HAL
open science

Oracles et qualification du test de transformations de modèles

Jean-Marie Mottu

To cite this version:

Jean-Marie Mottu. Oracles et qualification du test de transformations de modèles. Génie logiciel [cs.SE]. Université Rennes 1, 2008. Français. NNT : . tel-00514506

HAL Id: tel-00514506

https://theses.hal.science/tel-00514506v1

Submitted on 2 Sep 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° d'ordre 3820

THÈSE

présentée

DEVANT L'UNIVERSITÉ DE RENNES 1
pour obtenir

le grade de : *DOCTEUR DE L'UNIVERSITE DE RENNES 1*

Mention : Informatique

par

JEAN-MARIE MOTTU

Équipe d'accueil : IRISA/TRISKELL

École doctorale : Matisse

Composante universitaire : IFSIC

Titre de la thèse :

Oracles et qualification du test de transformations de modèles

soutenue le 26 novembre 2008 devant la commission d'Examen

COMPOSITION DU JURY :

M. :	Olivier	RIDOUX	Président
MM. :	Jean-Luc	DEKEYSER	Rapporteurs
	Christian	PERCEBOIS	
MM. :	Yves	LE TRAON	Examinateurs
	Benoit	BAUDRY	
	Jean-Marc	JEZEQUEL	

Remerciements

Tout d'abord, je remercie la personne qui a le plus contribué à la réussite de ce travail : Benoit Baudry. Son encadrement m'a permis d'avancer malgré les difficultés et toutes les incertitudes. Ses conseils et encouragements m'ont toujours été utiles et je me suis beaucoup enrichi à ses côtés. Je remercie également Yves le Traon. Sous sa direction j'ai découvert beaucoup d'aspects de la recherche, l'esprit de synthèse, et j'ai compris l'intérêt de prendre du recul sur ce travail.

Je remercie Olivier Ridoux qui m'a fait l'honneur de présider le jury, ainsi que Jean-Luc Dekeyser et Christian Percebois qui ont rapporté cette thèse. J'ai apprécié leurs observations et questions.

Je souhaiterais aussi remercier Jean-Marc Jézéquel de m'avoir accueilli dans l'équipe Triskell. Le dynamisme de ses membres a été particulièrement motivant. Les discussions que nous avons eu ont renforcé ma curiosité et m'ont convaincu que nous menons nos activités avec passion. Un remerciement particulier pour Franck Fleurey qui était le premier d'une nouvelle espèce de testeurs : les testeurs de transformations de modèles. J'en suis une mutation (l'opérateur de mutation PF s'est attaqué à mon œil gauche) et j'apporte mes encouragements à tous les doctorants qui nous suivent.

Je remercie tous les ingénieurs de l'équipe qui m'ont assisté (et supporté), et qui ont découvert à leur dépend qu'un testeur détecte des erreurs sans forcément les localiser et qu'un doctorant-testeur a toujours un besoin urgent qu'elles soient corrigées. Je remercie tous ceux qui ont contribué à rendre mon environnement de travail beaucoup plus simple à gérer que le travail lui-même, en particulier le personnel de l'IRISA.

Je remercie également les chercheurs de Colorado State University et les partenaires du projet Domino qui m'ont permis d'envisager mes recherches avec d'autres perspectives que cette thèse.

Je remercie mes amis qui m'ont vu passer du côté obscur de la science sur mon tabouret.

J'ai également une pensée pour des personnes qui m'ont permis de tenir moralement et physiquement tout en résistant aux sirènes de la caféine : le groupe de D de mon répertoire téléphonique, les groupes de Thom, Roger, Craig, Brian, et bien d'autres encore.

Finalement, je remercie ma famille de m'avoir soutenu pendant ces années de thèse, en étant souvent aussi dépassée que moi ; avec une pensée particulière pour Marie.

1 Introduction *9*

2 Contexte et état de l'art *13*

2.1 Ingénierie Dirigée par les Modèles **13**

2.1.1 Modèles, méta-modèles, méta-méta-modèles, méta-modélisation 14

2.1.2 Mise en application de développement logiciel dans l'IDM 16

2.2 Transformation de modèles **19**

2.2.1 Principe de la transformation de modèles 20

2.2.2 Un exemple de transformation de modèles class2rdbms 22

2.2.3 Langages et outils de transformations de modèles 25

2.3 Test de logiciel **28**

2.3.1 Principe et principales problématiques du test de logiciel 29

2.3.2 Analyse de mutation 32

2.3.3 Oracle de test de logiciel 34

2.4 Contraintes et composants **38**

2.4.1 Conception par contrat 38

2.4.2 Exploitation des contraintes pour le test 39

2.4.3 Composant dans l'IDM 39

2.5 Test de transformations de modèles **40**

2.5.1 Travaux sur le test de transformations de modèles 41

2.5.2 Comparaison de modèles pour l'oracle 44

3 Analyse de mutation *47*

3.1 Adaptation de l'analyse de mutation à l'ingénierie dirigée par les modèles **48**

3.1.1 Une technique pour la qualification et l'amélioration des modèles de test 48

3.1.2 Limitations des opérateurs de mutation classiques 51

3.1.3 Les transformations de modèles d'un point de vue sémantique pour l'analyse de mutation 52

3.2 Opérateurs de mutation dédiés aux transformations de modèles **54**

3.2.1 Opérateurs de mutation concernant la navigation: 55

3.2.2 Opérateurs de mutation concernant le filtrage: 57

3.2.3 Opérateur de mutation concernant la création:.....59

3.3 Simplification des activités non automatisables 62

3.3.1 Simplification de l'amélioration des modèles de test.....63

3.3.2 Réduction du nombre de modèles de test.....65

3.4 Application de l'analyse de mutation adaptée au test d'une transformation 70

3.4.1 Un ensemble initial de modèles de test 70

3.4.2 Création des mutants 70

3.4.3 Évaluation du score de mutation initial.....70

3.4.4 Amélioration des modèles de test en exploitant le point de vue sémantique.....71

3.4.5 Minimisation du nombre de modèles de test.....73

3.5 Conclusion.....73

4 Oracles de transformations de modèles.....75

4.1 La problématique de la construction d'oracles pour le test de transformations de modèles 76

4.2 Construction de six fonctions d'oracle sur la base de trois techniques d'analyse de modèles 79

4.2.1 Trois techniques pour analyser les modèles 79

4.2.2 Six fonctions d'oracle pour le test de transformations de modèles 80

4.3 Construction de données d'oracle.....84

4.3.1 Illustration de la vérification de la transformation d'un modèle de test 85

4.3.2 Vérification complète de la transformation sous test 90

4.4 Permettre de qualifier les fonctions d'oracle.....91

4.4.1 Deux qualités pour l'oracle de transformation de modèles 91

4.4.2 Cinq propriétés d'oracle 93

4.4.3 Influence des propriétés d'un oracle sur sa qualification 96

4.5 Qualification de chaque fonction d'oracle 100

4.5.1 Propriétés d'oracle.....100

4.5.2 Qualités des fonctions d'oracle 106

4.6 Conclusion.....110

5 Mise en œuvre et qualification de composants de transformation de modèles *111*

5.1 Construction de composants de transformation de modèles de confiance **112**

5.1.1 Formation de composants de transformation de modèles 112

5.1.2 Qualification des composants de transformations de modèles..... 114

5.1.3 Augmentation du niveau de confiance 115

5.2 Développements pour la définition, la qualification et l'amélioration de composants de confiance **118**

5.2.1 Utilisation du langage Kermeta..... 118

5.2.2 Intégration d'un support des contraintes dans Kermeta 119

5.2.3 Harnais de test supportant les différentes fonctions d'oracle..... 124

5.2.4 Plate-forme pour l'application de l'analyse de mutation 126

5.3 Expérimentations pour le développement d'un composant de confiance **127**

5.3.1 Étape 1 : Qualification et amélioration des modèles de test 127

5.3.2 Étape 2 : Test et correction d'erreur..... 128

5.3.3 Seconde itération..... 129

5.3.4 Étape 3: Améliorer les contrats 130

5.3.5 Conclusion des expérimentations..... 134

5.3.6 Une adaptation de la méthodologie pour l'utilisation d'un générateur de modèles..... 135

5.4 L'analyse de mutation pour l'expérimentation de critères fonctionnels..... **137**

5.4.1 Critères de couverture du domaine d'entrée 137

5.4.2 Mise à l'épreuve de modèles satisfaisant les critères 138

5.4.3 Validation des critères proposés 139

5.5 Conclusion **140**

6 Conclusions et perspectives *141*

6.1 Contributions..... **141**

6.1.1 Analyse de mutation pour le test de transformations de modèles 142

6.1.2 Oracles pour le test de transformations de modèles..... 142

6.1.3 Qualification de composants de transformations de modèles 142

6.2 Perspectives..... 143

Annexe A 146

A.1 Six modèles de test..... 147

A.2 Six modèles attendus 149

A.3 Contrats..... 151

A.4 Assertions de patterns de model snippets..... 162

A.5 Assertions OCL..... 168

Annexe B 170

Annexe C 172

Annexe D..... 174

Annexe E 176

Glossaire..... 178

Bibliographie..... 180

1
Introduction

L'informatique occupe une place de plus en plus importante dans notre quotidien. Si bien qu'elle envahit tous nos équipements, des plus simples aux plus complexes. Même si l'utilisation de nombreux appareils est éloignée de celle d'un classique ordinateur, il est certain que des programmes informatiques régissent leur fonctionnement ou du moins ont permis leur mise au point. Les systèmes développés sont donc très nombreux, de plus en plus grands, sophistiqués, et complexes. Cette augmentation induit des besoins en techniques modernes pour le développement de systèmes fiables et à coût maîtrisé.

Capitaliser et réutiliser les artefacts du développement logiciel est crucial pour maîtriser l'augmentation de la complexité des systèmes développés, les nombreux changements des exigences tout au long du développement, ou encore la mise au point de lignes de produits logicielles. Dans l'Ingénierie Dirigée par les Modèles (IDM[1]), l'utilisation des modèles n'est plus restreinte à la documentation et à la phase de conception en amont du développement. Les modèles ne sont plus seulement informatifs mais productifs. Ils sont désormais exploités automatiquement dans le développement de logiciel. Dans ce contexte, au lieu de tenter une difficile réutilisation de code logiciel, les modèles deviennent des entités de première classe et sont réutilisés. Nous espérons que cette exploitation des modèles pour l'implémentation apporte un gain de productivité et de fiabilité pour les différentes versions du code. Cette nouvelle approche a conduit l'OMG (Object Management Group) à promouvoir le standard MDA (Model Driven Architecture) pour l'exploitation automatique et la réutilisation d'entité de développement à un haut niveau d'abstraction. Le MDA (et l'IDM en général) permet d'exploiter directement les modèles, alors qu'UML, malgré son adoption ces dernières années, servait essentiellement à l'expression de modèles en amont du développement.

Les transformations de modèles manipulent et créent de tels modèles productifs. Elles sont utilisées tout au long du développement pour en automatiser certaines étapes. Elles ont différentes préoccupations : l'introduction des décisions concernant la conception, le réusinage d'un modèle, la rétro-conception, ou encore la génération d'une forme exécutable de ce modèle

[1] Ou MDE (Model Driven Engineering)

(ce qui est la finalité du MDA). Ces transformations sont déployées dans différents développements logiciels et à différentes étapes. Des industriels les utilisent pour générer du code embarqué (Airbus) ou pour migrer des systèmes en transformant leurs modèles plutôt qu'en recodant leurs implantations (Sodifrance [Fleurey'07b]).

Dans un développement guidé par des modèles, il est crucial que chaque transformation soit testée car, si elle est incorrecte, elle peut introduire des erreurs dans les modèles qu'elle produit. Si une erreur n'est pas détectée et corrigée, elle se propagera dans plusieurs modèles à différentes étapes du développement :

- dans une succession de transformations, il y a *transmission* de l'erreur qui devient plus difficile à détecter et surtout à localiser,
- à chaque réutilisation de la transformation erronée, il y a *diffusion* de l'erreur dans tous les modèles produits par cette transformation.

Les techniques existantes de test de programmes sont difficiles à appliquer et insuffisantes pour tester des transformations de modèles. Ces transformations sont mises en œuvre dans des programmes, mais elles ont des caractéristiques propres.

Les transformations de modèles manipulent des données *complexes*. Les modèles sont des graphes d'objets dont la taille peut être importante. La structure des modèles est décrite par un méta-modèle définissant le domaine d'entrée de la transformation. La *génération de données de test* doit être automatisée car le domaine d'entrée est trop vaste et trop complexe pour écrire manuellement des *modèles de test*. Il est également difficile d'élaborer des *critères de test* pour sélectionner des ensembles efficaces de modèles de test. Pour couvrir le domaine d'entrée, les critères de test sont basés sur la structure complexe définie par le méta-modèle source. Pour sélectionner les modèles de test selon leur pouvoir de détection d'erreur, il faut connaître les erreurs contenues dans les transformations. Mais la nouveauté de l'IDM nous donne peu de recul sur l'utilisation des transformations.

Les transformations produisent des modèles qui sont complexes à analyser. L'*oracle* détermine si le modèle retourné par un test respecte la spécification. Tout d'abord, il contrôle des modèles de sortie complexes. Ensuite, la réutilisation des transformations influence leur test. Les transformations sont réutilisées parce qu'elles automatisent des traitements répétitifs mais également parce qu'elles sont coûteuses à développer. Quand une transformation est réutilisée dans plusieurs développements, sa spécification est adaptée aux caractéristiques du nouveau domaine d'application [Siikarla'08a]. Ces changements imposent d'adapter les tests, en sélectionnant les modèles de test encore valables mais surtout en reconsidérant les oracles.

Par ailleurs, les langages de transformation sont nombreux, récents, et hétérogènes. Pour considérer des techniques de test structurelles se basant sur le code de la transformation, les langages devraient être considérés individuellement, sans approche globale de la nature d'une transformation. Finalement, les environnements de développement orienté-modèles sont encore en développement, instables. C'est une difficulté supplémentaire car il est difficile de

développer des outils spécifiques pour le test, surtout sans avoir d'assurance sur la pérennité des technologies choisies.

L'analyse de ces caractéristiques permet de comprendre les limites des techniques de test existantes et d'identifier les besoins spécifiques pour le test de transformations. Dans nos travaux, nous mettons en évidence les caractéristiques et qualités des transformations de modèles ainsi que l'approche réalisée pour leur développement : autant d'éléments que nous voulons considérer pendant les différentes phases du test[Mottu'05, Baudry'06a]. Quand cela a été nécessaire et possible nous avons adapté les techniques existantes et proposé de nouvelles méthodes pour mettre en œuvre le test de transformation de modèles. Le travail que nous présentons dans ce rapport s'articule autour de trois contributions principales. Ainsi, dans cette thèse, nous contribuons à la fiabilisation de développements exploitant des modèles en étudiant le test des transformations de modèles.

Notre première contribution est l'**adaptation de l'analyse de mutation au test de transformations de modèles**. La connaissance des fautes potentiellement commises pendant la mise en œuvre d'une transformation est importante pour plusieurs raisons : les modèles de test doivent être mis à l'épreuve vis-à-vis de ces erreurs, les oracles doivent détecter les défaillances dues à ces erreurs. Ainsi nous proposons différents opérateurs de mutation, définissant les erreurs injectées. Nous les définissons en considérant les opérations réalisées par les transformations à un niveau sémantique sans considération du langage d'implémentation. Cet aspect de nos travaux a été publié dans [Mottu'06a]. Nous utilisons ensuite ces opérateurs pour évaluer l'efficacité des modèles de test, et celle des oracles, par analyse de mutation.

Notre seconde contribution consiste à **proposer différentes fonctions d'oracle et à étudier leur emploi pour le test de transformations de modèles**. Nous utilisons une fonction d'oracle pour vérifier que le modèle de sortie d'un test satisfait la spécification. Il s'agit d'un problème récurrent de l'activité de test car il est peu automatisable. Les fonctions proposées prennent en considération la complexité des modèles devant être analysés. Par ailleurs, l'évolution d'une transformation influence les vérifications des modèles de sortie d'un test. Nous avons étudié si les fonctions d'oracle proposées permettent de ne pas redéfinir systématiquement les vérifications des modèles produits par chaque nouvelle version d'une transformation. Cet aspect de nos travaux a été publié dans [Mottu'08b, Mottu'08a].

Dans notre troisième contribution nous **intégrons nos travaux dans une approche globale de qualification d'un composant de transformation de modèles** [Mottu'06b]. Nous exploitons un modèle de composant qui intègre l'implantation de la transformation, les tests (ensembles de couple modèle de test/oracle) mais aussi la spécification sous une forme exécutable. Sous cette forme, les composants embarquent une interface et garantissent leur exactitude et leurs conditions d'utilisation, ce qui en fait une unité réutilisable. Un assemblage de ces composants s'inscrit dans la chaîne de production du logiciel/système final. Le niveau de confiance du composant est alors mesuré et amélioré par une méthodologie globale exploitant l'analyse de mutation. De cette manière, nous validons directement les deux autres contributions de la thèse. La mise en œuvre de ces composants et l'application de la méthode étudiée est

possible grâce aux différents développements effectués pour le support de nos travaux dans la plateforme Kermeta : l'intégration d'un support d'un système de contraintes [Mottu'07] et la fourniture d'une plateforme expérimentale pour la mise en œuvre de l'analyse de mutation. Cette mise en œuvre de notre première contribution constitue une base importante pour l'expérimentation d'autres aspects de la problématique du test de transformation.

Nos travaux s'intègrent dans une recherche à longue échéance pour la validation des transformations de modèles. Les travaux de thèse de Franck Fleurey [Fleurey'06] ont initié cette recherche en fournissant Kermeta, le langage permettant la mise en œuvre de transformations et de nos contributions. Le test de transformations y était abordé en considérant la problématique de la qualification fonctionnelle de données de test pour les transformations. Nous exploitons notre adaptation de l'analyse de mutation et son implantation pour poursuivre cette étude. Nos travaux forment également une base pour les travaux de thèse de Sagar Sen auxquels nous participons [Sen'08].

Ce travail s'inscrit également dans différentes collaborations industrielles et universitaires qui ont déjà permis de disséminer une partie de nos résultats. Nous participons au projet ANR Domino[2] et nous collaborons avec l'équipe du professeur France (Colorado State University). L'équipe Triskell exploite aussi nos travaux dans différentes activités : le projet européen Speeds[3] par exemple.

Dans le chapitre 2, nous présentons le contexte de cette thèse et nous précisons la problématique de ce travail, en particulier vis-à-vis des travaux existants. Nous détaillons les points qui sont importants pour notre travail : le test dans l'IDM et de transformations de modèles, l'analyse de mutation, l'oracle.

Le chapitre 3 est consacré à la première contribution de notre travail qui est l'adaptation de l'analyse de mutation pour les transformations de modèles.

Le chapitre 4 est consacré à notre seconde contribution. Nous y proposons différentes fonctions d'oracles. Leurs différences et leurs propriétés sont étudiées pour qualifier chaque fonction dans le but d'évaluer leur intégration et leur impact sur l'emploi de transformations dans un développement dirigé par les modèles.

Le chapitre 5 est consacré à la dernière contribution avec l'étude et l'expérimentation d'une méthodologie globale de qualification et d'augmentation du niveau de confiance de composants de transformations de modèles. Cette technique nous permet de démontrer l'applicabilité de nos autres contributions. La mise en œuvre de notre travail est possible grâce aux différents développements que nous avons réalisés et que nous détaillons.

Le chapitre 6 nous servira à conclure ces travaux et à en présenter des perspectives.

---

[2] http://www.domino-rntl.org/
[3] http://www.speeds.eu.com/

2
Contexte et état de l’art

Dans ce chapitre, nous détaillons le contexte de cette thèse et nous dressons un état de l’art des travaux qui ont précédé et accompagné cette thèse. Cette partie attire l’attention sur les techniques reprises dans les chapitres suivants ainsi que sur les raisons qui ont appuyé nos études. Dans cette thèse, nous traitons la problématique du *test de transformations de modèles*. Cet état de l’art suit donc deux axes : l’*ingénierie dirigée par les modèles* et le *test de logiciel*. Dans la section 2.1, nous présentons l’ingénierie dirigée par les modèles d’un point de vue général. Dans la section 2.2, nous présentons les transformations de modèles et les travaux qui traitent ces programmes. Dans la section 2.3, nous présentons les principes du test de logiciel et nous détaillons un ensemble de travaux qui se sont intéressés à l’analyse de mutation et à l’oracle. Dans la section 2.4, nous regroupons ces deux axes en présentant les travaux qui exploitent des contrats dans le test et dans l’ingénierie dirigée par les modèles. Finalement, dans la section 2.5, nous présentons les travaux qui portent directement sur notre problématique : le test de transformations de modèles.

2.1 Ingénierie Dirigée par les Modèles

L’Ingénierie Dirigée par les Modèles (*IDM* [Favre'06]) permet de surmonter la croissance de la complexité des systèmes logiciels développés. L’IDM est une forme d’ingénierie générative par laquelle tout ou partie d’une application informatique est générée à partir de modèles. De cette manière, une partie importante du développement est réalisée à un niveau d’abstraction plus élevé que celui de la programmation classique. L’IDM permet alors d’automatiser, ou au moins de dissocier et de reporter, la part du développement qui est proprement technique et dédiée à telle plateforme d’implantation.

Dans cette nouvelle perspective, le développement du système se fait à partir d’une abstraction sous forme de modèles. Ces derniers occupent désormais une place de premier plan parmi les artéfacts de développement des systèmes. En contrepartie, ils doivent être suffisamment précis pour être interprétés ou transformés par des machines. Ce sont des transformations de modèles qui sont séquentiellement appliquées pour automatiser le processus de développement des systèmes. Chaque transformation prend des modèles en entrée et produit des modèles de sortie jusqu’à obtention d’artéfacts exécutables. Les modèles sont définis dans

des langages qu'il est nécessaire de décrire de manière précise : il s'agit de la méta-modélisation.

L'intérêt de l'IDM a été fortement amplifié, en novembre 2000, lorsque l'OMG (Object Management Group) a rendu publique son initiative MDA [Soley'00] qui vise à définir un cadre normatif pour l'IDM. Il existe des alternatives technologiques aux normes de l'OMG, par exemple Ecore dans la sphère Eclipse, mais aussi les grammaires, les schémas de base de données, les schémas XML.

Dans la sous-section 2.1.1, nous présentons les principaux concepts de l'IDM. Dans la sous-section 2.1.2, nous abordons la manière d'exploiter l'IDM. Tout d'abord nous présentons des approches de développement basé sur les modèles comme le MDA. Nous expliquons ensuite l'activité de méta-modélisation qui permet de créer des modèles.

2.1.1 Modèles, méta-modèles, méta-méta-modèles, méta-modélisation

Dans cette sous-section, nous définissons les concepts de base de l'IDM et les relations qui existent entre ces concepts. La suite de ce document s'appuie fortement sur ces définitions. Elles sont principalement issues des résultats de l'action spécifique MDA du CNRS [Estublier'05] qui ont été publiés dans le livre [Favre'06]. Nous remarquons qu'il s'agit de concepts généraux qui ne sont pas propres à l'IDM, sauf du point de vue du vocabulaire employé. Nous précisons dans la sous-section suivante l'emploi de ces concepts dans l'IDM et dans cette thèse.

a- Modèles et systèmes

Le premier concept important dans l'IDM est le concept de *modèle*. Quelle que soit la discipline scientifique abordée, un modèle est une abstraction d'un système construite dans un but donné. Une première relation est définie : un modèle *représente* un système (comme illustré dans la figure 2-1). Un modèle est une abstraction car il contient un ensemble restreint d'informations sur un système, sous une forme synthétique. Il est construit dans un but précis en ne contenant que les informations pertinentes vis-à-vis de l'utilisation qui sera faite du modèle. Par exemple, une carte routière est un modèle qui représente un territoire géographique dans le but de faciliter des trajets.

b- Méta-modèle : langage de modélisation

Un modèle n'est utilisable que si son langage de modélisation est précisé. Le langage de modélisation est *représenté par* un méta-modèle qui définit les éléments et la structure d'un modèle ainsi que sa sémantique. En d'autres termes, le méta-modèle exprime le lien existant entre un modèle et le système modélisé. Ces modèles *appartiennent au* langage représenté par le méta-modèle. Dans l'exemple d'une carte géographique, le méta-modèle utilisé est donné par la légende qui définit la signification des différents symboles utilisés ou encore l'échelle. La carte appartient au langage représenté par la légende et qui inclut toutes les cartes qu'il est possible d'élaborer à partir de cette légende.

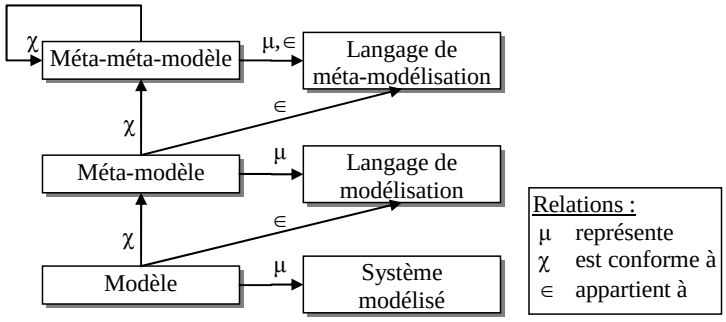


Figure 2-1 - Relations entre système modélisé, modèle, méta-modèle, et langage

Un modèle est *conforme à* un méta-modèle si tous les éléments du modèle sont définis par le méta-modèle. Cette notion de conformité est essentielle à l'ingénierie des modèles mais n'est pas nouvelle : un mot est conforme à une grammaire, un programme JAVA est conforme à la grammaire de JAVA et un document XML est conforme à sa DTD. Les deux relations *conforme à* et *appartient à* sont illustrées dans la figure 2-1.

c- Méta-méta-modèle : langage de méta-modélisation

Un méta-modèle est un modèle, il est donc conforme à son propre méta-modèle. Dans ce cas, il s'agit d'un *méta-méta-modèle*. Pour limiter le nombre de niveaux d'abstraction, et éviter de définir un méta-méta-méta-modèle, les méta-méta-modèles sont généralement conçus pour être *auto-descriptifs*, c'est-à-dire capables de se décrire eux-mêmes. Dans le domaine des grammaires, EBNF est spécifié par une grammaire et dans le cas d'XML, c'est une DTD XML qui joue le rôle de méta-méta-modèle pour les DTD.

En pratique, un méta-méta-modèle détermine l'*espace technique* de ses méta-modèles et de leurs modèles. Un espace technique est l'ensemble des outils et techniques issus d'une pyramide de méta-modèles dont le sommet est occupé par une famille de (méta)méta-modèles similaires [Favre'06]. A titre d'exemple, nous citons trois espaces techniques différents :

- EBNF appartient à l'espace technique des grammaires (grammarware). Il permet de définir des grammaires qui jouent le rôle de méta-modèles et dont les mots jouent le rôle de modèles. La compilation permet de passer d'un niveau d'abstraction à l'autre.
- XML appartient à l'espace technique des documents (docware) pour lequel la DTD joue le rôle de méta-modèle pour des documents XML pouvant alors être considérés comme des modèles.
- UML appartient à l'espace technique des modèles (modelware) représentant des systèmes logiciels.

Dans la suite du document, nous considérons les développements logiciels dans l'espace technique modelware.

2.1.2 Mise en application de développement logiciel dans l’IDM

Dans cette sous-section, nous présentons d’abord différentes approches de développement basées sur les modèles, puis nous présentons l’activité de modélisation qui se place en face des transformations pour la production de modèles.

a- Approches de développement basées sur les modèles

Nous avons retenu quatre approches de développement basé sur les modèles qui sont parmi les principales développées depuis une décennie. Nous nous intéressons principalement à la dernière, le MDA, car elle appartient à la même partie de l’espace technique modelware (mêmes méta-modèles, même outillage) que l’ensemble des travaux de cette thèse.

i. Rational Unified Process (RUP)

Le Rational Unified Process (RUP) [Kruchten'99] est un processus de développement fondé sur le cycle de vie en spirale. Il comble les lacunes du classique cycle en V (figure 2-10) pour limiter le risque de remise en question de la totalité (ou d’une trop grande part) du développement en cas de problème (coût sous-estimé, réalisation incohérente ou incorrecte, etc.). Chaque itération de la spirale est un incrément du système pour successivement passer d’une phase du projet à une autre jusqu’à la livraison du système (il y a quatre phases qui se nomment : *inception* puis *élaboration* puis *construction* puis *transition*). Chaque phase comprend un certain nombre d’itérations et selon la phase d’une itération l’effort est porté différemment sur l’analyse, la conception, l’implémentation, le test, etc. Le RUP exploite directement la modélisation en utilisant les diagrammes UML (Unified Modeling Language) [OMG'07] pour toutes les étapes d’analyse et de conception. Dans cette approche, les modèles sont exploités pour limiter l’utilisation d’un cahier des charges textuel pour le codage du logiciel mais également pour être facilement retravaillés et améliorés à chaque nouvelle itération. Les modèles assistent le développement, facilitent le travail d’équipe, la répartition des rôles, le développement itératif.

ii. Model-Integrated Computing (MIC)

Le Model-Integrated Computing [Sztipanovits'95] est une méthodologie qui décompose le développement logiciel en deux phases. La première phase est réalisée par des ingénieurs logiciels et systèmes. Elle consiste à étudier le domaine d’application visé pour produire un environnement de modélisation spécifique. Dans une seconde phase, les ingénieurs du domaine utilisent cet environnement pour modéliser l’application. L’implantation est ensuite générée automatiquement à partir du modèle et des fonctionnalités de l’environnement.

La suite d’outils Generic Modeling Environment (GME) [Davis'03] permet la mise en œuvre du MIC. GME est intégré dans l’environnement de développement logiciel Visual Studio .NET 2003 de Microsoft. Cependant la génération du logiciel à partir du modèle n’est pas complètement supportée car les langages de modélisation développés n’ont pas de sémantique.

iii. Usines logicielles

Microsoft propose sa propre vision de l'IDM avec les usines logicielles (Software Factories) [Greenfield'03]. L'idée consiste à adapter pour le développement logiciel des caractéristiques du développement matériel ayant fait leur preuve (d'où le nom d'usine). La première caractéristique retenue est la spécialisation des fournisseurs et des développeurs de logiciels. La deuxième est l'utilisation d'outils dédiés au domaine d'application sous la forme de transformations de modèles, de langages et d'outils de modélisations. La troisième est la réutilisation de composants logiciels mis à disposition. Ces idées sont intégrées dans l'environnement de développement logiciel Visual Studio .NET 2005 de Microsoft.

iv. MDA

L'OMG a proposé le MDA (Model Driven Architecture) en 2000 [Soley'00] pour mettre en avant de bonnes pratiques de modélisation et d'exploitation des modèles. Le MDA avance plusieurs idées. Tout d'abord, pour construire un système, il faut abstraire son développement en séparant les spécifications fonctionnelles et les contraintes de la plate-forme d'implantation. Dans cet objectif, le MDA définit une architecture de spécifications à plusieurs niveaux : d'un coté le système est modélisé avec des modèles indépendants de la plate-forme (PIM : Platform Independent Models), de l'autre côté le système est modélisé avec les caractéristiques de la plate-forme prises en compte (PSM : Platform Specific Models). Ensuite, des transformations de modèles sont utilisées pour automatiser le passage d'un PSM à partir d'un PIM. Des transformations servent également à modifier et améliorer les PSM et les PIM.

L'approche MDA permet de réutiliser le même modèle pour l'implémenter sur différentes plates-formes. Elle permet de relier les modèles de différentes applications pour les faire interagir. Elle supporte l'évolution des plates-formes et des techniques.

L'initiative de l'OMG a entrainé plusieurs normalisations de technologies pour l'ingénierie des modèles. Le MDA initial proposait d'utiliser le langage UML (Unified Modeling Language) comme unique langage de modélisation. Ce langage a été étendu avec la possibilité de créer des *profils* pour l'ajout de nouveaux concepts de modélisation. Ce mécanisme ne suffisant pas à la communauté MDA, le langage UML ne fut plus le sommet de la pyramide de méta-modélisation, mais un méta-modèle parmi d'autres. Ces méta-modèles sont des langages de méta-modélisation qui sont créés dans un contexte particulier et qui sont conformes à un méta-méta-modèle : le MOF (Meta-Object Facility).

b- Méta-modélisation orientée objet

Dans ce document, nous nous restreignons aux approches orientées objet. Dans ce contexte, les principaux langages de méta-modélisation sont MOF [OMG'97a], CMOF [OMG'04], EMOF [OMG'04], ECore [Ecore]. L'OMG a normalisé le MOF 1.4 en même temps qu'UML 1.4. EMOF (Essential MOF) et CMOF (Complete MOF) sont des évolutions du MOF 1.4 qui se différencient selon le nombre de concepts qu'elles définissent. Elles ont été normalisées dans le MOF 2.0 [OMG'04] au moment de la création d'UML 2.0. Le langage de modélisation UML [OMG'07] est conforme au MOF (selon leurs versions respectives). IBM a proposé Ecore

[Ecore] comme langage de méta-modélisation pour le framework de modélisation EMF (Eclipse Modeling Framework) [EMF] intégré dans la plate-forme de développement Eclipse. Ecore a été aligné sur la norme EMOF.

Nous avons retenu EMOF pour tous les développements de cette thèse car c'est un standard et qu'il est adopté dans la communauté française et supporté dans les outils de l'équipe Triskell. Dans ce point, nous présentons les principales constructions de méta-modélisation qui permettent d'obtenir et de manipuler des modèles.

