'objets et d'états-transitions données dans la figure 11.1.

D'après le diagramme de classes 11.1a, le modèle comporte deux classes. Un gestionnaire, nommé Scheduler, qui est chargé de changer les états des processus et des les regrouper au travers d'associations en fonction de leur état. Le gestionnaire est la classe majeure du modèle. Un processus est modélisé par la classe Thread.

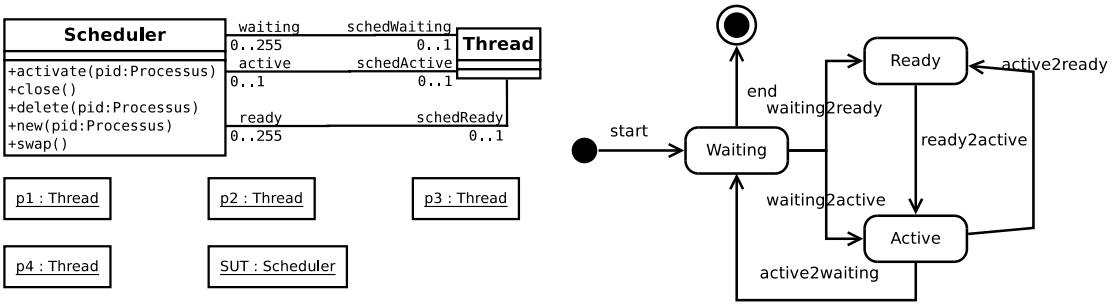
Le diagramme d'objets contient l'ensemble des instances utilisées durant les animations pour respecter les contraintes du langage UML4MBT. Ainsi les animations concerneront un gestionnaire et quatre processus.

Pour décrire les comportements du modèle, le choix a été fait d'employer prioritairement les transitions du diagramme d'états-transitions. Ce diagramme liste les transitions permettant de changer d'état parmi l'attente, l'activation, la disponibilité.

11.1.2 ECinema

Le modèle eCinema représente un système de vente de billets de cinéma au travers d'un système informatique (site web ou borne automatique). Le processus d'achat usuel comporte les étapes suivantes ;

1. Authentification d'un utilisateur.
2. Achat d'un ou plusieurs billets.
3. Visualisation des transactions.
4. Suppression des billets achetés par erreur.



(a) Diagramme de classes et d'objets

(b) Diagramme d'états-transitions

FIGURE 11.1 – Modèle UML4MBT - PID

5. Déconnexion.

La modélisation s'appuie sur un diagramme de classes 11.2 et un diagramme d'objets 11.3.

Ce premier diagramme modélise le système avec quatre classes et cinq énumérations. La classe majeure, nommée eCinema, représente l'interface permettant à un utilisateur d'acheter des billets. La description des fonctionnalités principales de l'application est réalisée au travers des opérations de cette classe. Pour réaliser l'authentification, cette classe dispose d'une liste d'utilisateurs pré-enregistrés. De la même façon, une liste de films est stockée via une association. La dernière classe, nommé Ticket, est le billet de cinéma. Elle permet d'associer un utilisateur à un film.

La présence d'énumération s'explique par la contrainte du langage UML4MBT d'interdire les chaines de caractères. Ainsi ces dernières sont remplacés par des énumérations. Les valeurs des énumérations représentent des cas de test. Par exemple, dans le cas de l'énumération Password, les valeurs ne sont pas directement des mots de passes tels *190285* ou *sophie22* mais les cas à tester : *INVALID_PWD* représente les mots de passes invalides, *REGISTERED_PWD* modélise les mots de passes enregistrés dans le système et *UNREGISTERED_PWD* regroupe les mots de passe absent du système. Ce type de modélisation permet de limiter la taille du modèle et donc le nombre de variables lors de l'animation.

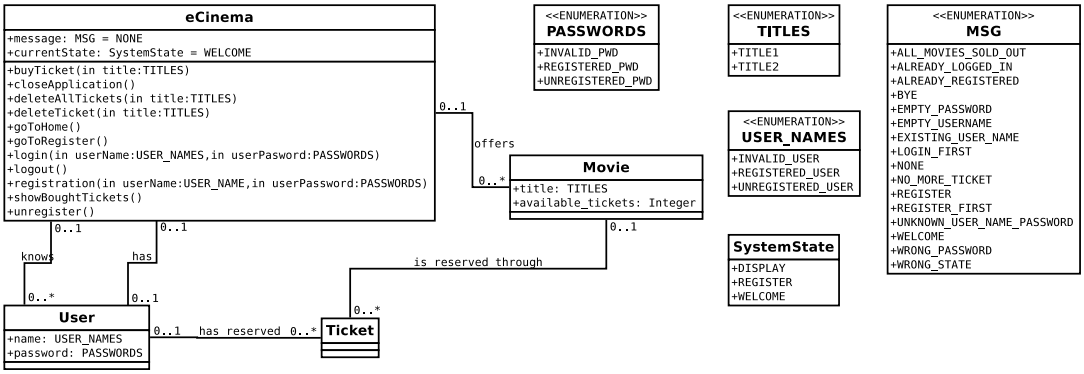


FIGURE 11.2 – Diagramme de classes du modèle eCinema

Durant l'animation, le modèle dispose d'une instance de la classe eCinema, de deux utilisateurs, l'un étant déjà enregistré dans le système et l'autre non, de deux films et d'une dizaine de billets d'après le diagramme d'objets. Initialement les utilisateurs ne sont pas authentifiés sur le système et les billets ne sont pas vendus.

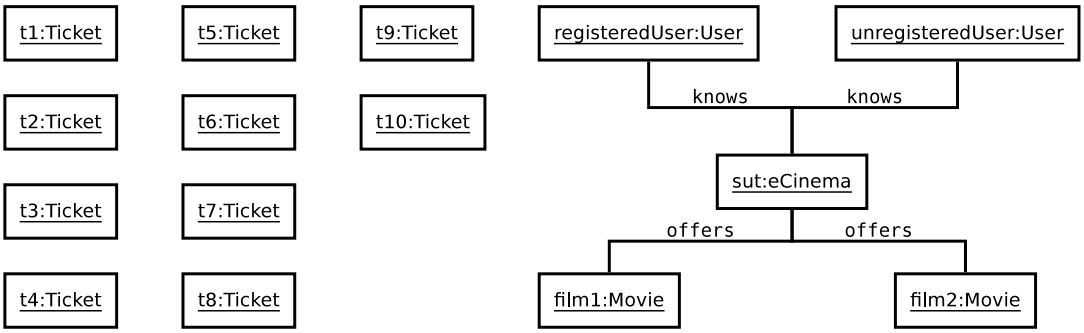


FIGURE 11.3 – Diagramme d'objets du modèle eCinema

11.1.3 ServiDirect

Le modèle ServiDirect représente un site internet de vente de polices d'assurances. Ce modèle a été créé lors du projet TestIndus [1] et correspond à un site réel de la société ServiDirect. Malheureusement ce site n'est plus disponible car la société a arrêté son activité.

La modélisation est réalisée au travers des diagrammes d'états-transitions 11.5, de classes 11.6 et d'objets 11.4.

Le diagramme d'objets décrit un état initial où le processus de souscription n'est pas commencé. Ce diagramme contient une instance de la classe majeure et des instances des différentes classes représentant les informations en mémoire dans le système telles les localisations avec les domiciles et cantons ou l'entourage familial avec les instances des classes enfants et adultes.

Le diagramme d'états-transitions concentre la majorité de la logique de l'application. Ce diagramme décrit le processus de souscription à une police d'assurance. Le processus a les étapes suivantes :

1. Saisie des informations générales décrivant la situation du souscripteur potentiel.
2. Choix de la police d'assurance et de ses options.
3. Saisie d'informations spécifiques en fonction de la police et de la situation du souscripteur.
4. Prise de décision sur l'acceptation du dossier.
5. Appel téléphonique si le dossier est refusé ou création du contrat s'il est accepté.

1. http ://lifc.univ-fcomte.fr/test_indus/

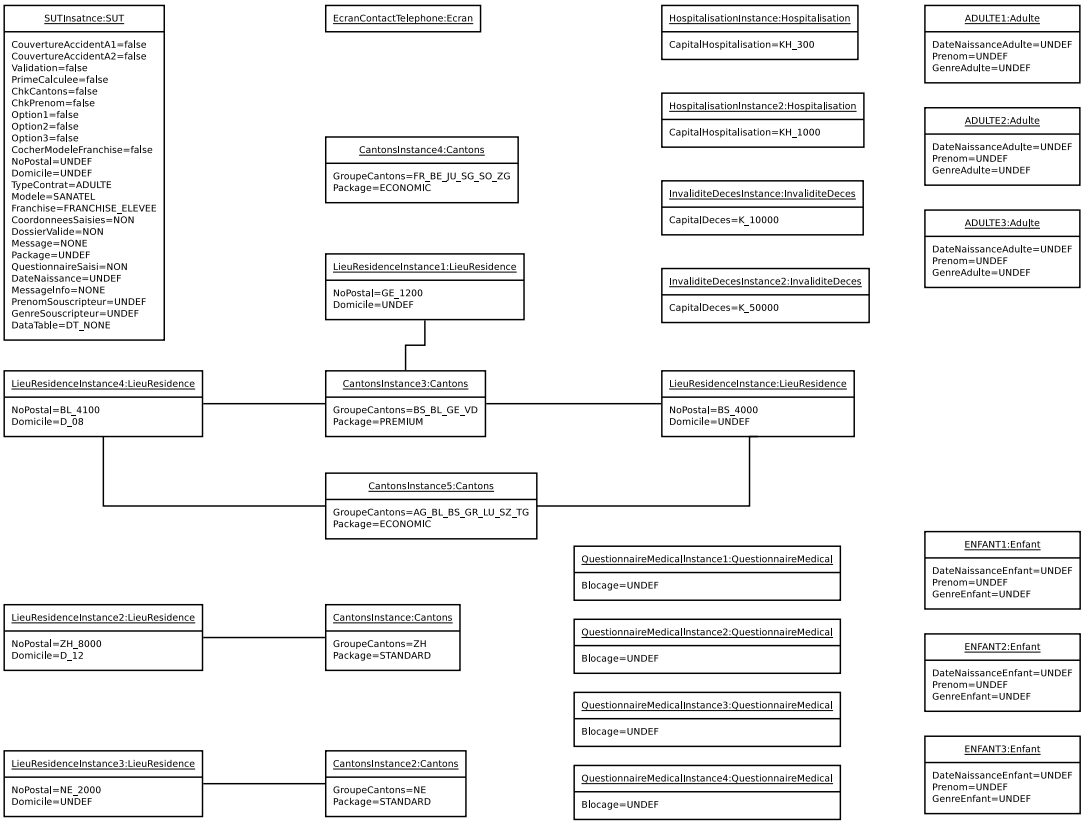


FIGURE 11.4 – Diagramme d'objets du modèle ServiDirect

La structure décrite par le diagramme de classes est similaire à celle du modèle eCinema. La classe majeure, nommée SUT pour System Under Test, correspond à l'interface du système qui est dans ce cas un site internet. Cette classe mémorise l'avancement de la procédure de souscription au travers de ses attributs. Elle mémorise également les informations décrivant directement le souscripteur. Les informations sur la famille sont conservées dans des classes dédiées. Le choix de la police est restreint par la localisation du domicile car chaque canton a une liste de polices disponible.

Les énumérations jouent un rôle similaire à celles du modèle eCinema. En plus, des informations représentées naturellement par des chiffres tel un capital sont également modélisées sous forme d'énumérations. Ceci permet de regrouper les cas de test et de les expliciter. En effet, si le capital entraine une différence en fonction d'un seuil, alors il suffit d'une énumération avec deux valeurs, l'une dessous et l'autre dessus le seuil, pour modéliser les différents comportements.