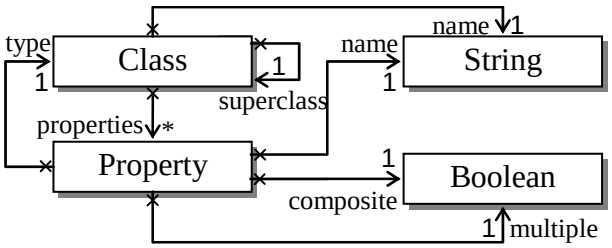


Figure 2-2 - Diagramme de classes d'un méta-méta-modèle orienté objet simplifié

La figure 2-2 représente un méta-méta-modèle orienté objet simplifié avec la notation UML des diagrammes de classes. Ce méta-méta-modèle permet de définir des classes qui ont des propriétés. Ces classes et propriétés ont un nom. Une propriété est également typée par une classe, ses deux booléens spécifient si la propriété référence un ou plusieurs objets et si la propriété contient cet(ces) objet(s). Une classe peut hériter d'une autre classe.

Ce méta-méta-modèle permet de définir de nombreux langages de méta-modélisation. Par exemple, le méta-modèle de la figure 2-3 présente un méta-modèle simplifié pour représenter des machines à états. Une machine à états contient différents états dont certain peuvent être initiaux, finaux, ou composites. Un état composite contient d'autres états. Les états sont reliés par des transitions correspondant à des évènements. La figure 2-3 représente le méta-modèle avec une syntaxe concrète. Une *syntaxe concrète* permet de représenter de manière plus concise et compréhensible un modèle, ce qui permet sa manipulation. Il peut exister différentes syntaxes concrètes. Par opposition, avec une syntaxe abstraite le modèle n'est représenté que de manière structurelle. Il s'agit typiquement du diagramme des instances des méta-classes. Par exemple, le méta-modèle de machines à états composites est représenté avec un diagramme d'instances dans la figure 2-4. Cette syntaxe illustre la conformité d'un modèle avec son méta-modèle et révèle la complexité des modèles.

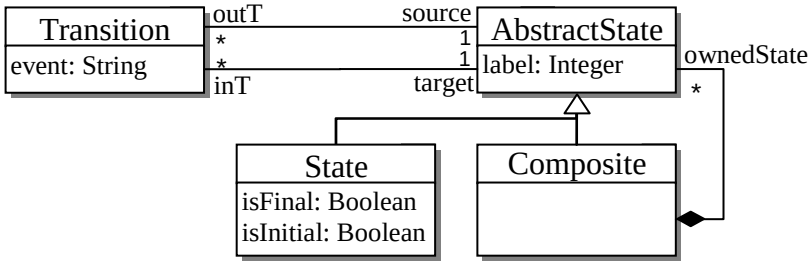


Figure 2-3 - Méta-modèle de machines à états composites

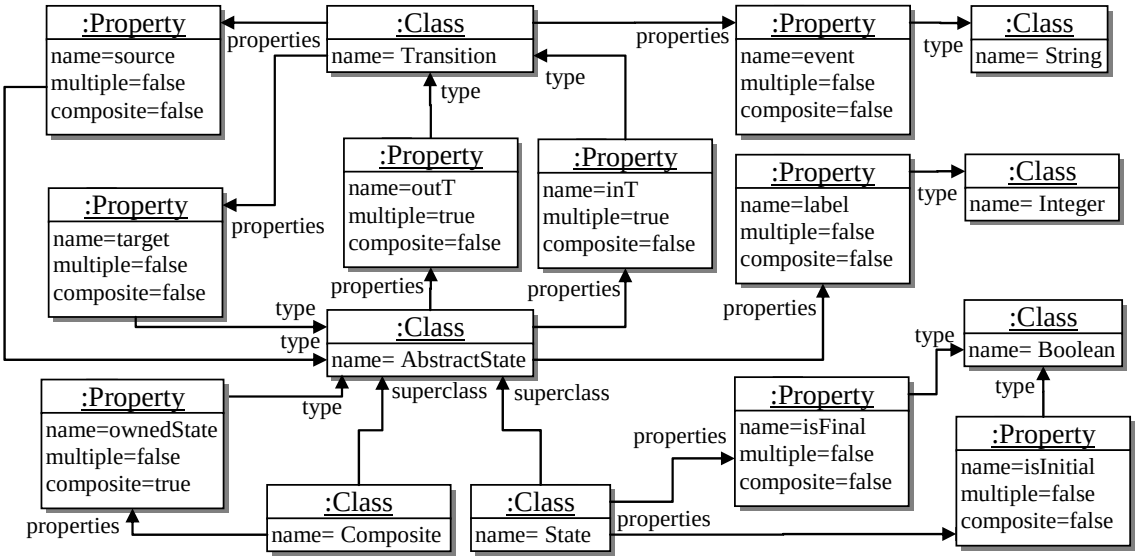


Figure 2-4 - Diagramme d'instances correspondant au méta-méta-modèle de machines à états composites

Avec ce méta-modèle, il est possible de créer des modèles en instanciant les méta-classes du méta-modèle. La figure 2-5 illustre deux exemples de modèles de machines à états dans une syntaxe concrète inspirée d'UML.

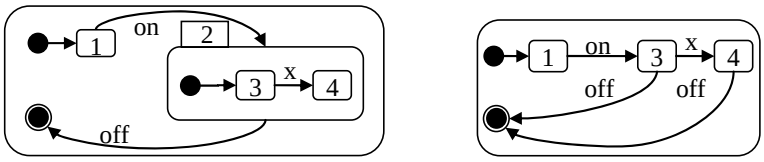


Figure 2-5 - Exemples de modèles de machines à états

2.2 Transformation de modèles

Dans l'IDM, les transformations de modèles ont beaucoup d'importance. Elles permettent de manipuler automatiquement des modèles. Cette automatisation est essentielle pour plusieurs raisons :

- La figure 2-4 illustre la complexité des modèles et donc la difficulté de les traiter manuellement.
- Les modèles traités peuvent être nombreux ou les mêmes traitements doivent être appliqués plusieurs fois.
- La séparation des rôles dans un cycle de développement complexifie la tâche si le spécialiste de la plateforme doit traiter manuellement le modèle produit par le spécialiste des fonctionnalités du système développé.
- Le gain obtenu grâce à l'abstraction fournie par les modèles (en productivité, en coût, en diminution des risques, en maîtrise de la complexité, etc.) est moindre si le passage vers différents niveaux d'abstraction n'est pas automatisé.

Les transformations sont utilisées pour réaliser différents traitements sur des modèles et sont employées dans différents domaines qui exploitent l’IDM.

Dans le MDA, les transformations sont utilisées pour changer de niveau d’abstraction et produire le modèle spécifique à une plateforme [Gerber'02]. Cette approche permet d’utiliser des transformations pour produire automatiquement des modèles qui prennent en compte des contraintes complexes liées à la plate-forme qu’il est préférable d’abstraire pour modéliser les fonctionnalités. De cette manière l’utilisateur peut se concentrer sur la vision abstraite du système développé et réutiliser des transformations qui ajoutent des fonctionnalités telles que le parallélisme [Yu'07] ou des contraintes pour embarquer des systèmes sur puces [Bondé'04, Bondé'06].

Les lignes de produits logiciel sont un important champ d’investigation pour l’utilisation de transformations de modèles. Oldevik et al. [Oldevik'07] proposent d’utiliser des transformations pour fournir la variabilité à une ligne de produits. A partir d’un modèle commun à l’ensemble de la chaîne, chaque produit peut être le résultat d’une transformation. Chaque transformation peut prendre en compte la variabilité de la plate-forme cible ou la variabilité fonctionnelle des produits.

Un système en développement peut être amélioré au niveau modèle, avant son codage (ou la génération de son code). Les améliorations les plus courantes consistent à appliquer des patrons de conception (design pattern). Un patron représente une solution à un problème de conception courant et sous une forme permettant sa mise en œuvre dans un système existant. De manière pratique, appliquer un patron de conception consiste à transformer un modèle, il est donc possible de spécifier des transformations qui automatisent l’application des patrons de conception [France'03].

L’intérêt de l’IDM réside donc en grande partie sur les transformations de modèles [Sendall'03]. Le gain de l’IDM ne réside pas seulement dans l’abstraction fournie par les modèles. L’automatisation des transformations de modèles permet d’exploiter les traitements complexes et répétés des modèles.

2.2.1 Principe de la transformation de modèles

Il existe différents types de transformations utilisées de différentes manières. Le processus général d’une transformation consiste à transformer un modèle `Ma` en un modèle `Mb` qui sont conformes à leurs méta-modèles respectifs `MMa` et `MMb` (figure 2-6). La transformation est *endogène* si `MMa` = `MMb`, sinon elle est *exogène* [Mens'05]. Une classification des différents types de transformation peut aussi être faite selon le changement de niveau d’abstraction. Une transformation *horizontale* produit un modèle du même niveau d’abstraction alors qu’une transformation verticale produit un modèle de niveau d’abstraction différent du modèle d’entrée. Dans le cas d’une transformation endogène (plutôt horizontale), il est possible que le modèle de sortie ne soit pas un nouveau modèle mais seulement une modification du modèle d’entrée.

Nous avons classé dans la figure 2-6 différentes catégories de transformation de l'IDM. Par exemple, l'approche MDA s'appuie sur les transformations de la catégorie « génération » qui ajoutent les préoccupations propres à une plateforme au modèle d'entrée pour produire des modèles de sortie [Gerber'02]. Dans le cas des transformations de migration de systèmes ou de sérialisation, il est possible de changer d'espace technique. Par exemple, la sérialisation d'un modèle UML en un fichier XMI (XML Metadata Interchange) permet de passer de l'espace modelware au docware (2.1.1c-).

.	niveaux d'abstraction z y x Ma transfo Mb horizontale	niveaux d'abstraction z y x w Ma transfo Mb verticale
MM endogène conformes à Ma transfo Mb	Réusinage (refactoring) Normalisation Application de patrons de conception Ligne de produits (nouvelle préoccupation fonctionnelle)	Raffinement
MMa exogène MMb confor-me à confor-me à Ma transfo Mb	Sérialisation Migration de systèmes Ligne de produits (nouvelle préoccupation fonctionnelle)	Génération (MDA : PIM vers PSM) Rétro-conception Ligne de produits (nouvelle plateforme)

Figure 2-6 - Classification des différentes catégories de transformation

De manière générale, une transformation est créée à partir d'une spécification de la transformation à effectuer qui comprend : les méta-modèles source et cible, ainsi qu'une description du comportement de la transformation. La réalisation d'une transformation nécessite une complète compréhension des méta-modèles d'un point de vue syntaxe abstraite [Gerber'02]. La transformation s'appuie sur la structure définie dans le méta-modèle pour analyser le modèle d'entrée et pour construire le modèle de sortie. Judson et al. [Judson'03] considèrent directement les transformations au niveau méta-modèle pour définir les relations à établir entre les modèles d'entrée et de sortie. Un méta-modèle de transformations décrit l'ensemble des relations à partir de la structure définie dans les méta-modèles source et cible. Un modèle conforme à ce méta-modèle de transformations définit une transformation de modèles particulière.

Différentes approches peuvent être employées pour développer des transformations. Plusieurs chercheurs ont relevé les propriétés des transformations qui permettent de classifier les différentes approches. Czarnecki et al. ont réalisé un travail important [Czarnecki'03, Czarnecki'06] qui a été repris dans d'autres travaux comme ceux de Mens et al. [Mens'05]. Une partie des propriétés retenues par Czarnecki et al. est :

- Les *relations entre l'entrée et la sortie* : s'agit-il de deux modèles ou d'une mise-à-jour du modèle d'entrée (comme dans les approches VIATRA, GReAT [Agrawal'03] par exemple), etc.

- la *portée* de la transformation : tout ou partie d'un modèle d'entrée peut être transformé,
- l'*ordre* des règles de la transformation : sont-elles ordonnées, interactives, ou encore le choix d'une règle est-il déterministe, etc.
- la *direction* des règles : unidirectionnelles, bidirectionnelles (comme dans l'approche MTF [Tratt'07] par exemple),
- etc.

Bézivin et al. [Bézivin'03c] proposent l'approche réflexive qui consiste à appliquer l'approche MDA au développement de transformations de modèles. Tout d'abord, elle consiste à définir une transformation abstraite qui soit indépendante de toute plateforme : PIT (Platform Independent Transformation). Ensuite, il est possible de dériver de façon plus ou moins automatique une PST (Platform Specific Transformation) spécifique à l'environnement de modélisation utilisé. La transformation elle-même est un modèle, décrite par son méta-modèle. Cela revient à appliquer aux transformations les mêmes principes qu'aux modèles et à en tirer les mêmes avantages : abstraction, généricité, réutilisabilité. UML semble être approprié pour décrire ce genre de transformation, mais il est restreint en matière d'exécutabilité : c'est pourquoi il est proposé de leur rajouter des fonctionnalités exécutables [Pollet'02, Clark'04] (en particulier au langage de contraintes OCL d'UML dont nous reparlons dans la section 2.4).

Les transformations de modèles sont souvent utilisées pour injecter de l'information dans un modèle (refactoring, etc.). Cette information est exprimée dans la spécification et intégrée dans le code de la transformation. Une alternative à cette approche consiste à formuler cette information dans un modèle qui est composé avec le modèle étudié pour produire le nouveau modèle. Cette approche est utilisée dans le MDA par exemple. Pour produire un modèle PSM à partir d'un modèle PIM, les informations sur la plate-forme peuvent être intégrées dans un PDM (Platform Description Model) qui est composé avec le PIM pour produire le PSM. La composition de modèles fait l'objet de différents travaux [Hovsepyan'07, France'07], en particulier dans le contexte de la modélisation par aspect.

2.2.2 Un exemple de transformation de modèles class2rdbms

Avant de lister un certain nombre de langages de transformation et leurs approches respectives, nous présentons une transformation que nous employons dans nos travaux et qui a été implémentée dans différents langages. Nommée `class2rdbms`, elle a été spécifiée pour le workshop MTIP (Model Transformation In Practice) [Bézivin'05]. Elle a été proposée par les organisateurs du workshop pour être la plus représentative possible des transformations de modèles pouvant exister dans l'IDM. L'objectif du workshop était d'évaluer sur une transformation commune les différentes approches de transformations de modèles. Les contributeurs de ce workshop ont proposé leur propre implantation de la transformation et ils ont expliqué leur approche et le langage utilisé [Taentzer'05, Murzek'05, Muller'05b, Lawley'05, Konigs'05, Kalnins'05, Jouault'05, Akehurst'05].

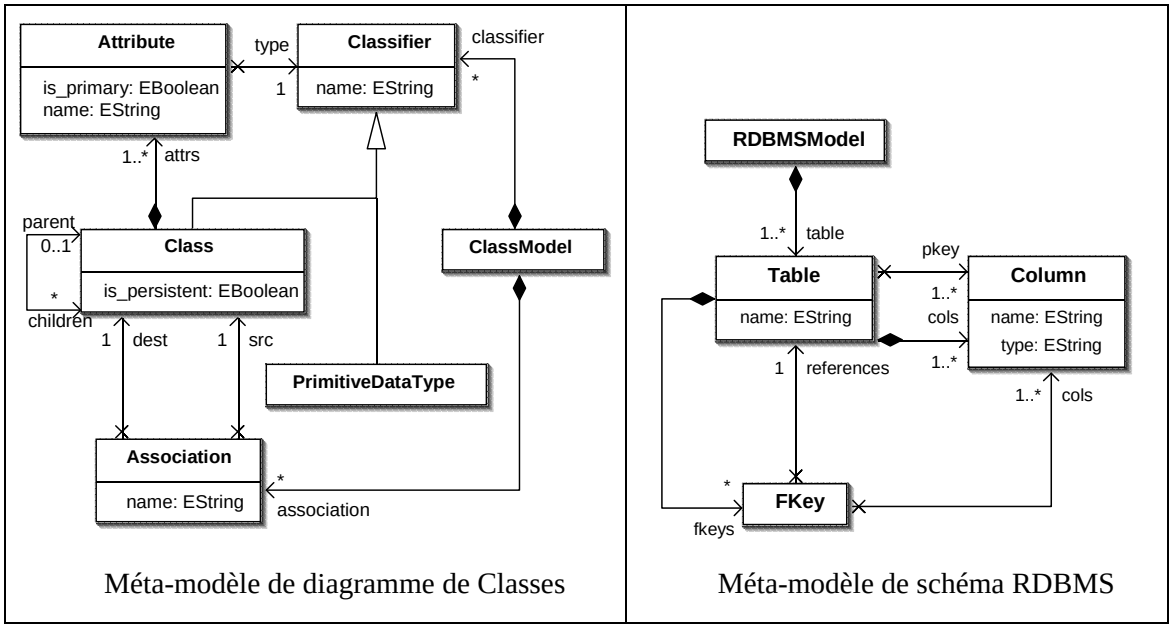


Figure 2-7 - Méta-modèles source et cible de la transformation class2rdbms

La spécification de la transformation class2rdbms est fournie dans l'appel à proposition du workshop MTIP [Bézivin'05]. Elle comporte plusieurs éléments : des méta-modèles source et cible et un ensemble de règles textuelles spécifiant le comportement de la transformation.

Le méta-modèle source définit le langage d'entrée de la transformation. Les modèles d'entrée sont conformes à ce méta-modèle, illustré dans la partie gauche de la figure 2-7. Ces modèles représentent des diagrammes de classes, tels que celui de la figure 2-8. Les modèles de sorties sont conformes au méta-modèle de schéma RDBMS illustré dans la partie droite de la figure 2-7. Par exemple, la transformation du modèle de la figure 2-8 produit le modèle RDBMS de la figure 2-9.

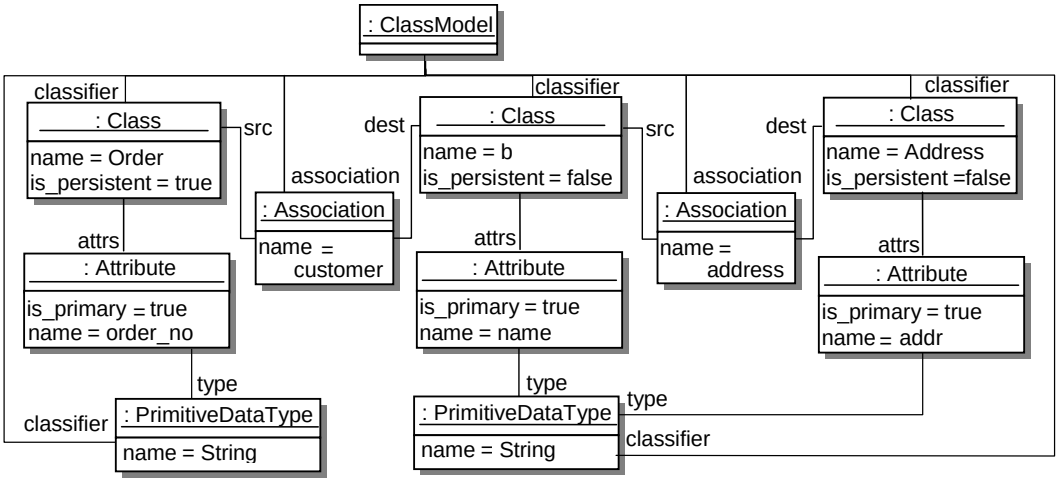


Figure 2-8 - Exemple de modèle d'entrée de la transformation class2rdbms

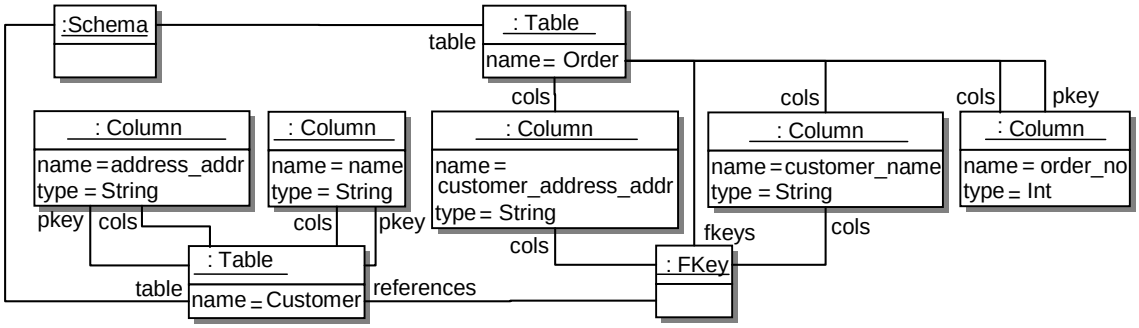


Figure 2-9 - Exemple de modèle de sortie de la transformation class2rdbms

La spécification du comportement de la transformation a été écrite avec ces sept règles [Bézivin'05] :

1. Classes that are marked as persistent in the source model should be transformed into a single table of the same name in the target model. The resultant table should contain one or more columns for every attribute in the class, and one or more columns for every association for which the class is marked as being the source. Attributes should be transformed as per rules 3 - 5.

2. Classes that are marked as non-persistent should not be transformed at the top level. For each attribute whose type is a non-persistent class, or for each association whose `dst` is such a class, each of the classes' attributes should be transformed as per rule 3. The columns should be named `name_transformed attr` where *`name`* is the name of the attribute or association in question, and `transformed attr` is a transformed attribute, the two being separated by an underscore character. The columns will be placed in tables created from persistent classes.

3. Attributes whose type is a primitive data type (e.g. `String`, `Int`) should be transformed to a single column whose type is the same as the primitive data type.

4. Attributes whose type is a persistent class should be transformed to one or more columns, which should be created from the persistent classes' primary key attributes. The columns should be named `name_transformed attr` where `name` is the attributes' name. The resultant columns should be marked as constituting a foreign key; the `FKey` element created should refer to the table created from the persistent class.

5. Attributes whose type is a non-persistent class should be transformed to one or more columns, as per rule 2. Note that the primary keys and foreign keys of the translated non-persistent class need to be merged in appropriately, taking into consideration that the translated non-persistent class may contain primary and foreign keys from an arbitrary number of other translated classes.

6. When transforming a class, all attributes of its parent classes (which must be recursively calculated), and all associations which have such classes as a `src`, should be considered. Attributes in subclasses with the same name as an attribute in a parent class are considered to override the parent attribute.

7. In inheritance hierarchies, only the top-most parent class should be converted into a table; the resultant table should however contain the merged columns from all of its subclasses.

Cette transformation est suffisamment complexe pour être représentative. Sa spécification comporte des règles qui impliquent des traitements non triviaux (clôture transitive, récursivité). Certaines règles sont dépendantes les unes des autres, et leur ordre d'exécution peut varier selon les modèles d'entrée. Nous remarquons que si ce cas d'utilisation est souvent employé pour illustrer des travaux, il ne l'est pas avec la même complexité. Par exemple, Podnieks [Podnieks'05] utilise une spécification de seulement trois règles simplifiées :

1. Transform each persistent class into a single table.
2. Transform each class attribute of a primitive type into a column of the corresponding table.
3. "Drill down" class attributes of complex types to leaf-level primitive attributes; transform these primitive attributes into columns of the corresponding table.

2.2.3 Langages et outils de transformations de modèles

La transformation class2rdbms peut être implémentée avec différents langages. Les propositions au workshop MTIP représentent un panel des langages utilisables et de leurs outils respectifs : AGG, AToM^3, VIATRA2, et VMTS [Taentzer'05], BMT [Murzek'05], MTL et Kermeta [Muller'05b], Tefkat [Lawley'05], TGG [Konigs'05], MOLA [Kalnins'05], ATL [Jouault'05], KMT [Akehurst'05]. De nombreux outils supportent différents langages de transformations de modèles. Ils sont commerciaux, libres, open source. Ils sont désormais trop nombreux pour être tous détaillés mais nous pouvons en introduire certains en les classifiant en quatre catégories :

1. les outils génériques de transformation qui peuvent être appliqués sur des modèles,
2. les langages de scripts intégrés à la plupart des ateliers de génie logiciel,
3. les langages dédiés à la transformation de modèles et dont la plupart s'intègrent dans des environnements de développement,
4. les outils dédiés à la méta-modélisation qui supportent les transformations de modèles et qui les considèrent comme des méta-programmes.

Dans un dernier point, nous introduisons la nouvelle norme de l'OMG pour la transformation de modèles QVT.

a- Outils génériques

Dans la première catégorie, les outils génériques de transformation regroupent principalement les techniques de transformation d'arbres et de graphes. Il y a notamment les outils de la famille XML, comme XSLT [W3C'07b] ou Xquery [W3C'07a]. Ils sont largement utilisés dans l'espace technique XML et permettent de transformer des modèles qui sont sérialisés dans un arbre. En particulier, les modèles conforment au MOF sont sérialisés dans un fichier XMI (XML Metadata Interchange) qui est la norme définie par l'OMG et qui intègre XML.

Les outils de transformation de graphes sont la plupart du temps issus du monde académique [Taentzer'05]. Dans ce contexte, les modèles prennent la forme de graphes dirigés, attribués, à nœuds et arcs étiquetés. Il existe plusieurs systèmes de transformation de graphes qui permettent de réaliser des transformations de modèles, par exemple : VIATRA2 (Visual Automated model TRAnsformations) [Varro'07], ATOM3 [de Lara'02], AGG [Taentzer'04], VMTS [Levendovszky'04], TGG [Konigs'05], MOLA [Kalnins'04],GReAT (Graph Rewriting And Transformation) [Agrawal'03], etc.

b- Langages intégrés aux ateliers de génie logiciel

Dans la seconde catégorie, une famille d’outils proposés par des vendeurs d’ateliers de génie logiciel permet de réaliser des transformations de modèles. Par exemple, l’outil Arcstyler [Hubert] de Interactive Objects propose la notion de MDA-Cartridge qui encapsule une transformation de modèles écrite en JPython (langage de script construit à l’aide de Python et de Java). D’autres outils commerciaux existent comme ADONIS qui propose le langage de transformations BMT [ADONIS'05], ou encore Objecteering [Objecteering Software] qui propose, pour transformer des modèles, le langage de script OptimalJ [Compuware]. D’autres ateliers sont proposés en open source, par exemple Fujaba [Burmester'04] (From UML to Java and Back Again).

L’intérêt de ce support des transformations de modèles est leur intégration dans ces ateliers qui sont matures et pas seulement expérimentaux ou démonstrateurs (car certains d’entre eux comme le J d’Objecteering sont développés depuis plus d’une décennie). Cette intégration poussée a un inconvénient : ces langages de transformation de modèles sont la plupart du temps propriétaires, non standardisés. De plus, ces langages ne sont que des fonctionnalités additionnelles pour leurs ateliers de génie logiciel. Ils n’ont pas été définis spécifiquement pour l’application de développements dirigés par les modèles.

c- Langages spécifiques

Dans la troisième catégorie, des outils sont spécifiquement conçus pour la transformation de modèles et sont parfois prévus pour être intégrés dans les environnements de développement standards. Mia Transformation [Mia-software] de Mia-Software (filiale de Sodifrance) est un outil commercial qui exécute des transformations de modèles prenant en charge différents formats d’entrée et de sortie (XMI, tout autre format de fichiers, API, dépôt). Les transformations sont exprimées comme des règles d’inférence semi-déclaratives. PathMATE [Pathfinder Solutions] de Pathfinder Solutions est un autre environnement de transformations de modèles qui s’intègre à des ateliers de modélisation UML comme par exemple IBM Rational Rose. Dans le monde académique, de nombreux projets s’inscrivent dans cette approche : les outils ATL [Bézivin'03a] (qui s’intègre à la plate-forme open-source TopCased [Vernadat'06] ou la plate-forme commerciale MDworkbench de Sodius [Sodius]) et MTL [Pollet'05] de l’INRIA, l’extension d’OCL pOCL [Millan'08] (qui s’intègre à la plateforme Neptune II), ModTransf [West Team] (qui est exploité dans la plate-forme Gaspard2 [Cuccuru'05]). Ces langages spécifiques sont nombreux : AndroMDA [AndroMDA], BOTL [Braun'03] (Bidirectional Object oriented Transformation Language), Coral [Alanen'04] (Toolkit to

create/edit/transform new models/modeling languages at run-time), Epsilon [Kolovos'08], Tefkat [Lawley'05], MT [Tratt'07], KMT [Akehurst'05], SiTra [Akehurst'06], et d'autres encore.

d- Outils de méta-modélisation

Dans la quatrième catégorie, des outils de méta-modélisation supportent la transformation de modèles en l'interprétant comme un méta-programme. MetaEdit+ [Tolvanen'03, Smolander'91] de MetaCase est le premier de ces outils. Il permet de définir explicitement un méta-modèle et de programmer au même niveau les outils pour éditer les modèles, générer du code, et transformer des modèles. L'outil MetaGen [Revault'96] propose un environnement pour l'édition de méta-modèles et un langage à base de règles pour exprimer des transformations de modèles. L'environnement XMF-Mosaic [Xactium] de Xactium est un environnement complet pour la définition de langage. XMF-Mosaic comprend un noyau exécutable pour définir des langages. Son langage de méta-modélisation est orienté objet. Il a toute la puissance d'un langage de programmation complet ce qui en fait un outil puissant pour la transformation de modèles.

Le langage Kermeta est statiquement typé, libre et conforme aux normes de méta-modélisation actuelles ce qui le distingue des précédents outils de méta-modélisation. Kermeta a été défini pour prendre en compte les besoins liés à la validation et à la vérification de méta-modèles et de transformations de modèles. Dans le chapitre 5, nous présentons différents développements, que nous avons réalisés pour Kermeta et avec Kermeta, qui contribuent au test de transformations de modèles en particulier.

e- Une norme pour la transformation de modèles QVT

En définissant le standard QVT (Query/View/Transformation) [OMG'08], l'OMG propose une norme pour les langages de transformations de modèles. Le méta-modèle de QVT est conforme au MOF et exploite OCL pour parcourir les modèles. QVT définit trois langages de transformations de modèles. Chacun est basé sur un paradigme différent pour l'implémentation d'une transformation (déclaratif, impératif, hybride). Les langages `Relations` et `Core` sont déclaratifs. La sémantique de `Relations` est spécifiée à partir de `Core` et se présente comme une transformation de `Relations` vers `Core`. Puisque toutes les transformations ne peuvent pas être entièrement déclaratives, QVT étend `Relations` et `Core` par deux mécanismes : le troisième langage de QVT (`Operational Mappings`) et un mécanisme d'invocation de fonctionnalités de transformation implémentées dans un langage non défini dans la norme. `Operational Mappings` permet d'implémenter les transformations les plus complexes avec des constructions impératives et des constructions OCL permettant les effets de bord.

Plusieurs langages ont été proposés à l'appel à proposition de QVT [OMG'05], ATL [Jouault'06] par exemple. De nouveaux langages correspondent à la version finale de la norme. Par exemple, Belaunde et al. proposent le support de QVT Operational dans SmartQVT [Belaunde'06]. Dans le projet M2M d'Eclipse, QVTR est une implémentation de QVT Relational (développé par Obeo) [Obeo] alors que QVTO sera une implémentation de QVT

Operational. Des outils commerciaux supportent aussi des parties du standard QVT : Borland propose le support de QVT Operational dans Borland Together.

Face à cette multitude de langages, Etien et al. [Etien'07] définissent le langage TrML qui propose une notation graphique unifiée des transformations de modèles. Une transformation est définie en TrML et exécutée dans un moteur de transformation existant (celui d’ATL par exemple) grâce à une transformation de TrML vers un langage cible.

Nous ne détaillons pas davantage les différences et points communs de ces approches et langages et nous ne les comparons pas pour plusieurs raisons :

- Il y a beaucoup de langages et pas encore de consensus sur la définition et la mise en œuvre d’une transformation. QVT a été normalisé pendant cette thèse (appel à proposition en 2005, la norme évoluant jusqu’en 2008 [OMG'08]), et les premiers langages supportant officiellement la norme ne sont apparus que très récemment (fin 2006 pour la version préliminaire de SmartQVT [Belaunde'06])
- Nous avons davantage travaillé sur le test fonctionnel. Cette approche ne considère pas la structure de l’implantation de la transformation et ne doit pas s’appuyer sur la syntaxe du langage utilisé. Nous montrerons que même pour l’analyse de mutation nous avons travaillé à un niveau sémantique et pas syntaxique alors que cette technique s’appuie traditionnellement sur la syntaxe du langage pour l’insertion d’erreur (nous le présentons dans la sous-section 2.3.2). Les principales particularités d’une transformation qui influencent son test résident dans les manipulations qu’elle réalise sur des modèles.

Dans nos travaux, nous avons étudié des techniques en les préservant indépendantes des langages et des technologies de transformation. Seules les expérimentations, les réalisations, et les illustrations de nos contributions ont du être réalisées avec un langage de transformation de modèles (Kermeta en l’occurrence).

2.3 Test de logiciel

Dans cette section, nous présentons le test de logiciel et ses principales problématiques. Nous appuyons cette présentation sur un ensemble de travaux du domaine du test de logiciel.

Un programme a besoin d’être vérifié tout au long de son cycle de vie et pas uniquement dans le cas de logiciels hautement critiques. Le test apparaît comme une technique incontournable pour la vérification de logiciels [Xanthakis'00, Myers'79, Binder'99, Beizer'90]. En effet, le test est un complément indispensable aux techniques de vérification statique ou de preuves mises en œuvre au cours du développement. Dans cette section, nous présentons tout d’abord brièvement le processus du test de logiciel. Nous insistons ensuite sur deux problématiques de test traitées dans cette thèse :

- la qualification de données de test avec la technique de l’*analyse de mutation*,
- la production du verdict d’un test avec l’*oracle*.

2.3.1 Principe et principales problématiques du test de logiciel

Le but du test de logiciel est de détecter les erreurs des programmes. Il existe une confusion entre les termes *faute*, *erreur*, et *défaillance*. Nous retenons la définition de Laprie [Laprie'95] qui se comprend en deux phrases :

> « Le programmeur effectue une *faute* en introduisant une *erreur* dans le programme. Cette *erreur* entraîne une *défaillance* du fonctionnement du programme. »

Cependant Xanthakis inverse l'emploi des termes faute (défaut) et erreur [Xanthakis'92]. Nous ne porterons pas attention à cette confusion car nous nous intéressons uniquement à vérifier que l'implantation d'un programme est *correcte*.

Les tests d'un logiciel sont liés aux techniques de développement utilisées. Ainsi, il est nécessaire de faire évoluer conjointement les techniques de développement et les techniques de test qui leur sont associées. Par exemple, dans un cycle classique en V, quatre phases de test correspondent à quatre phases de développement (figure 2-10). Les tests unitaires vérifient l'implantation du programme par unité. La taille d'une unité dépend du paradigme du langage de programmation : une classe dans les programmes orientés objet ou une procédure. Les tests d'intégration vérifient les interactions entre plusieurs unités d'un assemblage. Les tests systèmes vérifient la totalité du système en intégrant tous les groupes d'unités utilisés lors du test d'intégration. Les tests de recette valident le système par rapport à la spécification initiale et aux attentes du client. Chaque paradigme de programmation a ses propres spécificités pour le test. Par exemple, l'introduction des classes dans la programmation par objet nécessite de vérifier les interactions entre classes dans les tests d'intégration. Dans cette thèse, nous étudions les spécificités du test de transformations de modèles.

Certaines conditions influencent la testabilité d'un programme. Voas [Voas'92a] identifie les trois principales : une partie du programme doit pouvoir être exécuté, si cette partie est erroné il faut que son exécution modifie l'état normal des données, et ce changement d'état doit être observable. Les tests d'une transformation de modèles doivent solliciter ses erreurs pour que les modèles de sortie soient différents de ce qui est spécifié.

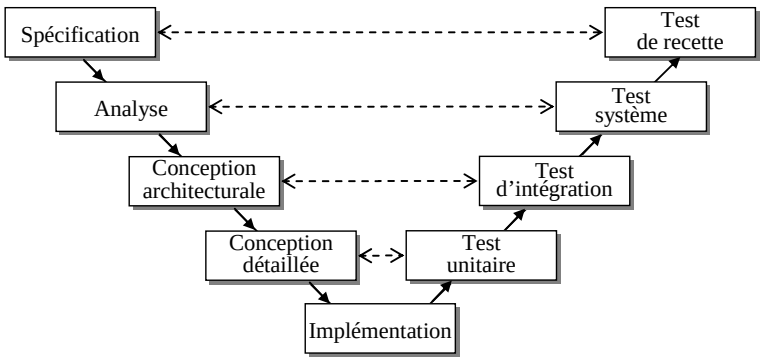


Figure 2-10 - Cycle de développement en V

L'exécution des tests est statique ou dynamique. Un test est statique s'il est réalisé sans exécuter le programme testé. Le programme est vérifié manuellement ou par des techniques

d’analyse statique de code. Un test est dynamique quand le programme est exécuté pour provoquer sa défaillance et révéler la présence d’une erreur. Dans cette thèse, nous étudions le test dynamique.

a- Processus du test dynamique de logiciel

D’un point de vue synthétique, le processus du test dynamique de logiciel consiste à exécuter un programme avec un ensemble de données en entrée et à vérifier que les résultats obtenus sont ceux attendus. Si ce n’est pas le cas, ce test a détecté une erreur qu’il faut localiser et corriger. Il faut systématiquement réappliquer les tests après chaque correction. Généralement, il existe une infinité de données d’entrée, donc il faut définir et appliquer un critère de test pour décider quand le programme a été suffisamment testé.

Dans la figure 2-11, nous illustrons ce processus en employant la même terminologie [Xanthakis'00] que dans le reste de la thèse. Le programme exécuté par les tests est le *programme sous test.* Les données d’entrée d’un programme sous test sont des *données de test*. Un *oracle* vérifie l’exactitude d’une donnée produite par l’exécution d’une donnée de test. L’oracle produit le *verdict* du test : le test *passe* ou *échoue*. Il y a *régression* quand corriger une erreur introduit une autre erreur. Finalement, un *cas de test* regroupe : une donnée de test, l’oracle correspondant, d’autres informations comme l’état du système avant l’exécution du test [Binder'99].

Les principales problématiques du test dynamique de logiciel portent sur ces activités (figure 2-11). Les techniques de test pour mettre en œuvre ces activités sont de deux types : fonctionnel ou structurel [Beizer'90] :

- Le test fonctionnel n’exploite pas la structure du programme qui est considéré dans une « boîte noire ». De cette manière, les activités du test ne sont pas affectées par les changements effectués dans le code du programme.
- Le test structurel exploite la structure du programme qui est considéré dans une « boîte blanche ».

Nous abordons la *création de données de test*, l’*élaboration de critères d’arrêt*, la *localisation des erreurs* dans les points suivants. Nous consacrons les sous-sections suivantes à deux problématiques traitées dans le reste du document : l’*analyse de mutation* (participant aux activités de critère d’arrêt et de création de données de test) et l’*oracle*.

b- Critères de test

Le test ne peut pas être exhaustif car les données de test appartiennent généralement à des domaines d’entrée non bornés ou trop grands. La valeur du test repose sur la sélection d’un ensemble de données de test *suffisant*. Pour cela les données de test doivent satisfaire un critère de test (ou critère d’arrêt). Le critère de test s’assure que les cas de tests ont un pouvoir de détection d’erreur suffisant. L’élaboration de critères de test s’appuie sur la structure du programme (si le test est structurel) ou uniquement sur la spécification (si le test est fonctionnel).

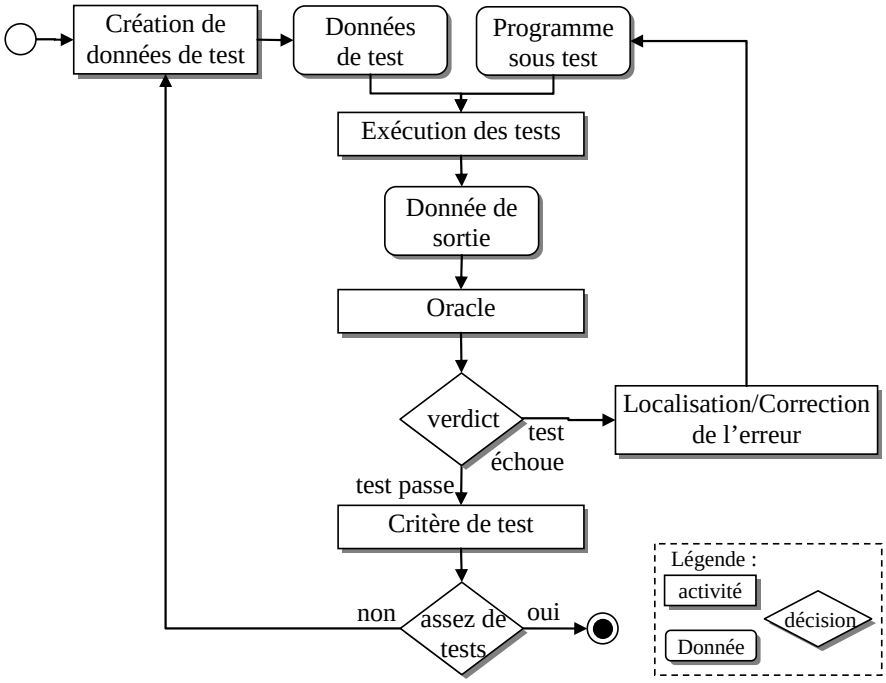


Figure 2-11 - Processus de test dynamique de logiciel

La plupart des travaux élaborent des critères de test structurel qui s'appuient sur la couverture : du code, du graphe de flot de contrôle, ou encore du graphe de flot de données. Par exemple, chaque instruction doit être exécutée au moins une fois, ou chaque chemin du graphe de flot de contrôle/données doit être parcouru un certain nombre de fois [Harrold'94].

Les critères de test fonctionnels exploitent uniquement la spécification. Dans ce cas, des critères déterminent la couverture du domaine d'entrée du programme sous test [Ostrand'88]. Si le comportement du programme est spécifié par des machines à états, il existe des critères de test qui déterminent leurs couvertures [McGregor'01]. Les machines à états sont similaires à des graphes, les exploiter pour mettre au point des critères de test est similaire à l'exploitation des graphes de flot de contrôle/données (couverture de nœuds, d'arcs, de chemins).

D'autres approches existent, comme l'*analyse de mutation* dont nous introduisons l'approche et les travaux dans la sous-section suivante et que nous étudions en détail dans le chapitre 3.

c- Localisation d'erreurs

La localisation d'erreurs (ou *diagnostic*) consiste à trouver la position des erreurs dans l'implantation. C'est une activité coûteuse car elle est majoritairement réalisée à la main. Dans ce cas, le but est d'assister la localisation. Les principaux travaux proposent d'exploiter la trace d'exécution du programme. Elle est fournie par des outils de débogage et informe sur l'évolution de l'état du programme pendant son exécution.

Les techniques d'assistance au diagnostic déterminent un ensemble d'instructions suspectes du programme sous test. Pour cela, elles réalisent plusieurs exécutions du programme. Dans [Agrawal'95], Agrawal et al. proposent une technique qui exploite deux exécutions : une

correcte et une défaillante. Les instructions qui sont parcourues uniquement par l'exécution défaillante sont identifiées comme suspectes. D'autres travaux ont amélioré cette technique [Jones'02a, Eagan'01] par l'exploitation de toutes les exécutions disponibles et le classement des instructions dans un ordre de suspicion. Des travaux de Baudry et al. [Baudry'06b] contribuent également à l'amélioration de ces techniques. Ils considèrent la création d'un minimum de cas de test permettant d'obtenir suffisamment d'exécutions tout en réduisant le coût global du test.

Les travaux de « reverse engineering » permettent d'obtenir un modèle structurel du programme à partir de son implantation. Dans ce cas, l'appréhension du programme au niveau modèle facilite la correction des erreurs. Les travaux de Delamare et al. [Delamare'06] participent en plus à la localisation des erreurs en fournissant un modèle *comportementale* du programme à partir de traces d'exécution.

2.3.2 Analyse de mutation

L'analyse de mutation est une technique pour la qualification de données de test fondée sur l'injection d'erreurs. Nous étudions l'adaptation de l'analyse de mutation au test de transformations de modèles dans le chapitre 3. Par ailleurs, l'insertion systématique d'erreurs dans un programme peut être utilisée pour la vérification expérimentale de techniques de test. La connaissance du nombre, de l'emplacement, et de la nature des erreurs dans le programme permet de contrôler l'efficacité de techniques de test ou de diagnostic [Jones'02b]. Nous réalisons une telle exploitation dans le chapitre 5 pour qualifier des oracles et des composants de transformation de modèles.

L'analyse de mutation a été proposée par DeMillo [DeMillo'78]. Cette technique peut être employée dans deux buts : évaluer la qualité d'un ensemble de cas de test et assister la génération de test. Dans les deux cas, la technique consiste à créer un ensemble de versions erronées du programme sous test, appelées *mutants*, puis à exécuter un ensemble de cas de test sur chacun des mutants. Un mutant est une copie du programme sous test qui diffère par l'injection d'une seule erreur. En pratique, l'ensemble des mutants est créé en appliquant sur le programme des *opérateurs de mutation* qui correspondent à différents types d'erreur (remplacement d'opérateurs arithmétiques ou logique, changement de signe des constantes et variables, remplacement d'appel de méthode…). Les erreurs sont injectées de manière maîtrisée et systématique dans le programme.

Le résultat de l'exécution des cas de test avec les mutants permet d'une part d'évaluer l'efficacité de l'ensemble de cas de test par la proportion de mutants détectés (appelée le *score de mutation*). D'autre part, l'analyse des erreurs qui n'ont pas été détectées permet de guider la génération de nouveaux cas de test. Ceci consiste à analyser les erreurs des mutants, puis à créer des données de test spécifiques pour provoquer un comportement différentiable du mutant et du programme sous test. Si un cas de test peut détecter l'erreur d'un mutant, il *tue le mutant*, sinon le *mutant est vivant*.

Chaque mutant ne comporte qu'une seule erreur simple. Cette limitation est fondée sur deux hypothèses :

- le programmeur est compétent,
- l'effet de couplage.

La première hypothèse est fondée sur la compétence du programmeur. Il peut écrire un programme incorrect mais qui est néanmoins « proche » de la version correcte (les modifications pour le corriger restent mineures). Dans le cas d'erreurs grossières, les tests les plus simples mettraient facilement à jour l'incompétence du programmeur. L'hypothèse sur l'effet de couplage dit que si des tests peuvent détecter les erreurs simples, alors ils pourront détecter des erreurs plus complexes. Offutt définit les fautes simples et complexes commises par le programmeur dans [Offutt'92] :

Faute simple, faute complexe : *Une faute simple est une faute qui peut être corrigée par une seule modification d'instruction [du programme]. Une faute complexe est une faute qui ne peut pas être corrigée par une seule modification d'instruction.*

Dans ce même article, Offutt étudie l'hypothèse sur l'effet de couplage. Il compare l'exécution de cas de test efficaces avec des mutants ne comportant qu'une seule erreur et des mutants comportant plusieurs erreurs. Les cas de test détectant des erreurs simples détectent aussi des erreurs combinant plusieurs erreurs simples, ce qui tend à valider l'hypothèse sur l'effet de couplage.

Plusieurs travaux portent sur la comparaison de l'efficacité de l'analyse de mutation avec des critères de couverture de flots de données [Wong'93, Offutt'96b, Mathur'94, Frankl'97]. Par exemple, Offutt et al. mettent en évidence que les cas de test construits avec l'analyse de mutation satisfont plus facilement les critères de flot de données. Ils découvrent également plus d'erreurs alors que dans les deux cas l'effort est le même pour créer des données de test. Dans [Andrews'05], les auteurs ont expérimentés l'analyse de mutation avec des programmes contenant de véritables erreurs. Ces erreurs n'avaient pas été injectées à des fins expérimentales mais avaient été insérées pendant le développement de huit logiciels existants. Les résultats qu'ils obtiennent confirment l'intérêt de l'analyse de mutation pour la qualification de données de test capable de détecter les véritables erreurs. Ils appuient également l'hypothèse qui considère que le programmeur est compétent.

Les erreurs injectées en appliquant les opérateurs de mutation correspondent à différents types d'erreurs de programmation. Ces types ont été identifiés en observant les pratiques des programmeurs et en analysant les erreurs détectées lors de tests. La plupart des travaux produisent des opérateurs de mutation dédiés aux langages d'implantation des programmes sous test : C [Barbosa'01, Agrawal'89], ADA [Offutt'96c], Java [Ma'02]. Ces opérateurs s'appuient sur la syntaxe des langages pour injecter les erreurs. La mise au point des opérateurs dépend du paradigme de programmation utilisé : procédural [Agrawal'89], orienté objet [Ma'02, Kim'01, Chevalley'01, Alexander'02], ou plus récemment la programmation par aspect [Ferrari'08]. Les opérateurs peuvent être dédiés à un niveau de test : unitaire [Baudry'00a], intégration [Hoijin'98, Ghosh'01, Delamaro'01]. Dans le chapitre suivant, nous proposons des opérateurs de mutation spécifiques aux transformations de modèles.

Face à l’augmentation du nombre d’opérateurs de mutation, certains travaux se sont intéressés à minimiser ce nombre. Ainsi, Offutt étudie le sous-ensemble suffisant pour les langages procéduraux dans [Offutt'96a], de même Borbosa effectue ce travail pour le langage C [Barbosa'01].

Des travaux récents ont mis au point des opérateurs pour l’ingénierie des modèles. Dans [Trung'05], Trung et al. dressent une taxonomie des fautes qui peuvent être commises dans les diagrammes UML. Dans [Sen'06], Sen et al. les exploitent pour la génération de modèles en entrée de transformations de modèles.

Des travaux ont montré qu’augmenter le score de mutation d’un ensemble de données de test au-delà de 95% est généralement difficile [Frankl'97, DeMillo'93]. Leurs auteurs en déduisent que ces ensembles sont suffisamment efficaces. D’autres expériences montrent que les cas de test fournis par le testeur détectent souvent entre 50 et 70% des mutants [Baudry'03]. La création des données de test pour tuer près de la moitié des mutants (de 50% à 95%) est fastidieuse, puisqu’il faut analyser pourquoi leurs erreurs ne sont pas révélées, puis produire une donnée de test pour tuer chacun d’entre eux.

Dans [DeMillo'91], les auteurs proposent une technique pour la génération automatique de cas de test en exploitant l’analyse de mutation. Ils développent l’idée de fixer comme objectif de test la position de l’erreur dans le programme, et de remonter toutes les contraintes sur les données entre cette position et l’entrée du programme. Avec cette méthode, ils générent une donnée de test qui atteint la position de l’erreur et tue le mutant. Un outil pour le langage Fortran supporte ce travail [DeMillo'93]. Dans [Chevalley'01] les auteurs détaillent une technique de génération automatique pour des mutants de programmes JAVA fondée sur le mécanisme d’introspection du langage.

De nombreux outils ont été proposés pour mettre en œuvre l’analyse de mutation : Mugamma [Kim'06], Mujava [Ma'05], CREAM system [Derezinska'08], NMutator [Baudry'05], JMutator [Baudry'06b], Jester [Moore'01].

2.3.3 Oracle de test de logiciel

L’oracle produit le verdict du test (passe ou échoue) en détectant les défaillances dans l’exécution d’une donnée de test. La problématique de l’oracle est très peu traitée dans la littérature. Elle forme une part très faible des travaux qui traitent du test [Xanthakis'00].

Il existe différents moyens de construire l’oracle d’un cas de test. Ils dépendent de l’information disponible sur laquelle il peut se baser, de l’effort que doit réaliser le testeur pour le construire. Dans cette section nous énumérons ces différents moyens et les oracles correspondants en se basant sur la littérature existante.

La définition d’un oracle dans un contexte « idéal » est la suivante :

Soit P un programme sous test qui implante une spécification S. Soit In(P) et out(P) les domaines d’entrée et de sortie de P. L’oracle est une fonction of_S *de* $In(P) \times Out(P)$ *qui renvoie le verdict sous la forme d’un booléen tel que :*

$$\forall x \in In(P), of_S(x, P(x)) = vrai \Leftrightarrow P(x) = S(x)$$

Il s'agit d'une utilisation « idéale » de l'oracle car elle dépend de la possibilité d'exécuter la spécification S. Cet idéal est utopique car si S était exécutable le développement de P aurait peu d'utilité.

La détection d'une défaillance peut se faire à plusieurs niveaux :

- Au niveau de la donnée de sortie : le résultat obtenu par l'exécution d'une donnée de test doit correspondre à ce qui a été spécifié.
- Au niveau du comportement du programme : pendant l'exécution du test, l'état du système peut être important et doit être contrôlé par l'oracle. Par exemple, il ne s'agit pas seulement de vérifier le résultat d'un programme sans effet de bord mais il faut aussi vérifier qu'il n'a pas modifié le reste du système. Autre exemple : l'exécution d'un programme parallèle doit avoir été répartie sur plusieurs ressources.

La construction de l'oracle est basée sur la spécification. Il s'agit d'extraire de la spécification le moyen de vérifier que l'exécution d'une donnée de test est correcte. La formation d'oracles à partir de ces informations est réalisée plus au moins automatiquement. Nous classons les différentes approches dans l'ordre de leur niveau d'automatisation.

a- Manuel

Le premier oracle disponible est manuel : le testeur vérifie par lui-même qu'il n'y a pas eu de défaillance. Cette solution n'est praticable qu'à faible échelle quand le nombre de cas de test est faible (le test unitaire de fonction élémentaire par exemple). Il est donc nécessaire d'automatiser l'oracle pour éviter que le coût du test n'explose.

b- Comparaison avec la sortie attendue

De nombreux travaux considèrent que l'oracle est le résultat attendu. Le verdict est produit en comparant cet oracle au résultat obtenu [Panzl'78, Hamlet'77]. Pour cette raison, la plupart de ces travaux considèrent que les cas de test se composent d'une donnée de test et du résultat attendu [IEEE'04]. Dans ce cadre, l'exécution de l'oracle est automatique : la comparaison du résultat obtenu avec le résultat attendu. Cependant, produire le résultat attendu manuellement peut être coûteux, en particulier si les cas de test sont nombreux.

Weyuker a retenu cette définition dans [Weyuker'82]. Elle précise que dans de nombreux cas l'oracle (en tant que résultat attendu) n'existe pas. Nous retenons certains cas :

- Le résultat attendu ne peut pas être prédit. Par exemple, si la spécification n'est pas complètement déterministe, l'exécution d'une même donnée de test pourrait renvoyer plusieurs sorties.
- Le résultat attendu pourrait être obtenu mais pas avec un coût raisonnable. Cette hypothèse est particulièrement vraie aujourd'hui où les systèmes testés sont extrêmement grands, complexes, et produisent des données grandes et complexes également. De très

nombreux cas de test sont nécessaires et l'effort pour construire les résultats n'est raisonnable que pour des systèmes hautement critiques.

Elle remarque que même sans le résultat attendu il est possible de détecter des défaillances. Pour cela, il est possible d'employer des *oracles partiels*. Par exemple, si le solde d'un compte positif après plusieurs crédits positifs ne peut être prédit, il est possible de vérifier que le solde ne soit pas négatif. Un *oracle partiel* permet de déterminer si un résultat est incorrect ou partiellement correct.

c- Oracle partiel

La plupart des approches sont semi-automatiques et réclament le travail du testeur pour écrire des oracles qui réalisent des vérifications partielles. Un oracle ne vérifie pas l'intégralité des données produites mais il porte sur certaines propriétés de ces données en considérant une sous-partie de la spécification.

Dans [Hoffman'99], Hoffman propose d'utiliser des oracles en les basant sur des heuristiques. Il s'agit de définir des heuristiques pour sélectionner uniquement des tests significatifs à vérifier. Par exemple, pour la vérification d'une fonction sinus il suffit de vérifier les valeurs significatives 0, π, 2 π, 3π, 4π, et la tendance (augmentation, diminution) de leurs intervalles. Le choix de l'heuristique est primordial et risqué car de nombreuses erreurs peuvent être omises. Dans l'exemple, une fonction en dent de scie qui a les mêmes valeurs significatives et les mêmes tendances peut induire en erreur l'oracle. Dans [Hoffman'99], Hoffman essaie de réaliser une taxonomie des oracles qui prédisent les résultats attendus. Sa classification ne se base pas sur le moyen de créer un oracle à partir de la spécification et de la donnée de test, mais plutôt de sélectionner un sous-ensemble de données de test pour minimiser l'effort d'écriture d'oracles en maintenant la qualité des tests à détecter des erreurs.

Une autre grande famille d'oracle partiel exploite des contraintes, nous la détaillons dans la section suivante. Nous consacrons une section de ce chapitre aux contraintes car elles interviennent dans différents travaux importants de cette thèse (l'oracle au chapitre 4, l'outillage et l'intégration aux composants de la spécification au chapitre 5).

d- Automatique à partir d'une spécification formelle

Les travaux les plus avancés qui traitent de la génération automatique d'oracles se basent sur une spécification formelle. Certains travaux proposent de générer les résultats attendus mais la plupart considèrent l'oracle au sens le plus large : un programme analysant le résultat pour produire le verdict. En effet, si certains chercheurs considèrent l'oracle au sens littéral du terme : un « devin » qui prédit (ou qui est) le résultat attendu [Chen'03], ils reconnaissent qu'en l'absence de résultat attendu, le déroulement du test doit être vérifié par une méthode produisant différemment le verdict autrement que par comparaison. Dans ce cas, l'oracle analyse le résultat pour produire le verdict.

TOG (Test Oracle Generator) [Peters'98] génère des oracles à partir d'une spécification du programme sous forme de relations entrée/sortie appelées « tabular expressions ». Le

programme sous test est spécifié avec sa signature, les variables externes qu'il emploie, et sa sémantique sous la forme de relation entre les états d'entrée et de sortie de l'exécution du programme. L'oracle est produit sous la forme de procédures C qui analysent les données renvoyées par l'exécution de données de test.

En se basant sur une interprétation symbolique de la spécification, Richardson et al. [Richardson'92] présentent une méthode pour générer des oracles pour les systèmes réactifs à partir de leur spécification. Ils peuvent appliquer leur méthode quand dans la spécification la partie manipulation de données est écrite en Z et les aspects temporels sont exprimés en RTIL.

De la même manière, la plupart des travaux qui proposent des méthodes de génération d'oracle se base sur une spécification formelle du programme sous test. Les spécifications utilisées pour la génération d'oracle sont exprimés avec $MITL_{[0,d]}$ [Wang'05], Graphical Interval Logic (GIL) [Dillon'94], en Z [Yang'94, Richardson'92]. Le passage d'une spécification textuelle à cette spécification formelle est pourtant l'étape la plus critique car sa complexité est proche de celle de l'implémentation.

Dans [Pickin'02], les auteurs proposent une technique exploitant une spécification fonctionnelle du programme sous test. Cette technique utilise les diagrammes de classes et les machines d'états spécifiant le comportement du programme pour construire un système de transitions représentant le comportement global du programme. Un simulateur génère alors des cas de tests (donnée de test et oracle) à partir d'objectifs de test décrits par des diagrammes de séquence. Cette technique est supportée dans l'atelier de génie logiciel UMLAUT [Ho'99] qui utilise TGV [Jéron'98] pour la génération de cas de test exécutables.

L'inconvénient de toutes ces approches est de se baser sur des formes particulières de spécification : définies de manière formelle. Elles ne traitent pas du passage d'une spécification textuelle, ce qui est la majorité des cas, à une version formelle.

e- Automatique

Il existe peu de moyen pour automatiser complètement l'oracle :

- avec un programme de référence [Chapman'82] quand les changements dans le système sont non fonctionnels. Ce cas se présente dans deux situations particulières : la migration de système logiciel, l'amélioration non fonctionnelle (performance, etc.) d'un programme, dans ce cas la non régression est vérifiée.
- avec un programme inverse [Xanthakis'00]. Par exemple, quand le programme calcule la racine carrée, le carré du résultat doit renvoyer la donnée de test. Cette solution n'est pas toujours applicable car tous les programmes ne sont pas inversibles. Une condition nécessaire est que le programme soit injectif, ce qui n'est pas une condition suffisante.

En revanche ces deux solutions ne sont automatiques que si le programme de référence ou inverse est disponible. Sinon le testeur doit développer (ce qui à priori n'est pas son rôle) ce programme, ce qui rend le test du système aussi coûteux que son développement.

Dans le chapitre 4, nous proposons plusieurs fonctions d’oracle pour le test de transformations de modèles. Elles appartiennent à ces différentes catégories. Nous n’abordons pas l’approche manuelle car dans ce cas aucune fonction (d’oracle) n’est nécessaire, et surtout parce que le niveau de complexité des transformations ne permettraient pas de réaliser la vérification des nombreux modèles de sortie manuellement. Nous n’exploitons pas non plus d’oracle généré à partir d’une spécification formelle de transformation, car il n’y a pas encore de consensus sur la forme de cette formalisation et surtout sur la manière de l’obtenir.

2.4 Contraintes et composants

2.4.1 Conception par contrat

La notion de contrat a été définie pour exprimer les droits et les devoirs des classes dans le paradigme orienté objet. L’expérience a montré que l’expression des contraintes de fonctionnement d’un logiciel sous forme de contrats non ambiguës est une approche de conception valide [Jézéquel'97]. Bertrand Meyer a nommé cette approche la conception par contrats (Design by Contract) [Meyer'92a]. La conception par contrats impose aux concepteurs de séparer les responsabilités du client et celles de l’implantation d’une méthode. Ces contrats sont définis par des expressions booléennes nommées pré-condition et post-condition d’une méthode, et invariant d’une classe. Le client d’une classe appelle les méthodes en respectant les pré-conditions et les invariants de cette classe. Dans ce cas, la méthode assure qu’après son exécution, ses post-conditions et les invariants de sa classes sont respectés. Ces contrats sont vérifiés dynamiquement pendant l’exécution du système. Ils signalent une erreur quand ils sont violés.

Plusieurs langages de programmation supportent la conception par contrat : nativement dans Eiffel, avec des extensions comme Contract4J pour Java, par exemple.

Dans l’ingénierie des modèles, le standard pour l’expression de contrats est OCL (Object Constraint Language [OMG'97b]. Il a été développé en même temps qu’UML pour contraindre les différents diagrammes UML. En particulier il permet de définir des invariants de classes, des pré et post-conditions de méthodes. Une contrainte OCL est une expression booléenne sans effet de bord qui est fortement typée et écrite dans un contexte. Un contexte peut être soit une classe, soit une opération. Les trois contraintes suivantes présentent un exemple d’utilisation basique d’OCL dans un modèle de programme bancaire :

```
context CompteBancaire::créditer(montant : Integer)
   pre : montant > 0 //avant l’opération de crédit le montant doit être positif
   post : solde = solde@pre + montant   //après l’opération, le nouveau solde
                                        //doit être = à l’ancien + le montant
context CompteBancaire
   inv : self.solde >= self.autorisationDécouvert
           //à tout instant le solde d’une instance de CompteBancaire ne doit
           //pas être inférieur à l’autorisation de découvert
```

Le formalisme introduit par OCL est préférable au langage naturel car il supprime les ambiguïtés et peut être exploité automatiquement dans les programmes. L'introduction du MOF et d'UML 2.0 ont conduit à l'adaptation du méta-modèle d'OCL [OMG'03] pour permettre d'associer des contraintes aux éléments des différents méta-modèles conformes au MOF.

2.4.2 Exploitation des contraintes pour le test

Les contrats obtenus par conception par contrats sont utiles pour le test de programmes. Ils représentent une version exécutable de la spécification et permettent de relever la présence d'erreurs quand ils sont violés à l'exécution. Leitner et Ciupa ont réalisé différents travaux dans l'équipe de Meyer pour exploiter les contrats comme oracle du test [Meyer'07, Leitner'07, Ciupa'08, Ciupa'05]. Leur approche de test exploite la présence des contrats créés pendant le développement d'un système suivant la conception par contrat. Cette approche permet de s'affranchir de la construction des oracles pendant la phase de test et de considérer uniquement l'automatisation de la génération de données de test. L'inconvénient des contrats issus de la conception par contrat réside dans le fait qu'ils sont écrits pendant le développement, par le développeur. Si celui-ci commet des fautes il y a des risques qu'il les fasse à la fois dans le programme et dans les contrats. Il n'y a pas la séparation nécessaire entre le développement et le test.

L'exploitation des contrats permet néanmoins de récupérer à moindre coût des oracles à condition de s'assurer de leur correction par rapport à la spécification. Cette utilisation des contrats a été retenue dans différents travaux étudiant l'oracle et considérant les contrats au sens plus large comme des contraintes [Le Traon'06, Briand'03]. Elles peuvent aussi bien être des assertions inclus dans un programme de test, que des contrats d'un programme sous test, embarqués ou non dans le code du programme.

Les langages permettant la définition d'assertions pour la vérification d'états du programme existent depuis longtemps : Anna [Luckman'85] pour Ada, APP (Annotation Pre-Processor) [Rosenblum'95] pour C, Eiffel [Meyer'92b]. D'autres systèmes qui intègrent le support d'assertions pour le test unitaire ont été développés plus récemment : JUnit pour Java [Cheon'02], SPARK pour ADA [Barnes'03], Spec# pour C# [Barnett'05].

Il existe également des travaux sur l'utilisation d'OCL pour la vérification statique de propriétés sur des modèles. Un exemple est pOCL [Millan'08] qui propose une extension du langage OCL pour permettre de mettre en relation plusieurs modèles dans une même contrainte par la définition de contexte multiple.

2.4.3 Composant dans l'IDM

La notion de composant MDA est apparue comme une entité essentielle pour le succès du déploiement du MDA. Dans [Bézivin'03d], ce type de composant est défini comme "a packaging unit for any artifact used or produced within an MDA-related process". Bézivin et al. introduisent plusieurs concepts et entités présents dans un contexte MDA et fournissent plusieurs exemples de composants MDA. S'ils présentent les transformations de modèles comme le plus important composant, ils considèrent également les méta-modèles, les profils

UML. Bendraou et al. [Bendraou'08] proposent aussi de considérer les composants MDA pour factoriser les artéfacts du développement dirigé par les modèles. Fondement et al. [Fondement'04] proposent d’utiliser des composants IDM (MDE component) pour répondre à des besoins méthodologiques. Ils analysent quels sont les différentes technologies disponibles pour améliorer la réutilisation et définir des entités qui permettent d’automatiser un développement dirigé par les modèles. Ils considèrent les dépendances entre packages, les profils, les transformations, les méta-modèles, et ils montrent que ces mécanismes permettent l’assemblage de composants mais ont des limites quant à l’adaptation de ces composants. Comme dans les travaux de Bézivin, ce travail considère une large définition des composants IDM. Nous parlerons dans le chapitre 5 uniquement de composants de transformations de modèles.

Nous traiterons ce dernier point en exploitant une méthodologie définie dans l’équipe Triskell pour la définition de composants autotestables [Jézéquel'01]. Cette méthodologie a déjà été étudiée dans le paradigme orienté objet [Baudry'03]. Nous l’adapterons à l’ingénierie des modèles et l’utiliserons pour l’expérimentation de nos travaux dans le chapitre 5. Cette solution consiste à embarquer dans un composant son implantation, ses tests, et sa spécification. Les composants ont la possibilité d’exécuter leurs tests. Pour être autotestable, le composant embarque également sa spécification sous forme exécutable avec des contrats. Le Traon et al. [Le Traon'06] définissent qu’un composant est *vigilant* s’il embarque des contrats suffisamment précis pour détecter les états erronés du composant pendant l’exécution. Il est possible alors de dynamiquement contrôler que le composant accepte les modèles qu’on lui présente et produit des modèles corrects. En particulier, les contrats assurent de l’appartenance des modèles manipulés au domaine d’entrée et au domaine de sortie de la transformation, ce qui permet d’assembler plusieurs composants.

2.5 Test de transformations de modèles

Dans cette section, nous étudions quelques travaux qui concernent le test spécifiquement de transformations de modèles.