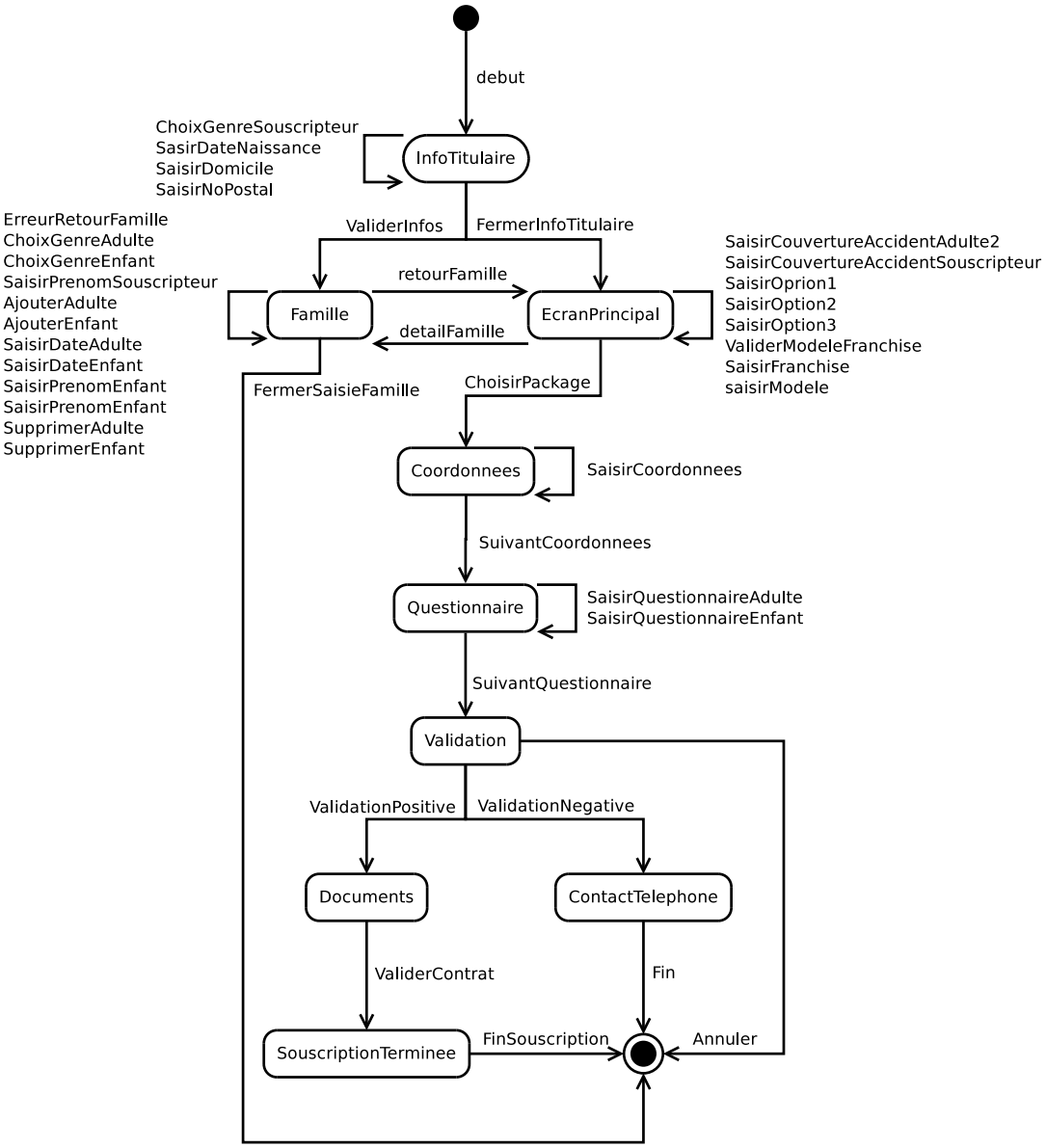


FIGURE 11.5 – Diagramme d'états-transitions du modèle ServiDirect

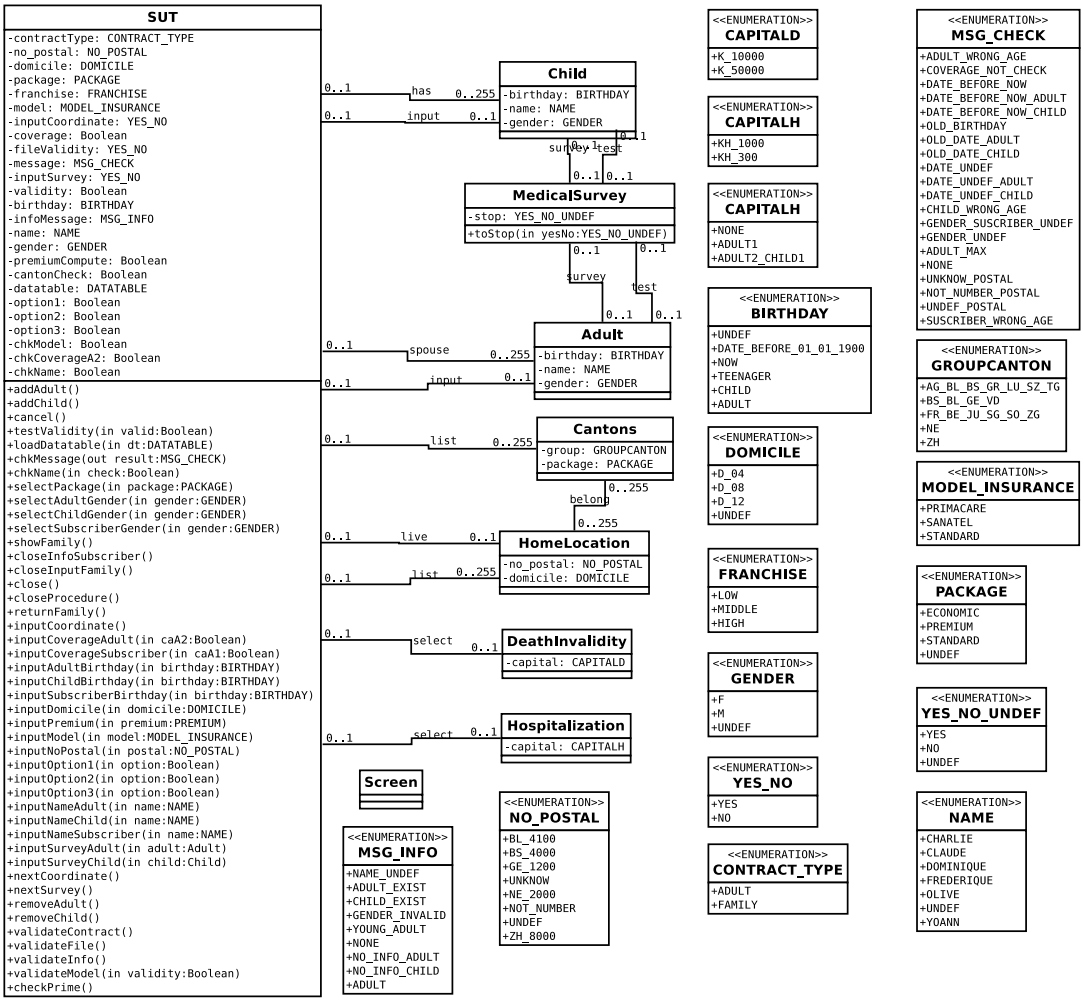


FIGURE 11.6 – Diagramme de classes du modèle ServiDirect

11.1.4 Comparaison

Le tableau 11.1 caractérise les modèles et les scénarios d'animations en découlant. La première partie donne une indication sur la taille des modèles en listant les entités UML4MBT constituant les modèles. Ainsi les quatre modèles sont du plus complexe au plus simple : Servidirect, eCinema, Robot et PID. D'un point de vue des choix de modélisation, deux modèles, PID et Servidirect, utilisent les diagrammes d'états-transitions en conjonction des diagrammes de classes pour décrire les comportements. Les deux autres modèles, eCinema et Robot, utilisent dans ce but uniquement les diagrammes de classes.

La deuxième partie du tableau indique le nombre de cibles de test créées pour couvrir les comportements élémentaires tel que décrit à la section 6.3 pour le modèle du robot. L'ensemble des cibles considérées sont atteignables depuis l'état du modèle décrit dans le diagramme d'objets. Ainsi la présence des diagrammes d'états-transitions accroit naturellement le nombre de cibles puisque certains comportements sont décrits dans les opérations des classes et d'autres dans les transitions.

La dernière partie du tableau donne une indication sur la taille des scénarios d'animations créés depuis ces modèles. Ces scénarios utilisent le motif par défaut, c-à-d que l'ensemble des transitions du système de transitions sont présentes dans les scénarios. Le nombre de contraintes et de fonctions dépend de la longueur des scénarios. Il est calculable par les formules suivantes :

$$NbFonctions = FP * NbPas + FI$$
$$NbContraintes = CP * NbPas + CI$$

Table 11.1 – Caractéristiques des modèles

	ServiDirect	eCinema	Robot	PID
Classes	9	1	3	2
Attributs	38	6	5	1
Enumerations	14	5	3	3
Associations	14	13	3	3
Instances	26	14	4	5
Slots	70	14	6	4
Opérations	45	13	6	4
Cibles de test	87	20	8	18
Contraintes initiales (CI)	79	48	17	15
Contraintes par pas (CP)	936	352	48	62
Fonctions initiales (FI)	469	99	44	50
Fonctions par pas (FP)	1077	219	108	88

11.2 Mesures

Dans cette section, chaque stratégie disponible dans notre méthode de génération de test est appliquée sur les modèles introduits à la section précédente. Les mesures donnent le temps nécessaire pour obtenir l'ensemble des tests en fonction du nombre de threads et du paramétrage de la stratégie.

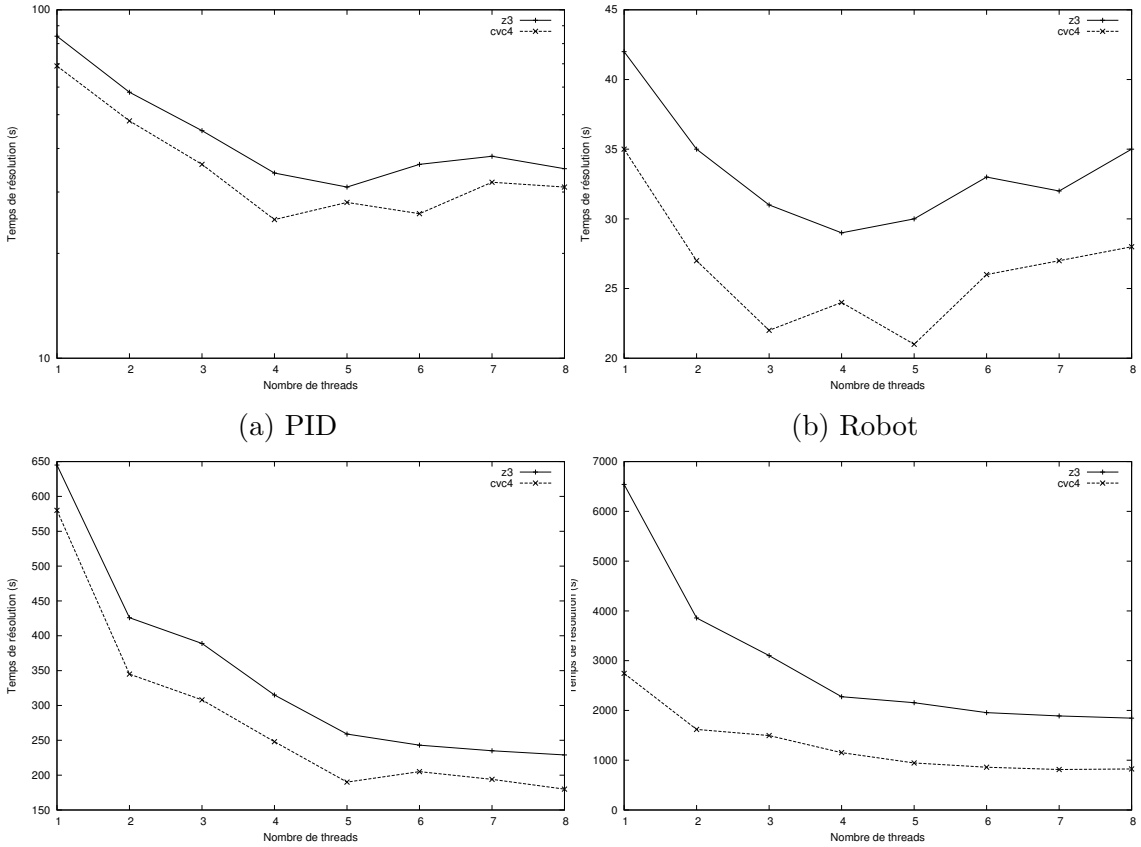
Les mesures sont effectuées sur un ordinateur ayant un processeur 64 bits avec 8 cœurs à 2.20 Ghz et 6 Go de RAM. Parmi les prouveurs SMT utilisés dans les expérimentations du chapitre 9, seul Z3 et CVC4 ont été retenu pour effectuer ces mesures. MathSat 5 a été éliminé car il a des performances largement inférieure dans la logique employée par notre démarche. CVC3 est éliminée car son successeur est disponible et il n'accepte pas le langage SMT-lib v2.0 en entrée à la différence de Z3 et CVC4. Ainsi les scénarios d'animations sont convertis dans le même langage de sortie pour les deux prouveurs SMT retenus.

11.2.1 Stratégie Basic

Les résultats de l'application de la stratégie *Basic*, décrite à la section 7.2, sont donnés au travers des graphiques contenus dans la figure 11.7. Durant cette application, le paramétrage de la stratégie a été adapté à chaque modèle. Ainsi la longueur des scénarios d'animations est de 10 pas pour le modèle Robot, 15 pas pour le modèle PID, 30 pas pour le modèle ECinema et 40 pas pour le modèle ServiDirect. Ces valeurs permettent d'atteindre l'ensemble des cibles de test et elles ont été choisies grâce à notre connaissance des modèles.