Si l’utilisateur final d’une transformation constate des erreurs dans les modèles générés, il va en chercher la provenance dans ses modèles sources et pas dans la transformation qu’il n’a pas de raison de connaître. Il faut donc avoir confiance dans la transformation et celle-ci doit subir une phase de vérification pendant son développement comme tout système logiciel. Cette phase de vérification est d’autant plus importante pour les programmes de transformations de modèles. En effet :

- Une erreur dans la transformation peut se propager dans les modèles jusqu’à l’implantation, son origine sera très difficile à localiser depuis cet endroit.
- Une transformation de modèles doit être réutilisée de nombreuses fois pour justifier l’effort qu’on consacre à sa création et à sa mise au point. Si la transformation est défectueuse alors elle risque de générer les mêmes erreurs un certain nombre de fois.

La réalisation de phases de test est donc justifiée [Fleurey'04, Bézivin'03b]. Des techniques spécifiques doivent être mises au point pour cela car les programmes de transformations de modèles sont d'emblée des programmes complexes : ils se basent sur des méta-modèles et les données d'entrée sont des modèles. Nous sommes donc confrontés à de nouveaux problèmes pour la génération, la sélection, et la qualification des données d'entrée ainsi que sur la construction d'oracles.

2.5.1 Travaux sur le test de transformations de modèles

Si les transformations de modèles sont un sujet dont l'étude s'est bien développée, il y a peu de travaux traitant de la vérification de ce type de programme. Nous les présentons dans cette sous-section.

Dans [Steel'04] les auteurs présentent un retour d'expérience issue de la validation d'un moteur de transformation de modèles déclarative. Pour le test de ce moteur, chaque donnée de test est une transformation et un modèle d'entrée. La transformation est exécutée par le moteur avec ce modèle et le modèle de sortie obtenu est comparé avec le modèle de sortie attendu. Les auteurs remarquent que ce processus de test est très similaire à un processus de vérification de transformations de modèles. Les auteurs discutent des différents problèmes rencontrés et des solutions envisagées pour les résoudre. La première difficulté identifiée est la complexité de manipuler des modèles comme donnée de test qui impose de disposer d'outils pour éditer et sérialiser les modèles. Le problème du critère de test est également évoqué mais la solution adoptée par les auteurs n'est ni systématique ni automatisable.

Küster a réalisé différents travaux sur le test de transformations de modèles. Il a commencé par aborder la validation de transformations de modèles UML en exploitant une spécification formelle de la transformation sous forme de règles de transformation de graphes attribués et typés [Küster'03]. Dans [Küster'04, Küster'06a], l'auteur identifie le besoin de techniques pour la vérification et le test de transformations de modèles. La technique qu'il permet la vérification de propriétés syntaxiques pour des transformations de modèles décrites par un ensemble de règles de transformations. Les vérifications effectuées dans son approche considèrent la correction syntaxique des règles, la convergence, et la terminaison d'un ensemble de règle.

Dans [Küster'06b], Küster et al. choisissent d'utiliser une approche boîte blanche (test structurel). Ils définissent un langage de template pour générer des modèles de test en se basant sur la structure des règles de la transformation. Cette méthode est fortement dépendante du langage de l'implantation, elle doit être adaptée ou complètement redéfinie pour un autre langage. L'avantage d'une approche boîte blanche est d'imposer que chaque étape de la transformation soit individuellement considérée. Le testeur doit créer des modèles de test pour chacune de ces étapes. De cette manière, le critère de test est plus détaillé et efficace que des critères boîte noire qui ne peuvent pas s'appuyer sur la mécanique interne de la transformation.

Dans [Podnieks'05], Podnieks étudie l'exactitude d'une version simplifiée de la transformation de modèles class2rdbms. Il impose que les modèles d'entrée (diagramme de classes) et de sortie (schéma RDBMS) soient sémantiquement équivalents. Cette condition

permet de produire une transformation inversible (contrairement à la spécification complète de la transformation class2rdbms) qui facilite son test. Narayanan et al. étudient [Narayanan'08] la vérification de transformations de modèles sous forme de graphes mais ils ne considèrent qu’un type de transformations : celles dont les modèles d'entrée et de sortie sont sémantiquement équivalentes. Nous ne restreignons pas notre étude aux transformations de cette nature particulière. Nous considérons pour nos expériences une version complexe de class2rdbms qui ne permet pas d’exploiter cette condition (d’équivalence sémantique) par exemple.

Différents travaux traitent de la création de modèles de test :

Dans [Fleurey'04, Fleurey'07a], Fleurey et al. proposent de transposer l’approche de [Andrews'03] pour le test fonctionnel de transformations de modèles. Une transformation de modèles est créée à partir des méta-modèles sources et cibles. Puisque les méta-modèles sont décrits par le MOF, ils sont semblables aux diagrammes de classes UML (avec l’utilisation de classes, d’attributs, d’héritage, d’associations). Les critères de couverture proposés dans [Fleurey'04, Fleurey'07a] peuvent être utilisés pour couvrir les méta-modèles. Ceci nous donne des critères génériques pour produire des données pour le test de n’importe quelle transformation. Cependant ces critères sont uniquement basés sur la capacité des modèles de test à couvrir les méta-modèles sources de transformations. Même si ces critères fournissent une première mesure de la qualité des modèles de test, cette approche doit être améliorée pour considérer l’intention de la transformation testée.

Dans [Brottier'06], Brottier et al. présentent un processus et un outil pour la génération automatique de modèles de test satisfaisant les critères de test de [Fleurey'07a]. Le processus se base sur les critères pour générer des modèles incomplets car non conformes au méta-modèle source. Puis il complète les modèles pour en faire des modèles de test appartenant au domaine d’entrée de la transformation. Ce domaine d’entrée est spécifié par un méta-modèle effectif qui est un sous-ensemble du méta-modèle source dans lequel seules les méta-classes impliquées dans la transformation sont conservées. Lamari et al. [Lamari'07] proposent un outil génère automatiquement ce méta-modèle effectif. Pour appliquer cette méthode, la spécification de la transformation doit être écrite formellement en MTSpecL, le langage proposé dans l’article.

Dans [Sen'07], le travail pour obtenir des modèles complètement conforme à un méta-modèle [Brottier'06] est repris avec une approche différente. La méthodologie proposée permet en plus de combiner différentes contraintes exprimées en Prolog. De cette manière, il est possible d’obtenir des modèles complets à partir des critères définies par Fleurey et al., du méta-modèle source d’une transformation, et de contraintes complémentaires comme les pré-conditions de la transformation. Cependant Prolog ne permettant pas de contraindre suffisamment la génération, une alternative est fournie avec l’outil de génération de modèles Cartier [Sen'08] qui exprime l’ensemble des paramètres de génération en Alloy. Alloy est un langage avec une logique de premier ordre relationnelle qui supporte la clôture transitive et les quantifieurs.

Dans [Anastasakis'07], Anastasakis et al. exploitent le solveur de contraintes d’Alloy [Jackson'08] directement pour vérifier des transformations de modèles. Ils réalisent la

vérification en deux étapes. Dans la première étape, les méta-modèles source et cible sont transformés en Alloy ainsi que les règles de la spécification de la transformation sous test. Dans la deuxième étape, si le solveur n'est pas capable d'obtenir une instance de la transformation à partir d'un modèle de test et du modèle correspondant, alors une défaillance est révélée. Si la deuxième étape du processus est intéressante, les hypothèses de la première étape limitent la portée de cette méthode de test. Tout d'abord, le travail pour transformer les règles de la spécification en Alloy est difficile et donc source d'erreur. Cet obstacle est similaire aux précédents travaux cités sur la génération de test qui se base sur une spécification formelle. Ensuite, Alloy utilise une logique de premier-ordre qui peut être insuffisante pour exprimer la totalité de la spécification d'une transformation. Dans ce cas, il n'est pas certain que la méthodologie produira des résultats exacts (sans faux-négatif, ni faux-positif) ou alors en étant fortement limité par la complexité de la transformation à tester.

Dans [Ehrig'06], Ehrig et al. proposent également une technique pour générer automatiquement des ensembles de modèles conformes à un méta-modèle. Leur approche est basée sur une dérivation automatique de grammaires de graphes à partir du méta-modèle. Ils analysent le méta-modèle et génèrent des règles pour créer et associer des instances des méta-classes concrètes du méta-modèle. Cette approche a un certain nombre de limitations. Les valeurs des attributs des objets générés ne sont pas initialisées. Les auteurs proposent d'utiliser un solveur de contrainte pour vérifier que le modèle produit satisfait les contraintes OCL définies dans le méta-modèle, mais ils n'exploitent pas les contraintes pour la génération, il s'agit d'une sélection à posteriori.

Dans [Darabos'06], Darabos et al. considèrent des règles de transformation de graphes comme spécification de la transformation sous test. Ils proposent de générer des modèles de test à partir de cette spécification. Leur technique se focalise sur le test du pattern matching qu'ils considèrent comme particulièrement critique dans le processus d'une transformation. Ils proposent plusieurs classes de fautes qui peuvent être commises pendant la mise en œuvre du pattern matching. Ils développent aussi une technique de génération de tests visant certaines erreurs.

D'autres travaux permettent d'aborder le test de transformations avec des approches formelles :

Dans [Poernomo'08], Poernomo exploite une spécification formelle de la transformation sous test pour en réaliser la preuve. La spécification de la transformation doit pouvoir être transcrite dans le langage formel. La spécification de la transformation ne doit donc pas imposer de construction impérative pour pouvoir appliquer cette méthode. Le travail de formalisation est important et peut entraver la mise en œuvre de ce type de preuve si la transformation est complexe. L'exemple de l'article ne considère d'ailleurs qu'une version de la transformation class2rdbms dont la spécification textuelle ne fait que trois lignes.

Dans leurs travaux, Varró et al. [Varró'03, Varro'02] proposent également de réaliser la vérification des transformations de modèles avec une approche formelle. Ils exploitent une

technique de « model checking » qu’ils appliquent à des transformations implantées avec des règles de transformation de graphes.

Dans [Giese'06], Giese et al. s’intéressent à la vérification formelle de propriétés critiques de transformations de modèles. Les auteurs utilisent des grammaires "triple graph grammars" (TGG) pour représenter le système et les transformations dans le but d’utiliser des méthodes formelles et représenter les propriétés critiques vérifiées par un outil de démonstration de théorème.

Les approches à base de formalisation peuvent être utiles en complément du test quand la spécification formelle est disponible ou peu coûteuse à obtenir. Nous ne considérons pas cette approche formelle dans nos travaux car il nous parait essentiel de disposer de moyens pour obtenir la version formelle de la spécification avant d’essayer de l’exploiter.

Dans le chapitre 4, nous traitons la problématique de l’oracle pour le test de transformations. Nous avons vu dans la section 2.4 que l’oracle de test de logiciels peut être réalisé à partir de contrats ou de résultat attendu. Pour cette raison, nous introduisons ici différents travaux qui exploitent les contrats. Puis nous consacrons le point suivant à la comparaison de modèles.

Dans [Cariou'04], Cariou et al. montrent qu’OCL est adapté pour spécifier les transformations de modèles (UML particulièrement). Il s’agit de pré et post-conditions pour la transformation et aussi de contraintes liant les modèles d’entrée et de sortie. Cela montre que les hypothèses que nous faisons sur les possibilités de formalisation de la spécification sont justifiées et nous permettent d’envisager l’utilisation des contrats pour l’oracle.

Dans [Solberg'06], Solberg et al. proposent des patterns pour exprimer les pré et post-conditions d’une transformation. Ces patterns sont des templates qui décrivent des propriétés attendues des modèles d’entrée et de sortie et qui peuvent être vus comme une spécialisation des méta-modèles source et cible. Les patterns de pré-condition sont utilisés pour contraindre la génération de modèles de test, tandis que les patterns de post-condition sont utilisés pour vérifier l’exactitude des modèles de sortie.

2.5.2 Comparaison de modèles pour l’oracle

Dans [Lin'05], Lin et al. identifient trois problèmes pour mettre au point des transformations de modèles. Le premier est la comparaison entre modèles qui permet de réaliser l’oracle lors de l’exécution des tests. Le second est la visualisation des erreurs pour réaliser le diagnostic après l’échec d’un test. Le troisième est le développement de techniques de correction d’erreurs adaptées aux langages de transformations de modèles pour permettre de corriger les erreurs détectées. Dans l’article, les auteurs proposent leur solution aux deux premiers problèmes avec un algorithme de comparaison de modèles inspiré de techniques de comparaison de graphes. Ils détaillent également un framework permettant d’organiser le test de transformations de modèles. Ils illustrent les différentes étapes du test d’une transformation de modèles avec un exemple. [Lin'04, Lin'07]

Les dépositaires de modèles et les technologies utilisées actuellement dans l’IDM stockent et manipulent les modèles comme des graphes d’objets. La complexité de ces structures de

données rend difficile la mise au point de techniques de comparaison de modèles efficaces et fiables. En effet, en considérant les modèles comme des graphes d'objets, leur comparaison atteint la complexité d'un algorithme NP-complet. L'approche de Alanen et Porres dans [Alanen'03] simplifie le problème en comparant deux versions d'un même modèle, elle profite ainsi des identifiants des objets. Leur méthode n'est pas utilisable dans une phase de test car les modèles comparés par l'oracle viennent de sources différentes et leurs objets n'ont donc pas les mêmes identifiants. La comparaison des modèles doit donc se baser sur leur structure et leurs attributs. D'autres algorithmes exploitent les méta-modèles et les structures qu'ils définissent. De tels algorithmes sont proposés dans [Xing'05, Lin'07]. La comparaison de modèles commence ainsi à être outillée par des outils comme EMF Compare [Eclipse Foundation'07].

Un outil de suivi des versions des modèles bénéficie au test s'il n'exploite pas leurs identifiants. Dans ce cas, il peut servir d'oracle en mettant en évidence les conflits entre le modèle obtenu et le modèle attendu. Ces conflits peuvent aider la détection des erreurs de la transformation. De plus, le suivi des versions des méta-modèles source et cible est utile pour les tests de régression : si nous connaissons les parties des méta-modèles qui évoluent, nous pouvons détecter quels modèles de test sont encore exploitables. CVS Model est un projet open source qui propose ce type d'outil de suivi de versions de modèles. Ses mécanismes de détection de conflits sont basés sur le travail de Reiter et al.[Reiter'07]

Dans les travaux de [Cicchetti'07, Cicchetti'08], Cicchetti et al. s'intéressent au retour d'information en cas de détection d'inégalité entre modèles. Ces informations sont utiles pour l'activité de localisation d'erreur. Ils étendent le méta-modèle des modèles comparés en ajoutant de nouvelles méta-classes correspondant aux différences possibles des objets : le changement, l'ajout, la suppression.

Dans [Kolovos'06], Kolovos et al. présentent une alternative à la comparaison complète du modèle de sortie avec le modèle attendu. Ils définissent par des règles la comparaison d'éléments du modèle de test avec le modèle de sortie. Cette méthode permet de faire des vérifications sur les mappings entre entrée et sortie. Par exemple dans le cas de la transformation class2rdbms, la comparaison d'une classe persistante avec une table est définie dans une règle vérifiant l'égalité de leurs noms, elle permet de tester la transformation des classes.

Nous n'avons pas connaissance de travaux traitant des modèles de faute d'une transformation. Dans le chapitre suivant, nous présentons notre adaptation de l'analyse de mutation en étudiant les erreurs que l'on peut trouver dans l'implantation d'une transformation.

3
Analyse de mutation

Utilisée à l'origine pour la qualification d'ensemble de modèles de test, nous employons l'analyse de mutation dans plusieurs travaux de cette thèse. Cette technique se base sur la création de versions erronées de la transformation pour mettre à l'épreuve des modèles de test. Ces versions erronées, nommées *mutants*, nous permettent également de mettre à l'épreuve les fonctions d'oracle que nous étudions au chapitre suivant. Enfin, l'approche en triangle des composants de transformations de modèles (introduite section 2.4 et étudiée en détail dans le chapitre 5) utilise l'analyse de mutation pour produire une mesure du niveau de confiance du composant [Mottu'06b]. Ainsi, l'analyse de mutation joue un rôle majeur pour l'avancée des travaux réalisés au cours de cette thèse. Elle forme également une base fondamentale pour les travaux en cours sur le test de transformation de modèles. Par exemple, nous l'exploitons pour comparer des critères de test.

Afin d'améliorer la pertinence de l'analyse de mutation pour le test de transformations de modèles, nous devons prendre en compte les spécificités des transformations. La principale contribution de ce chapitre est la proposition d'une adaptation de l'analyse de mutation au test de transformations de modèles. Une telle adaptation consiste à proposer de nouveaux *opérateurs de mutation* [Mottu'06a].

Les spécificités des transformations de modèles induisent des erreurs représentées par les opérateurs proposés. Pour identifier ces spécificités, nous nous affranchissons des préoccupations du langage d'implémentation. En effet, il n'y a pas de langage unanimement adopté pour implémenter des transformations de modèles. Il est plus important de considérer les transformations à un niveau sémantique que syntaxique. Ainsi, nos opérateurs sont relatifs aux traitements réalisés par la transformation sur les modèles d'entrée et de sortie conformément à leurs méta-modèles.

L'analyse de mutation peut être coûteuse en temps « humain » car il faut étudier les mutants pour créer manuellement les données qui vont mettre en évidence leurs erreurs. De même, un ensemble de modèles de test satisfaisant peut provenir d'une création excessive de nouveaux modèles de test. La réduction des étapes manuelles et du nombre de modèles de test produit sont deux autres problèmes traités dans ce chapitre.

Dans la section 3.1, nous étudions les spécificités de l'application de l'analyse de mutation au test de transformations de modèles pour mettre en évidence le besoin de l'adapter. Dans la section 3.2, nous introduisons dix opérateurs de mutation spécifiques. Dans la section 3.3, nous traitons les problèmes d'optimisations. Finalement, dans la section 3.4, nous illustrons cette section avec l'expérimentation menée sur la transformation `class2rdbms`.

3.1 Adaptation de l'analyse de mutation à l'ingénierie dirigée par les modèles

3.1.1 Une technique pour la qualification et l'amélioration des modèles de test

L'analyse de mutation est une technique de test qui a été mise au point pour évaluer un ensemble de données de test. L'évaluation porte sur la capacité des données de test à détecter des erreurs. La technique permet alors d'en améliorer l'efficacité et la capacité pour détecter des erreurs [Voas'92b, Offutt'96b]. Proposée en 1978 [DeMillo'78], la technique consiste à créer volontairement, de manière systématique et non aléatoire, un ensemble de versions erronées du programme sous test. Si un ensemble de données de test est capable de distinguer le comportement du programme de celui de ses versions erronées, il est satisfaisant sinon il doit être amélioré. Avec le vocabulaire de l'analyse de mutation, nous nommons ces versions erronées des « mutants ». Si une donnée de test met en évidence l'erreur d'un mutant alors elle « tue » ce mutant. Un mutant qui n'est pas « tué » est considéré « vivant » tant que l'erreur qu'il contient n'a pas été mise en évidence.

Dans cette sous-section, nous présentons en détail l'analyse de mutation, son processus, et les données utilisées. Nous rapprochons la technique de base de celle employée dans le contexte des transformations de modèles. Nous insisterons sur l'importance des opérateurs de mutation qui forment la base qualitative de la technique.

a- Processus global de l'analyse de mutation pour le test d'une transformation de modèles

L'analyse de mutation prend en entrée un *programme sous test*, et un *ensemble initial de données de test*. Pour le test de transformations de modèles, le programme sous test est la *transformation de modèles sous test*, alors que les données de test sont les *modèles de test*. La figure 3-1 représente le processus global de l'analyse de mutation sous la forme d'un pseudo-diagramme d'activité. Les parties grisées de la figure représentent les données essentielles intervenant dans le processus : la transformation sous test, l'ensemble de modèles de test, les mutants (de la transformation, vivants, et tués), et le résultat de l'analyse de mutation (le score de mutation).

La première étape consiste à créer un ensemble de mutants en appliquant au programme les *opérateurs de mutation*. Les opérateurs modélisent l'ensemble des erreurs à injecter. En pratique, chacun représente une classe d'erreur et l'injection d'une seule erreur produit un mutant. Un même opérateur peut être appliqué plusieurs fois au programme, autant de fois que l'erreur peut-être injectée, ce qui crée autant de mutants. Par exemple, en appliquant l'opérateur

de mutation AOR (Arithmetic Operator Replacement), les opérateurs logiques sont remplacés dans cette instruction d'un programme Java :

```
for(int i=0 ; i!=5; i=i+1){        Mutant 1:    for(int i=0 ; i!=5; i=i-1){
                                   Mutant 2:    for(int i=0 ; i!=5; i=i*1){
```

Les mutants sont ensuite exécutés un à un. Un modèle de test tue un mutant si la transformation réalisée par ce mutant se comporte différemment de la transformation sous test. Un *oracle* compare le modèle de sortie produit par un mutant avec le modèle produit par la transformation sous test. En cas d'inégalité, l'oracle statue que le mutant est tué par le modèle de test. Le mutant est également tué s'il n'est pas capable de transformer un modèle de test, alors que la transformation sous test en est capable. Si aucun modèle de test ne révèle l'erreur d'un mutant, alors celui-ci reste vivant.

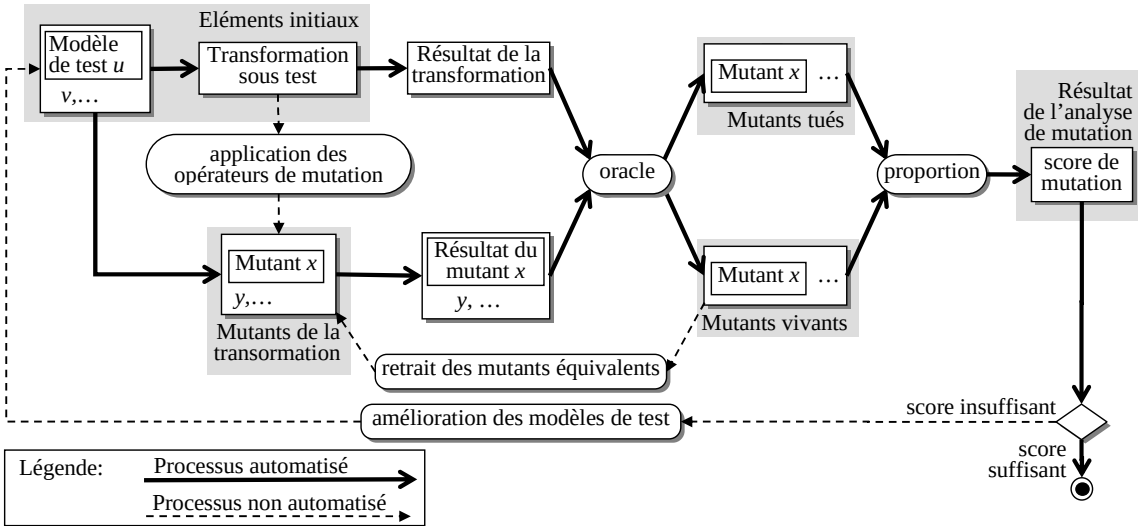


Figure 3-1 – Processus global de l'analyse de mutation pour le test de transformations de modèles

Les ensembles de mutant tués et vivants permettent de calculer le *score de mutation* de l'ensemble de modèles de test. Il s'agit de la proportion de mutants tués par rapport au nombre de mutants qu'il fallait tuer :

$$Score\ de\ mutation = \frac{nombre\ de\ mutants\ tués}{nombre\ de\ mutants\ tués + nombre\ de\ mutants\ vivants}$$

Ce score qualifie l'ensemble de modèles de test et mesure son efficacité en pourcentage de mutants tués et donc d'erreurs injectées que les modèles de test ont pu révéler. Ainsi un ensemble de modèles de test est relativement adéquat s'il distingue la transformation originale de tous ses mutants (non-équivalents). Le score de mutation est bénéfique même si par la suite aucune erreur n'a été révélée par les modèles de test qualifiés. En effet, il mesure à quel point la transformation a été testée en informant l'utilisateur de la qualité des modèles de test.

Si des mutants sont vivants à la fin du processus, il y a deux possibilités :

- Des mutants peuvent être *équivalents* à la transformation sous test et ils ne pourront donc pas être différenciés. Ces mutants sont supprimés de la liste des mutants à tuer. Nous en reparlons au point suivant.
- Soit l'ensemble courant de modèles de test peut être amélioré. De nouveaux modèles de test peuvent être générés ou ceux existants améliorés. Cette amélioration est nécessaire si le score de mutation est insuffisant.

b- Mutants équivalents

Tous les mutants générés ne sont pas des versions erronées de la transformation sous test. Il est possible de générer des mutants équivalents à la transformation sous test. Ils ne pourront pas être tués car ils produisent les mêmes modèles de sortie que la transformation quelque soit le modèle d'entrée.

Mutant équivalent *: Soit E le domaine d'entrée de la transformation sous test et S le domaine de sortie de la transformation. Si M est un mutant de T, la transformation sous test alors M est équivalent à T si et seulement si:* $\forall x \in E, M(x) = T(x)$ *avec :*

$$\begin{array}{l} M : E \to S \\ \quad y \mapsto M(y) \end{array} \quad et \quad \begin{array}{l} T : E \to S \\ \quad z \mapsto M(z) \end{array}$$

Par exemple, si dans un programme Java, l'opérateur de mutation LOR (Logical Operator Replacement) est appliqué sur cette instruction du programme original :

```
for(int i = 0 ; i != 5; i = i + 1){
```

alors un mutant équivalent est produit avec:

```
for(int i = 0 ; i < 5; i = i + 1){
```

Le remplacement de `!=` par `<` n'a pas d'influence puisque la variable i est incrémentée à partir de 0 et devra dans les deux cas être égale à 5 pour sortir de la boucle.

Il est nécessaire de mettre en évidence les mutants équivalents. En effet, ne pouvant pas être tués ils pénalisent le score de mutation. Ils doivent donc être retirés de l'ensemble des mutants.

La détection des mutants équivalents n'est pas une tâche entièrement automatisable. Dans des travaux orientés objet [Offutt'97], Offut et al. proposent une technique pour détecter automatiquement certains types de mutants équivalents. Il s'agit de considérer la génération de données de test comme un problème de résolution de contraintes. Pour tuer un mutant, il faut générer une donnée de test satisfaisant une contrainte particulière. Si la contrainte ne peut être satisfaite alors le mutant est équivalent. Cependant dans la plupart des analyses de mutation effectuées, cette détection est réalisée manuellement, ce qui augmente le coût.

3.1.2 Limitations des opérateurs de mutation classiques

La valeur de l'analyse de mutation est fondée sur la pertinence des mutants, donc sur la pertinence des erreurs introduites par l'application d'opérateurs de mutation. Dans cette sous-section, nous expliquons pourquoi les opérateurs existants ne sont pas appropriés pour les transformations de modèles. Selon le langage utilisé pour coder la transformation, les opérateurs classiques ou orientés objet peuvent être employés, cependant leur pertinence est limitée dans ce contexte particulier pour différentes raisons :

- *Pertinence des mutants produits* : les erreurs injectées représentent les erreurs qu'un *programmeur compétent* de transformations de modèles peut commettre. Les opérateurs de mutation classiques se basent sur les instructions du programme pour y injecter des erreurs qui ne sont pas systématiquement détectées à la compilation ou à l'exécution. Les transformations de modèles sont élaborées à plus haut niveau. Elles mettent en relation les méta-classes définies dans les méta-modèles source et cible. Dans class2rdbms, les `classes persistantes` et les `tables` sont mises en relation par exemple. Les erreurs d'une transformation sont d'abord commises au niveau sémantique de la définition de la transformation avant d'être implanter au niveau syntaxique de son code. Par exemple, il est possible de manipuler de mauvaises relations en se trompant dans la structure définie par le méta-modèle. Un mutant est plus pertinent s'il contient des erreurs qui reflètent le niveau de la définition de la transformation et pas seulement les spécificités de son codage.
- *Indépendance d'un langage d'implémentation* : les opérateurs de mutation doivent refléter les erreurs qui se trouvent dans l'implantation d'une transformation de modèles. Une contrainte pour définir ces opérateurs est qu'ils doivent définir les erreurs dans la syntaxe du langage utilisé. C'est de cette manière que sont définis les opérateurs de mutation classiques (orientés objet ou non, comme les exemples de la sous-section 3.1.1). Cependant, il existe aujourd'hui de nombreux langages de transformations de modèles qui ont leurs propres spécificités et qui sont très hétérogènes (orientés objet, déclaratifs, fonctionnels, mixtes). Il est donc préférable de rester indépendant d'un langage spécifique pour considérer les opérations effectuées par les transformations à un niveau sémantique et pas syntaxique.

En conséquence, pour créer des mutants pertinents sans contrainte de langage, nous nous concentrons sur la partie sémantique de la transformation plutôt que sur la syntaxe du langage de l'implantation. Dans ce cas, les opérateurs de mutation spécifiques représentent des *erreurs sémantiques* qui ne prennent en compte que la nature d'une transformation sans considération du code de l'implantation. A ce niveau sémantique les erreurs respectent l'hypothèse du couplage car elles sont « simples » (comme définie sous-section 2.3.2). Comme les erreurs classiques, elles ne sont pas systématiquement détectées à la compilation ou à l'exécution car elles sont introduites en se basant sur les méta-modèles.

Les opérateurs de mutation classiques (orientés objet ou non selon le langage d'implantation) restent utiles, pour vérifier directement le code ou la couverture d'instructions par exemple. Ils peuvent être employés en complément des opérateurs de mutation proposés dans la section 3.2.

3.1.3 Les transformations de modèles d'un point de vue sémantique pour l'analyse de mutation

Afin de proposer des opérateurs spécifiques aux transformations de modèles, nous analysons les activités impliquées dans le développement qui sont susceptibles d'entrainer des erreurs. La figure 3-2 illustre l'application d'une transformation de modèles. Un modèle d'entrée est transformé en un modèle de sortie. Chacun est conforme à son méta-modèle. La transformation ainsi que ses mutants sont basés sur ces méta-modèles pour opérer sur les modèles. Ainsi les opérateurs de mutation sont directement connectés à la notion de méta-modèle.

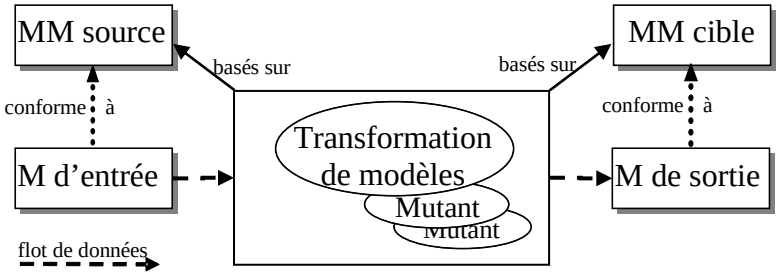


Figure 3-2 – Processus de transformation de modèles avec une transformation et ses mutants

Pour définir les opérateurs, nous essayons de répondre à la question :

De quelles natures sont les erreurs introduites dans l'implantation d'une transformation de modèles ?

Si nous considérons une transformation d'un point de vue sémantique nous retenons :

- qu'elle traverse un modèle d'entrée en naviguant les associations définies dans le méta-modèle d'entrée
- qu'elle sélectionne des éléments en fonction d'attributs, de types
- qu'elle produit et modifie les objets des modèles de sorties conformément au méta-modèle cible.

Par exemple, si une transformation traverse le modèle d'entrée à la recherche d'éléments à transformer alors une erreur peut entraîner la navigation d'une mauvaise association du méta-modèle ou la sélection des mauvais éléments d'une collection. L'analyse des erreurs d'une transformation s'appuie sur quatre opérations abstraites. Elles constituent les quatre traitements possibles sur les modèles d'entrée et de sortie d'une transformation :

Opération de navigation : *Une opération de navigation parcourt le modèle grâce à une relation définie sur son méta-modèle, et un ensemble d'éléments du modèle est obtenu. Si le méta-modèle MM définit deux méta-classes MC_x et MC_y (pouvant être la même) et une relation α de cardinalité n depuis MC_x vers MC_y alors la navigation de α à partir d'une instance de MC_x renvoie n instances de MC_y :*

$$\alpha : MC_x \rightarrow MC_y^{\,n}$$
$$x \quad \mapsto Y$$

$$\begin{array}{ll} \textit{alors pour } x, \textit{ instance de } MC_x, \textit{ la navigation de } \alpha \textit{ à partir de } x \ (x.\alpha) \textit{ renvoie } Y, & \\ \textit{tel que} & \forall y \in Y, y \textit{ instance de } MC_y \wedge y \in x.\alpha \\ \textit{et} & \forall y \in x.\alpha,\ y \in MC_y \end{array}$$

Opération de filtrage : *Une opération de filtrage sélectionne un ensemble d'éléments qui satisfont une condition. Si le méta-modèle MM définit une méta-classe MC alors une opération de filtrage sur un ensemble X renvoie un sous-ensemble Y, dont les éléments satisfont une condition C sur MC. f[C] est une opération de filtrage telle que :*

$$\begin{array}{c} f : MC^m \rightarrow MC^n \\ X \quad \mapsto \quad Y \end{array}$$

$$\textit{si et seulement si } \forall z \in Y, z \in X \wedge z \textit{ satisfait } C.$$

Opération de création : *Une opération de création crée des objets et initialise leurs attributs. C'est ainsi que les éléments du modèle de sortie sont créés à partir d'élément(s) extrait(s) du modèle d'entrée.*

Opération de modification: *Une opération de modification modifie les éléments existants du modèle de sortie. Cette opération est employée en particulier quand la transformation modifie le modèle d'entrée pour le retourner en sortie. Des éléments de modèles peuvent être supprimés ou leurs attributs modifiés.*

Nous illustrons l'utilisation de ces opérations sur la transformation class2rdbms. La figure 3-3 représente les modèles d'entrée et de sortie (respectivement à droite et à gauche des lignes verticales) successivement (de `(a)` à `(i)`) à différentes étapes de la transformation. La figure 3-3 représente les modèles sous la forme de diagrammes utilisant des syntaxes concrètes (proche de la notation UML avec le symbole * associé à un attribut pour exprimer que `is_primary` est à `true`, et les tables grisées avec le nom d'une colonne souligné pour indiquer que c'est une clé primaire `pkey`). Cependant les manipulations effectuées concrètement sur ces modèles se font à partir de leur structure telle qu'elle est définie par leurs méta-modèles (figure 2-7). Ainsi les modèles d'entrée et de sortie utilisés comme illustration ici dans la figure 3-3 (complètement en haut à gauche et en bas à droite respectivement) correspondent aux diagrammes d'instances de la figure 3-4.

Dans le diagramme de classes du modèle d'entrée, les classes persistantes sont sélectionnées et des tables correspondantes sont créées avec des colonnes correspondant aux attributs. Tout d'abord (a), le modèle d'entrée est navigué pour trouver les `classifiers` (b). Ces instances sont filtrées pour sélectionner celles de type `Class` (c). Ces instances de `Class` sont filtrées pour trouver celles qui ont l'attribut `is_persistent` à `true` (d). Une `table` est créée pour chacune d'elles (e). Puis les instances de `Class` persistantes sont naviguées pour trouver toutes leurs instances d'`Attribute` (f), celles avec `PrimitiveDataType` comme `type` sont filtrées (f aussi) et les instances de `Column` correspondantes sont créées (g). Finalement chaque colonne

est filtrée (h) selon sa correspondance avec une instance `Attribute` dont `is_primary` est à `true` pour créer la relation `pkey` (i).

Une transformation de modèles peut être interprétée en termes de navigation, filtrage, création, modification. C'est donc sur ces opérations que peuvent apparaître des erreurs spécifiques. De plus, ces opérations sont séquentiellement dépendantes : quand la navigation retourne des éléments, ils sont souvent filtrés avant d'être utilisés pour la création (ou la modification). La décomposition d'une transformation en séquences d'opération offre un point de vue sémantique de la transformation. Il est utile pour la définition de modèles de fautes et également pour la construction de modèles devant tuer des mutants (3.3.1b-).

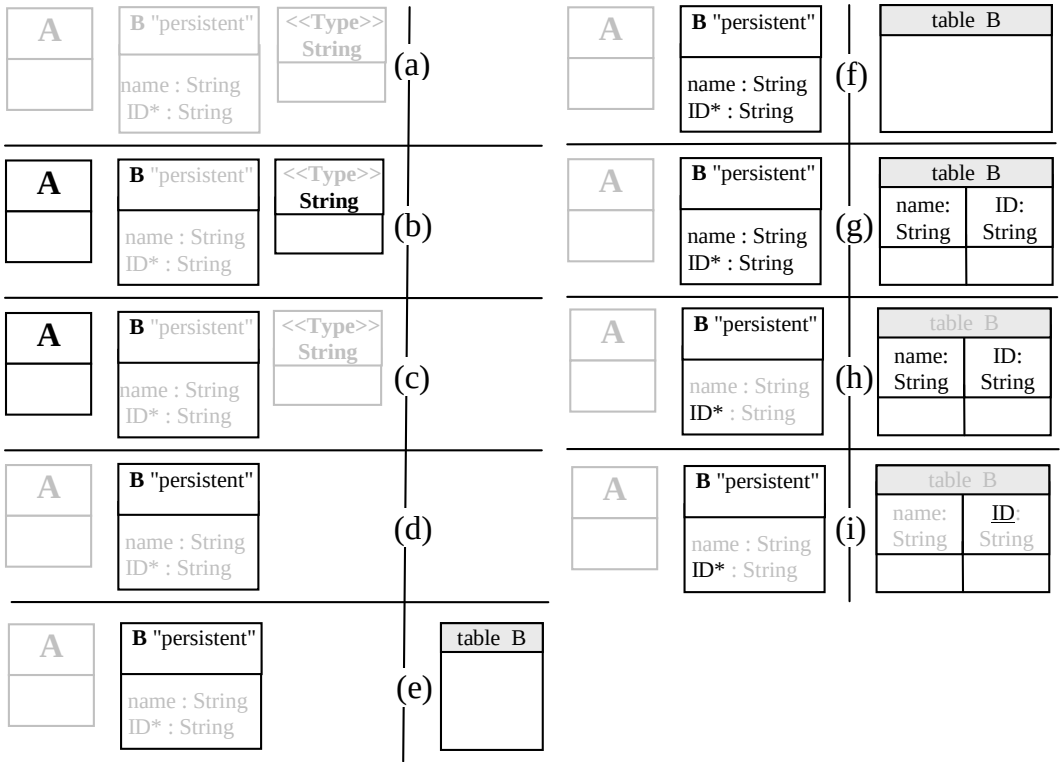


Figure 3-3 – Succession d'opérations pendant le déroulement de class2rdbms

3.2 Opérateurs de mutation dédiés aux transformations de modèles

En se basant sur l'analyse des opérations qui constituent une transformation de modèles, nous proposons plusieurs nouveaux opérateurs de mutation. Ils agissent sur la navigation et le filtrage réalisés par la transformation de modèles sur les modèles d'entrée et de sortie et sur la création du modèle de sortie.

Nous présentons nos opérateurs en commençant par leurs noms et abréviations. Puis nous donnons une explication concise de leur fonctionnalité. Ensuite nous introduisons une définition précise de leur mise en application pour créer les mutants. Chacun est ensuite illustré en se basant sur les méta-modèles de `Class` et `RDBMS` de la figure 2-7 et avec le langage OCL. Nous

n'avons pas choisi un langage de transformation parce que ni les opérations ni les mutants ne sont définis pour un langage de transformation mais pour une représentation abstraite des transformations. OCL n'est pas un langage de transformation mais il permet de réaliser des requêtes sur les modèles. Par ailleurs, il comporte une expression « `let` » permettant de définir des variables *temporaires* qui peuvent être affectées avec l'opérateur « = » [OMG'03]. Nous les utilisons pour que les opérations de création créent des objets.

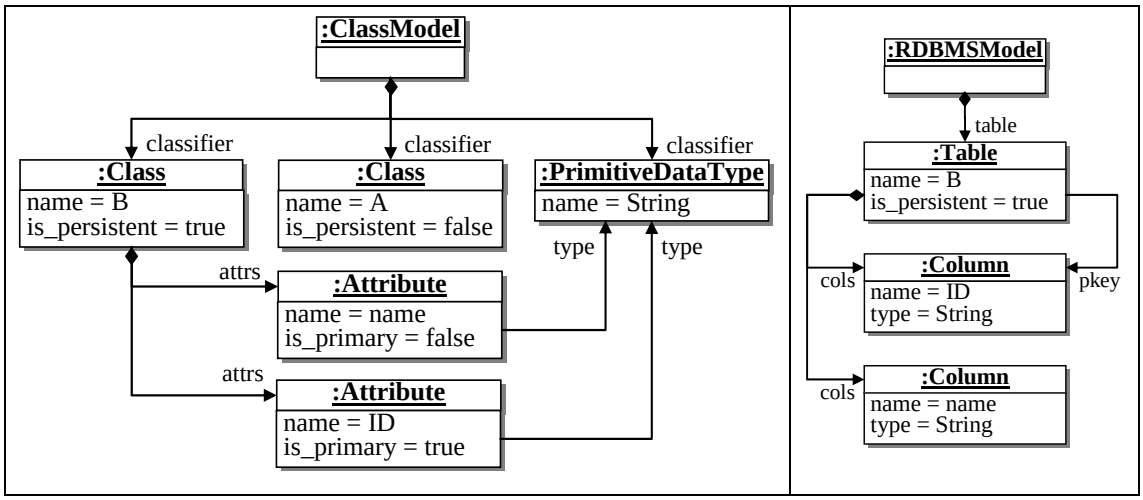


Figure 3-4 - Diagrammes d'instances d'un modèle de classes et du modèle RDBMS correspondant

3.2.1 Opérateurs de mutation concernant la navigation:

a- Remplacement d'une relation vers une même classe (RRMC) :

Cet opérateur remplace la navigation d'une relation vers une classe par la navigation d'une autre relation vers la même classe.

RRMC : *Si la transformation comporte une opération de navigation d'une relation φ définie de la classe X à la classe Y et s'il existe une autre relation σ entre X et Y alors la navigation de φ est remplacée par la navigation de σ. Tous les remplacements possibles sont effectués, chacun créant un mutant.*

Par exemple, si `uneClass`, `uneAssoc`, et `uneTable` sont respectivement des instances de `Class`, de `Association`, et de `Table` :

Opération de navigation originale	Opération de navigation mutante
`uneClass.parent`	`uneClass.src`
`uneClass.children`	`uneClass.parent`
`uneAssoc.dest`	`uneAssoc.src`
`uneTable.pkey`	`uneTable.cols`

Tableau 3-1- Exemples d'application de RRMC

La navigation d'une mauvaise relation définie vers la même classe peut avoir différentes conséquences en fonction de la cardinalité des relations (celle naviguée par le mutant et celle

qui aurait dû l'être). Le remplacement de `uneClass.parent` par `uneClass.children` conduit à une différence de cardinalité puisque la navigation de la relation `parent` renvoie une seule instance de `Class` alors que la navigation de `children` renvoie un ensemble d'instance de `Class`. Selon l'opération qui sera appliquée sur le résultat de cette navigation et selon le langage d'implantation utilisé, la différence de cardinalité peut bloquer la suite de la transformation et être mise en évidence à l'exécution sans attendre que l'oracle compare les résultats de la transformation et du mutant. Par contre le remplacement de `uneTable.pkey` par `uneTable.cols` conduit à une erreur pertinente : les deux retournant une collection d'instances de `Column`. C'est aussi le cas pour `uneAssoc.src` qui comme `uneAssoc.dest` retourne une instance de `Class`. Si ces fautes étaient commises par le développeur, elles ne seraient pas systématiquement révélées pendant la mise au point du programme car le reste de la transformation n'est pas affecté.

b- Remplacement d'une relation vers une autre classe (RRAC) :

Cet opérateur remplace la navigation d'une association vers une classe par la navigation d'une autre association vers une autre classe.

RRAC : *Si la transformation comporte une opération de navigation d'une relation φ définie de la classe X à la classe Y et s'il existe une relation σ entre X et une classe Z différente de Y alors la navigation de φ est remplacée par la navigation de σ. Tous les remplacements possibles sont effectués, chacun créant un mutant.*

Par exemple, si `uneClass` et `uneTable` sont respectivement des instances de `Class` et de `Table` :

Opération de navigation originale	Opération de navigation mutante
`uneClass.children`	`uneClass.attrs`
`uneClass.parent`	`uneClass.attrs`
`uneTable.pkey`	`uneTable.fkeys`

Tableau 3-2- Exemples d'application de RRAC

La navigation incorrecte d'une relation définie vers une autre classe peut avoir des conséquences plus importantes que celle vers une même classe (`RRMC`). En effet, les objets retournés par les opérations originales et mutantes sont des instances de classes de types différents. Ainsi il est plus complexe pour la transformation de poursuivre si les traitements effectués sur les objets résultants de la navigation sont spécifiques à leur type. Cependant, les classes définies dans les méta-modèles peuvent avoir des attributs identiques. C'est le cas du premier exemple puisque les classes `Class` et `Attribute` ont le même attribut `name`. Ces cas sont fréquents si les classes appartiennent au même arbre d'héritage. Dans ce cas elles héritent des mêmes propriétés et des mêmes méthodes dont le comportement peut changer en étant redéfini. Dans ces différents cas, cet opérateur est pertinent en ne bloquant pas l'exécution des opérations qui suivraient l'opération mutée. Nous retrouvons cependant comme précédemment (RRMC) les préoccupations de différences de cardinalités des relations naviguées.

c- Modification d'une succession de navigation avec manque (MSNM) :

Pendant la navigation, la transformation peut naviguer successivement plusieurs relations. Cet opérateur supprime la dernière étape d'une navigation composée.

MSNM : *Si la transformation comporte une succession d'opérations de navigation non interrompue par une opération d'un autre type alors la dernière opération de navigation est retirée. Tous les retraits possibles sont effectués, chacun créant un mutant.*

Par exemple, si `uneAssoc` est une instance de `Association` :

Succession originale d'opérations de navigation	Succession mutante d'opérations de navigation
`uneAssoc.dest.parent`	`uneAssoc.dest`
`uneAsso.dest.attrs.type`	`uneAsso.dest.attrs`

Tableau 3-3- Exemples d'application de MSNM

Cet opérateur conduit aux mêmes cas de figure que l'opérateur RRAC, ce qui justifie sa pertinence.

d- Modification d'une succession de navigation avec ajout (MSNA) :

Cet opérateur réalise le contraire de MSNM, il ajoute une étape à une navigation déjà composée d'une ou plusieurs opérations de navigation ininterrompues.

MSNA : *Si la transformation comporte une opération de navigation ou une succession d'opérations de navigation non interrompue par une autre sorte d'opération, si la dernière opération de navigation renvoie des instances d'une classe X, s'il existe une relation σ définie sortante de X, alors une opération de navigation naviguant σ est ajoutée. Tous les ajouts possibles sont effectués, chacun créant un mutant, produisant autant de mutant que de relation sortant de X.*

Par exemple, si `uneAssoc` et `uneClass` sont des instances de `Association` et de `Class` :

Opération ou succession originale d'opérations de navigation	Succession mutante d'opérations de navigation
`uneAssoc.dest`	`uneAssoc.dest.children`
	`uneAssoc.dest.parent`
	`uneAssoc.dest.attrs`
`uneClass.parent.attrs`	`uneClass.parent.attrs.type`

Tableau 3-4- Exemples d'application de MSNA

La pertinence de cet opérateur est justifiée car il conduit aux mêmes cas que RRAC.

3.2.2 Opérateurs de mutation concernant le filtrage:

Le filtrage manipule des collections et sélectionne seulement les éléments utiles à la transformation. D'une manière générale, un filtrage peut être considéré comme une garde sur une collection, dépendant d'une condition spécifique. Deux types de filtrage sont considérés. D'abord, des instances d'une classe donnée peuvent être sélectionnées en fonction de leurs

propriétés (attributs…) : filtrage de propriété. Ensuite, des instances peuvent être sélectionnées parmi une collection d'instance de classe générique en fonction du type : filtrage de type.

a- Modification du filtrage d'une collection avec perturbation (MFCP) :

Cet opérateur cherche à modifier un filtrage existant, en influençant sa condition. Cette condition porte sur les propriétés ou le type des objets de la collection filtrée.

MFCP : *Si la transformation comporte une opération de filtrage f[P] sur une collection d'objet U alors ce filtrage est perturbé en altérant P de trois manières : une négation est ajoutée à la condition P, la condition P est altérée pour sélectionner le premier élément de U, et si la condition porte sur le type des objets de la collection alors le type souhaité d'une classe C est remplacé par un autre type d'une classe D appartenant au même arbre d'héritage que C. Toute perturbation possible est effectuée, chacune produisant un mutant.*

Par exemple, si `desClasses` et `desClassifiers` sont respectivement des collections d'instances de `Class` et de `Classifier` :

Succession originale d'opérations de filtrage	Succession mutante d'opérations de filtrage
`desClasses.select(c|` `  c.is_persistent=true)`	`desClasses.select(c|` `  not c.is_persistent==true)`
	`desClasses.first()` `//sélectionne le premier élément`
`desClassifiers.select(c|` `  c.oclIsTypeOf(PrimitiveDataType))`	`desClassifiers.select(c|` `  c.oclIsTypeOf(Class))`
	`desClassifiers.select(c|` `  not c.oclIsTypeOf(PrimitiveDataType))`
	`desClassifiers.first()`

Tableau 3-5- Exemples d'application de MSNM

Dans le cas le plus simple, le filtrage dépendant d'une propriété est viable parce que la collection renvoyée a des instances de même type. Les collections voulues et erronées ont seulement des tailles différentes, ainsi la transformation peut se poursuivre même si elle est influencée. Dans les cas plus complexes, nous considérons le filtrage en fonction du type. Filtrer une collection d'instances d'une classe générique en fonction de cette classe ou de n'importe laquelle de ses classes filles n'a pas d'influence. Toutes ces classes partagent les mêmes propriétés héritées ce qui permet à la transformation de se poursuivre.

b- Modification du filtrage d'une collection avec manque (MFCM) :

Cet opérateur supprime un filtre sur une collection, l'opération de filtrage mutée retourne la collection qu'elle était supposée filtrer.

MFCM : *Si la transformation comporte une opération de filtrage f[P] sur une collection d'objet U alors ce filtrage est supprimé et U ne subit pas de filtrage. Tous les filtrages existants sont supprimés, chaque suppression produit un mutant.*

Par exemple, si `desClasses` et `desClassifiers` sont respectivement des collections d'instances de `Class` et de `Classifier` :

Opération de filtrage originale	Opération de filtrage mutante
`desClasses.select(c|c.ispersistent=true)`	`//instruction supprimée`
`desClassifiers.select(c|` `c.oclIsTypeOf(PrimitiveDataType))`	`//instruction supprimée`

`desClassifiers`
`.select(c|`
`c.oclIsTypeOf(PrimitiveDataType))`
`.select(c|c.name = "String")`
	`desClassifiers .select(c|`
	`c.oclIsTypeOf(PrimitiveDataType))`

	`desClassifiers` `.select(c|c.name = "String")`

Tableau 3-6- Exemples d'application de MFCM

Cet opérateur conduit aux mêmes cas que l'opérateur MFCP, ce qui justifie sa pertinence.

c- Modification du filtrage d'une collection avec ajout (MFCA) :

Cet opérateur réalise le contraire de MFCM. Il utilise une collection et lui applique un filtrage inutile. Cet opérateur pourrait retourner un nombre infini de mutants, nous devons en restreindre les possibilités. Nous choisissons de prendre une collection et d'en retourner un élément choisi arbitrairement.

MFCA : *Si une opération de la transformation renvoie une collection d'objets (opération de navigation, de filtrage ou de création) alors un filtrage f[P] lui est appliqué avec P qui contraint la sélection du premier élément de la collection.*

Par exemple, si `uneClass` et `desClass` sont respectivement une instance de `Class` et une collection d'instances de `Class` :

Opération de filtrage originale	Opération de filtrage mutante
`uneClass.children`	`uneClass.children.first()`
`uneClass.children.select(c|` `c.is_persistent=true)`	`uneClass.children.select(c|` `c.is_persistent=true).first()`
`let dautresClass : Collection(Class)` `= desClass`	`let dautresClass : Collection(Class)` `= desClass.first()`

Tableau 3-7- Exemples d'application de MFCA

Cet opérateur conduit aux mêmes cas que l'opérateur MFCP, ce qui justifie sa pertinence.

3.2.3 Opérateur de mutation concernant la création:

Ces opérateurs sont basés sur deux opérations abstraites : la création des éléments du modèle de sortie et la partie concernant la création pour l'opération de modification du modèle d'entrée.

a- Remplacement de la création d'une classe par une autre compatible (RCCC) :

Cet opérateur remplace la création d'un objet par la création d'un objet d'un type compatible. Il peut s'agir d'une instance d'une classe fille, d'une classe parente ou d'une classe avec un parent commun.

RCCC : *Si une opération de la transformation crée un objet δ instance d'une classe C, si il existe une classe F tel que F et C appartiennent au même arbre d'héritage, alors la création de δ est remplacée par la création d'une instance de F. Tous les remplacements possibles sont effectués, chacun produisant un mutant.*

$$\text{Les classes } F \text{ et } C \text{ appartiennent au même arbre d'héritage si et seulement si}$$
$$\exists P \text{ une classe telle que } F = P \text{ ou } F \text{ hérite de } P \text{ et } C = P \text{ ou } C \text{ hérite de } P$$

Par exemple, si l'expression OCL `let` est utilisée pour illustrer la création d'un objet :

Opération de création originale	Opération de création mutante
`let unObjet : Classifier`	`let unObjet : Class`
`let unObjet : Class`	`let unObjet : Classifier`
`let unObjet : Class`	`let unObjet : PrimitiveDataType`

Tableau 3-8- Exemples d'application de RCCC

Cet opérateur est pertinent parce que la mauvaise classe créée et la bonne ont des propriétés héritées communes (relations, méthodes et attributs), donc le reste de la transformation peut être affecté tout en se poursuivant.

L'opérateur contraire qui consisterait à supprimer la création d'un objet a peu de sens. En effet, un objet créé est ensuite forcément utilisé. Il sera contenu par un autre objet, conteneur d'autres objets ou mis en relation avec d'autres objets. Ainsi l'absence de sa création serait systématiquement constatée dans la suite du développement ou dès la mise au point de la transformation de modèle. S'il est imaginable que son utilisation soit aussi absente par erreur, alors l'hypothèse du programmeur compétent n'est plus satisfaite et le cadre d'utilisation de l'analyse de mutation est dépassé.

b- Modification d'une mise en relation de classes avec retrait (MMRR) :

Cet opérateur supprime la création d'une association entre deux instances.

MMRR : *Si une opération de la transformation affecte une relation δ entre une instance d'une classe C et une instance d'une classe D, δ étant une relation définie de C vers D, alors l'affectation est supprimée. Chaque suppression possible est effectuée, chacune produisant un mutant.*

Par exemple, si `uneClass`, `uneautreClass`, `uneTable`, et `uneColumn` sont respectivement des instances de `Class`, `Table`, et `Column` :

Opération de création originale	Opération de création mutante
`uneClass.parent = uneautreClass`	`//instruction supprimée`
`uneTable.cols.add(uneColumn)`	`//instruction supprimée`

Tableau 3-9- Exemples d'application de MMRR

Si l'affectation non créée concerne une relation définie avec une cardinalité `1` (ou une autre supérieure fixée) alors son absence peut affecter le reste de la transformation. En effet, il est probable qu'à un moment de la transformation cette affectation soit attendue pour être utilisée.

En revanche, si la cardinalité est `n` alors l'utilisation de l'ensemble reviendrait à naviguer la relation en obtenant un élément de moins ce qui n'est pas bloquant pour la poursuite de la transformation. Par exemple, un modèle peut avoir une instance de `Table` connectée à plusieurs instances de `Column` par la relation `cols`. Si une relation n'est pas créée, la navigation `aTable.cols` retournera juste une collection privée d'un élément, sans conséquence systématiquement détectable.

c- Modification d'une mise en relation de classes avec ajout (MRCA) :

Cet opérateur ajoute une création inutile d'une relation entre deux instances existant dans modèle de sortie, quand le méta-modèle le permet.

MRCA : *Si au cours de la transformation, deux instances a et b de classes A et B (pouvant être une même classe) sont disponibles et s'il existe une relation δ définie de A vers B, alors la relation δ est affectée entre a et b. Toutes les relations définies entre des classes dont des instances sont disponibles sont affectées, chaque création d'affectation produisant un mutant.*

Par exemple, si `uneClass`, `uneautreClass`, `uneTable`, et `uneColumn` sont respectivement des instances de `Class`, `Table`, et `Column` :

Opération de création originale	Opération de création mutante

`let uneColumn : Column`
`let uneautreColumn : Column`
`let uneTable : Table`
`uneTable.cols.add(uneColumn)`
	`let uneColumn : Column`
	`let uneautreColumn : Column`
	`let uneTable : Table`
	`uneTable.cols.add(uneColumn)`
	`uneTable.cols.add(uneautreColumn)`

`let uneClass : Class`
`let uneautreClass : Class`
`let uneAssociation : Association`
`uneAssociation.src=uneClass`
	`let uneClass : Class`
	`let uneautreClass : Class`
	`let uneAssociation : Association`
	`uneAssociation.src =uneClass`
	`uneAssociation.src = uneautreClass`
	`//l'écrasement ne pose pas problème`

Tableau 3-10- Exemples d'application de MRCA

Dans notre exemple, le méta-modèle de `Class` (figure 2-7) permet deux relations différentes entre une instance de `Class` et une autre (`parent`, `children`). Si une seule instance de `Class` existe alors deux mutants peuvent être créés, même si une telle relation existe déjà (la relation peut alors être écrasée).

Les erreurs injectées par cet opérateur ne sont pas systématiquement détectées en bloquant le déroulement de la transformation : des collections vont simplement avoir un élément supplémentaire, et les relations de cardinalité 1 seront simplement créées ou écrasées.

opération abstraite	opérateur de mutation	altération réalisée
navigation	RRMC	Remplacement d'une relation vers une même classe
	RRAC	Remplacement d'une relation vers une autre classe
	MSNM	Modification d'une succession de navigation avec manque
	MSNA	Modification d'une succession de navigation avec ajout
filtrage	MFCP	Modification du filtrage d'une collection avec perturbation
	MFCM	Modification du filtrage d'une collection avec manque
	MFCA	Modification du filtrage d'une collection avec ajout
création	RCCC	Remplacement de la création d'une classe par une autre compatible
	MMRR	Modification d'une mise en relation de classes avec retrait
	MRCA	Modification d'une mise en relation de classes avec ajout

Tableau 3-11 - Liste des dix opérateurs de mutation proposés

Pour conclure sur ces opérateurs : toutes ces erreurs sont directement liées à la manière dont nous définissons les transformations de modèles qui sont divisées en quatre opérations abstraites (navigation, filtrage, création, modification). Ces opérateurs permettent de générer des mutants viables avec des erreurs pertinentes pour améliorer la valeur de l'analyse de mutation. Cela permet alors d'améliorer la capacité de l'ensemble de modèles de test à détecter les erreurs du programmeur. Le tableau 3-11 synthétise tous ces nouveaux opérateurs.

3.3 Simplification des activités non automatisables

Dans cette section, nous étudions les activités non automatisables du processus de l'analyse de mutation. Dans la figure 3-5, nous schématisons les trois temps de ce processus :

1. **la création des mutants** : c'est une tâche *automatisable* si un langage de transformations de modèles est choisi,
2. **l'exécution du processus pour obtenir le score de mutation** : c'est une tâche qui est *automatisée* (nous présentons l'outil que nous avons réalisé dans la section 5.2.3),
3. **l'amélioration du score de mutation** : les mutants équivalents sont détectés et les modèles de test sont améliorés pour tuer les mutants vivants *manuellement*.

Pour améliorer la pratique de l'analyse de mutation, il est nécessaire de réduire le coût des activités manuelles qui composent le troisième temps. Nous considérons également la création des oracles à partir des modèles de test (figure 3-5). Ce n'est pas une activité de l'analyse de mutation, mais elle dépend directement du nombre de modèles qui sont produits pour obtenir un score de mutation élevé. Dans cette section, nous présentons une méthode pour :

- *faciliter la création de nouveaux modèles de test* pour tuer les mutants vivants,
- *diminuer le nombre de modèles de test* pour diminuer l'*effort de création des oracles*.

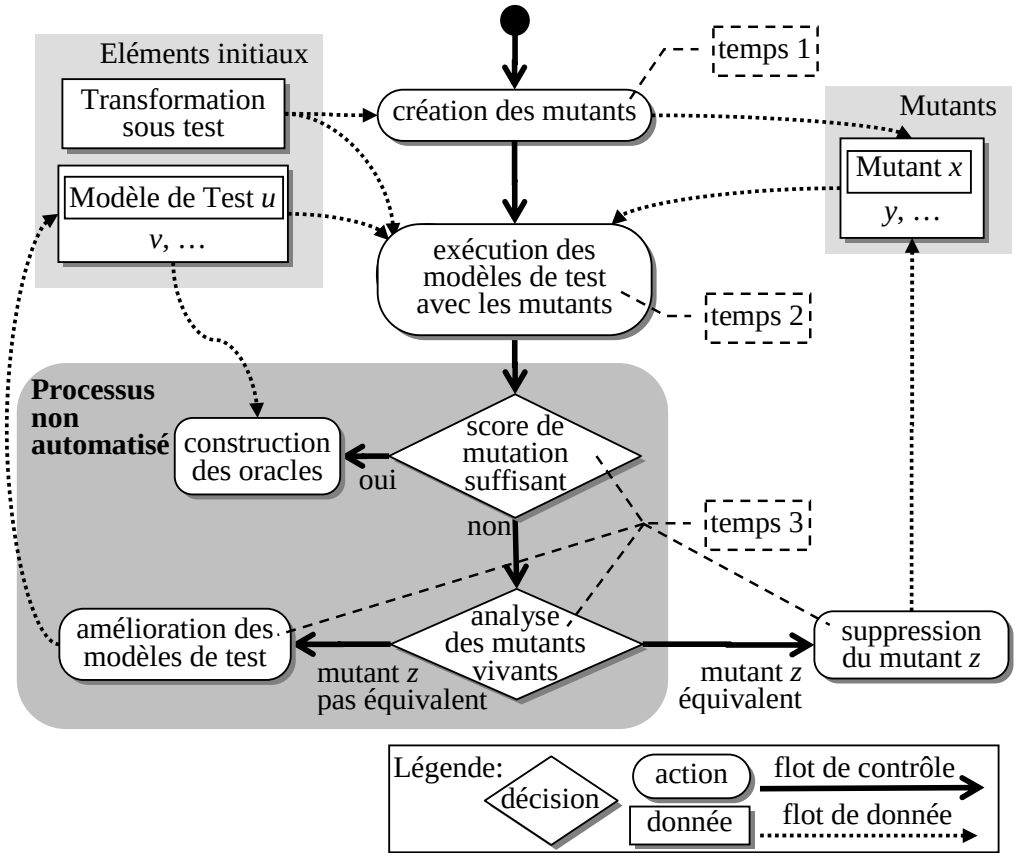


Figure 3-5- Processus global en trois temps de l'analyse de mutation mettant en évidence les activités manuelles et la construction d'oracle qui suit l'analyse

3.3.1 Simplification de l'amélioration des modèles de test

a- Création de modèles de test contre amélioration de modèles de test existants

Plusieurs stratégies sont envisageables pour améliorer l'ensemble de modèles de test :

- Soit en améliorant les modèles de test existants
- Soit en créant de nouveaux modèles de test
- Soit en combinant l'amélioration et la création

Nous privilégions la seconde possibilité. Une expérience a été menée avec plusieurs groupes d'étudiants de niveau master pour valider ce choix. Deux années de suite, nous avons proposé à trois groupes différents d'étudiants un sujet de travaux pratiques sur l'analyse de mutation. Le premier groupe a dû créer de nouvelles données de test, le second a dû améliorer celles existantes et le troisième a eu le choix. La majorité des étudiants de ce dernier groupe a choisi spontanément de créer de nouvelles données de test. Les élèves devant améliorer les modèles existants ont eu plus de difficultés que ceux qui ont créé de nouveaux modèles. Ces difficultés se sont révélées par un temps plus long pour résoudre l'exercice, certains groupes n'arrivant pas à améliorer les modèles pour obtenir le score de 100% dans le temps imparti.

Le choix de créer de nouveaux modèles de test est justifié car un modèle amélioré pour tuer des mutants risque de ne plus tuer ceux qu'il tuait auparavant. Ce risque n'est pas présent avec la création de nouveaux modèles de test.

b- Exploitation des opérateurs de mutation définis à un niveau sémantique pour la création de modèles de test

Il faut analyser les mutants vivants pour créer de nouveaux modèles de test. Nous considérons l'erreur et l'opération altérée de chaque mutant. Les opérations composant une transformation sont utilisées séquentiellement : quand la navigation retourne des éléments, ils sont filtrés avant d'être utilisés pour la création (ou la modification). L'opération altérée d'un mutant s'inscrit dans une séquence d'opérations qui se termine par une création ou une modification dans le modèle de sortie. En observant cette séquence et en connaissant l'erreur injectée avec l'opérateur de mutation, il est possible de déterminer quel modèle est nécessaire pour le tuer.

Il est possible de suivre ces trois étapes pour créer un nouveau modèle de test :

- identifier la séquence d'opérations dans laquelle s'inscrit l'opération altérée,
- définir les objets nécessaires dans le modèle pour réaliser cette séquence,
- créer un modèle contenant ces objets.

Par exemple, l'opération de la figure 3-6 est extraite d'un mutant de la transformation class2rdbms. A la ligne de code 10, l'application de l'opérateur MFCA (Modification du filtrage d'une collection avec ajout) sur l'opération de filtrage a introduit une erreur. Il s'agit de l'ajout d'un filtrage arbitraire sélectionnant le premier élément de la collection renvoyé par le filtrage.

```
1  /**
2   * Get all the associations of the children classes of the parameter class
3   */
4  operation getSubClassesAssociation(cls : Class) : Association[0..*] is do
5       var model : ClassModel model ?= cls.container
6       result := OrderedSet<Association>.new
7       // get all the sub-classes
8       getAllClasses(model).select{ c | c.parent == cls }.each{ c |
9          result.addAll(model.association
10                       .select{ asso | asso.src == c }.subSequence(0,0))
11         result.addAll(getSubClassesAssociation(c))
12      }
13 end
```

Figure 3-6 - Extrait d'un mutant de la transformation class2rdbms

L'opération de filtrage altérée s'inscrit dans cette séquence d'opérations :

1. **navigation** du modèle pour collecter les objets de type `classifier`
2. **filtrage** pour sélectionner les objets de type `class`

3. **filtrage** pour sélectionner les classes `X` dont `is_persistent` est à `true`
4. **navigation** du modèle pour collecter les objets de type `classifier`
5. **filtrage** pour sélectionner les objets de type `class`
6. **filtrage** pour sélectionner celles `Y` dont le parent est la classe `x` (répétitions des opérations suivantes pour tout x appartenant à X)
7. **navigation** pour collecter toutes les associations du modèle
8. **filtrage** pour sélectionner les associations dont la source est une de ces classes `y` (`y`∈`Y`)
9. **création** des colonnes correspondantes

C'est le dernier filtrage (8.) qui est altéré. En reprenant cette séquence d'opération, un modèle peut être construit : un modèle avec une classe persistante `cp` (opérations 1. 2. 3.), `cp` a une classe fille `cf` (opérations 4. 5. 6.) qui a des associations sortantes `as` (opérations 7. 8.). Le mutant ne sélectionne plus toutes les associations mais une seule. Il faut donc un modèle ayant plusieurs de ces associations `as` pour qu'une seule colonne soit créée (opérations 9.) au lieu de plusieurs : cette différence est détectée et le mutant tué. Cette séquence d'opérations permet de déterminer un model snippet, illustré figure 3-7, (la notion de model snippet est détaillée dans le chapitre suivant et définie dans le glossaire) que doit contenir un modèle de test. Nous construisons ce modèle de test (figure 6-4).