Ces résultats décrivent les faits suivants :

- Les performances de CVC4 sont supérieures aux performances de Z3. Ceci est conforme aux attentes entrainées par les expérimentations conduites à la section 10.2. De plus, l'écart de performance s'accroit avec la taille des scénarios d'animations. Ainsi il y a un écart moyen de 23% dans le cas du modèle PID qui a une taille limitée et un écart de 100% pour le modèle industriel ServiDirect.
- Le gain de performance apporté par la parallélisation du processus de génération est divisé en deux selon le nombre de threads. Le gain est extrêmement important jusqu'à 5 threads puis il devient faible voire négatif après 5 threads. Par exemple dans le modèle ServiDirect, le temps est réduit de 65% entre l'utilisation de 1 et 5 threads avec le prouveur CVC4. Par contre, il est réduit uniquement de 14% entre 5 et 8 threads pour le même prouveur. Une explication de la limitation du gain apporté par la parallélisation est le choix d'arrêter un thread lorsque l'objectif de tests du scénario d'animations exécuté est satisfait par un autre scénario exécuté par un thread parallèle.
- La différence de performance entre les modèles PID et Robot qui sont de taille et complexité comparable s'explique par la nécessité d'employer des scénarios plus long pour le PID. En effet ce dernier utilise un diagramme d'états-transitions et l'activation d'une de ces transitions est possible uniquement si l'opération idoine du diagramme de classes a été exécuté précédemment. Par conséquent, la longueur des scénarios s'accroit.



(a) PID (b) Robot

(c) ECinema (d) ServiDirect

FIGURE 11.7 – Temps de génération des tests avec la stratégie Basic en fonction du nombre de threads

11.2.2 Stratégie Step-Increase

Les temps de génération des tests pour les différents modèles en appliquant la stratégie *Step-Increase*, décrite à la section 7.2, à l'aide des prouveurs CVC4 et Z3 sont donnés par les graphiques de la figure 11.8. Pour rappel, cette stratégie est configurable à l'aide de trois paramètres qui sont la longueur initiale des scénarios d'animations notée *init* dans les graphiques, l'incrément de cette longueur noté *inc* et la longueur maximale des scénarios d'animations.

Ce dernier paramètre est constant pour les différentes mesures en fonction du modèle. Ainsi la longueur maximale est de 10 pas pour le modèle Robot, 15 pas pour le modèle PID, 30 pas pour ECinema et 40 pas pour ServiDirect. Ces valeurs correspondent à celles utilisées durant l'évaluation de la stratégie *Basic* dans la sous-section précédente.

Le taux de couverture n'est pas précisé dans les graphiques car l'ensemble des cibles de test sont atteintes. Ce résultat était attendu car ce taux était déjà atteint lors de la dernière stratégie, les longueurs maximums des scénarios sont égales et cette stratégie garantit d'atteindre l'ensemble des cibles si elles sont atteignables par des animations d'une longueur donnée.

Les faits suivants ressortent de ces résultats :

- Les performances de CVC4 sont meilleures que celle de Z3. Cependant cet écart devient quasiment négligeable pour les modèles de taille restreinte, PID et Robot. En effet, les temps de résolutions dans ces modèles ne sont pas quasi équivalent aux temps de génération. Les temps nécessaires pour la conversion, l'écriture et l'analyse des problèmes sont à considérer. De plus, cette stratégie a pour propriété qu'un nombre important de scénarios sont crées et écrits au cours de son exécution. En effet, le processus de conversion du modèle UML4MBT en un modèle SMT4MBT est relancé après chaque accroissement de la longueur des scénarios.
- Pour le prouveur CVC4, une tendance se dégage. Les meilleures performances sont obtenues par ordre décroissant à l'aide de longueurs initiale et incrémentale de 5, 10 et 1 pas. Ce résultat démontre la nécessité de trouver un équilibre entre la longueur des scénarios, minimisé avec une longueur initiale et incrémentale de 1 et maximisé avec une longueur initiale et incrémentale de 10, et le nombre de scénarios, minimisé avec une longueur initiale et incrémentale de 10 et maximisé avec une longueur initiale et incrémentale de 1.
- Pour le prouveur Z3, les meilleures paramètres dépendent de la taille du modèle. Dans le cas de modèles complexes tels ServiDirect et eCinema, l'utilisation d'une longueur initiale et incrémentale de 10 pas conduit à une performance inférieure à une longueur de 1 pas. Ceci résulte que le temps de résolution ne s'accroit pas linéairement avec le nombre de pas comme l'a montré la section 10.2.
- Le gain apporté par la parallélisation se divise en deux parties. Ce gain est important jusqu'à 4-5 threads et faible avec 6 threads et plus.

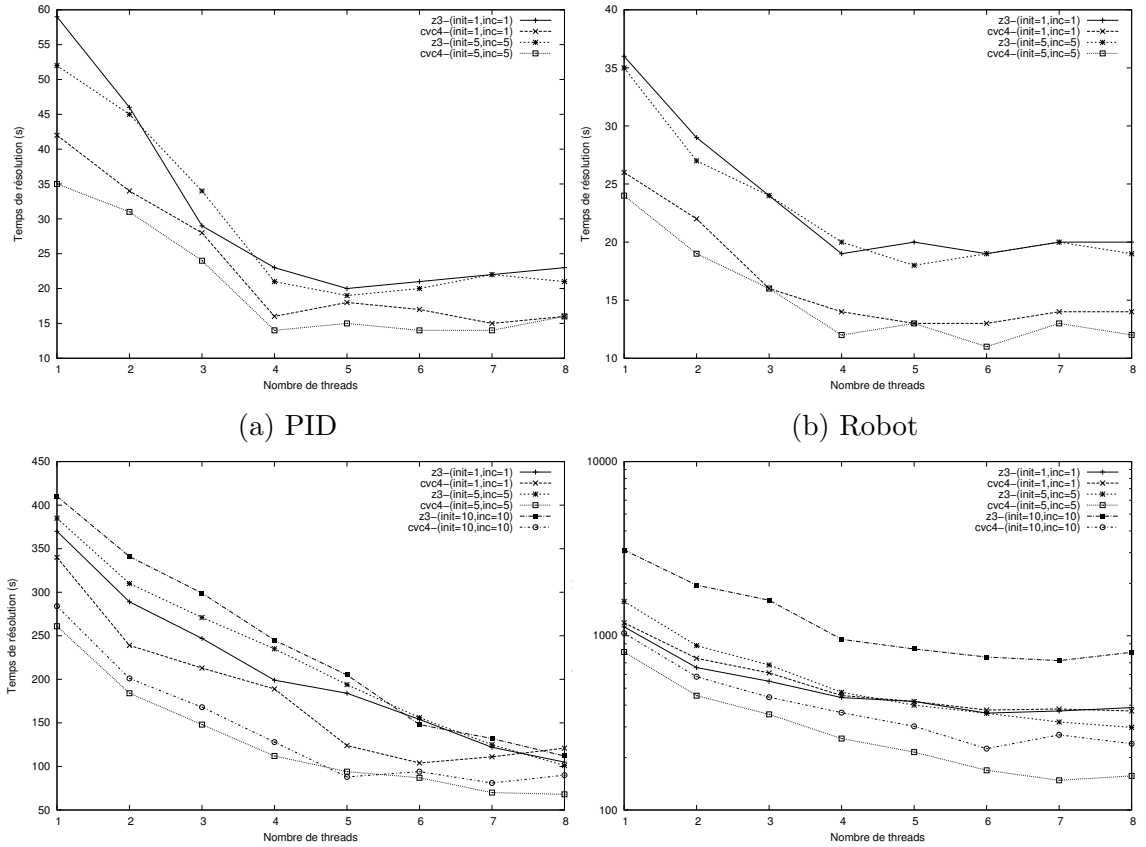
11.2.3 Stratégie Space-search : depth

Les temps de génération des tests pour les différents modèles en appliquant la stratégie *Space-search-depth* , décrite à la section 7.2, à l'aide des prouveurs CVC4 et Z3 sont donnés par les graphiques de la figure 11.8. Pour rappel, cette stratégie est configurable à l'aide de deux paramètres qui sont la longueur des scénarios d'animations notée dans les légendes des graphiques et le nombre maximum de scénarios successifs noté entre parenthèse. Deux scénarios sont successifs si l'état initial du second scénario est l'état final du premier.

Ce dernier paramètre est choisi pour obtenir une longueur maximale pour les scénarios au moins équivalente à celle utilisée lors des mesures précédentes. Ainsi, ce paramètre vaut 10 pour des scénarios de longueurs 4 dans le cas du modèle ServiDirect afin d'obtenir une longueur maximale de $4 * 10 = 40$ pas.

Les faits suivants sont déductibles des mesures :

- La prédominance du prouveur CVC4 sur Z3 est conservé sur les modèles de tailles conséquentes, ECinema et ServiDirect.
- Les gains de la parallélisation respecte également le schéma usuel avec une séparation autour de 4-5 threads. Avant cette limite, le gain est important, après il est faible ou négatif.



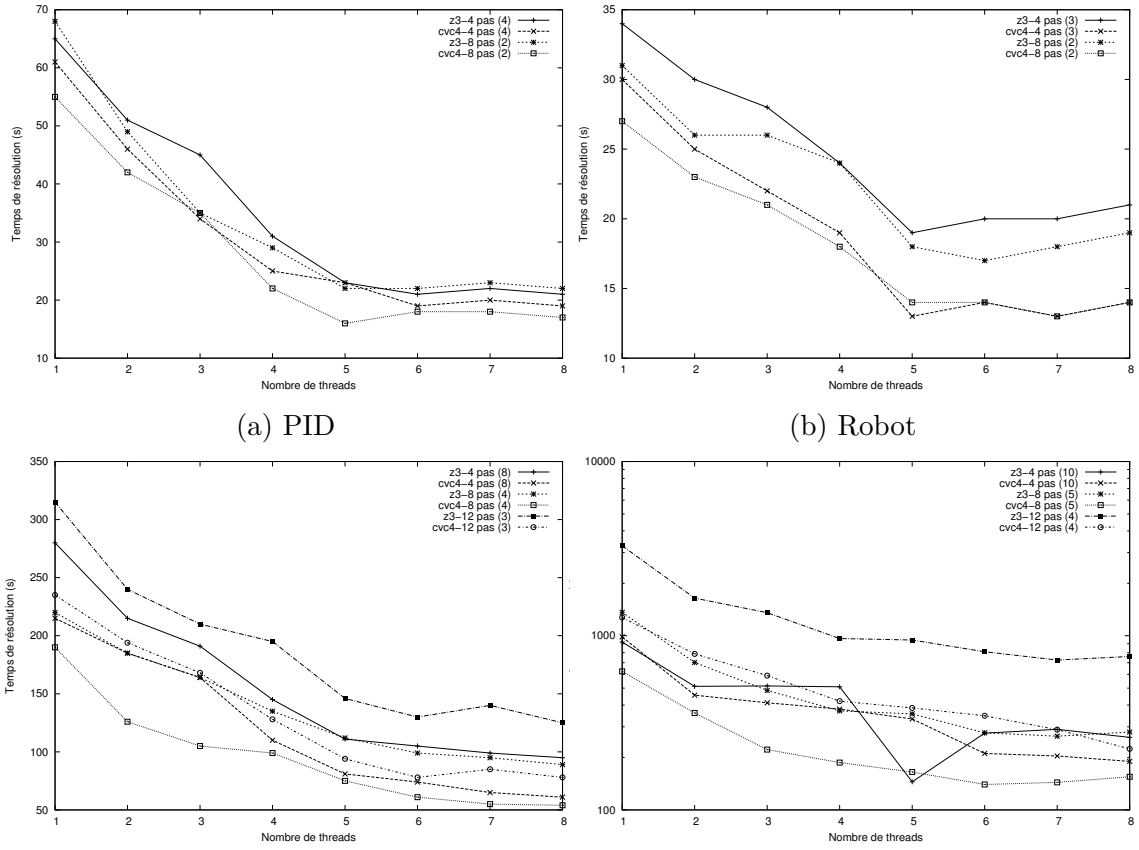
(a) PID (b) Robot

(c) ECinema (d) ServiDirect

FIGURE 11.8 – Temps de génération des tests avec la stratégie Step-Increase en fonction du nombre de threads

– Pour le prouveur CVC4, les meilleures configurations par ordre décroissant de la stratégie en terme de temps de génération des tests utilisent des scénarios de longueur 8, 4 et 12 pas. Pour expliquer ce résultat, nous pouvons considérer un test nécessitant une animation de 40 pas sur le modèle ServiDirect. Le fractionnement de cette dernière durant la stratégie conduit à exécuter une séquence de 10, (resp. 5 ou 4) animations de longueur 4 (reps. 8 ou 10) pas. Cependant l'exploration de l'espace de recherche au cours de la stratégie ne garantit pas d'obtenir immédiatement la séquence désirée. Par conséquent, le nombre de scénarios d'animations sera généralement supérieur à la longueur de la séquence. Ainsi le temps de résolution gagné par l'obtention de scénarios plus courts est contrebalancé par l'augmentation du nombre de scénarios.
– Les performances obtenues sur les modèles ServiDirect et ECinema sont à mettre en concordance avec l'adéquation des particularités des systèmes modélisés à la stratégie. Les comportements de ces systèmes correspondent à la navigation dans des formulaires. La majorité des opérations se divisent entre la saisie d'information, leur validation et la navigation entre les formulaires. Ce type de système est particulièrement adapté à la stratégie *Space-search-depth*

car elle privilégie des cibles de test ayant des liens forts.