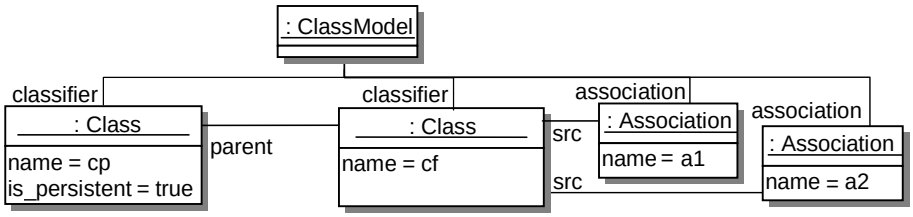


Figure 3-7 - Model snippet permettant de créer un modèle tuant un mutant vivant

3.3.2 Réduction du nombre de modèles de test

L'ensemble de modèles de test obtenu en fin d'analyse de mutation peut contenir des modèles de test ayant peu d'utilité, tout du moins au regard du score de mutation :

- certains modèles ont été créés pour tuer des mutants qui se sont révélés être équivalents,
- l'ensemble initial de modèles de test n'est pas construit en fonction des mutants,
- l'utilisation d'un générateur de modèles va créer des modèles sans lien avec les mutants.

L'étape suivant l'analyse de mutation dans le processus de test est la construction d'oracle pour chacun des modèles de test pour former les cas de test. Nous verrons dans le chapitre suivant que c'est une étape manuelle coûteuse. Les modèles tuent plusieurs mutants, donc il y a des recouvrements avec des mutants tués par plusieurs modèles de test. Nous proposons donc de réduire l'ensemble de modèles pour conserver un ensemble minimal suffisant pour obtenir le même score de mutation que l'ensemble total.

a- Exploitation d'une matrice pour réduire le nombre de modèles de test

Nous utilisons un algorithme de réduction de matrice pour retirer de l'ensemble de modèles de test ceux dont l'action est déjà assurée par d'autres modèles de test. La matrice utilisée contient les résultats de l'exécution de chaque modèle de test avec chaque mutant. Chaque ligne correspond à un mutant et chaque colonne au résultat de chaque modèle de test : un 1 signifie que le modèle de test a tué le mutant, et un 0 signifie le contraire.

Modèle de test	Mt 1	Mt 2	Mt 3	Mt 4
Mutant 1	0	0	1	1
Mutant 2	1	0	1	0
Mutant 3	0	0	0	0
Mutant 4	1	1	0	0

Tableau 3-12 - Matrice résultat de l'analyse de mutation

Le tableau 3-12 illustre une matrice obtenue avec quatre mutants et quatre modèles de test. Une ligne sans 1 signifie que le mutant est vivant : c'est le cas du troisième mutant. Une ligne avec au moins un 1 signifie que le mutant est tué : c'est le cas des premier, deuxième, et quatrième mutants. Cet ensemble de modèles obtient donc un score de 75% (trois mutants tués sur quatre). On constate dans cette matrice qu'il est possible de se passer du modèle de test 1 pour obtenir le même score de mutation puisque les mutants qu'il tue sont aussi tués par d'autres modèles de test. C'est d'ailleurs le cas pour chacun des modèles de test. Ainsi cet ensemble de modèle de test peut être réduit par le retrait de n'importe lequel de ses modèles de test. Ce qui peut réduire l'effort de construction des oracles de 25%.

Pour appliquer la réduction, il faut produire la matrice pendant l'analyse de mutation. L'analyse de mutation étant une technique coûteuse, il est commun que dès qu'un mutant est tué par une donnée de test, il ne soit pas exécuté avec les autres données. Pour obtenir la matrice il faut étudier chaque mutant avec tous les modèles de test.

Dans un premier temps, nous appliquons le processus de mutation pour tuer tous les mutants et obtenir la matrice des mutants tués par chaque modèle de test. L'algorithme est le suivant :

```
1 soit M l'ensemble des mutants de taille nb_mutant
2 soit Màtuer l'ensemble des mutants à tuer de taille nb_mutant_à_tuer
3 soit Mtué l'ensemble des mutants tués initialement vide
4 soit MoT l'ensemble des modèles de test de taille nb_modèles
5 soit Mat la matrice de nb_mutant lignes et nb_modèles colonnes remplie de 0
```

```
6  pour tous les mutants m_u de M avec u de 1 à nb_mutant
7  faire   pour tous les modèles de test mt_x de MoT avec le mutant m_u
9          faire
10            exécuter mt_x avec m_u contre l'oracle
11            si m_u est tué
12            alors   m_u est retiré de l'ensemble des mutants à tuer Màtuer
13                    // nb_mutant_à_tuer est donc décrémenté de un
14                    m_u est ajouté dans l'ensemble des mutants tués Mtué
15                 (m_u , mt_x) est mis àr 1 dans la matrice Mat
16            fin de si
17         fin de faire
18 fin de faire

19 pour tous les mutants m_s de Màtuer avec s de 1 à nb_mutant_à_tuer
20 faire   étudier la mutation de m_s
21         si m_s est équivalent
22         alors   m_s est retiré de l'ensemble des mutants à tuer Màtuer
23         sinon
24            créer un modèle de test mt_z dans MoT pour tuer m_s
25            pour tous les mutants m_t de M avec t de 1 à nb_mutant
26            faire exécuter mt_z avec le mutant m_t
27               si m_t est tué
28               alors   retirer m_t de l'ensemble des mutants à tuer Màtuer
29                       // nb_mutant_à_tuer est donc décrémenté de un
30                       m_t est ajouté dans l'ensemble des mutants tués Mtué
31                       (m_t, mt_z) est mis à 1 dans la matrice Mat
32               fin de si
33            fin de faire
34         fin de si
35 fin de faire
```

L'algorithme se compose de deux boucles principales. Dans la première, l'ensemble initial de modèles de test est employé. Dans la deuxième, de nouveaux modèles de test sont créés pour tuer les mutants vivants et retirer les mutants équivalents. A la fin de ce processus, nous obtenons la matrice et l'ensemble de modèles de test.

Diagnostiquer qu'un mutant est équivalent est une activité manuelle coûteuse et difficile. Il est possible qu'un nouveau modèle de test ne tue pas le mutant visé (ligne 24). En conséquence, il faut analyser plus attentivement le mutant pour : soit produire un autre modèle de test plus efficace, soit diagnostiquer l'équivalence du mutant. L'échec dans la création d'un nouveau modèle n'est cependant pas problématique. En effet, tous les modèles de test sont exécutés avec chaque mutant pour produire la matrice complète (et pas seulement les vivants). Par effet de bord, le modèle va tuer d'autres mutants que son objectif initial. Il est possible qu'il tue d'autres mutants vivants et contribue quand même à améliorer l'ensemble de modèles de test. La phase suivante de minimisation de la matrice produite, retirera de l'ensemble.

b- Algorithme de réduction de la matrice retournée par l'analyse de mutation

Nous avons implémenté un algorithme qui réduit la matrice retournée par l'analyse de mutation. Le résultat d'une réduction est une autre matrice contenant autant de lignes (une par mutant) et moins de colonnes (une par modèle de test) que la matrice initiale. Les modèles de test d'une réduction obtiennent le même score de mutation. Il peut exister plusieurs réductions possibles. Un modèle de test est retiré si les mutants qu'il tue sont tués par les autres modèles. Donc une colonne est retirée si les 1 qu'elle contient sont contenus dans les autres colonnes.

La figure 3-8 schématise le déroulement de l'algorithme de réduction de base. Il considère chaque colonne (identifiée `a` par exemple) et recherche si les 1 qu'elle contient sont contenus dans les autres colonnes (par exemple `a` $\subset$ `bcde ?`). Si c'est le cas la colonne peut être retirée: c'est une *minimisation possible* (`a`) et nous obtenons une matrice minimisée en retirant la colonne `a` de la matrice initiale. Puis la colonne suivante est considérée (`b` $\subseteq$ `cde ?`) : une nouvelle minimisation est obtenue (`ab`). Nous obtenons un arbre dont les nœuds sont les minimisations possibles.

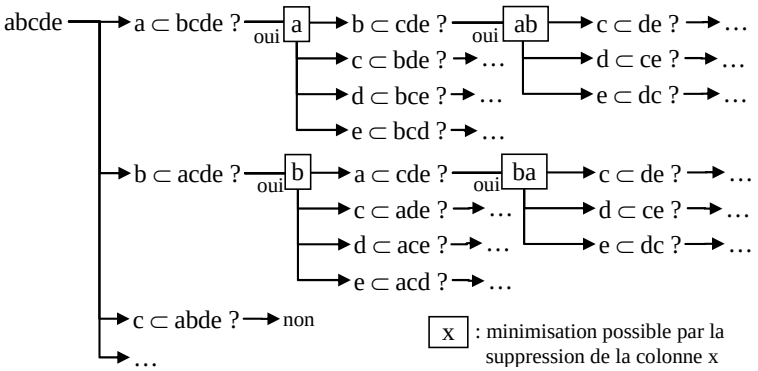


Figure 3-8 - Processus de base pour la minimisation

Ce processus de base est coûteux et injustement répétitif : nous obtenons par exemple les minimisations `ab` et `ba` qui sont les mêmes. En effet, l'ordre du retrait des colonnes n'a pas d'importance. La figure 3-9 illustre l'optimisation que nous avons réalisée du processus de base. Nous avons exploité deux propriétés :

- Quand une réduction X, composée de colonnes (xyz…) est déjà trouvée, si dans une branche nous trouvons la même réduction alors il n'est pas nécessaire de poursuivre cette branche. Dans la figure 3-9, nous observons qu'une fois la réduction `ab` trouvée, quand la réduction `b` est trouvée il n'est pas nécessaire de chercher à retirer `a` car cela conduirait à la même réduction (`ba`). Il n'est pas non plus nécessaire de poursuivre dans cette branche car on constate que les recherches effectuées dans les zones 3 et 5 sont les mêmes et conduiraient aux mêmes réductions.

- Quand une colonne x ne peut pas être retirée parce que x $\not\subset$ Y (Y un ensemble de colonnes) alors x ne peut pas être retirée d'aucune partition de Y. Dans la figure 3-9, nous observons que la colonne `c` n'est pas contenue dans `abde` donc il ne peut pas y

avoir de réduction c. Si c n'est pas contenue dans abde, elle ne peut être contenue dans aucune de ces partitions (a,b,d,e,ab,ad,ae,…)

Pour obtenir ces optimisations, nous parcourons l'arbre en largeur. Par exemple, nous avons numéroté les zones grisées de la figure 3-9. Une zone grisée n'est pas numérotée quand elle n'est pas exécutée grâce à la première optimisation.

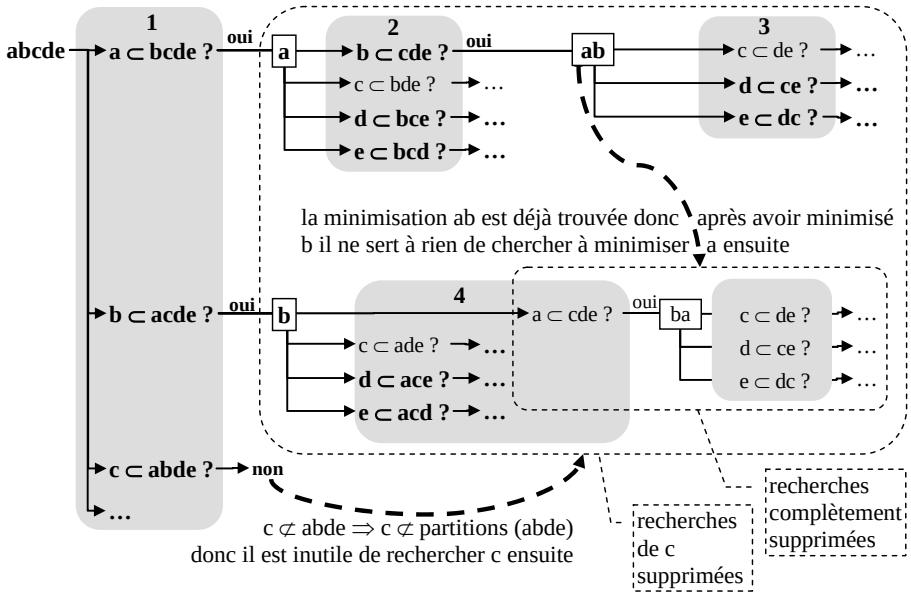


Figure 3-9 - Optimisation du processus de minimisation

Dans le pire des cas, toutes les colonnes sont égales et toutes les minimisations sont possibles pour obtenir une matrice de minimisation maximale contenant une colonne. Dans le cas non optimisé, pour n colonnes, le nombre maximal de minimisations trouvées est de :

$$n + n(n-1) + n(n-1)(n-2) + \ldots + n!$$

ce qui correspond à n minimisations d'une colonne plus n(n-1) minimisations de deux colonnes plus etc. Notre première optimisation permet de ne pas considérer l'ordre des minimisations. Le nombre maximal de minimisations trouvées devient :

$$C_n^1 + C_n^2 + \ldots + C_n^{n-1} = n + C_n^2 + \ldots + C_n^{n-2} + n$$

ce qui correspond à n combinaisons d'une colonne plus $C_n^2 = \frac{n!}{2!(n-2)!}$ combinaisons de deux colonnes plus etc. Ainsi pour n = 5, le nombre maximal de minimisations passe de 205 à 30. La complexité du processus pour obtenir ces minimisations est donc fortement réduite.

Nous réalisons une optimisation supplémentaire en commençant la minimisation en comparant les colonnes par paires pour retirer chaque colonne qui est incluse dans une autre colonne. Par exemple, si d ⊂ e, la colonne d est retirée et non considérée dans l'algorithme. Cela présente deux avantages :

- toutes les vérifications de l'inclusion de d dans un ensemble de colonnes contenant e sont supprimées. Cette suppression permet d'économiser la comparaison des colonnes de l'ensemble (autres que e) avec d,
- les vérifications de l'inclusion d'une colonne x dans un ensemble de colonnes contenant d et e sont simplifiées car :

$$si\ d \subset e\ et\ si\ U\ contient\ d\ et\ e, alors\ x \subset U \Leftrightarrow x \subset (U - d)$$

Ces optimisations permettent d'obtenir le code Java de l'Annexe D pour la réduction du nombre de modèles de test d'un ensemble qualifié par analyse de mutation.

3.4 Application de l'analyse de mutation adaptée au test d'une transformation

Dans cette section, nous appliquons l'analyse de mutation sur notre exemple (class2rdbms) en utilisant les opérateurs de mutation introduits section 3.2.

3.4.1 Un ensemble initial de modèles de test

L'analyse est appliquée sur un ensemble initial de modèles de test. Ici, nous utilisons sept modèles de classes écrits pendant le développement de la transformation. Ces modèles sont plus ou moins compliqués et les développeurs les considéraient suffisamment significatifs pour la mise au point de la transformation. La figure 6-7 illustre un de ces modèles de test initiaux tandis que le tableau 3-13 résume la taille de ces modèles de test.

modèle de test	1	2	3	4	5	6	7
nombre d'objets	4	5	11	7	9	7	9
nombre de propriétés	9	13	33	20	28	20	28

Tableau 3-13 – Taille des modèles de test initiaux

3.4.2 Création des mutants

Le tableau 3-14 présente la répartition des mutants créés pour class2rdbms. L'opérateur CCCD n'est pas appliqué dans cette étude car il dépend de l'héritage et le méta-modèle cible n'en contient pas. Nous obtenons cent soixante-six mutants ce qui est un nombre suffisant pour considérer la transformation choisie comme suffisamment appropriée pour expérimenter la technique. Les erreurs représentées par les opérateurs de mutation ont été insérées manuellement.

3.4.3 Évaluation du score de mutation initial

Les modèles de test initiaux sont exécutés contre les mutants. Nous utilisons deux oracles différents pour tuer les mutants :

- **L'échec de la transformation** : quand un mutant n'est pas capable de produire un modèle de sortie à partir d'un modèle de test

- **La différence des modèles produits** : quand le mutant produit un modèle différent du modèle produit par la transformation originale. Pour ce dernier oracle, nous avons créé un outil dédié à la comparaison de modèles RDBMS. Quand nous avions entrepris ces expériences, nous ne disposions pas d'outil générique de comparaison de modèles.

opération abstraite	nombre de mutants par opération	opérateur de mutation	nombre de mutants par opérateur
navigation	87	RRMC	9
		RRAC	8
		MSNM	10
		MSNA	60
filtrage	59	MFCP	26
		MFCM	16
		MFCA	17
création	20	MMRR	11
		MRCA	9

Tableau 3-14 - Répartition des mutants

Avec les sept modèles de test initiaux, 78% des mutants ont été tués (comme illustré dans la première colonne de la figure 3-10).

3.4.4 Amélioration des modèles de test en exploitant le point de vue sémantique

Selon le niveau d'exigence du testeur, le score de mutation de 78% pourrait être suffisant. Pour cette démonstration, nous essayons d'atteindre le score maximal de 100%.

Nous avons exploité la définition des mutants à un niveau sémantique (comme expliqué au point 3.3.1b-) pour tuer tous les mutants avec neuf nouveaux modèles de test. La figure 3-10 illustre cette progression et la dernière colonne du tableau 3-15.

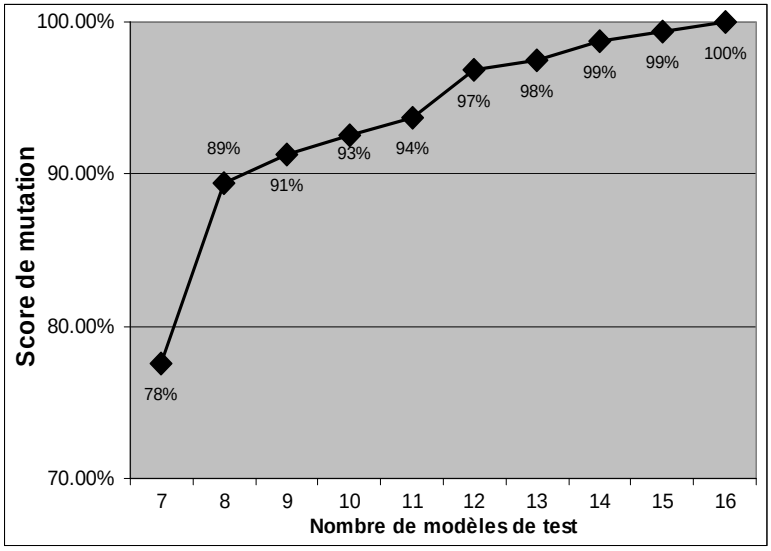


Figure 3-10 – Évolution du nombre de mutants tués.

	avant amélioration	après amélioration
nombre de modèle de test	7	16
score de mutation	78%	100.0%

Tableau 3-15 – Progression du score de mutation

Pour calculer ces scores de mutation, nous ne considérons pas les six mutants équivalents. Certains mutants ne sont pas tués par les modèles créés pour cela. En revanche ces modèles tuent d'autres mutants par effet de bord. Dans la figure 3-11, nous observons que le modèle de test 8 destiné à tuer un mutant de création (dont une opération de création est altérée) en tue plusieurs ainsi que des mutants de navigation et de filtrage. Le modèle de test 10 n'a pas pu tuer le mutant de filtrage pour lequel il était destiné mais il tue des mutants de navigation et de création. Ainsi il diminue le nombre de mutants vivants restant à la fin du processus. Finalement l'analyse des mutants vivants révèle que six sont équivalents. Ils ont été retirés de l'ensemble des mutants et les scores de mutation ont été recalculés avec cent soixante mutants au lieu de cent soixante-six. La proportion de mutants équivalents est faible : moins de 4%. La détection des mutants équivalents ne va pas être excessivement coûteuse avec notre adaptation de l'analyse de mutation.

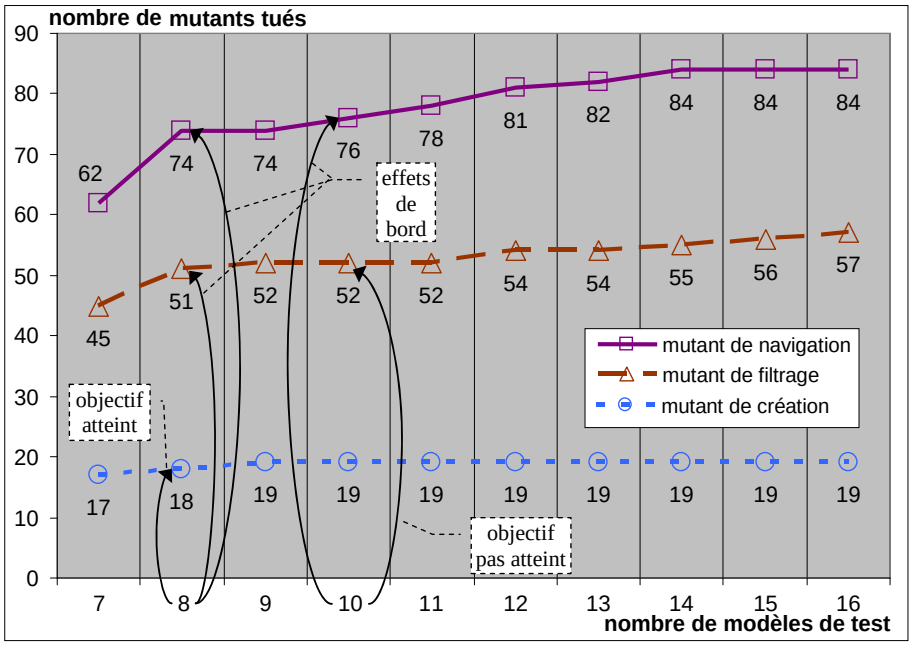


Figure 3-11 - Évolution du nombre de mutants tués illustrant l'effet de bord

Finalement, nous avons produit neuf nouveaux modèles de test pour atteindre le score de 100%. Nous pouvons remarquer que nous avons eu besoin de plus de modèles de test que ceux de l'ensemble initial (neuf autres contre sept initiaux) pour améliorer le score de mutation de 22%. Nous en déduisons qu'il n'est pas compliqué de détecter la majorité des erreurs mais que la détection d'une minorité plus subtile pose plus de problème et nécessite pus d'effort du testeur. Cette déduction avait déjà été faite dans des travaux hors de l'ingénierie dirigée par les modèles [Frankl'97, DeMillo'93, Briand'03, Baudry'03].

3.4.5 Minimisation du nombre de modèles de test

Dans cette dernière étape, nous minimisons l'ensemble de modèles de test. Cela conduit à sélectionner six modèles parmi seize : les 7, 8, 9, 14, 15, et 16ièmes. L'effort de création des oracles est donc diminué. Nous observons que sur les sept modèles initiaux, un seul est conservé. Cela montre deux choses :

- l'ensemble initial des modèles de test n'avait pas été généré en fonction de la mutation,
- les modèles créés pour tuer des mutants et avec notre méthode ont de meilleurs scores de mutation. Ils sont donc plus performants à détecter des erreurs.

3.5 Conclusion

L'efficacité des modèles de test pour garantir leur capacité à révéler des erreurs, est essentielle pour la qualité d'une transformation. Dans ce chapitre, nous avons étudié l'analyse de mutation pour qualifier un ensemble de modèles de test.

L'utilisation de cette technique dans un nouveau contexte impose la définition de nouveaux modèles de fautes sous la forme d'opérateurs de mutation. Ainsi, nous avons proposé des opérateurs de mutation pour le test de transformations de modèles. Cependant, contrairement aux approches classiques, nous n'avons pas défini ces opérateurs en fonction de la syntaxe d'un langage de transformation, même si plusieurs nouveaux langages dédiés aux transformations de modèles ont été développés. La première raison vient justement de la nouveauté de ces langages : ils sont nombreux et aucun consensus n'a encore été trouvé dans la communauté IDM. La seule norme existante (QVT) définit elle-même non pas un mais trois langages dans différents paradigmes. La deuxième raison est le niveau auquel nous considérons les transformations : leurs spécificités sont essentiellement dues aux opérations qu'elles réalisent sur des modèles en se basant sur leurs méta-modèles. C'est ainsi que nous avons défini dix opérateurs de mutation au niveau sémantique des opérations abstraites réalisées par une transformation de modèles.

Dans un second temps, nous avons étudié la diminution de la part des activités manuelles du processus de l'analyse. En particulier, nous proposons de minimiser le nombre de modèles de test pour réduire l'effort dans la construction d'oracle (étudié dans le chapitre suivant).

Finalement, nous avons expérimenté ces opérateurs pour la qualification de modèles de test pour la transformation class2rdbms. Nous avons réalisé ces expériences avec le langage de transformations de modèles Kermeta. A l'avenir, si un langage de transformation était adopté dans la communauté, il serait envisageable de définir un mapping des opérateurs sémantiques vers la syntaxe de ce langage.

4
Oracles de transformations de modèles

L'oracle produit le *verdict* d'un cas de test. Contrairement à ce que suggère son nom, l'oracle n'est pas restreint à la prédiction du résultat attendu. Il vérifie si la « transformation d'un modèle de test » est correcte en analysant le modèle de sortie produit. Il analyse des modèles de sortie sur la base des modèles de test et d'informations extraites de la spécification de la transformation. Le verdict peut être partiel si les vérifications réalisées ne portent pas sur tous les éléments du modèle analysé ou si une seule partie de la spécification est considérée.

La construction d'oracle est une tâche difficile pour le test de logiciel. Cette difficulté est particulièrement présente pour le test de transformations de modèles. D'une part, les modèles de sortie analysés sont des structures de données complexes pouvant être assimilées à des graphes d'objets. D'autre part, les paramètres considérés pour les vérifications sont aussi de nature complexe. Ils comprennent le domaine de sortie (défini dans nos études par un méta-modèle et des contraintes), le modèle de test utilisé, et des informations extraites de la spécification. Cette extraction d'informations est une étape critique dans la construction de l'oracle. Elle ne peut être réalisée automatiquement (en particulier quand la spécification est textuelle).

L'oracle utilisé pour le test de logiciel est généralement une fonction qui compare le résultat obtenu au résultat attendu en sortie. Nous retrouvons cette approche dans différentes études, dont certaines portent sur le test de transformations de modèles [Heckel'03]. Dans ce contexte, le résultat attendu est un modèle appartenant au domaine de sortie de la transformation. Certains travaux [Lin'07] transposent ainsi le problème de l'oracle en un problème de comparaison de modèles. Cependant, spécifier le résultat attendu est extrêmement complexe quand il s'agit d'un modèle. Même si cette approche doit être prise en compte, il est trop restrictif de considérer l'oracle sous ce seul angle. Plusieurs autres techniques peuvent être envisagées pour obtenir un verdict : nous étudions ces alternatives pour les qualifier et mettre en avant les plus appropriées pour le test de transformations de modèles.

Dans cette section, nous proposons six fonctions d'oracle différentes. Elles varient selon le nature de l'information qu'elles exploitent et que nous nommons « donnée d'oracle ». Le modèle attendu ne constitue qu'une seule sorte de donnée d'oracle parmi d'autres. En fournissant des transformations, des contraintes, des modèles, ou des patterns à différentes

fonctions d'oracle, nous définissons des oracles qui n'ont pas les mêmes propriétés et qui ne nécessitent pas le même travail d'extraction d'informations de la spécification.

Nous étudions les propriétés de ces fonctions pour les évaluer suivant deux critères qualitatifs. Le premier critère lié à la complexité, est le *risque d'introduire des erreurs dans la fonction d'oracle*. Ces fonctions d'oracles analysent des modèles complexes, ce qui implique un grand nombre de vérifications de propriétés et accroit le risque d'erreur (error-proneness) dans la définition de l'oracle. Le second critère est la *réutilisabilité de l'oracle* dans différents cas de test et lorsque la transformation de modèles évolue. Dans le but principal de qualifier chaque fonction d'oracle selon ces deux qualités, nous évaluons différentes propriétés caractérisant chacune des fonctions proposées dans ce chapitre.

Notre analyse va montrer que les oracles qui réalisent des vérifications partielles sont de meilleure qualité. En effet, l'utilisation d'un oracle dédié à une vérification particulière sur un modèle précis simplifie sa définition et permet sa réutilisation. En revanche, l'emploi d'un seul oracle pour vérifier la transformation de n'importe quel modèle de test entraîne une complexité importante et peu de réutilisabilité. Pour une intégration cohérente du test au développement dirigé par les modèles, nous montrons dans cette section que les oracles doivent pouvoir être modularisés, chaque module assurant des vérifications partielles.

Dans la section 4.1, nous présentons la problématique de l'oracle pour le test de transformation de modèles. Dans la section 4.2, nous proposons six fonctions d'oracle différentes qui exploitent différentes données d'oracle. Dans la section 4.3, nous traitons la construction de données d'oracle avec l'exemple class2rdbms. Dans la section 4.4, nous introduisons les qualités que nous voulons évaluer et les propriétés d'un oracle qui influencent sa qualification. Dans la section 4.5, nous comparons ces fonctions selon leur risque d'erreurs, ainsi que sur leur réutilisabilité.

4.1 La problématique de la construction d'oracles pour le test de transformations de modèles

Pendant la phase de test, un ensemble de cas de test est créé. Chacun se compose principalement d'une donnée de test et d'un oracle. La transformation d'un modèle de test produit un modèle de sortie. Un oracle analyse alors ce modèle de sortie pour vérifier si le cas de test passe ou échoue. Pour définir un *oracle*, il faut disposer d'une *fonction d'oracle* et **instancier** ses paramètres avec des *données d'oracle* :

Fonction d'oracle : *Une fonction d'oracle analyse un modèle de sortie et fournit un verdict. Elle est paramétrée par le modèle de sortie à vérifier et par des informations nécessaires à la production du verdict du test.*

Dans ce chapitre, nous proposons plusieurs fonctions d'oracle différentes. Elles varient selon les traitements qu'elles réalisent et le type d'information qu'elles exploitent sous la forme de données d'oracle :

Donnée d'oracle : *Une donnée d'oracle est utilisée comme paramètre d'une fonction d'oracle. Elle fournit des informations nécessaires à l'analyse d'un modèle de sortie. Elle définit les vérifications effectuées sur la transformation d'un modèle de test.*

Nous avons identifié six genres de donnée d'oracle différents, un modèle de sortie attendu en est un exemple. Nous avons défini une fonction d'oracle par genre de donnée d'oracle. Nous proposerons dans la section suivante les six fonctions d'oracle correspondantes. Dans le chapitre 5, nous présenterons l'outil qui permet d'appliquer ces différentes fonctions.

Dans le domaine du test, il est habituellement considéré que l'oracle est obtenu par une vérification de l'égalité entre le résultat attendu et le résultat obtenu. Dans ce cadre, une fonction de comparaison constituerait une fonction d'oracle et un résultat attendu serait une donnée d'oracle. Cette fonction d'oracle est générique et permet de produire le verdict à partir de la comparaison de n'importe quel couple (sortie attendue, sortie obtenue). Une fois mise en œuvre, elle peut être employée pour le test de n'importe quel programme. Chaque résultat attendu dépend du programme testé et de la donnée de test utilisée. De cette façon, nous mutualisons la partie fonction de l'oracle. Nous proposons cela pour permettre le développement d'un harnais de test indépendant de la transformation testée. Il s'agit d'une contribution technique réalisée et expliquée dans le chapitre suivant (sous-section 5.2.3). Par ailleurs, nous souhaitons évaluer la meilleure façon de réaliser l'oracle pour le test de transformations. Il n'est pas utile d'étudier chaque oracle puisqu'il est propre à la transformation d'un modèle de test donnée. En revanche, la considération d'une fonction d'oracle par rapport à une autre permet de déterminer laquelle est la plus utile.

Formellement, si E et S sont respectivement les domaines d'entrée et de sortie d'une transformation T, telle que :

$$\begin{aligned} T : E &\rightarrow S \\ me &\mapsto T(me) \end{aligned}$$

alors une fonction d'oracle *Of* est telle que :

$$\begin{aligned} Of : S \times D &\rightarrow Booléen \\ ms \times do &\mapsto Of(ms, do) \end{aligned}$$

avec D le domaine des données d'oracle utilisées par cette fonction d'oracle donnée. Finalement pour chaque cas de test Φ composé d'un modèle de test $Mt \in E$, nous définissons une donnée d'oracle $\delta \in D$ pour paramétrer la fonction d'oracle *Of* et définir un oracle Ω. Nous obtenons alors un cas de test Φ tel que $\Phi = (Mt, \Omega) avec\ \Omega = Of(T(Mt), \delta)$.

La figure 4-1 illustre le processus de construction d'oracle. Il est divisé en deux tâches réalisées indépendamment. Notre travail a consisté à construire et implémenter différentes fonctions d'oracle qui sont désormais proposées pour le test de n'importe quelles transformations de modèles. Le travail d'un testeur consiste à choisir une de ces fonctions dans

un premier temps, puis à la paramétrer avec une donnée d'oracle qui est construite à partir de la spécification et du modèle de test dont on veut vérifier la transformation.