(a) PID (b) Robot

(c) ECinema (d) ServiDirect

FIGURE 11.9 – Temps de génération des tests avec la stratégie depth en fonction du nombre de threads

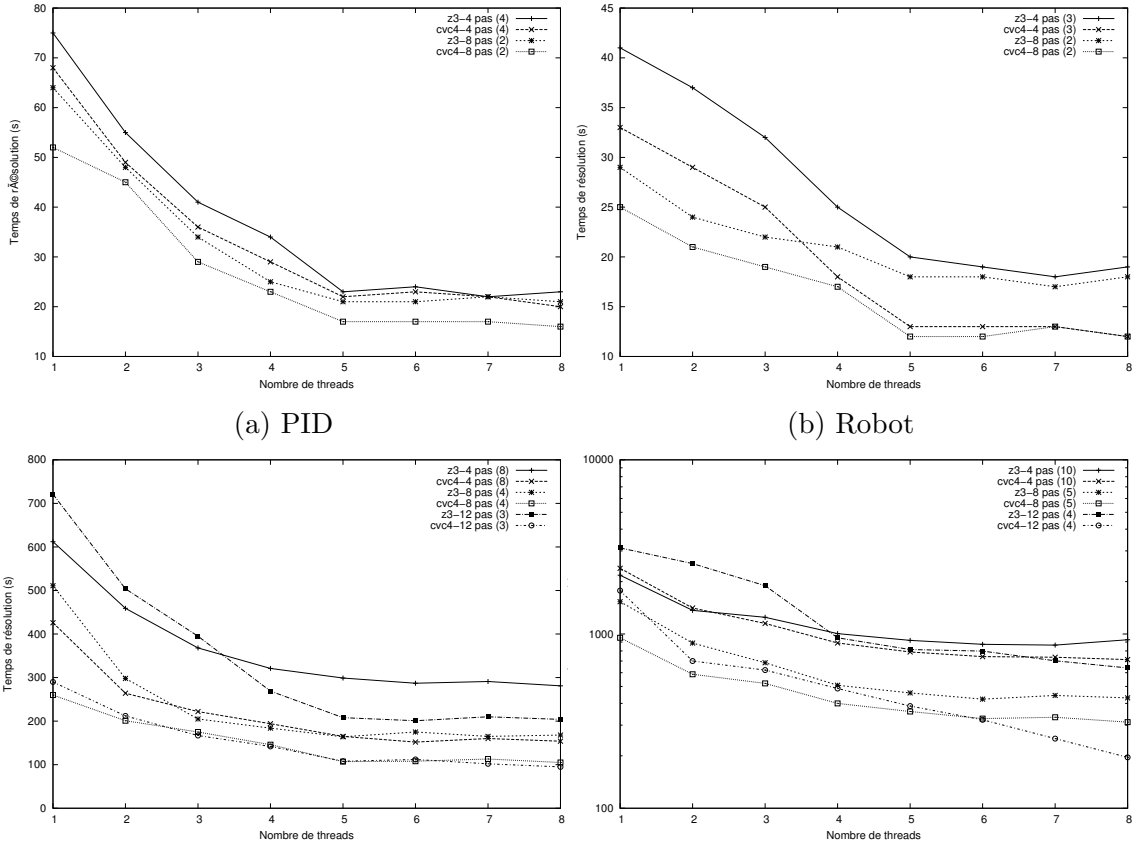
11.2.4 Stratégie Space-search : breadth

Les temps de génération des tests pour les différents modèles en appliquant la stratégie *Space-search breadth*, décrite à la section 7.2, à l'aide des prouveurs CVC4 et Z3 sont donnés par les graphiques de la figure 11.10. Cette stratégie est une variante de la précédente, différant seulement par l'heuristique d'exploration de l'espace de recherche durant la génération des tests. Par conséquent, la stratégie est configurée par les mêmes paramètres. Ainsi les valeurs de ces paramètres sont conservées pour ces mesures.

Les faits suivants sont observables dans les mesures :

- La prédominance du prouveur CVC4 sur Z3 est conservée sur les modèles de tailles conséquentes, ECinema et ServiDirect.
- Les gains de la parallélisation respecte également le schéma usuel avec une séparation autour de 4-5 threads. Avant cette limite, le gain est important, après il est faible ou négatif.

– Les meilleures performances par ordre décroissant pour le prouveur CVC4 sont obtenues avec des scénarios de longueurs 12, 8 et 4 pas. Pour le prouveur Z3 l'ordre s'établit sur des scénarios de longueurs 8, 4 et 12 pas pour optimiser le temps de génération. Cette différence s'explique par deux points. Le premier est l'influence supérieure de la longueur de l'animation sur les performances dans le cas de Z3. Le second point est que pour Z3, la résolution de scénarios d'animations lorsqu'ils conduisent a des animations invalides est plus rapide que les scénarios valides. Ainsi Z3 utilise des scénarios de longueurs inférieurs à CVC4. Ceci entraine une augmentation du nombre de scénarios mais cependant une majorité de ces scénarios conduit à des animations invalides et donc à des temps de résolutions inférieurs. La validité des scénarios ayant peu d'influence sur les performances de CVC4, ce prouveur va privilégier des scénarios de longueurs supérieures qui conduisent à une probabilité plus élevée d'animations valides.



(a) PID (b) Robot

(c) ECinema (d) ServiDirect

FIGURE 11.10 – Temps de génération des tests avec la stratégie Breadth en fonction du nombre de threads

11.2.5 Stratégie Path

Les temps de génération des tests pour les différents modèles en appliquant la stratégie *Path*, décrite à la section 7.2, à l'aide des prouveurs CVC4 et Z3 sont donnés par les graphiques de la figure 11.11. Pour rappel, cette stratégie est configurable à l'aide d'un unique paramètre qui est la longueur maximale des scénarios d'animations. Cette longueur est de 10 pas pour le modèle Robot, 15 pas pour le modèle PID, 30 pas pour ECinema et 40 pas pour ServiDirect. Ces valeurs correspondent à celles utilisées durant l'évaluation de la stratégie *Basic* et *Step-increase*.

D'après ces mesures, les observations suivantes sont possibles :

- L'écart de performance entre les prouveurs Z3 et CVC4 est extrêmement faible. Sur le modèle PID, Z3 arrive dans certains cas à obtenir de meilleurs temps de générations. Une hypothèse expliquant ces mesures est la particularité des scénarios d'animations lors de cette stratégie. En effet, toutes les autres stratégies conduisent à cause de la duplication des variables par le processus d'encodage à obtenir des contraintes qui ne différent que par l'indice des stepVariables. Hors cette structure particulière est exploitée par CVC4 lors d'une phase de traitement dédiée à la ré-écriture des contraintes. Ainsi l'impossibilité de ré-écrire aussi efficacement les contraintes dans le cas de la stratégie *Path* pourrait expliquer la diminution de l'écart de performance entre les prouveurs.
- Un autre fait mis en évidence par ces mesures est que l'efficacité de la stratégie croît avec la taille du modèle. En effet cette stratégie entraîne une multiplication du nombre de scénarios. Par conséquent, le temps de création de ces scénarios doit être contrebalancé par un temps de résolution plus rapide. Hors le gain augmente avec le nombre de transitions présentes dans le système. Ainsi, les modèles ServiDirect et eCinema, ayant plus de transitions que les modèles PID et Robot sont plus adaptés à cette stratégie.

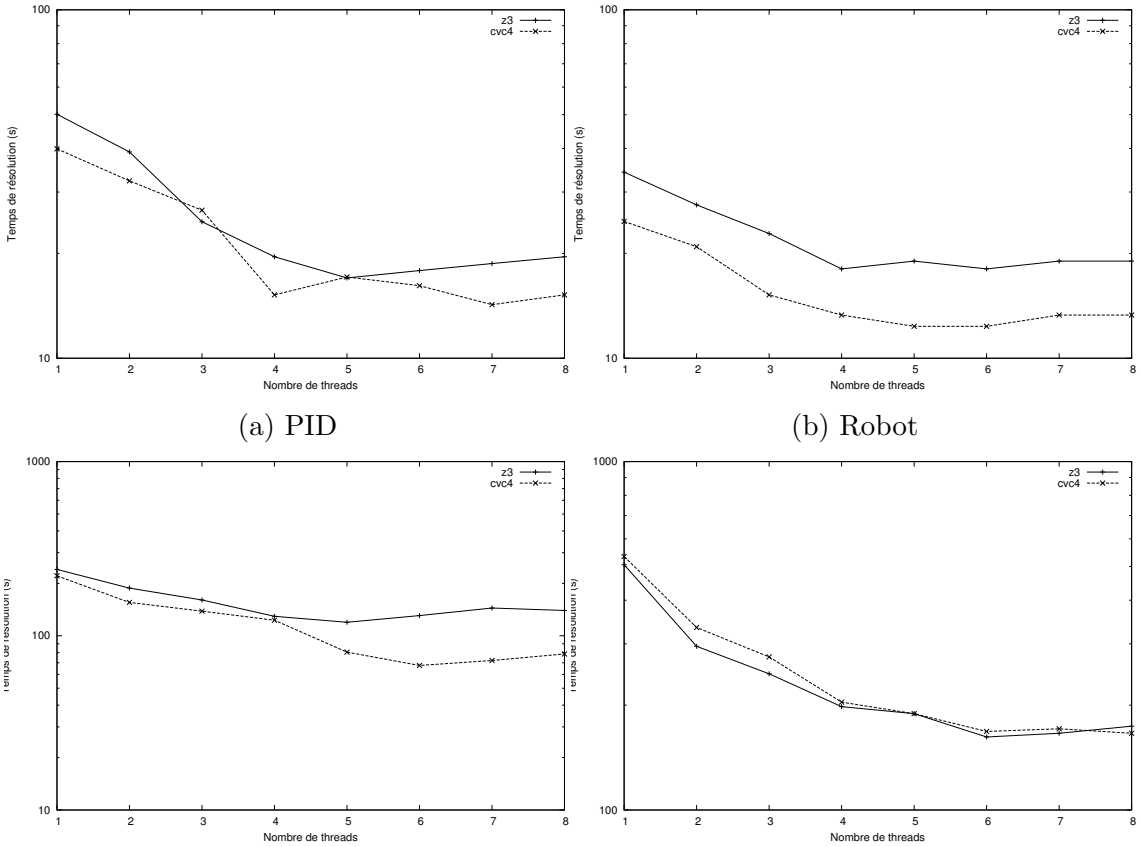
11.2.6 Stratégie Step-increase collaborative

Les temps de génération des tests pour les différents modèles en appliquant la stratégie *Step-Increase collaborative*, décrite à la section 7.2, à l'aide des prouveurs CVC4 et Z3 sont donnés par les graphiques de la figure 11.12. Pour rappel cette stratégie est construite sur la stratégie *Step-Increase*. Ainsi les paramètres de configuration (longueur initiale notée *init*, longueur incrémentale notée noté *init*, longueur maximale) utilisés pour l'évaluation sont les mêmes que ceux employés pour l'évaluation de la stratégie *Step-Increase*.

Pour résoudre les scénarios décrits à l'aide de modèles CSP4MBT, le solveur Gecode, présenté à la section 4.2, a été choisi. En effet, il a les meilleures performances et offre une excellente intégration du langage MiniZinc.

Les mesures contenues dans les graphiques montrent que :

- L'ajout d'une phase de résolution d'une instance CSP4MBT n'influe pas de manière notable sur le gain apporté par la parallélisation. Ceci s'explique par un temps négligeable en général par rapport au temps de résolution d'une instance SMT4MBT. En effet, dans cette stratégie le solveur CSP exécute



(a) PID

(b) Robot

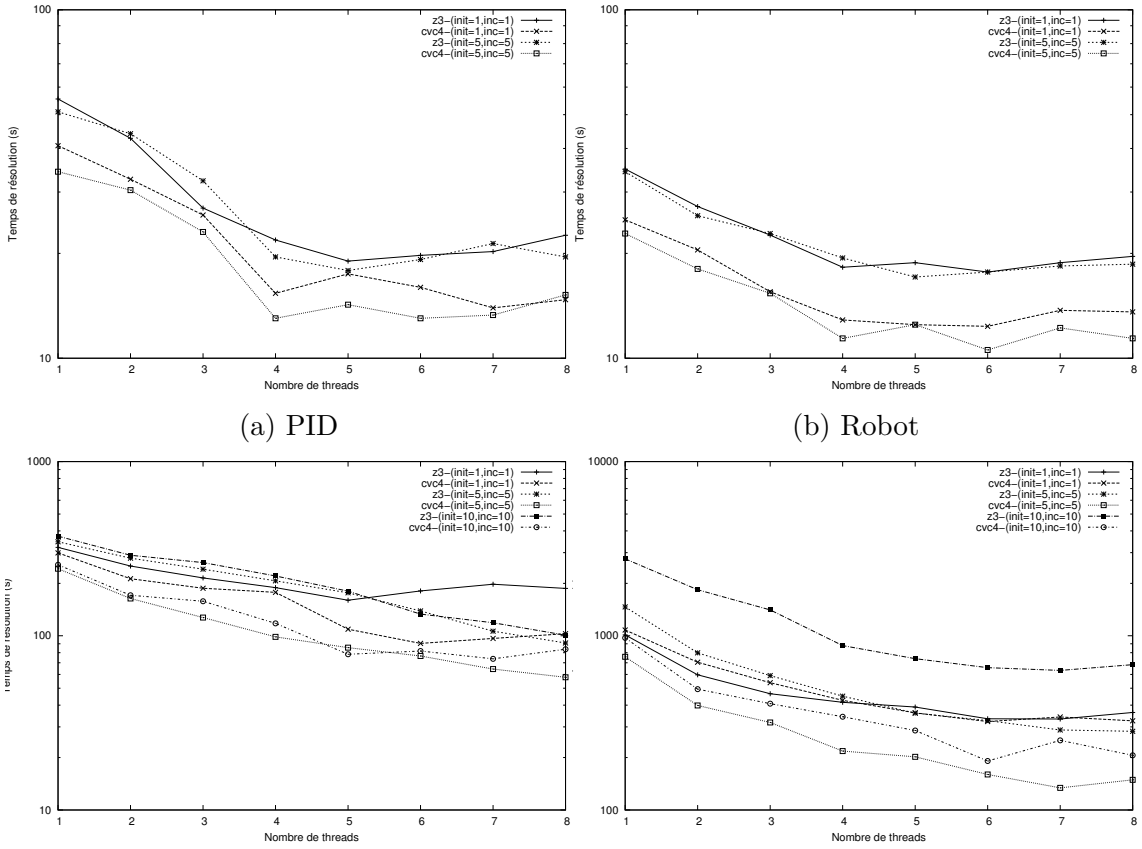
(c) ECinema

(d) ServiDirect

FIGURE 11.11 – Temps de génération des tests avec la stratégie Path en fonction du nombre de threads

uniquement des séquences de transitions découvertes précédemment par le prouveur. Par conséquent, l'espace de recherche est extrêmement réduit.

– L'écart de performance entre Z3 et CVC4 demeure mais l'introduction du solveur conduit à accroitre les écarts par rapport à la tendance établie par l'introduction de la parallélisation.
– Le gain apporté par la collaboration est supérieur pour les modèles de tailles conséquentes comme ServiDirect et eCinema. Par rapport à l'application de la stratégie *Step-Increase*, le gain est en moyenne de 10 pour cent. Par contre, les modèles de tailles inférieures comme PID et Robot conduisent à des gains moyens de 4 pour cent. Une des raisons expliquant ce comportement est que pour obtenir des gains, la sequence rejouée par le solveur doit contenir un maximum de point de choix. Par exemple, les paramètres des opérations doivent avoir un choix important de valeurs. Une autre raison est que le temps de création et de résolution des instances CSP4MBT est proportionnellement plus important sur les modèles de petites tailles.



(a) PID

(b) Robot

(c) ECinema

(d) ServiDirect

FIGURE 11.12 – Temps de génération des tests avec la stratégie Step-increase collaborative en fonction du nombre de threads

11.2.7 Stratégie Space-search collaborative

Les temps de génération des tests pour les différents modèles en appliquant la stratégie *Space-search collaborative*, décrite à la section 7.2, à l'aide des prouveurs CVC4 et Z3 sont donnés par les graphiques de la figure 11.13. Cette stratégie est construite autour de la stratégie *Space-search*. Par conséquent, elle se décline également en deux variantes, *depth* and *breadth* selon l'algorithme d'exploration choisi. Lors de ces mesures, l'exploration utilise un algorithme *depth* car cette variante a obtenue de meilleurs résultats.

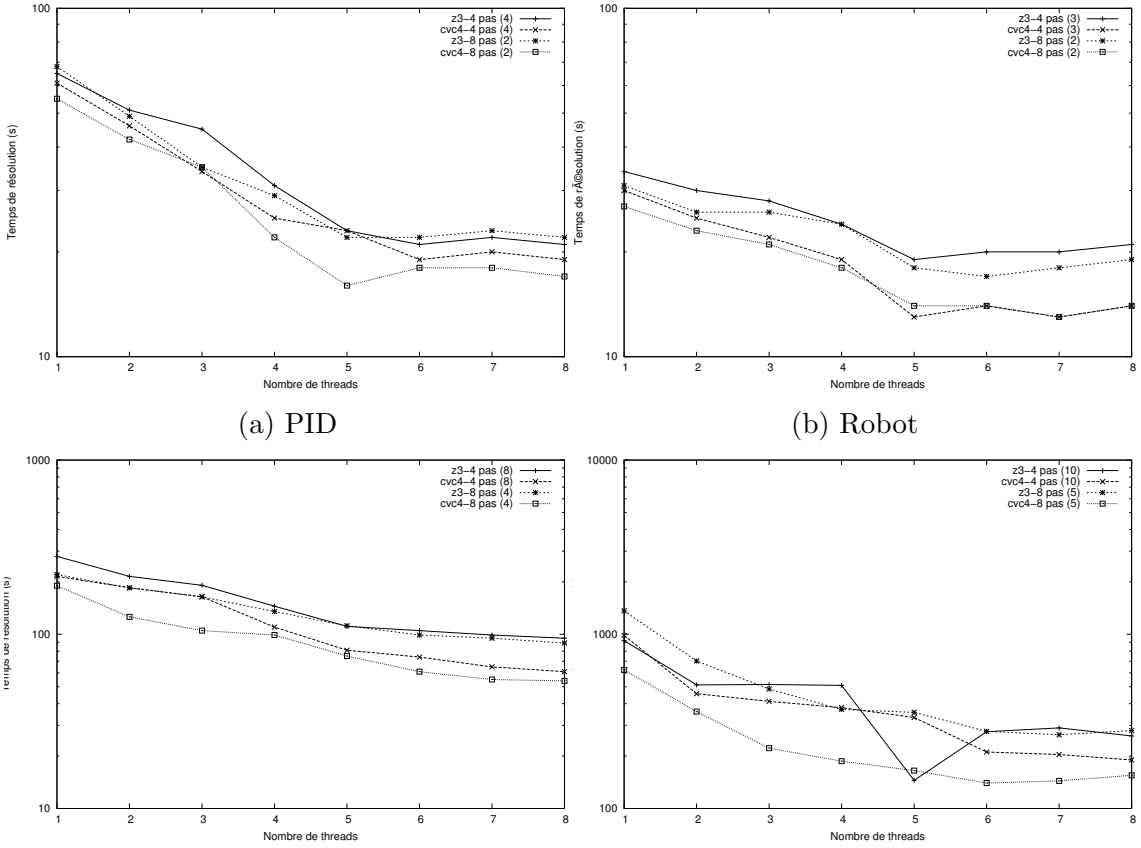
La résolution des instances CSP4MBT est confiée au solveur Gecode comme lors de l'évaluation de la stratégie précédente. Il a été configuré pour retourner, si possible, deux solutions à chaque instance.

Les autres paramètres de configurations, la longueur des scénarios d'animations notée en pas et le nombre maximum d'animations successives précisée dans la figure par un nombre entre parenthèse sont ceux appliqués lors de l'évaluation de la stratégie *Space-search : depth*.

Les mesures présentées dans les graphiques montrent que :

– L'écart de performance demeure entre CVC4 et Z3. CVC4 continue d'obtenir

de meilleures performances.
- Le gain de parallélisation respecte également le schéma usuel en apportant un gain important jusqu'à 5 threads et un gain faible après.
- L'intérêt de cette stratégie par rapport à la stratégie *Space-search : depth* n'est pas concluant sur les modèles évalués. En effet, un des intérêts majeurs est d'augmenter l'espace de recherche pour accroître la possibilité d'atteindre les cibles de test. Cependant toutes les cibles étant déjà atteintes avec la version non collaborative de la stratégie, cette stratégie n'est pas adaptée.



(a) PID (b) Robot

(c) ECinema (d) ServiDirect

FIGURE 11.13 – Temps de génération des tests avec la stratégie Space-search collaborative en fonction du nombre de threads

11.3 Analyse

L'analyse comparative des évaluations des différentes stratégies apportent des informations sur les prouveurs, la parallélisation et le choix des stratégies. Pour faciliter cette analyse, le tableau 11.2 regroupe le meilleur temps de génération des tests de chacune des stratégies en fonction du modèle et de l'utilisation de la parallélisation.

TABLE 11.2 – Les meilleurs temps de génération des tests en seconde en fonction de la stratégie appliquée et de l'utilisation de la parallélisation.

Parallélisé	Stratégie	ServiDirect	eCinema	Robot	PID
Non	Basic	2745	580	69	35
	Step-Increase	807	261	35	24
	Space-Search :depth	624	190	55	27
	Space-Search :breadth	952	260	52	25
	Path	506	221	40	23
	Step-Increase collaboratif	694	221	40	23
	Space-Search collaboratif	648	207	57	29
Oui	Basic	814	180	31	28
	Step-Increase	148	68	14	11
	Space-Search :depth	155	54	17	13
	Space-Search :breadth	196	95	16	12
	Path	162	67	14	12
	Step-Increase collaboratif	133	61	13	11
	Space-Search collaboratif	149	56	18	13

Prouveurs Les performances du prouveur CVC4 ont été nettement supérieures à celles du prouveur Z3 pour l'ensemble des stratégies et des modèles à l'exception de la stratégie *Path*. Comme expliqué lors de la présentation des résultats de cette stratégie, cet écart est sans doute explicable par la spécificité des scénarios d'animations dans ce cas. Les scénarios ne contiennent pas de contraintes dupliquées pour chaque pas à cause de l'utilisation de motifs d'animations. Ainsi la phase de prétraitement exécutée par CVC4 est moins efficace. En effet, dans les autres stratégies, ce prouveur est capable de factoriser les contraintes dupliquées.

L'écart de performance est évidemment plus notable lorsque le temps de résolution des instances SMT4MBT devient prépondérant par rapport aux temps de création et d'écriture de ces instances. Cet écart augmente lors de la résolution de problèmes plus complexes. Par conséquent, l'écart est plus important lors de la résolution de scénarios d'animations de longueur conséquentes portant sur des modèles complexes. Les mesures confirment cette hypothèse. Les temps de générations des tests avec CVC4 pour la stratégie *Basic* appliquée au modèle ServiDirect prend en moyenne 45 % du temps de génération avec Z3. Ce cas considère le modèle le plus complexe avec la stratégie employant les scénarios avec les longueurs les plus grandes. A l'inverse, les mesures sur le modèle le plus simple, le robot, avec la stratégie ayant les scénarios avec les longueurs les plus courtes, *Space-Search : depth*, indiquent que les temps de générations avec CVC4 nécessitent en moyenne 75 % des temps de génération avec Z3.

Parallélisation La parallélisation de notre démarche de génération de tests a permis d'obtenir des gains conséquents en terme de temps. Le gain maximum pour les stratégies appliquées au modèle ServiDirect est d'après le tableau 11.2 de 70 % pour

Basic, 82 % pour *Step-Increase*, 75 % pour *Space-Search : depth*, 80 % pour *Space-Search : breadth*, 68 % pour *Path*, 81 % pour *Step-Increase collaboratif* et 77 % pour *Space-Search collaboratif*. Ainsi sur le modèle le plus complexe, le gain maximum est d'au moins 68 % pour l'ensemble des stratégies.

Le gain apporté par la parallélisation dépend du nombre de threads utilisés. Cependant ce gain n'est pas linéaire. Il est important entre 2 et 5 threads et faible voire négatif au delà. Ainsi sur le modèle Servidirect, le temps de génération moyen pour l'ensemble des stratégies est divisé par un facteur 3,4 entre l'utilisation de 1 et 5 threads. Par contre, ce facteur n'est plus que de 1,22 entre l'utilisation de 5 et 8 threads. La raison expliquant ce comportement est que la résolution d'un scénario d'animations peut devenir obsolète si son objectif de tests est remplit par un autre scénario. Ainsi l'augmentation du nombre de threads augmente la probabilité de rendre obsolète un scénario et donc d'interrompre sa résolution.

Choix d'une stratégie D'après les mesures présentées par le tableau 11.2, le critère principal de sélection d'une stratégie sans l'utilisation de la parallélisation est la taille et la complexité du modèle. Le modèle eCinema correspond à une taille pivot. En dessous, les meilleures stratégies en terme de temps de génération sont par ordre décroissant : *Path*, *Step-Increase*, *Space-Search :breadth* et *Space-Search :depth*. Quand la taille et la complexité augmentent, les performances des stratégies *Space-Search :breadth* et *Step-Increase* se dégradent. A contrario, les stratégies *Space-Search :depth* et *Path* ont de meilleurs résultats. La seule constante vérifiée sur l'ensemble des modèles est que la stratégie *Basic* a systématiquement les temps de génération les plus élevés.