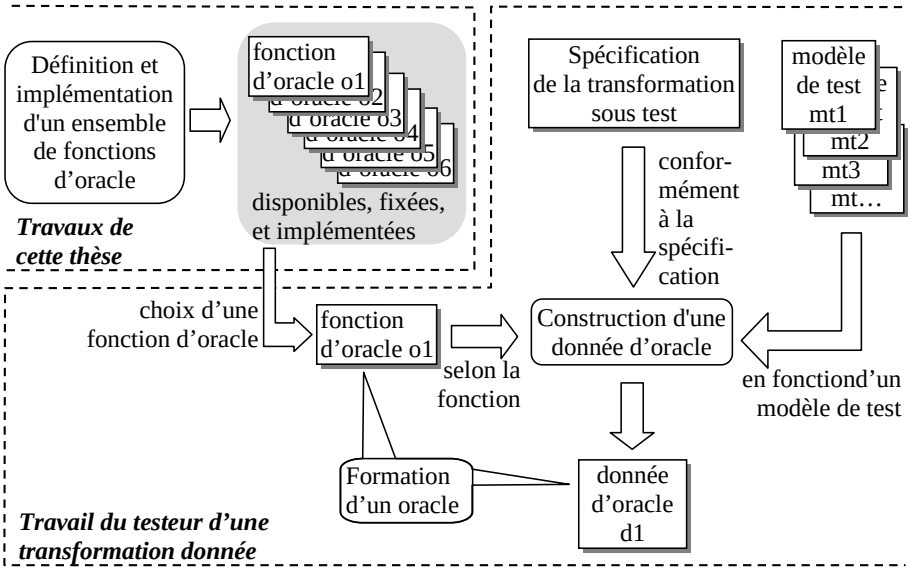


Figure 4-1 -Mise en œuvre d'oracle par constructions successives de fonctions et de données d'oracle

La problématique de la définition d'oracle se divise en deux :

- La problématique de la définition et l'outillage de fonctions d'oracle génériques
- La problématique de la construction de données d'oracle

Une fonction d'oracle doit analyser au moins un modèle : le modèle de sortie. Ce modèle de sortie est analysé en fonction du modèle de test transformé et de la spécification sous forme de données d'oracle. La complexité réside dans les paramètres utilisés pour la vérification. Il s'agit de mettre en rapport deux modèles selon la spécification de la transformation testée. Les fonctions d'oracle sont construites une fois pour toute et réutilisées en leur fournissant uniquement de nouveaux paramètres. L'utilisation d'une seule fonction d'oracle permet de vérifier plusieurs transformations avec différents modèles de test. Comme contribution de ce chapitre, nous définissons ces différentes fonctions d'oracles. Elles varient en fonction de leur méthode d'analyse des modèles de sortie et du type de donnée d'oracle qu'elles exploitent.

La deuxième problématique est la définition des données d'oracles. Elles sont écrites ou récupérées pour paramétrer les fonctions d'oracle, formant ainsi les oracles des cas de test. Une donnée d'oracle est un extrait plus ou moins complet de la spécification. Elle doit être exploitable de manière automatique. La spécification de la transformation de modèles prend rarement une forme exécutable. Si c'était le cas, elle constituerait une seule donnée d'oracle qu'une fonction d'oracle se contenterait d'exécuter. Il s'agit alors d'un oracle idéal mais irréaliste. Par exemple la spécification de la transformation class2rdbms ne comprend que deux contraintes OCL (donc exécutables), le reste étant fourni sous forme textuelle. Il n'existe d'ailleurs pas aujourd'hui de forme standardisée pour la définition de spécification de

transformations de modèles, cela même sans considération sur l'exécutabilité de celle-ci. C'est pourquoi la principale tâche dans la définition d'oracle est d'extraire de l'information de la spécification et de la retranscrire en donnée d'oracle sous une forme automatiquement exploitable. Pour la problématique de la définition de données d'oracle, notre contribution consiste à qualifier les différentes fonctions pour conseiller leur utilisation les données d'oracle qu'il est possible d'obtenir dans un contexte.

4.2 Construction de six fonctions d'oracle sur la base de trois techniques d'analyse de modèles

Dans cette section, nous proposons six fonctions d'oracle. Elles permettent de définir différemment des oracles en utilisant des techniques d'analyse de modèles et en exploitant différents types de données d'oracle. La définition de ces fonctions permet par la suite de proposer un harnais de test supportant ces fonctions et à restreindre le travail d'élaboration d'oracles à la construction de données d'oracle.

4.2.1 Trois techniques pour analyser les modèles

Trois techniques peuvent être utilisées en Ingénierie Dirigée par les Modèles (IDM) pour leur capacité à analyser des modèles. Nous les exploitons pour implémenter des fonctions d'oracle de transformations de modèles.

a- Comparaison de modèles pour l'oracle

Différentes fonctions d'oracle peuvent employer la comparaison de modèles (introduite sous-section 2.5.2). Pour utiliser cette technique, la fonction d'oracle doit comparer deux modèles : l'un obtenu après l'exécution du test, l'autre obtenu à partir de la donnée d'oracle. Elles peuvent ainsi comparer le modèle renvoyé par la transformation du modèle de test avec un modèle de référence disponible ou obtenu par le testeur.

b- Contrats pour l'oracle

Les pré et post-conditions forment les contrats d'une méthode. Ce sont des assertions qui sont évaluées avant et après l'exécution d'une méthode [Meyer'92a]. Des contrats peuvent être définis pour une transformation de modèles. Les pré-conditions contraignent l'ensemble des modèles en entrée et les post-conditions spécifient un ensemble de propriétés attendues sur les modèles de sortie.

Les travaux présentés dans [Le Traon'06, Ciupa'08, Briand'03] montrent l'intérêt de l'utilisation des contrats comme fonction d'oracle dans des systèmes orientés objet. Les bénéfices de ces approches peuvent être adaptés à l'ingénierie des modèles.

La mise en œuvre de contrats peut être réalisée de différentes manières comme :

- Des contraintes OCL.
- Des contraintes Kermeta, le langage supporte nativement la transformation de modèles et les contrats (fonctionnalité que nous avons mise en œuvre, sous- section 5.2.2).

- D'autres systèmes de contraintes appropriées au paradigme choisi pour l'implémentation de la transformation testée : ATOM3 par exemple pour la transformation de graphes [de Lara'02].

c- Pattern matching pour l'oracle

Dans l'ingénierie des modèles, le pattern matching permet d'identifier les ensembles d'objets d'un modèle qui correspondent au pattern.

La création de patterns est utile à la définition d'oracle pour vérifier dans un modèle de sortie la présence de structures attendues. Un pattern permet de signaler la présence, l'absence d'éléments, et permet de récupérer des objets trouvés dans un modèle de sortie. Dans ce cas, une simple assertion permet de faire des vérifications sur ces objets pour produire le verdict.

La définition des patterns et la mise en œuvre de leur recherche peut être réalisée de différentes manières :

- *Assertions OCL* : Il est possible de naviguer un modèle et d'en sélectionner les éléments au moyen d'une contrainte OCL. Dans le cadre de l'expression de patterns pour l'oracle, il s'agit d'assertions associées à un modèle donné, celui issue de la transformation d'un modèle de test donné.
- Patterns basés sur des *modèles snippets* [Ramos'07] : Les model snippets contiennent des objets qui sont des instances des méta-classes du méta-modèle et associés selon les relations définies dans le méta-modèle mais ils ne satisfont pas les cardinalités et les contraintes du méta-modèle. Le framework développé dans l'équipe Triskell permet d'écrire, de définir, et de rechercher des patterns en vérifiant la présence des model snippets dans un modèle, ainsi que de récupérer des pointeurs vers les éléments du modèle respectant le pattern. Des assertions peuvent alors procéder à des vérifications sur ces objets, comme leur nombre, leurs propriétés, etc., pour produire le verdict.

Ainsi exprimés, les assertions utilisant des patterns doivent être satisfaites après la transformation d'un modèle de test particulier. Chaque assertion ou une conjonction d'assertions peut être associée à un cas de test comme donnée d'oracle d'une fonction d'oracle.

4.2.2 Six fonctions d'oracle pour le test de transformations de modèles

Ces trois techniques sont utilisées pour réaliser six fonctions d'oracles. Ces fonctions sont paramétrées par le modèle de sortie devant être analysé et une donnée d'oracle. Elles retournent le verdict du test sous la forme d'un booléen. Ainsi certaines fonctions exploitent la même technique différemment en fonction du type de donnée d'oracle. Ces fonctions sont utilisées pour former des oracles pour le test d'une transformation de modèles `Tr` telle que :

$$\begin{aligned} Tr : E &\rightarrow S \\ me &\mapsto Tr(me) \end{aligned}$$

a- Fonction d'oracle utilisant une version de référence de la transformation sous test :

L'oracle produit le verdict en effectuant une comparaison entre :

- le *modèle de sortie* (`ms`) renvoyé par la transformation d'un modèle de test
- un *modèle de référence* renvoyé par une transformation de référence.

La donnée d'oracle qu'il faut construire est une transformation de référence (`Tref`). Il s'agit d'une transformation fonctionnellement similaire à la transformation de modèles sous test. Elle est telle que :

$$\begin{aligned} Tref &: E \to S \\ me &\mapsto Tref(me) \end{aligned}$$

Elle est utilisée en couple avec le modèle de test (`mt`) d'un cas de test pour paramétrer la fonction. Cette transformation de référence produit le modèle de référence à partir du modèle de test (`mt`). Cette fonction d'oracle O_1 est telle que :

```
O1(ms , (Tref,mt)) : Boolean is do
   result := compare(ms , Tref(mt)).booleanResult
end
```

b- Fonction d'oracle utilisant une transformation inverse

L'oracle produit le verdict en effectuant une comparaison entre :

- le *modèle de test* (`mt`)
- le modèle d'entrée obtenu après deux transformations successives du modèle de test : avec la transformation sous test, puis avec sa *transformation inverse*.

La donnée d'oracle qu'il faut construire est une transformation inverse (T^{-1}). Elle ne peut exister que si la transformation sous test `T` est une application injective (ce qui n'est pas systématique mais nécessaire) telle que pour une transformation $Tr : E \to S, \forall (x, y) \in E^2, (x \neq y \Rightarrow Tr(x) \neq Tr(y))$. Une transformation inverse T^{-1} est telle que :

$$\begin{aligned} T^{-1} &: S \to E \\ ms &\mapsto T^{-1}(ms) \end{aligned}$$

Elle est utilisée en couple avec le modèle de test (`mt`) d'un cas de test pour paramétrer la fonction. Cette fonction d'oracle O_2 est telle que :

```
O2(ms , (T-1,mt)) : Boolean is do
   result := compare(mt , T-1(ms))
end
```

c- Fonction d'oracle utilisant un modèle de sortie attendu

L'oracle produit le verdict en effectuant une comparaison entre :

- le *modèle de sortie* (ms) renvoyé par la transformation d'un modèle de test
- le *modèle attendu* (ma)

La donnée d'oracle qu'il faut construire est un modèle attendu (ma). Un modèle attendu appartient au domaine de sortie S de la transformation. Il est donc conforme au méta-modèle cible et satisfait les post-conditions de la transformation qui définissent S. Ce modèle attendu est utilisé pour la vérification de la transformation du modèle de test d'un cas de test précis. Ainsi le modèle de test ne paramètre pas la fonction. Cette fonction d'oracle O_3 est telle que :

```
O3(ms,ma) : Boolean is do
   result := compare(ms, ma)
end
```

Cette fonction d'oracle est la solution retenue dans plusieurs travaux sur le test de transformation de modèles. Notre contribution consiste à présenter des alternatives et à qualifier toutes ces fonctions pour les comparer.

d- Fonction d'oracle utilisant un contrat générique

L'oracle produit le verdict en vérifiant que le modèle de sortie satisfait le contrat générique Cg en fonction du modèle de test (mt).

La donnée d'oracle qu'il faut construire est un contrat générique Cg. Un contrat générique est une post-condition de la transformation. Il contraint n'importe quel modèle de sortie (ms) en fonction du modèle de test (mt) transformé. Ce contrat peut vérifier l'exactitude de ce modèle de sortie vis-à-vis d'une partie plus ou moins grande de la spécification. Dans l'exemple class2rdbms, un contrat générique vérifie qu'aucune classe non persistante de n'importe quel modèle de test ne correspond à une table du modèle de sortie. Un contrat générique peut être : extrait de la spécification, fournit par le développeur ayant suivi une méthodologie design-by-contract, ou produit par le testeur. Il est tel que :

$$Cg : E \times S \rightarrow [vrai/\ faux]$$
$$me\ , ms \mapsto \beta$$

avec β à vrai si et seulement si (me,ms) satisfait Cg. Cg est utilisé en couple avec le modèle de test (mt) d'un cas de test pour paramétrer la fonction. Cette fonction d'oracle O_4 est telle que :

```
O4(ms, (Cg,mt)) : Boolean is do
   result := (ms,mt).satisfies(Cg)
end
```

e- Fonction d'oracle utilisant une assertion OCL

L'oracle produit le verdict en vérifiant que le modèle de sortie satisfait l'assertion OCL (Aocl).

La donnée d'oracle qu'il faut construire est une assertion OCL (`Aocl`). Il s'agit d'une contrainte qui, à la différence d'un contrat générique, est dédiée à la vérification d'un modèle de sortie donnée (`msd`). Dans l'exemple class2rdbms, une assertion OCL vérifie qu'un modèle de sortie donné ne contient pas de table nommée « D ». Une assertion OCL est telle que :

$$\begin{aligned} Aocl : S &\rightarrow [vrai/\ faux] \\ msd &\mapsto \beta \end{aligned}$$

avec β à vrai si et seulement si `msd` satisfait `Aocl`. Cette contrainte est utilisée pour la vérification de la transformation du modèle de test d'un cas de test précis. Ainsi le modèle de test ne paramètre pas la fonction. Le testeur va exprimer dans cette contrainte les propriétés requises dans le modèle de sortie. Il n'est pas imposer d'exprimer simultanément toutes les propriétés. Cette fonction d'oracle `O5` est telle que :

```
O5(ms, Aocl) : Boolean is do
   result := ms.satisfies(Aocl))
end
```

f- Fonction d'oracle utilisant un pattern fait de model snippets

L'oracle produit le verdict en vérifiant que le modèle de sortie contient les éléments définis dans le pattern.

La donnée d'oracle qu'il faut construire est une assertion basée sur un pattern fait de model snippets (`pattern_with_snippets`). L'opération de pattern matching renvoie la liste des « matchs » trouvés sous la forme de `PObjects`. Le framework du pattern matching ajoute une superclasse abstraite `PObject` à chaque classe du méta-modèle de sortie de la transformation. Les matchs retournés par le pattern matching sont donc bien des objets du modèle de sortie nous détaillons l'emploi des model snippets au point 4.3.1e-).

Une assertion (`Ams`) permet de vérifier la présence (ou l'absence) d'éléments parmi les matchs. Fournir un pattern revient à fournir aussi les model snippets qu'il utilise. Ainsi l'assertion est `Ams` telle que :

$$\begin{aligned} Ams : S, P^n &\rightarrow [vrai/\ faux] \\ ms, \{\rho\} &\mapsto Ams(ms, \{\rho\}) \end{aligned}$$

où P définit le domaine des PObjects, l'application de la recherche d'un pattern dans un modèle renvoyant une liste de PObject. Cette fonction d'oracle `O6` est telle que :

```
O6(ms, (Ams, pattern_with_snippets) : Boolean is do
   result := satisfied(Ams, pattern_with_snippets)
end
```

4.3 Construction de données d'oracle

Chaque fonction d'oracle manipule un type de donnée d'oracle spécifique (tableau 4-1). Dans cette section, nous illustrons la construction et l'utilisation des données d'oracles de chaque fonction. Pour cela nous utilisons la transformation class2rdbms que nous faisons évoluer pour étudier la réutilisabilité des données d'oracle.

La spécification de la transformation class2rdbms (`T`) a été présentée au chapitre 2 (section 2.2.2). Nous ne rentrons pas dans les détails de la totalité de la spécification mais nous nous intéressons à la règle suivante:

`Ru : « Les classes persistantes, et uniquement celles là, sont transformées en tables dont les noms correspondent, sauf si elles héritent directement ou indirectement d'une autre classe persistante »`

Par cette règle, à partir du modèle d'entrée de la figure 4-2, la transformation produit dans le modèle de sortie (figure 4-3) la table nommée B et uniquement celle-là.

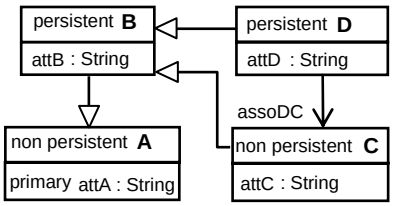


Figure 4-2. Modèle d'entrée Mt1

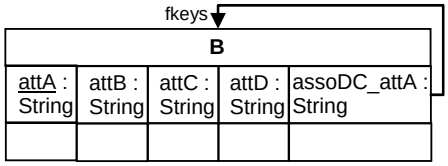


Figure 4-3. Modèle de sortie T(Mt1)

Dans cette section illustrative, nous considérons tout d'abord le modèle de test de la figure 4-2 et vérifions que le modèle de sortie produit par la transformation de modèles sous test respecte la règle `Ru`. Dans un deuxième temps, nous présentons l'ensemble des oracles correspondant aux modèles de test sélectionnés et qualifiés par l'analyse de mutation et les techniques étudiées dans le chapitre précédent. Les oracles complets sont donnés en Annexe A.

Nous étudions la réutilisabilité d'un oracle en créant une évolution `T'` de la transformation `T` par une modification de la règle `Ru` :

`Ru' : « Les classes persistantes, et uniquement celles là, sont transformées en tables dont les noms correspondent »`

Avec cette nouvelle spécification, la transformation `T'` devrait produire le modèle RDBMS de la figure 4-4 à partir du modèle de test `Mt1`.

Pour élaborer les oracles, nous avons utilisé chacune des fonctions d'oracle paramétrées avec des données d'oracle de différents types (tableau 4-1).

Fonction d'oracle	Donnée d'oracle	Type de donnée
o_1	Transformation de référence	Transformation de modèles
o_2	Transformation inverse	Transformation de modèles
o_3	Modèle attendu	Modèle de sortie
o_4	Contrat générique	Contrainte (OCL)
o_5	Assertion OCL	Contrainte OCL
o_6	Assertion basée sur des model snippets	Model snippet

Tableau 4-1 - Fonctions d'oracle avec leur donnée d'oracle principale et leur type

4.3.1 Illustration de la vérification de la transformation d'un modèle de test

a- Transformation de référence

Une transformation de référence est disponible pour l'oracle dans le cadre de travaux de migration ou d'amélioration d'implémentation. Dans ces contextes, le but est de changer l'implémentation de la transformation sans en changer le fonctionnement. Si aucune transformation de référence n'est disponible, il est nécessaire d'en développer pour utiliser la fonction d'oracle correspondante.

Dans notre cas, la transformation sous test a été implémentée en Kermeta [Muller'05b]. Une des autres implémentations réalisées pour le workshop MTIP peut être utilisée comme version de référence. Nous avons choisi celle mise en œuvre en Tefkat par Lawley et al. [Lawley'05] .

Pour vérifier la transformation du modèle de test avec la transformation adaptée T', il a fallu adapter la transformation de référence. Nous avons dû prendre en main le langage Tefkat, comprendre l'implantation de la transformation de référence, identifier et modifier la règle qui transforme les classes en colonnes.

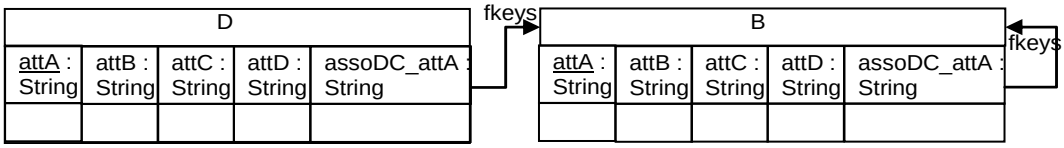


Figure 4-4. Modèle de sortie T'(Mt1)

b- Transformation inverse

La transformation étudiée n'est pas injective, deux modèles d'entrée peuvent être transformés en un même modèle de sortie. En effet des éléments d'un modèle d'entrée ne sont pas transformés (par exemple les classes non persistantes). En conséquence, les éléments non transformés du modèle de test ne peuvent pas être déduits du modèle de sortie. C'est le cas des classes nommées A et C du modèle Mt1. Ainsi il n'est pas possible d'écrire de transformation inverse pour cet exemple [Podnieks'05], ce qui est souvent le cas d'après notre expérience.

c- Modèle attendu

Le testeur doit écrire les modèles attendus illustrés figure 4-3 et figure 4-4.

d- Contrat générique

Il est possible d'écrire ce contrat générique en OCL :

```
1  post table_correctly_created :
2  result.table.size = inputModel.classifier.select(cr|cr.oclIsTypeOf(Class))
3   .select(cs | cs.oclAsType(Class).is_persistent)
4   .select(csp | not csp.oclAsType(Class).parents.exists(p | p.is_persistent))
5   .size
6  and //note: les classes ont des noms différents
7  inputModel.classifier.select(cr | cr.oclIsTypeOf(Class))
8   .select(cs | cs.oclAsType(Class).is_persistent)
9   .select(csp | not csp.oclAsType(Class).parents.exists(p | p.is_persistent))
10  .forAll(csp | result.table.exists(t | t.name = csp.name))
```

Ce contrat peut être adapté pour la nouvelle version de la transformation en enlevant les deux `select(…)` en gras (lignes 4 et 9)

D'autres contrats sont utilisés pour vérifier les autres parties de la transformation. Certains de ces contrats doivent aussi considérer cette modification de la transformation. Par exemple, la vérification de la correspondance entre les attributs et les colonnes prend en compte les classes et les tables qui les contiennent. La modification de la création des tables va donc influencer la vérification de la création de leurs colonnes. Ainsi les opérations de navigation et de filtrage sont répétées d'un contrat à un autre. Ce qui implique que la modification d'une règle peut impacter plusieurs contrats même s'ils ne sont pas directement dédiés à la vérifier.

e- Model snippets

Dans ce point, nous détaillons l'utilisation de model snippets pour la définition d'oracle. L'utilisation de cette catégorie de patterns pour l'oracle est originale à cette thèse, et sa mise en œuvre est moins commune que celle des autres oracles.

Des model snippets sont des modèles partiels au sens où ils sont partiellement conformes au méta-modèle utilisé. Ils sont conformes à une version de ce méta-modèle relâchée en supprimant les contraintes : les cardinalités 1 sont changées en 0..1, les 1..* en 0..*, les invariants sont supprimés, ainsi que les valeurs par défaut des propriétés. Ainsi la version relâchée du méta-modèle RDBMS est donnée en figure 4-5, ce méta-modèle n'a plus ni cardinalité à 1, ni invariant. Chacune de ses classes héritent de `PObject` qui vient du méta-modèle du framework de pattern matching. Ainsi en appliquant le pattern matching, seules des instances de `PObjects` sont manipulées et référencées. De cette manière le framework est générique et peut être employé avec n'importe quel méta-modèle.

Dans notre exemple, il est possible de créer des model snippets qui ne contiennent que des instances de `Table` ou qui contiennent des `FKey` sans colonne, ce qui ne serait pas conforme au méta-modèle RDBMS présenté figure 2-7.

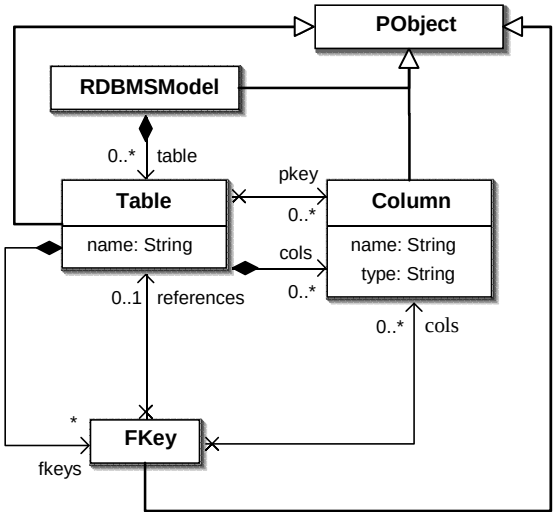


Figure 4-5 - Version relâchée du méta-modèle RDBMS

Ramos et al. [Ramos'07] ont proposé un framework complet permettant de générer ce méta-modèle relâché. L'outillage proposé permet de combiner des model snippets dans des patterns conformes au méta-modèle du framework [Ramos'07]. Un pattern fournit des fonctionnalités en plus du model snippet qui ne définie d'un modèle partiel. Le pattern permet de définir des « Rôle » qui doivent être considérés comme des variables permettant de consulter les objets du modèle trouvés grâce au pattern. Un pattern peut aussi exploiter d'autres model snippets pour définir des faux positifs. Les faux positifs sont des objets du modèle qui correspondent au model snippet qui sont rejetés. Le framework permet d'écrire des model snippets conformes à un méta-modèle relâché ainsi que des patterns basés sur ces model snippets.

Dans le cadre de notre exemple, en utilisant la version relâchée du méta-modèle RDBMS, nous écrivons le model snippet de la figure 4-6. Il contient une unique instance de `Table` seulement nommée « A », présentée avec une syntaxe concrète. Avec ce model snippet, nous écrivons le pattern de la figure 4-7. Il permet de trouver dans un modèle le nombre d'instance de `table` simplement nommée « A ». Ce pattern peut être précisé comme dans la figure 4-8 en ajoutant un suffixe au nom de l'objet considéré. Le suffixe est séparé par le symbole « @ » qui désigne un rôle du pattern. Dans le cas de la figure 4-8, la variable x permet de consulter toutes les instances de `Table` simplement nommées « A » contenues dans un modèle sur lequel nous appliquons le pattern. Alors que le pattern P1 ne permet que de connaître le nombre de `Table` simplement nommées « A » contenues dans un modèle. Dans le cas de la figure 4-9, il s'agit de trouver toutes les instances de Table sans considérer aucune de ses propriétés.

L'outillage proposé permet d'utiliser librement un éditeur de modèles. Il est par exemple possible d'utiliser la syntaxe concrète de l'éditeur réflexif d'EMF, ou une autre syntaxe concrète définie avec TopCased (par exemple). Cela présente l'avantage de permettre au concepteur et au testeur de pouvoir utiliser les outils avec lesquels ils sont familiers mais aussi de pouvoir s'affranchir de l'utilisation d'une syntaxe abstraite qui oblige à manipuler tous les concepts du méta-modèle. Il est ainsi possible pour le testeur d'étudier, d'écrire, et de modifier ses patterns

et ses modèles de test de la même manière et en utilisant une syntaxe concrète plus simple dès que cela est possible.

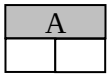


Figure 4-6 - Un modèle snippet MS1 de RDBMS

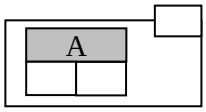


Figure 4-7 - Un pattern P1 de RDBMS basé sur MS1

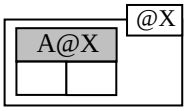


Figure 4-8 - Un pattern P2 de RDBMS basé sur MS1

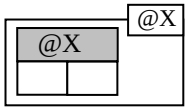


Figure 4-9 - Un pattern P3 de RDBMS basé sur un model snippet

A partir de cette définition des patterns, nous définissons un DSL (Domain Specific Language) pour l'écriture d'assertions utilisant ces patterns à base de model snippets. Avec ces assertions, nous vérifions la présence, l'absence, le nombre d'éléments dans un modèle. Nous avons défini ce DSL pour la définition de donnée d'oracle pour la fonction d'oracle O_6. Ce langage défini un nombre très limité de concepts pour consulter les propriétés des objets retournés par le pattern matching et vérifier la présence d'objets attendus dans les modèles de sorties.

Tout d'abord, il permet d'appeler l'opération de pattern matching définie dans le framework [Ramos'07]. Le pattern matching est appliqué sur un modèle appartenant à un domaine D, défini par un méta-modèle. Dans notre cas, ce domaine correspond au domaine de sortie de la transformation :

$$\begin{aligned} patternmatching : Pattern \times D &\rightarrow \left(\left(Rôle \times PObject\right)^{m}\right)^{n} \\ x \quad , y &\mapsto pattermatching(x, y) \end{aligned}$$

Cette opération `patternmatching` renvoie la liste des éléments trouvés dans le modèle qui correspondent au pattern. La différence d'utilisation des patterns P1 ou P2 est sensible. En effet, avec P2 le rôle X permet de consulter les éléments trouvés. Avec P1, seule la taille de l'ensemble des objets trouvés est consultable par l'opération `size`. La présence des éléments dans le modèle intéresse l'oracle, nous définissons donc une lambda expression `contain`. Elle utilise les rôles comme lambda et son expression est une conditionnelle qui vérifie les propriétés de type primitif ou du même type que le rôle :

$$\begin{array}{lll} contains : \left(\left(Rôle \times PObject\right)^m\right)^n, Rôle, expression\,booléenne & \rightarrow [vrai/\ faux] \\ \quad x \qquad\qquad\qquad\qquad , \quad y \quad , \quad z & \mapsto x.contains(y|z) \end{array}$$

La vérification de l'absence d'élément est aussi nécessaire et est réalisée avec l'opérateur unitaire `not`, utilisé avec l'opération `contains`.

L'assertion créée à partir de ces opérations peut se composer de plusieurs vérifications. Chacune renvoie un booléen. Elles sont mises en relation dans une opération logique utilisant les opérateurs binaires : `and`, `or`, et `xor`. Ces opérateurs sont également utilisés dans l'expression conditionnelle de `contains` ainsi que l'opérateur `not`. Pour éviter d'appeler l'opération `patternmatching` à chaque vérification il est possible d'affecter une variable avec l'opération `let in` et de l'utiliser dans le reste de l'assertion.

L'expressivité de ce langage est limitée car son but n'est que la définition d'assertions en ajoutant un supplément de modularité au framework de pattern matching. Il complète la modularité offerte par la définition de pattern à base de model snippet. Pour cela, nous pouvons utiliser un pattern et ses model snippets pour trouver plusieurs objets d'un modèle et les analyser dans l'assertion définie.

En utilisant le langage développé dans nos travaux et le framework existant, nous définissons des assertions pour vérifier la règle `Ru`. Les patterns P1, P2, ou P3 peuvent être utilisés. Néanmoins, l'utilisation de P1 et P2 est peu modulaire car elle impose la définition d'autres patterns avec des model snippets dont les tables seraient nommées explicitement « B », « C », « D », et sans nom. L'utilisation du pattern P3 est donc plus adaptée et permet l'écriture de quatre assertions pour former autant de données d'oracle :

```
do1 : let PM = patternmatching(P3, mout1) in
          not PM.contains(x | x.name=A) and
          not PM.contains(x | x.name=C)
do2 : patternmatching(P3, mout1).contains(x | x.name=B)
do3 : not patternmatching(P3, mout1).contains(x | x.name=D)
do4 : patternmatching(P3, mout1).size = 1
```

- La donnée d'oracle `do1` permet de vérifier que « Les classes persistantes […] sont transformées en tables dont les noms correspondent » : puisqu'il y a une classe persistante `B` dans le modèle de test, `do1` vérifie la présence d'une table `B` dans le modèle de sortie.
- `do2` permet de vérifier « et uniquement celles là » : puisqu'il y a deux classes non persistantes dans `Mt1`, `do2` permet de vérifier qu'il n'y a pas de table de même nom.
- `do3` permet de vérifier « sauf si elles héritent directement ou indirectement d'une autre classe persistante » : puisqu'il y a une classe persistante `D` qui hérite d'une classe persistante, `do3` permet de vérifier qu'il n'y a pas de table `D`.
- `do4` permet de vérifier aussi « et uniquement celles là » mais sans considérer spécifiquement l'attribut persistant : `do4` permet de vérifier qu'une seule table est créée dans le modèle de sortie.

Les model snippets simplifient l'écriture et la modularisation de l'oracle en fonction de la règle considérée. Ils sont facilement réutilisables avec la nouvelle version `Ru'` de la règle `Ru` : `do1` et `do2` sont similaires. `do3` est adapté en retirant le `not` puisqu'une classe persistante même héritant d'une classe persistante doit désormais être transformée. Le 1 de `do4` devient un 2 puisque le modèle de sortie doit contenir deux tables.

f- Assertions OCL

Les patterns des quatre précédents oracles peuvent être implémentés avec des assertions OCL. Par exemple le second pattern serait :

```
result.table.select(t|t.name=A).size()=0 and
result.table.select(t|t.name=C).size()=0
```

Ces assertions sont également réutilisables facilement. Elles sont un peu moins simples à écrire que les model snippets car, même si apprendre un langage tel qu'OCL n'est pas complexe, l'écriture de patterns de la même façon que les modèles de test (avec une syntaxe concrète et un éditeur de modèles) est un avantage des model snippets.

4.3.2 Vérification complète de la transformation sous test

Dans la précédente sous-section, nous avons illustré la construction de donnée d'oracle pour vérifier une règle de la spécification (et une adaptation de celle-ci) avec un modèle de test. Pour vérifier la transformation entière, il est nécessaire d'écrire un ensemble d'oracles qui vérifie les transformations d'un ensemble de modèles de test en considérant toutes les règles de la spécification.

Nous montrons ici que le passage à l'échelle se fait différemment selon la fonction utilisée et le type de donnée d'oracle qu'elle emploie.

a- Transformations

Tout d'abord, la définition des transformations de référence et inverse est réalisée une seule fois pour une version de la transformation testée. Elles sont utilisables dans n'importe quel cas de test, quelque soit le modèle de test.

L'implémentation de la transformation sous test class2rdbms [Muller'05b] se compose de cent treize lignes de code Kermeta en onze opérations. L'implémentation réalisée par Lawley et al. [Lawley'05] qui sert de version de référence est un programme fonctionnel de quatre-vingt-quatorze lignes de code Tefkat, contenant huit patterns et cinq règles. La complexité de cette seconde implémentation est importante, sa validité n'est pas assurée davantage que cette de la transformation sous test. De plus quand le testeur va devoir adapter cette transformation de référence à la nouvelle spécification, il devra apprendre un nouveau langage, comprendre la mise en œuvre et la modifier correctement. Ainsi, même si le testeur peut tirer avantage d'une transformation de référence existante, sa difficile réutilisation avec une nouvelle version et le risque dû à sa complexité n'assurent pas la qualité de cet oracle. L'écriture d'une version de référence est trop complexe, c'est une tâche de développeur plutôt que de testeur.

b- Contrats génériques

La définition des contrats génériques ne varie pas selon le modèle de test mais selon les règles qu'ils considèrent. Ainsi, il est nécessaire d'écrire différents contrats pour vérifier les différentes règles de la spécification.