La collaboration entre le solveur et le prouveur a conduit à des résultats mitigés. La stratégie *Step-Increase collaboratif* a des temps inférieurs de 10 % en moyenne à ceux de la stratégie *Step-Increase* sur l'ensemble des modèles. Par contre, la stratégie *Space-Search collaboratif* a des temps supérieurs à la stratégie *Space-Search :depth*. Cette stratégie doit être réservée au cas où la stratégie *Space-Search* n'atteint pas une couverture jugée suffisante puisqu'elle a n'apporte pas de gain en terme de temps de génération.

L'utilisation de la parallélisation dans notre démarche réduit l'importance du choix d'une stratégie car les temps de générations des tests sont plus rapprochés. En effet à part pour la stratégie *Basic* sur l'ensemble des modèles et pour la stratégie *Space-Search :breadth* sur les modèles conséquents, la parallélisation conduit à resserrer les temps et à modifier l'ordre de performance des stratégies.

11.4 Synthèse

Les sept stratégies présentes dans notre démarche de génération automatique de tests (*Basic, Step-Increase, Space-Search depth, Space-Search breadth, Path, Step-Increase collaboratif, Space-search-collaboratif*) ont été évaluées durant ce chapitre sur quatre modèles. Ces derniers sont représentatifs en terme de choix de modélisation avec la présence ou l'absence du diagramme d'états-transitions, de complexité

et de taille.

Un premier fait ressortant de ces évaluations est l'importance de la parallélisation sur le temps de génération. L'emploi de processus multiples a permis de diviser le temps par 4 ou 5 entre les modèles. Cependant le gain ne croît pas linéairement avec le nombre de processus. Les gains sont importants jusqu'à l'utilisation de 4 à 5 processus. Au-delà le gain est faible voire négatif. Cette limitation est due au fait que la résolution d'un scénario d'animations peut satisfaire l'objectif de tests d'un autre scénario. Dans ce cas, le processus gérant ce dernier est interrompu. Ainsi un nombre important d'interruption explique ces mauvaises performances.

Une deuxième constatation est l'importance de la configuration de la stratégie pour l'adapter aux caractéristique du modèle et des cibles de test. Ainsi un paramétrage adapté peut conduire à des gains supérieurs à 100% pour une même stratégie. De plus ce paramétrage varie en fonction du prouveur SMT employé.

Une troisième constatation est la différence importante de performance entre les stratégies en fonction des modèles. En particulier cette différence s'accroît quand la génération de tests n'est pas parallélisée. L'objectif est de déterminer un équilibre entre le nombre de scénarios d'animations employés, leurs longueurs, leurs tailles en terme de contraintes et de fonctions et le nombre de conversions entre les différents modèles. Ainsi même si notre démarche est possible sans connaissance sur le système, un mauvais choix de stratégie et de paramétrage entraîne des performances dégradées.

En conclusion, aucune stratégie ne doit être écartée a priori car elles permettent de répondre à des modèles et des cibles de test ayant des caractéristiques différentes. Ce choix peux être guidé par les critères fournis aux sections 7.3 et 11.3, une génération précédente et la connaissance du modèle apportée par l'ingénieur de tests.

Quatrième partie

Conclusion et perspectives

Chapitre 12

Conclusion

Sommaire

Les travaux de cette thèse ont porté sur la création d'une démarche de génération automatique de tests à partir de modèles écrits dans les langages UML4MBT et OCL4MBT, dérivés d'UML et d'OCL, à l'aide d'un prouveur SMT et d'un solveur CSP. La réalisation de cette démarche s'est effectuée au travers de trois étapes : la création d'un test à l'aide d'un prouveur SMT, la création d'un ensemble de tests via une stratégie et la création de tests grâce à une collaboration entre le solveur et le prouveur. Une implémentation de cette démarche a été réalisée au sein de la plate-forme Hydra afin d'évaluer les performances des différentes stratégies.

12.1 Méta-modèles et encodage

Les modèles employés dans notre démarche sont écrits dans des langages adaptés aux tests à partir de modèles nommés UML4MBT et OCL4MBT. La modélisation est réalisée au travers de trois diagrammes. Le diagramme de classes décrit la structure du modèle. Le diagramme d'objets capture l'état initial du système. Le diagramme d'états-transitions qui est optionnel, renseigne sur le comportement du modèle. Ainsi l'évolution du système est précisée aux travers des opérations des classes et des transitions du diagramme d'états-transitions. La caractéristique principale de ce langage est de définir un nombre fini d'états du système.

Pour représenter le modèle et son évolution, notre démarche le considère comme un système de transitions où l'état initial est défini par le diagramme d'objets et les transitions correspondent aux opérations du diagramme de classes et/ou aux transitions du diagramme d'états-transitions. Ainsi un test est une séquence de transitions

permettant d'atteindre une cible de test. La création de cette dernière est un processus externe à notre démarche. Une cible peut décrire un scénario, un état du système ou un comportement.

L'utilisation d'un prouveur SMT pour la génération conduit à représenter sous la forme d'une formule logique du premier ordre le modèle, son évolution et les cibles de test. Dans ce but, notre démarche a défini un méta-modèle nommé SMT4MBT (SMT for Model-Based Testing). Ce dernier a pour objectif d'associer les notions liées aux tests à partir de modèles telles les cibles de test ou les variables d'états avec le langage standard d'interaction nommé SMT-lib avec les prouveurs. Ainsi le méta-modèle ajoute une sémantique aux fonctions et aux formules décrivant un problème dans le cadre de la génération de tests. Les formules contenues dans ce méta-modèle entrent dans le cadre de la logique linéaire sur les entiers avec des fonctions non-interprétées et sans quantificateur.

Le processus de conversion entre le système de transitions représentant le modèle UML4MBT et un modèle SMT4MBT repose sur une duplication des variables d'états après l'exécution d'une transition. En conséquence, les performances de la démarche sont fonction de la longueur des séquences d'animations. Ce processus aboutit à la création de scénarios d'animations.

Un scénario d'animation regroupe l'ensemble des informations pour générer un cas de test. Un scénario est défini par un objectif de tests qui est une formule logique précisant les cibles de test à atteindre durant l'animation, un état initial, une longueur et un motif d'animations qui restreint les transitions autorisées durant l'animation. La résolution d'un scénario par un prouveur ou un solveur conduit à la création d'un cas de test si l'outil détermine l'existence d'une solution.

12.2 Stratégies

Cette première procédure, création d'un système de transitions suivi de la génération de scénarios d'animations grâce à la conversion dans un modèle SMT4MBT, permet de générer des cas de test. Cependant la génération de multiple cas de test n'est pas considérée dans cette procédure. En effet, la recherche de plusieurs cibles de test sur un même modèle conduit naturellement à sélectionner une stratégie de génération. Cette dernière est responsable du choix des scénarios d'animations et de leur ordonnancement.

Afin d'optimiser le temps de génération des tests, quatre stratégies exploitant les caractéristiques des prouveurs, des solveurs, des modèles, des cibles de test et des scénarios ont été créées. La stratégie *Basic* minimise le nombre de scénarios à exécuter mais nécessite des séquences de transitions de longueur importante. La stratégie *Step-increase* a pour objectif de minimiser la longueur des scénarios car la longueur a l'impact le plus fort sur le temps de résolution d'un scénario par un prouveur. La stratégie *Space-Search* fractionne les scénarios en une séquence de sous scénarios de longueur constante. Le fractionnement conduit à la création d'un espace de recherche constitué des états atteints après les animations. Cette stratégie se décline en deux variantes *depth* et *breadth* selon l'algorithmique d'exploration de

l'espace de recherche. La dernière stratégie, *Path*, emploie les motifs d'animations des scénarios pour restreindre les transitions autorisées et par conséquent le nombre de formules des scénarios. Cependant la création de ces motifs engendre la résolution d'un nombre important de scénarios.

12.3 Collaboration

La dernière partie de cette thèse se concentre sur la collaboration entre un prouveur SMT et un solveur CSP dans le cadre de la génération de tests. La première étape est d'établir un canal de communication entre ces outils. Dans ce but, un métamodèle associant solveur et tests nommé CSP4MBT (CSP for Model-Based Testing) est créé pour remplir un rôle similaire à SMT4MBT pour les prouveurs. Un processus de conversion est établi pour créer un modèle CSP4MBT depuis SMT4MBT. Ce choix s'explique par la place prise par le solveur dans notre démarche. En effet, le solveur est utilisé uniquement en soutien du prouveur.

Ainsi la collaboration consiste en la soumission d'un scénario d'animation a un prouveur puis à l'exploitation du résultat par un solveur. Cette collaboration se manifeste au sein de deux nouvelles stratégies. La première nommée *Step-increase collaboratif* améliore la stratégie *Step-Increase* en réanimant la séquence de transitions sans préciser leur paramètre et en utilisant une fonction objective pour maximiser le nombre de cibles de test atteintes. La deuxième stratégie dénommée *Space-Search collaboratif* augmente la taille de l'espace de recherche par rapport à la stratégie *Space-Search* en générant de multiples solutions aux scénarios, si possible, avec le solveur. Ceci permet d'augmenter la probabilité d'atteindre les cibles de test au prix d'un temps de génération plus élevé.

12.4 Implémentation et expérimentations

Cette démarche et ses stratégies ont été implémentées sur la plate-forme modulaire Hydra. Ainsi les performances des stratégies ont pu être évaluées sur plusieurs cas d'études. D'après ces observations, la parallélisation du processus de génération de tests a un apport conséquent en terme de performance. De plus, la stratégie et sa configuration doivent être adaptées sous peine d'une dégradation notable des temps de génération. Par conséquent, même si notre démarche est automatique, l'apport de connaissance sur le modèle par l'ingénieur de test est obligatoire pour optimiser les performances.

Chapitre 13

Perspective

Sommaire

A l'issue de ce travail, celui-ci peut être étendu dans plusieurs directions. L'objectif de ce chapitre est de vous présenter les trois qui nous semblent les plus prometteuses.

13.1 Au niveau du modèle

La première direction concerne le modèle utilisé pour la génération de tests. Ce dernier peut être analysé afin de guider la configuration de notre démarche, d'être adapté à notre approche et de vérifier sa validité.

13.1.1 Caractérisation du modèle

Des perspectives de voies de recherche existent pour optimiser notre démarche de génération de tests. Une fonctionnalité nouvelle et intéressante serait de disposer d'un processus automatique de configuration. Ce dernier fournirait la capacité de sélectionner une stratégie et son paramétrage en fonction du modèle et des cibles de test. La possibilité de créer ce processus est conditionnée par l'existence d'une analyse capable de caractériser de manière suffisamment précise le modèle et les cibles.

Une telle analyse aurait de nombreux avantages. Le premier est de réduire l'influence de la connaissance du modèle par l'ingénieur de tests sur les performances

en terme de temps de génération. En effet, une analyse permet d'automatiser le paramétrage de la démarche. Par conséquent, l'utilisateur de la démarche n'a plus besoin de posséder des connaissances approfondies dans le domaine du test et de la modélisation pour obtenir des performances acceptables.

Un deuxième avantage résultant de cette analyse est la possibilité d'obtenir une estimation a priori des performances de la démarche. Cet avantage prend tout son sens lors de la génération de tests sur des modèles de tailles et de complexités conséquentes où les temps de génération doivent être pris en compte lors du planning du projet. Par exemple, une estimation permet de savoir si et comment la démarche peut être intégrée dans un processus de développement continu.

Un troisième avantage est de permettre une comparaison facilitée entre les différentes méthodes de génération de tests à partir de modèles. En effet, actuellement l'établissement d'une comparaison est extrêmement compliqué car les méthodes utilisent des langages et paradigmes de modélisation variés. Même si deux méthodes emploient le même langage et ont les mêmes restrictions en terme d'expressivité, les choix de modélisations influent sur les performances. Pour ces raisons, la création d'une liste de modèles dans un but comparatif parait impossible. Une solution serait de la remplacer par une liste de cas de comparaison défini uniquement au travers d'une suite de caractéristiques. Ainsi chaque méthode peut être appliquée sur des modèles optimisés validant les cas en respectant les caractéristiques de ces derniers. Par contre ces comparatifs n'évaluent pas uniquement la méthode de génération de tests mais les méthodes de générations et de modélisations. La pertinence de cette évaluation est plus élevée dans le cas où le modèle a été créé spécifiquement pour la génération de tests par rapport au modèle utilisé conjointement pour les tests et le développement.

Un modèle pourrait ainsi être caractérisé par sa taille, sa complexité, son expressivité. La taille donne une indication sur le nombre d'entités constituant le modèle tels les slots ou les énumérations dans le cas du langage UML4MBT et sur la taille de l'espace de recherche délimité par les états du modèle. Par exemple cet espace est infini si des variables entières sont présentes. La complexité renseigne sur les évolutions possibles du modèle. Une mesure envisageable serait le nombre de comportements élémentaires définis dans le modèle. L'expressivité indique quel type d'information peut être modélisé tels des contraintes temporelles, des comportements non-déterministes, des états infinis ...

Pour obtenir une analyse complète, les cibles de test doivent également être considérées. Trois caractéristiques importantes concerne la longueur, la satisfiabilité et l'interdépendance. La longueur correspond à celle de la longueur des séquences d'animations atteignant la cible. L'analyse doit donc tendre à fournir la répartition des cibles en fonction de leur longueur. Ainsi elle peut donner au mieux une fonction de répartition et au moins la moyenne et l'écart-type. La deuxième caractéristique précise le pourcentage de cibles inatteignables. La dernière caractéristique renseigne sur le degré d'interdépendance entre les cibles. Cette notion indique une probabilité qu'une animation atteignant une cible atteigne également une autre cible.

13.1.2 Aide à l'écriture du modèle

Notre démarche, de part l'utilisation d'un langage de modélisation dédié aux tests, se situe clairement dans la catégorie du test à partir de modèles où ces derniers sont utilisés uniquement pour la génération de tests. Par conséquent, rien ne s'oppose à optimiser ces derniers pour améliorer les performances de notre méthode. Il est ainsi envisageable de fournir une analyse préliminaire du modèle afin de guider l'ingénieur de test lors des choix de modélisations.

Un exemple de choix est fourni lors de la modélisation du Robot. La première possibilité est de modéliser les différents pièces par une énumération. La présence d'une pièce sur un tapis roulant ou dans le bras du robot est ainsi encodée par un attribut. L'absence de pièce est modélisée par une valeur spéciale dans l'énumération. Ce choix de modélisation s'éloigne ainsi du système réel. La deuxième possibilité est d'utiliser une classe pour représenter une pièce où la catégorie est stockée par un attribut. Ce choix est plus proche du système réel mais nécessite un nombre supérieur d'instances. De plus, la présence ou l'absence de pièce dans les autres composants est modélisée par des associations. Ainsi la création de ces associations et de leurs liens conduit à obtenir des scénarios d'animations ayant plus de variables et de contraintes et donc des temps de résolutions supérieurs. Ainsi seul la première possibilité a été retenue.

La création de cet outil d'aide à la modélisation aura deux étapes. Dans un premier temps, l'outil fournira une évaluation du modèle via des critères portant sur les scénarios d'animations tels que le nombre de variables ou de contraintes. Dans un second temps, l'outil devra reconnaître des motifs afin de proposer à l'utilisateur de refactoriser son modèle pour l'optimiser.

13.1.3 Utilisation pour la vérification

Une dernière perspective pour étendre notre méthode au niveau du modèle est de l'adapter afin de créer un outil de vérification. En effet, d'après les travaux présentés à la section 3.2, les solveurs et les prouveurs peuvent être utilisés pour vérifier un certain nombre de propriétés du modèle telles que l'indépendance, la cohérence des contraintes OCL4MBT ou l'existence d'états valides du modèle.

Cette vérification n'a pas pour objectif de concurrencer les méthodes existantes mais simplement de proposer des fonctionnalités identiques pour les langages de modélisation UML4MBT/OCL4MBT afin d'aider l'ingénieur de tests lors de la phase de modélisation.

Pour réaliser l'outil, notre démarche doit avoir quatre propriétés. La première est de limiter l'évolution à l'exécution d'une unique transition du diagramme d'états-transitions ou d'opérations du diagramme de classes. Cette dernière est assurée par une limitation de la longueur des scénarios d'animations à 2 pas. En effet, 2 pas sont nécessaires car l'exécution d'une transition est conditionnée à la réalisation d'une opération. La seconde est de séparer les contraintes définissant l'état initial des autres. Ceci est déjà prévu car un scénario d'animations sépare ces contraintes. La troisième est de créer de nouveaux objectifs et cibles de tests couvrant les propriétés

à vérifier. La quatrième et dernière propriété est de maintenir un lien étroit entre les entités du modèle et les entités des scénarios d'animations afin de pouvoir fournir des retours intéressants pour l'ingénieur de tests. Dans l'état actuel, les liens sont trop faibles pour être en mesure de relier une contrainte fautive avec une erreur dans le modèle.

13.2 Amélioration de la collaboration

Une autre direction pour améliorer notre démarche est d'augmenter les possibilités d'interventions sur le processus de résolution des prouveurs SMT. En effet à la différence des solveurs CSP où l'utilisateur peut guider la résolution via des instructions et via l'ordre des contraintes décrivant le problème, le prouveur SMT n'offre pas de mécanisme pour aider la résolution. Lors de la description d'un problème, toutes les formules sont traitées sur le même plan. La hiérarchisation des formules offrirait la possibilité d'employer la sémantique associée aux formules en terme de scénarios d'animations pour réduire le temps du processus de résolution.

Un deuxième avantage de la hiérarchisation des formules serait d'ouvrir un canal de communication supplémentaire entre un prouveur et un solveur grâce à un échange d'informations sous forme de formules. Actuellement, l'ajout de formules limitant les domaines des variables dans un problème à destination d'un prouveur ne résulte jamais en un temps de résolution inférieur. Pourtant l'ajout de formules spécifiant une séquence de transitions précise permet de rendre caduque la majorité des formules présentes dans le scénario d'animations.

13.3 Optimisation de la génération de tests

Une dernière direction pour étendre notre démarche est la création de nouvelles stratégies de génération de tests. Une piste est la création d'une stratégie combinant deux processus de conversion pour obtenir un scénario d'animations depuis un modèle UML4MBT. Une des caractéristique du processus décrit dans notre démarche est de dupliquer la plupart des formules et des variables pour chacun des pas de l'animation. Ceci entraîne une dégradation des performances quand la longueur des scénarios d'animations augmente. Pour pallier cet inconvénient, un autre processus de conversion est envisageable où une logique avec des quantificateurs serait utilisée. Cependant, cette logique entraîne la possibilité que le prouveur soit incapable de conclure. Dans ce cas la stratégie pourrait se rabattre sur la logique sans quantificateur. Il faut noter que cette nouvelle stratégie serait beaucoup plus sensible au choix du prouveur car elle dépend de l'algorithme d'élimination des quantificateurs employé.

Bibliographie

[AAH05] Jean-Raymond Abrial, Jean-Raymond Abrial, and A Hoare. *The B-book : assigning programs to meanings*. Cambridge University Press, 2005.

[ABGR07] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. Uml2alloy : A challenging model transformation. In *Model Driven Engineering Languages and Systems*, pages 436–450. Springer, 2007.

[Ana] Alloy Analyzer. `http://alloy.mit.edu/alloy/`. Accessed : 2012-05-15.

[Arc] Rational Software Architect. `http://www-03.ibm.com/software/products/fr/fr/ratisoftarch/`. Accessed : 2012-05-15.

[AW06] Krzysztof R Apt and Mark Wallace. *Constraint logic programming using ECLiPSe*. Cambridge University Press, 2006.

[BB87] Tommaso Bolognesi and Ed Brinksma. Introduction to the iso specification language lotos. *Computer Networks and ISDN systems*, 14(1) :25–59, 1987.

[BB09] Robert Brummayer and Armin Biere. Boolector : An efficient smt solver for bit-vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177. Springer, 2009.

[BCD05] D. Berardi, D. Calvanese, and G. De Giacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, 168(1-2) :70–118, 2005.

[BCD+11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Computer Aided Verification*, pages 171–177. Springer, 2011.

[BDLN06] Fabrice Bouquet, Stéphane Debricon, Bruno Legeard, and Jean-Daniel Nicolet. Extending the unified process with model-based testing. In *MoDeVa'06, 3rd Int. Workshop on Model Development, Validation and Verification*, pages 2–15, Genova, Italy, October 2006.

[BDOS08] Clark Barrett, Morgan Deters, Albert Oliveras, and Aaron Stump. Design and results of the 4th annual satisfiability modulo theories competition (smt-comp 2008). *To appear*, 6, 2008.

[Bei90] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[Bel10] Axel Belinfante. Jtorx : A tool for on-line model-driven test derivation and execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 266–270. Springer, 2010.

[BELP10] Ulrik Brandes, Markus Eiglsperger, Jürgen Lerner, and Christian Pich. Graph markup language (graphml). *Handbook of Graph Drawing and Visualization (Discrete Mathematics and Its Applications). Chapman & Hall/CRC*, 2010.

[BFS05] Axel Belinfante, Lars Frantzen, and Christian Schallhart. 14 tools for test case generation. In *Model-Based Testing of Reactive Systems*, pages 391–438. Springer, 2005.

[BGL+07] Fabrice Bouquet, Christophe Grandpierre, Bruno Legeard, Fabien Peureux, Nicolas Vacelet, and Mark Utting. A subset of precise uml for model-based testing. In *Proceedings of the 3rd international workshop on Advances in model-based testing*, pages 95–104. ACM, 2007.

[BGM02] Marius Bozga, Susanne Graf, and Laurent Mounier. If-2.0 : A validation environment for component-based real-time systems. In *In Proceedings of Conference on Computer Aided Verification, CAV'02*, pages 343–348. Springer Verlag, 2002.

[BGN+03] Mike Barnett, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Model-based testing with asml .net. In *1st European Conference on Model-Driven Software Engineering*, pages 12–19, 2003.

[BHL03] Frédéric Boussemart, Fred Hemery, and Christophe Lecoutre. De ac3 à ac7. *Technique et Science Informatiques*, 22(1) :267–280, 2003.

[bilpcd] Les bugs informatiques les plus chers de 2011. `http://www.gizmodo.fr/2011/12/15/les-bugs-informatiques-les-plus-chers-de-2011.html`. Accessed : 2012-05-15.

[BKLW11] Achim D Brucker, Matthias P Krieger, Delphine Longuet, and Burkhart Wolff. A specification-based test case generation method for uml/ocl. In *Models in Software Engineering*, pages 334–348. Springer, 2011.

[BL03] F. Bouquet and B. Legeard. Reification of executable test scripts in formal specification-based test generation : the Java Card transaction mechanism case study. In *Proceedings of the International Conference on Formal Methods Europe (FME'03)*, volume 2805 of *LNCS*, pages 778–795, Pisa, Italy, September 2003. Springer-Verlag.

[BLP02] Fabrice Bouquet, Bruno Legeard, and Fabien Peureux. Clps-b–a constraint solver for b. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 188–204. Springer, 2002.

[BLS05] Mike Barnett, K Rustan M Leino, and Wolfram Schulte. The spec# programming system : An overview. In *Construction and analysis of*

safe, secure, and interoperable smart devices, pages 49–69. Springer, 2005.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The smt-lib standard : Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, 2010.

[BSV10] Miquel Bofill, Josep Suy, and Mateu Villaret. A system for solving constraint satisfaction problems with smt. In *Theory and Applications of Satisfiability Testing–SAT 2010*, pages 300–305. Springer, 2010.

[BT07] Clark Barrett and Cesare Tinelli. Cvc3. In *Computer Aided Verification*, pages 298–302. Springer, 2007.

[BW97] Martin Buchi and Wolfgang Weck. A plea for grey-box components. *Turku Centre for Computer Science*, 1997.

[BW08] AchimD. Brucker and Burkhart Wolff. Hol-ocl : A formal proof environment for uml/ocl. In JoséLuiz Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering*, volume 4961 of *Lecture Notes in Computer Science*, pages 97–100. Springer Berlin Heidelberg, 2008.

[BW09] Achim D Brucker and Burkhart Wolff. hol-testgen. In *Fundamental Approaches to Software Engineering*, pages 417–420. Springer, 2009.

[CCR07] Jordi Cabot, Robert Clarisó, and Daniel Riera. Umltocsp : a tool for the formal verification of uml/ocl models using constraint programming. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 547–548. ACM, 2007.

[CCR09] Jordi Cabot, Robert Clarisó, and Daniel Riera. Verifying uml/ocl operation contracts. In *Integrated Formal Methods*, pages 40–55. Springer, 2009.

[CDFP97] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The aetg system : An approach to testing based on combinatorial design. *Software Engineering, IEEE Transactions on*, 23(7) :437–444, 1997.

[CF89] R Cytron and J Ferrante. An efficient method of computing static single assignment form. *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1989.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. Springer, 2013.

[CJRZ02] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. Stg : A symbolic test generation tool. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 470–475. Springer, 2002.

[Cona] Concordion. `http://concordion.org/`. Accessed : 2012-05-15.

[conb] conformiq. `http://www.conformiq.com/products/`. Accessed : 2012-05-15.

[DDM06] Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http ://yices. csl. sri. com/tool-paper. pdf*, 2 :2, 2006.

[DH01] James B Dabney and Thomas L Harman. *Mastering Simulink 4*. Prentice Hall PTR, 2001.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7) :394–397, July 1962.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3 : An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[Eda95] Hisakazu Edamatsu. Finite state machine, February 15 1995. EP Patent 0,356,940.

[FB10] Elizabeta Fourneret and Fabrice Bouquet. Impact analysis for uml/ocl statechart diagrams based on dependence algorithms for evolving critical software. *Laboratoire d'Informatique de Franche-Comté, Besançon, France, Tech. Rep. RT2010-06*, 2010.

[Fer88] Jean-Claude Fernandez. *ALDEBARAN : un systeme de vérification par réduction de processus communicants*. PhD thesis, Université Joseph-Fourier-Grenoble I, 1988.

[FG09] Gordon Fraser and Angelo Gargantini. An evaluation of model checkers for specification based test case generation. In *ICST*, pages 41–50. IEEE Computer Society, 2009.

[FGC+06] Patrick Farail, Pierre GOUTILLET, Agusti Canals, Christophe Le Camus, David Sciamma, Pierre Michel, Xavier Crégut, and Marc Pantel. The topcased project : a toolkit in open source for critical aeronautic systems design. *Ingenieurs de l'Automobile*, (781) :54–59, 2006.

[FGK+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, and Mihaela Sighireanu. Cadp a protocol validation and verification toolbox. In *Computer Aided Verification*, pages 437–440. Springer, 1996.

[GBR07] Martin Gogolla, Fabian Büttner, and Mark Richters. Use : A uml-based specification environment for validating uml and ocl. *Science of Computer Programming*, 69(1) :27–34, 2007.

[GHR+03] Jens Grabowski, Dieter Hogrefe, György Réthy, Ina Schieferdecker, Anthony Wiles, and Colin Willcock. An introduction to the testing and test control notation (ttcn-3). *Computer Networks*, 42(3) :375–403, 2003.

[GM93] Michael JC Gordon and Tom F Melham. *Introduction to HOL : a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[GP95] JanFriso Groote and Alban Ponse. The syntax and semantics of μcrl. In A. Ponse, C. Verhoef, and S.F.M. Vlijmen, editors, *Algebra of Communicating Processes*, Workshops in Computing, pages 26–62. Springer London, 1995.

[Hal91] Joseph Y. Halpern. Presburger arithmetic with unary predicates is pi 1 1 complete. *Journal of Symbolic Logic*, 56 :56–2, 1991.

[HCRP91] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9) :1305–1320, 1991.

[HFT00] A. W. Heerink, J. Feenstra, and G. J. Tretmans. Formal test automation : The conference protocol with phact. In H. Ural, R. L. Probert, and G. von Bochmann, editors, *Proceedings of the IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems : Tools and Techniques*, volume 176 of *IFIP Conference Proceedings*, pages 211–220, Dordrecht, 2000. Kluwer Academic Publishers.

[HJLGP99] Wai Ming Ho, J-M Jézéquel, Alain Le Guennec, and François Pennaneac'h. Umlaut : an extendible uml transformation framework. In *Automated Software Engineering, 1999. 14th IEEE International Conference on.*, pages 275–278. IEEE, 1999.

[HK06] Antawan Holmes and Marc Kellogg. Automating functional tests using selenium. In *Agile Conference, 2006*, pages 6–pp. IEEE, 2006.

[HN04] Alan Hartman and Kenneth Nagin. The agedis tools for model based testing. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 129–132. ACM, 2004.

[HVFR04] Jean Hartmann, Marlon Vieira, Herb Foster, and Axel Ruder. Uml-based test generation and execution. *Präsentation auf der TAV21 in Berlin*, 2004.

[jac] jacop. `http://www.osolpro.com/jacop/index.php`. Accessed : 2012-05-15.

[Jac02] Daniel Jackson. Alloy : a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2) :256–290, 2002.

[jbs] jbsBug. `http://www.prweb.com/releases/2013/1/prweb10298185.htm`. Accessed : 2012-05-15.

[Jér09] Thierry Jéron. Symbolic model-based test selection. *Electronic Notes in Theoretical Computer Science*, 240 :167–184, 2009.

[JJ05] Claude Jard and Thierry Jéron. Tgv : theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer*, 7(4) :297–315, 2005.

[KFdB+05] Marcel Kyas, Harald Fecher, Frank S de Boer, Joost Jacob, Jozef Hooman, Mark Van Der Zwaag, Tamarah Arons, and Hillel Kugler. Formalizing uml models and ocl constraints in pvs. *Electronic Notes in Theoretical Computer Science*, 115 :39–47, 2005.

[KHG11] Mirco Kuhlmann, Lars Hamann, and Martin Gogolla. Extensive validation of ocl models by integrating sat solving into use. In *Objects, Models, Components, Patterns*, pages 290–306. Springer, 2011.

[KWKK91] E Kwast, H Wilts, H Kloosterman, and J Kroon. User manual of the conformance kit. *Dutch PTT Research, Leidschendam*, 1991.

[LBR98] Gary T Leavens, Albert L Baker, and Clyde Ruby. Jml : a java modeling language. In *Formal Underpinnings of Java Workshop*. Citeseer, 1998.

[LL97] Gérard Le Lann. An analysis of the ariane 5 flight 501 failure-a system engineering perspective. In *Engineering of Computer-Based Systems, 1997. Proceedings., International Conference and Workshop on*, pages 339–346. IEEE, 1997.

[LP00] H. Lötzbeyer and Alexander Pretschner. Autofocus on constraint logic programming. In *IN PROC. (CONSTRAINT) LOGIC PROGRAMMING AND SOFTWARE ENGINEERING (LPSE'2000*, 2000.

[LPU02] Bruno Legeard, Fabien Peureux, and Mark Utting. Automated boundary testing from z and b. In *FME 2002 : Formal Methods–Getting IT Right*, pages 21–40. Springer, 2002.

[MA00] Bruno Marre and Agnes Arnould. Test sequences generation from lustre descriptions : Gatel. In *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*, pages 229–237. IEEE, 2000.

[MC05] Rick Mugridge and Ward Cunningham. *Fit for developing software : framework for integrated tests*. Prentice Hall, 2005.

[min] minizinc. `http://www.minizinc.org/`. Accessed : 2012-05-15.

[NSB+07] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. Minizinc : Towards a standard cp modelling language. In *Principles and Practice of Constraint Programming–CP 2007*, pages 529–543. Springer, 2007.

[otISTQB12] Glossary Working Party of the International Software Testing Qualifications Board. Standard glossary of terms used in software testing, October 2012.

[PMV03] Tom Pender, Eugene McSheffrey, and Lou Varveris. *UML bible*. Wiley Chichester, 2003.

[PPW+05] Alexander Pretschner, Wolfgang Prenninger, Stefan Wagner, Christian Kühnel, Martin Baumgartner, Bernd Sostawa, Rüdiger Zölch, and Thomas Stauner. One evaluation of model-based testing and its

automation. In *Proceedings of the 27th international conference on Software engineering*, pages 392–401. ACM, 2005.

[Pro00] Stacy J. Prowell. Tml : A description language for markov chain usage models. *Information and Software Technology*, 42(12) :835–844, 2000.

[Pro03] Stacy J Prowell. Jumbl : A tool for model-based statistical testing. In *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, pages 9–pp. IEEE, 2003.

[RG98] Mark Richters and Martin Gogolla. On formalizing the uml object constraint language ocl. In *Conceptual Modeling–ER'98*, pages 449–464. Springer, 1998.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The*. Pearson Higher Education, 2004.

[Ros00] Ronald Rosenfeld. Two decades of statistical language modeling : Where do we go from here ? *Proceedings of the IEEE*, 88(8) :1270–1278, 2000.

[SBF10] Peter J Stuckey, Ralph Becket, and Julien Fischer. Philosophy of the minizinc challenge. *Constraints*, 15(3) :307–316, 2010.

[SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF : eclipse modeling framework*. Addison-Wesley Professional, 2008.

[SC] SMT-COMP2012. `http://smtcomp.sourceforge.net/2012/`. Accessed : 2012-05-15.

[SEG+96] M. Schmitt, A. Ek, J. Grabowski, D. Hogrefe, and B. Koch. Autolink - putting sdl-based test generation into practice. In *Testing of Communicating Systems*, pages 227–243. Kluwer Academic Publishers, 1996.

[Sep90] Approved September. Ieee standard glossary of software engineering terminology, 1990.

[SL10] Muhammad Shafique and Yvan Labiche. A systematic review of model based testing tool support. *Software Quality Engineering Laboratory, Department of Systems and Computer Engineering, Carleton University*, page 21, 2010.

[SLT06] Christian Schulte, M Lagerkvist, and G Tack. Gecode. *Software download and online material at the website : http ://www. gecode. org*, 2006.

[SORSC01] Natarajan Shankar, Sam Owre, John M Rushby, and Dave WJ Stringer-Calvert. Pvs prover guide. *Computer Science Laboratory, SRI International, Menlo Park, CA*, 1 :11–12, 2001.

[Spe06] OMG Available Specification. Object constraint language, 2006.

[SSD97] Simone Spitz, Frank Slomka, and Matthias Dörfel. Sdl* - an annotated specification language for engineering multimedia communication systems. In *6th Open Workshop On High Speed Networks, Institut für*

Nachrichtenvermittlung und Datenverarbeitung, Universität Stuttgart, 1997.

[SWD11] Mathias Soeken, Robert Wille, and Rolf Drechsler. Verifying dynamic aspects of uml models. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6. IEEE, 2011.

[SWK+10] Mathias Soeken, Robert Wille, Mirco Kuhlmann, Martin Gogolla, and Rolf Drechsler. Verifying uml/ocl models using boolean satisfiability. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1341–1344. European Design and Automation Association, 2010.

[Tas02] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, pages 02–3, 2002.

[TB03] Jan Tretmans and Ed Brinksma. Torx : Automated model-based testing. 2003.

[TDH08] Nikolai Tillmann and Jonathan De Halleux. Pex–white box test generation for. net. In *Tests and Proofs*, pages 134–153. Springer, 2008.

[TJ07] Emina Torlak and Daniel Jackson. Kodkod : A relational model finder. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647. Springer, 2007.

[Top] Topcased. `http://www.topcased.org/`. Accessed : 2012-05-15.

[Tre08] Jan Tretmans. Model based testing with labelled transition systems. In *Formal methods and testing*, pages 1–38. Springer, 2008.

[UL10] Mark Utting and Bruno Legeard. *Practical model-based testing : a tools approach*. Morgan Kaufmann, 2010.

[UPL12] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5) :297–312, 2012.

[V+06] FW Vaandrager et al. Does it pay off ? model-based verification and validation of embedded systems. *PROGRESS White papers*, pages 43–66, 2006.

[VCG+08] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. In *Formal methods and testing*, pages 39–76. Springer, 2008.

[Wei09] Stephan Weißleder. Parteg (partition test generator). *See http ://parteg. sourceforge. net*, 2009.

[WM08] Stephen A White and Derek Miers. *BPMN modeling and reference guide : understanding and using BPMN*. Future Strategies Inc., 2008.

Résumé

Les travaux présentés dans cette thèse proposent une méthode de génération automatique de tests à partir de modèles.

Cette méthode emploie deux langages de modélisations UML4MBT et OCL4MBT qui ont été spécifiquement dérivées d'UML et OCL pour la génération de tests. Ainsi les comportements, la structure et l'état initial du système sont décrits au travers des diagrammes de classes, d'objets et d'états-transitions.

Pour générer des tests, l'évolution du modèle est représente sous la forme d'un système de transitions. Ainsi la construction de tests est équivalente à la découverte de séquences de transitions qui relient l'état initial du système à des états validant les cibles de test.

Ces séquences sont obtenues par la résolution de scénarios d'animations par des prouveurs SMT et solveurs CSP. Pour créer ces scénarios, des méta-modèles UML4MBT et CSP4MBT regroupant formules logiques et notions liées aux tests ont été établies pour chacun des outils.

Afin d'optimiser les temps de générations, des stratégies ont été développé pour sélectionner et hiérarchiser les scénarios à résoudre. Ces stratégies s'appuient sur la parallélisation, les propriétés des solveurs et des prouveurs et les caractéristiques de nos encodages pour optimiser les performances. 5 stratégies emploient uniquement un prouveurs et 2 stratégies reposent sur une collaboration du prouveur avec un solveur.

Finalement l'intérêt de cette nouvelle méthode à été validée sur des cas d'études grâce à l'implémentation réalisée.

Mots-clés: Génération de tests à partir de modèles, UML, OCL, Solveur CSP, Pouveur SMT, Collaboration, Parallélisation, Stratégies de génération de tests

Abstract

This thesis describes an automatic test generation process from models.

This process uses two modelling languages, UML4MBT and OCL4MBT, created specifically for tests generation. Theses languages are derived from UML and OCL. Therefore the behaviours, the structure and the initial state of the system are described by the class diagram, the object diagram and the state-chart.

To generate tests, the evolution of the model is encoded with a transition system. Consequently, to construct a test is to find transition sequences that rely the initial state of the system to the states described by the test targets.

The sequence are obtained by the resolution of animation scenarios. This resolution is executed by SMT provers and CSP solvers. To create the scenario, two dedicated meta-models, UML4MBT and CSP4MBT have been established. Theses meta-models associate first order logic formulas with the test notions.

7 strategies have been developed to improve the tests generation time. A strategy is responsible for the selection and the prioritization of the scenarios. A strategy is built upon the properties of the solvers and provers and the specification of our encoding process. Moreover the process can also be paralleled to get better performance. 5 strategies employ only a prover and 2 make the prover collaborate with a solver.

Finally the interest of this process has been evaluated through a list of benchmark on various cases studies.

Keywords: Model-based Testing, UML, OCL, Solver CSP, Prover SMT, Collaboration, parallelization, Test generation strategies