Le contrat utilisé pour mettre en œuvre la vérification de la règle Ru n'est pas très complexe, mais la règle considérée est une des plus simples. Il est possible de l'écrire différemment en particulier avec un autre langage. Pour tester toute la transformation, un ensemble de contrats génériques doit être écrit pour considérer toutes les exigences. Nous avons eu besoin de quatorze contrats pour cela, ils sont présentés en Annexe A (A.3).

Une fois écrit, chaque contrat peut être utilisé dans des cas de test quel que soit le modèle de test. Ainsi le découpage des tests se fait uniquement selon les règles considérées.

c- Modèle attendu

Il est nécessaire d'écrire autant de modèles attendus que de modèles de test. Dans ce cas, aucune vérification précisément choisie n'est considérée et le découpage des tests se fait en fonction des modèles de test utilisés. Dans cet exemple, nous avons eu besoin d'un minimum de six modèles de test pour obtenir un score de mutation de 100%. Nous avons donc écrit six modèles attendus qui sont présentés en Annexe A - A.2.

d- Patterns

L'utilisation de patterns permet un découpage précis en fonction du modèle de test et de la vérification souhaitée. Ainsi en Annexe A, nous présentons l'ensemble des données d'oracle utilisant des model snippets et des assertions OCL. Ils sont en même nombre car ils expriment la même vérification de deux manières différentes. Nous verrons par la suite que cette différence n'est pas négligeable en termes de complexité et de réutilisation.

4.4 Permettre de qualifier les fonctions d'oracle

Nous mettons en avant deux qualités d'un oracle que nous voulons évaluer. Il s'agit du *risque d'erreur* de l'oracle, en association avec sa *complexité*, et de la *réutilisabilité* de l'oracle. Nous évaluons ces qualités pour mettre en avant les fonctions d'oracles les plus adaptées pour le test de transformations de modèles. Ces deux qualités sont nécessaires pour le test de transformation car elles concordent avec l'objectif de l'ingénierie dirigée par les modèles de permettre la réutilisation d'artéfacts de qualité.

Nous remarquons que nous évaluons le risque d'erreur qui est une qualité « négative ».

4.4.1 Deux qualités pour l'oracle de transformation de modèles

Les deux qualités évaluées portent sur les oracles définis à partir des différentes fonctions d'oracle et des différents types de données d'oracle.

a- Risque d'erreur

Tout d'abord, nous cherchons à évaluer le *risque d'erreur* introduit par l'oracle. En raison des difficultés dans la définition d'oracles, des erreurs sont commises. Les erreurs peuvent être de deux types :

– *faux-négatif* : l'oracle annonce une erreur qui n'en est pas une. Dans ce cas, une phase de diagnostic est nécessaire pour comprendre la défaillance annoncée, détecter le code qui serait erroné, corriger la faute suspectée. Dans le meilleur des cas, les conséquences sont un gaspillage de temps, de moyens, d'énergie. Dans le pire des cas, une correction inutile peut être une erreur introduisant une faute dans la transformation. Ce type d'erreur survient quand l'oracle utilisé est erroné.

– *faux-positif* : l'oracle annonce le succès d'un test alors qu'une erreur est présente. Ce type d'erreur survient quand la donnée d'oracle utilisée est erronée ou que les vérifications qu'elle définit sont incomplètes si elle ne considère qu'une partie du modèle de sortie ou de la spécification.

Pour augmenter la qualité d'un oracle, il doit relever toutes les vraies défaillances et indiquer les vraies fautes en procédant à tous les contrôles nécessaires. Néanmoins, il n'est pas nécessaire que l'oracle d'un unique cas de test réalise la totalité des vérifications, mais les oracles de l'ensemble de cas de test d'une transformation de modèles doivent le réaliser.

Le risque d'erreur est la conséquence de la *complexité* à construire des fonctions d'oracle et des données d'oracle. Nous faisons confiance aux fonctions d'oracles car nous les avons développées une fois et vérifiées. Nous présentons cette mise en œuvre dans le chapitre suivant. C'est un travail ne devant pas être réalisé à nouveau. En revanche le risque est élevé pour les multiples définitions de données d'oracle. Ainsi nous corrélons directement le risque d'erreur de l'oracle à la complexité des données d'oracle et de leur définition. Nous évaluons cette complexité à deux niveaux :

- au niveau de chaque donnée d'oracle et de sa définition, selon son type.
- au niveau des ensembles d'oracles utilisés pour la vérification de la transformation complète. Il n'est pas suffisant de comparer la complexité des données une à une. Il est nécessaire, par exemple, de créer plusieurs modèles attendus pour vérifier la transformation complète alors qu'une seule transformation de référence suffit.

b- Réutilisabilité

Les transformations de modèles dans un développement dirigé par les modèles sont sujettes à de nombreuses évolutions. Même si elles sont développées pour être réutilisées d'un projet à un autre, elles doivent généralement être adaptées au préalable. Les adaptations peuvent concerner des changements dans les méta-modèles sources et cibles, des modifications ou suppressions de règles de la spécification, l'ajout de nouvelles exigences. Un tel exemple de réutilisation de transformation de modèles est donné par Fleurey et al. dans [Fleurey'07b]. Ils présentent une série de transformations pour la migration d'application qui nécessitent d'importantes adaptations pour chaque nouveau projet de migration.

Réutiliser un cas de test pour une nouvelle version peut nécessiter l'adaptation de son oracle. C'est le cas si les vérifications qu'il effectue sont altérées par les changements de la spécification. Ces adaptations peuvent être coûteuses. Dans ce cas l'intérêt de la réutilisation de la transformation est atténué par le coût supplémentaire de l'adaptation de son test. Pour éviter cela, nous montrons que l'utilisation d'une fonction d'oracle plutôt qu'une autre permet de minimiser le nombre de modifications d'oracle. Le but est qu'une modification de la spécification affecte moins les cas de test comme illustré figure 4-10.

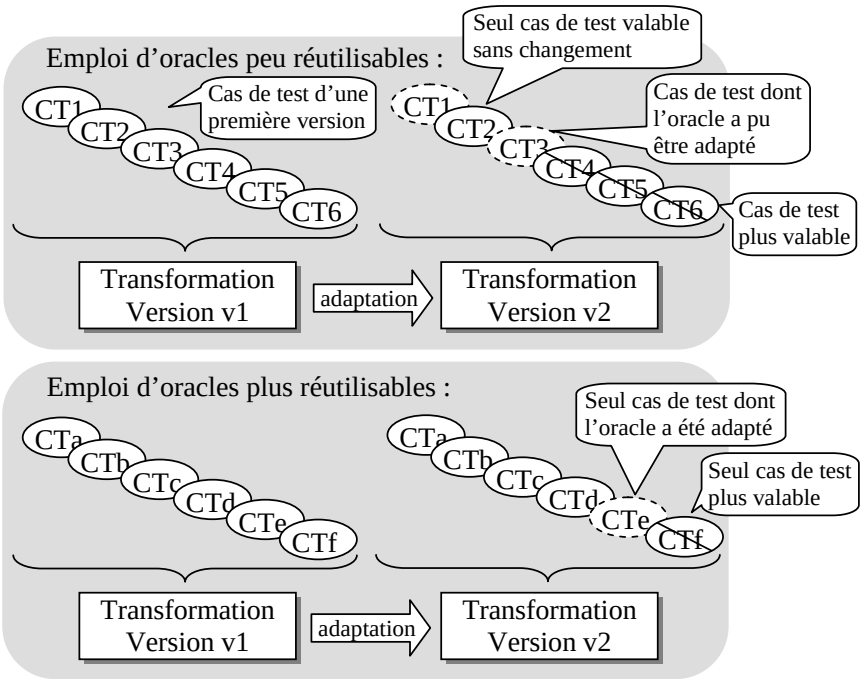


Figure 4-10 - Emploi d'oracles différemment réutilisables

D'une autre manière, nous considérons la réutilisation des oracles dans différents cas de test d'une même version de la transformation. Par exemple, si la transformation n'est pas injective, il est possible que deux modèles de test soient transformés en un même modèle de sortie. Ou si certaines propriétés sont communes à différents modèles de sortie, alors une vérification partielle de celles-ci peut être partagée par différents cas de test.

4.4.2 Cinq propriétés d'oracle

Nous identifions plusieurs propriétés d'oracle qui, selon leurs valeurs, influencent les qualités d'un oracle. Ces propriétés dépendent à la fois de la fonction d'oracle considérée ainsi que de la donnée d'oracle qu'elle utilise.

a- Généricité

Un oracle est générique s'il peut être utilisé pour vérifier la transformation de n'importe quel modèle de test. Un oracle générique utilise une donnée d'oracle générique et une fonction d'oracle générique. Nous n'avons pas identifié de fonctions d'oracle utilisant aussi bien des données d'oracle génériques et non génériques.

Une donnée d'oracle non générique est nécessairement associée à un modèle de test pour paramétrer une fonction d'oracle. Par exemple, une transformation de référence est générique mais elle ne permet pas seule de vérifier un modèle de sortie si elle n'est pas associée au modèle de test transformé.

Oracle générique : *un oracle est générique s'il n'est pas dédié à un modèle de test particulier mais peut être utilisé dans tous les cas de test d'une transformation de modèles. Dans ce cas il utilise comme paramètre une donnée d'oracle générique, qui est utilisable pour la vérification de la transformation de n'importe quel modèle de test. Une fonction d'oracle générique est telle que :*

$$\begin{aligned}
&\textit{pour une transformation } T \textit{ telle que} \quad T: \begin{array}{l} E \to F \\ u \mapsto T(u) \end{array} \\
&\textit{une fonction d'oracle } O \textit{ telle } O: F \times (G \times E) \to \textit{Boolean} \\
&\qquad\qquad T(x), (z, x) \mapsto \beta \\
&\textit{est générique ssi } \left(\exists z \middle| \forall x, \beta = O(T(x), (z, x)) \textit{ avec } \beta \textit{ le verdict de la vérification de } T(x)\right)
\end{aligned}$$

b- Complétude

La complétude d'un oracle mesure l'étendue des vérifications ou leur nombre. On distingue principalement trois classes de complétude :

- *totale* : lorsque l'oracle vérifie l'ensemble de la spécification quelque soit le cas de test et le modèle de test.
- *semi-totale* : lorsque l'oracle vérifie entièrement l'exactitude d'un modèle résultant de la transformation d'un modèle de test.
- *partielle* : lorsque l'oracle ne vérifie qu'un nombre limité de propriétés. Le verdict d'un tel oracle statut si la transformation d'un modèle de test satisfait une partie de la spécification. Ainsi plusieurs cas de test seront nécessaires avec différents oracles pour vérifier l'exactitude de la transformation d'un même modèle de test.

L'utilisation d'un oracle dont la complétude est totale ou semi-totale produit un *verdict complet* alors qu'un oracle de complétude partielle produit un *verdict partiel*. Pour vérifier l'intégralité de la transformation d'un modèle de test, il faut plusieurs cas de test produisant des verdicts partiels ou un seul cas de test au verdict complet.

Statuer qu'un oracle ou qu'un ensemble d'oracles a une complétude totale et fournit un verdict complet est une tâche complexe. Pour assurer ce niveau de complétude, il faut pouvoir assurer que la spécification dans sa forme originale a été transposée en totalité en données d'oracles. Il faut également assurer qu'elles soient utilisées correctement par les oracles des cas de test. Pour les mêmes raisons qu'il n'est pas possible d'exploiter automatiquement une spécification (surtout textuelle) pour la définition automatique d'oracles, il n'est pas possible de vérifier automatiquement cette complétude.

c- Redondance

Il peut y avoir de la redondance dans les données d'oracle. Selon le type de donnée, il peut s'agir de concepts qu'elles contiennent plusieurs fois, ou d'opérations qu'elles réalisent plusieurs fois. Les redondances peuvent être au sein d'une même donnée d'oracle ou entre plusieurs d'un ensemble. Ainsi, dans un même contrat de nombreuses opérations de navigation sont répétées pour accéder aux éléments du modèle à partir du contexte de la contrainte (généralement la racine du modèle).

d- Taille des données d'oracle

La taille des données d'oracle est difficile à évaluer pour deux raisons :

- Les données d'oracle sont de différents types d'une fonction à une autre.
- Les données d'oracle peuvent varier selon la vérification effectuée et le modèle de test utilisé.

En effet, nous proposons d'utiliser comme donnée d'oracle aussi bien des transformations que des modèles. Pour pouvoir réaliser une comparaison entre ces différents types, nous sérialisons toutes les données d'oracle sous forme de modèles pour comptabiliser leur nombre d'objets. Cette sérialisation est directe pour les modèles attendus, les model snippets. La plupart des outils supportant OCL réalisent cette sérialisation. Pour les transformations, cette sérialisation va dépendre du langage de transformation utilisé. Elle est native en Kermeta par exemple, ou nécessite un outil comme Spoon pour Java.

La taille des données peut varier selon la vérification effectuée et le modèle de test utilisé. Ainsi pour évaluer la taille des données d'oracle permettant de vérifier une transformation complète, nous additionnons le nombre d'objets des données d'oracle utilisées par l'ensemble de cas de test. Par exemple, la comparaison de la taille d'une transformation de référence avec celle d'un seul modèle attendu apporte peu d'information. En effet, la transformation permet de tout vérifier alors que le modèle attendu permet de vérifier la transformation d'un modèle de test. Dans ce cas, nous comparons la taille d'une transformation de référence avec l'ensemble des modèles attendus utilisés.

Ainsi nous réalisons deux mesures :

- la taille propre à une donnée d'oracle :

$$pour\ \Phi\ une\ donnée\ d'oracle : taille\ donnée(\Phi) = nombre\ d'objets(sérialisation(\Phi))$$

- la taille de l'ensemble des données d'oracle des cas de test utilisés pour la vérification d'une version complète de la transformation :

$$pour\ T\ la\ transformation\ sous\ test, \Lambda\ l'ensemble\ des\ cas\ de\ test\ utilisé\ pour\ tester\ T :$$
$$taille\ de\ l'ensemble\ des\ données(\Lambda) = \sum_{\Phi\ chaque\ donnée\ d'oracle} taille\ donnée(\Phi)$$

e- Couplage donnée d'oracle/méta-modèles

Une donnée d'oracle est plus ou moins couplée à la transformation testée si elle contient un nombre plus ou moins grand d'utilisation des méta-classes et relations définies dans les méta-modèles d'entrée et de sortie de la transformation.

Couplage **:** $\begin{array}{ll} Pour\, T : E \rightarrow F & telle\, que\, E\, est\, défini\, par\, MM_{in}\, et\, des\, contrats \\ x \mapsto T(x) & et\, F\, est\, défini\, par\, MM_{out}\, et\, des\, contrats \end{array}$

Si MM_{in} contient m classes comprenant n propriétés (attributs et relations sortantes)
et si MM_{out} contient o classes comprenant p propriétés (attributs et relations sortantes)
et si la donnée d'oracle utilise q classes et r propriétés
alors le couplage entre cette donnée d'oracle et la transformation dont elle sert le test est mesuré par la proportion de classes et de propriétés des méta-modèles utilisées par la donnée d'oracle :

$$C = \frac{q}{m+o} + \frac{r}{n+p}$$

Une donnée d'oracle de C=2 est couplée au maximum avec la transformation alors que le couplage minimal approche de C=0.

Le couplage d'un ensemble de données d'oracle, utilisé pour la vérification complète d'une transformation, ne varie pas selon les différents types de données d'oracle d'une même généricité. Il n'est donc pas nécessaire de considérer à la fois la généricité et le couplage de l'ensemble des oracles.

4.4.3 Influence des propriétés d'un oracle sur sa qualification

Les propriétés de chaque oracle prennent différentes valeurs (tableau 4-2). Ces propriétés représentent des facteurs qui permettent de qualifier chaque fonction d'oracle. Nous qualifions chaque fonction d'oracle selon la combinaison des valeurs prises par les propriétés de chacun de ses oracles.

Propriétés d'un oracle	Généricité	Complétude	Redondance	Taille des données	Couplage données/MM
Valeurs d'une propriété	Générique, Non générique	Totale, Semi-totale, Partielle	$\mathbb{N}$	$\mathbb{N}$	[0 ; 2]

Tableau 4-2- Propriétés d'un oracle et leurs valeurs

Avant de présenter les conséquences des propriétés sur la qualification, nous présentons la manière dont les valeurs de chaque propriété varient, et les corrélations qu'elles ont les unes avec les autres. Ainsi nous considérons l'impact des propriétés sur la qualification en ne les considérant pas individuellement mais en combinant leurs valeurs. Cela permettra par la suite d'obtenir les qualités de chaque fonction d'oracle de manière globale.

a- Variation des valeurs des propriétés

Les cinq propriétés d'oracle que nous avons définies dans la sous-section précédente varient différemment. Tout d'abord leurs plages de valeurs sont différentes : allant de valeurs binaires (la généricité ou la non généricité), ternaire (complétude totale, semi-totale, ou partielle), à des plages infinies (de 0 à 2 pour le couplage, des valeurs entières pour la redondance et la taille). Ensuite, elles ne varient pas autant. Certaines propriétés varient pour chaque nouveau cas de test : c'est le cas de la taille des données d'oracle si elles ne sont pas génériques. D'autres propriétés sont identiques quelque soit le cas de test s'il utilise une même fonction d'oracle (la généricité).

Toutes ces différences dans les plages de valeurs des propriétés et leur variation compliquent la qualification des fonctions d'oracle. Il est ainsi difficile de qualifier chaque fonction d'oracle car elle permet de définir des oracles qui ont des propriétés différentes pour chaque nouvelle donnée d'oracle utilisée.

b- Corrélation des propriétés

Nous relevons que certaines propriétés d'un oracle ne sont pas orthogonales.

Un oracle avec une complétude totale est forcément générique (pas forcément l'inverse) puisqu'un seul suffit pour vérifier toute la transformation et donc tous les modèles qui lui sont passés. Par ailleurs, cette unicité limite le nombre de redondances contrairement à la décomposition des vérifications en plusieurs cas de test: chaque vérification ne peut pas forcément se faire seule dans un cas de test (selon la fonction d'oracle utilisée), donc certaines vont se recouvrir, entrainant des redondances.

Par ailleurs, nous relevons une corrélation entre la complétude et la taille des données. Plus la complétude d'un oracle est totale, et plus la taille de sa donnée d'oracle sera proche de celle de la transformation sous test. Tandis qu'un oracle avec une complétude partielle est moins grand qu'un modèle de sortie puisqu'il n'en vérifie qu'une partie.

Finalement, le couplage d'un oracle générique augmente avec la valeur de la complétude. En effet, la complétude totale implique la considération de tout le méta-modèle cible effectif. Le couplage augmente aussi si l'oracle est générique puisque dans ce cas la couverture concerne les méta-modèles d'entrée et de sortie.

c- Les propriétés d'un oracle comme facteurs de qualité

Le *risque d'erreur* augmente en fonction des valeurs prises par les propriétés d'un oracle:

- L'augmentation de la *taille* des données augmente la complexité et accroît le risque d'erreur. La taille des données n'est pas considérée individuellement mais par ensemble car certaines données d'oracle seules ne permettent pas de vérifier la transformation complète.. Néanmoins, la mesure de la taille individuelle des données reste intéressante. En effet, il est moins complexe et moins risqué d'écrire plusieurs données d'oracle plus petites qu'une seule grande. Nous estimons que la difficulté pour construire l'oracle et le

risque d'erreur augmentent avec l'augmentation de la taille de l'ensemble des données d'oracle et de la taille moyenne de chacune d'elles.

- Les *redondances* sont aussi un point négatif puisqu'elles augmentent le nombre de vérifications et donc le risque de faire une erreur dans l'une d'entres elles. Elles sont considérées au niveau de chaque donnée d'oracle et de l'ensemble des données d'oracles utilisé pour la vérification complète d'une transformation.
- Plus le *couplage* est important dans une donnée d'oracle et plus le nombre de concepts différents (définis dans les méta-modèles source et cible) manipulés simultanément est important. Cela augmente la complexité et donc le risque d'erreur d'un oracle. Pour la même raison, la complexité augmente avec l'importance du couplage de l'ensemble des oracles.
- Comme un oracle *générique* analyse et met en rapport les modèles de test et de sortie correspondant, cela augmente le risque d'erreur par rapport à un oracle considérant uniquement le modèle de sortie. Nous ne tenons pas directement compte de cette propriété comme facteur de risque car elle est déjà considérée dans le couplage.
- Puisque la *complétude totale* implique une grande taille des données et que nous considérons déjà cette propriété, nous ne tenons pas compte de la complétude comme facteur de risque d'erreur.

Ainsi, nous remarquons que plus les valeurs des propriétés d'un oracle sont élevées et plus la complexité et le risque d'erreur sont élevés. La taille des données utilisées par l'oracle est la propriété la plus influente. Nous représentons cela dans le graphique de la figure 4-11. Ce graphique nous permet d'illustrer la qualité d'une fonction d'oracle en remplissant cette zone de référence avec la surface définie par les valeurs des propriétés. Si la surface symbolisant les propriétés d'une fonction d'oracle remplit la zone de référence alors son risque d'erreur est plus important.

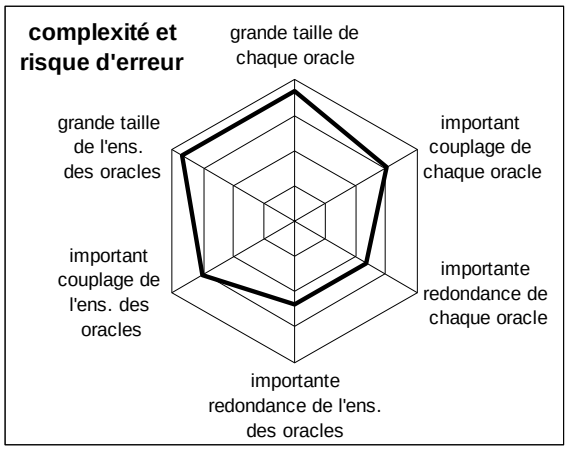


Figure 4-11 - Influence des valeurs des propriétés sur le risque d'erreur dans la définition d'oracles

Plusieurs critères diminuent les possibilités de *réutilisation* d'un oracle dans différentes versions de la transformation de modèles :

- Si un oracle est de grande *taille*, il sera compliqué pour le testeur de le réutiliser. En effet, une taille importante implique un nombre important de propriétés vérifiées dans le modèle de sortie. Plus le nombre de vérifications est important et plus il y a de risques qu'au moins une ou plusieurs soient impactées par une modification de la transformation. Par ailleurs, une donnée est analysée pour décider si elle reste valable après modification de la transformation. Cette tâche est plus complexe avec une grande donnée d'oracle. De plus, elle est plus difficile à adapter si sa taille est grande.
- Plus la *complétude* d'un oracle est totale et plus il vérifie d'exigences de la spécification alors plus il va être affecté par les modifications de la spécification.
- Les *redondances* compliquent la réutilisation de cas de test avec une version modifiée de la transformation de modèles. En effet, une modification d'une exigence de la spécification qui serait considérée plusieurs fois devrait être adaptée autant de fois.
- Plus le *couplage* est important, plus une modification d'un méta-modèle entravera la réutilisation des oracles. Le couplage d'un oracle indique également quelles méta-classes il considère. Si les variations de la spécification concernent ces méta-classes alors l'oracle peut ne pas être réutilisable. C'est également vrai même si son objectif ne portait pas directement sur ces méta-classes.
- La généricité n'entre pas directement en jeu pour la réutilisation.

Ainsi, nous remarquons que plus les valeurs des propriétés d'un oracle sont élevées, alors sa réutilisation sera plus difficile. Comme nous recherchons les oracles les plus réutilisables, nous représentons les valeurs des propriétés bénéfiques pour la réutilisabilité dans le graphique de la figure 4-12.

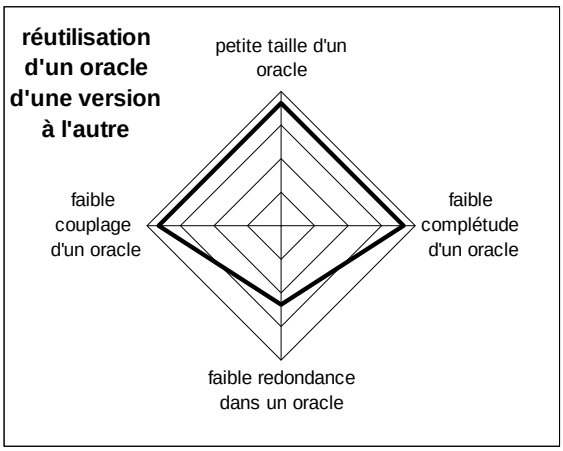


Figure 4-12 - Influence des valeurs des propriétés sur la réutilisabilité d'un oracle

En revanche, la généricité affecte positivement la *réutilisation* d'un oracle dans différents cas de test d'une *même version* de la transformation de modèles sous test.

Dans les sections suivantes, nous illustrons l'emploi des différentes fonctions d'oracle et les valeurs prises par les propriétés des oracles définis. Cela nous permet finalement de qualifier les fonctions d'oracle.

4.5 Qualification de chaque fonction d'oracle

Dans cette section, nous utilisons les expériences menées et nos analyses pour qualifier les différentes fonctions d'oracle. Tout d'abord, nous évaluons les propriétés des différents oracles, puis nous classifions les différentes fonctions selon leurs qualités.

4.5.1 Propriétés d'oracle

a- Généricité

Les *transformations de référence et inverse*, ainsi que les contrats génériques sont des données d'oracles génériques. Elles paramètrent des fonctions d'oracle génériques pour définir des oracles génériques qui vérifient la transformation de n'importe quel modèle de test. En revanche, les *modèles attendus*, les *assertions OCL*, et les *assertions avec des patterns de model snippets* sont des données d'oracle non génériques. Elles sont utilisées pour la transformation d'un modèle de test défini. Elles paramètrent des fonctions d'oracle non génériques pour définir des oracles non génériques utilisables dans un cas de test défini.

Le tableau 4-3 énumère la généricité des différentes fonctions d'oracle.

Fonctions d'oracle génériques	Fonctions d'oracle non génériques
o1 (transformation de référence)	o3 (modèle attendu)
o2 (transformation inverse)	o5 (assertion OCL)
o4 (contrat générique)	o6 (assertion avec patterns de model snippets)

Tableau 4-3 - Généricité des fonctions d'oracle

b- Complétude

Fonctions d'oracle	Complétude
o1 (transformation de référence)	Totale
o2 (transformation inverse)	Totale
o3 (modèle attendu)	Semi-totale
o4 (contrat générique)	Totale
	Partielle
o5 (assertion OCL)	Partielle
o6 (assertion avec patterns de model snippets)	Partielle

Tableau 4-4 - Complétude des fonctions d'oracle

Les trois valeurs différentes de complétude sont représentées dans les fonctions d'oracle. Les oracles utilisant des transformations de référence et inverse ont une complétude totale. Il est possible qu'un seul contrat générique permette de vérifier la totalité de la transformation. Ainsi,

il est utilisé pour définir un oracle de complétude totale. Un tel oracle est réalisable si la transformation accomplit peu de traitements. Dans le cas contraire, cet oracle se compose d'une conjonction de vérifications qu'il est plus simple de réaliser séparément dans différents oracles. Alors chacun a une complétude partielle. Les oracles utilisant des assertions OCL ou des assertions avec patterns de model snippets, ont aussi une complétude partielle.

Finalement, un oracle exploitant un modèle attendu a une complétude semi-totale. La complétude semi-totale peut être considérée comme un cas limite de la complétude partielle. En effet, il est possible d'exprimer la totalité du contenu d'un modèle par une contrainte (contrat générique ou assertion OCL) ou une assertion avec patterns de model snippets. Cependant, nous remarquons qu'un model snippet est un modèle partiel et que pour atteindre la complétude partielle, il devient un modèle attendu. Ainsi, le seul oracle ayant du sens pour la complétude semi-totale est celui utilisant un modèle attendu.

Le tableau 4-4 énumère la complétude des différentes fonctions d'oracle.

c- Redondance

L'utilisation de *transformations de référence* ou *inverse* est peu sujette aux redondances. En effet, elles ne sont écrites qu'une fois pour la vérification de la transformation complète. Des redondances peuvent venir de particularités définies dans la spécification ou du langage/type d'implantation utilisé.

Les *contrats génériques* ont ce dernier défaut car ils effectuent des navigations et sélections qui sont répétées (comme illustré sous-section 4.3.1d-). De nombreuses répétitions sont nécessaires pour naviguer le modèle et sélectionner des éléments utiles à chacun des contrats, cela même avant d'avoir abordé les vérifications propres à un contrat donné. Par exemple, pour vérifier la création des colonnes d'une table il est d'abord nécessaire de mettre en relation au sein du contrat les classes persistantes et leurs tables correspondantes (ce qui est fait avec un contrat proche de celui vérifiant Ru) puis ensuite de considérer la création des colonnes. C'est le cas dans l'exemple de contrat de la figure 4-13 qui vérifie la création des colonnes correspondant aux attributs de type primitif. Dans ce contrat, de la ligne 2 à la ligne 7, il s'agit d'abord de faire le lien entre les tables et les colonnes. Seulement ensuite, la correspondance est vérifiée entre les attributs et les colonnes.

Pour cela, les redondances entre contrats de complétude partielle sont nombreuses. Elles existent directement dans le contrat de complétude totale, et se trouvent entre ses différentes parties. Nous ne faisons pas de distinction entre les redondances d'un ensemble de contrats de complétude partielle et un seul contrat de complétude totale. Ainsi la valeur de leur redondance est la même. Nous leur attribuons donc la même valeur maximale « A » pour la redondance (comme listé dans le tableau 4-5). Dans le cas d'un seul contrat de complétude partielle, les redondances sont moins importantes. Elles existent aussi et sont dues aux opérations de navigation du modèle qui doivent commencer à sa racine. Dans ce cas des répétitions sont nécessaires pour atteindre les éléments qui intéressent précisément le contrat. Nous attribuons la valeur « B » à la redondance d'un contrat de complétude partielle et de la fonction qui l'utilise.

```
1 post attribute_primitive_type
2  inputModel.classifier
3  .select(cr | Class.isInstance(cr))
4  .select(cs | cs.asType(Class).is_persistent)
5  .select(csp | not csp.asType(Class).allParents.exists(p | p.is_persistent))
6    .forAll
7    (csp|result.table.select(t | t.name == csp.name)
8         .exists
9         (tn |csp.asType(Class).attrs
10             .select(at | PrimitiveDataType.isInstance(at.type))
11             // the attributes which type is a primitiveType
12             .forAll
13             (atp|tn.cols
14                  .select
15                  (ctn|ctn.name==atp.name and ctn.type==atp.type.name)
16                  .size==1
17                  //have a single corresponding columns in the table
18             )
19        )
20   )
```

Figure 4-13 - Exemple de contrat pour la transformation class2rdbms

Un *modèle attendu* ne permet pas de vérifier uniquement une partie précise de la transformation d'un modèle de test. Le modèle attendu contient des éléments qui sont inutiles pour la vérification que ce soit d'une seule partie du modèle de sortie ou d'une seule fraction de la spécification. Pourtant ces éléments sont nécessaires pour que la comparaison réussisse. Par exemple, au point 4.3.1c-, nous remarquons qu'il est nécessaire de créer aussi des colonnes et des clés alors que nous ne considérons qu'une règle de la spécification portant sur la création des tables. Ainsi, dans l'ensemble des modèles attendus, des vérifications similaires sont effectuées, entrainant des redondances. Pour cela, nous attribuons la valeur « B » à la redondance de l'ensemble des modèles attendus. En revanche, au sein d'un même modèle attendu, les redondances sont très faibles. Nous leur attribuons la valeur minimale « D ».

L'utilisation des *assertions OCL* se rapproche de celle des contrats de complétude partielle au niveau de la redondance. Avec une assertion OCL, les opérations OCL se font également depuis la racine. Cela entraine des répétitions pour atteindre les éléments à vérifier. En revanche, les assertions OCL n'ont pas besoin de reconsidérer la mise en correspondance d'éléments des modèles d'entrée et de sortie pour aller plus loin dans les vérifications comme c'était le cas des contrats génériques. Cette fois, seule la vérification de l'absence ou de la présence d'éléments dans un modèle de sortie est considérée. Par exemple, il s'agira seulement de vérifier la présence de la colonne nommée « `attA` » dans la table nommée « `B` » dans l'assertion suivante. Ainsi la redondance est plus limitée puisqu'elle ne concerne que la vérification de la présence de la table nommée « `B` ». Nous le constatons dans l'assertion

suivante où seule la première ligne est redondante. Pour ces raisons nous attribuons à la redondance des *assertions OCL* les valeurs minimales juste majorées d'une unité pour une donnée, et également pondérées d'un « + » pour un ensemble de données (tableau 4-5).

```
result.table.select(t | t.name == "B")
                .collect{cols}.exists(c | c.name == "attA")
```

Finalement, l'utilisation d'*assertion avec des patterns de model snippets* est la solution qui présente le moins de redondance. Les model snippets, contrairement aux contrats et assertion OCL, ne nécessitent pas d'opération de navigation superflue du modèle. Comme les assertions OCL, les redondances entre model snippets se limitent à la précision de combinaisons d'éléments. C'est le cas dans l'exemple où pour vérifier la présence de la colonne nommée « `attA` » dans la table nommée « `B` », il est nécessaire de préciser la table. Nous attribuons donc la valeur minimale à la redondance des *assertions avec des patterns de model snippets* juste majorée d'un « + » pour un ensemble (tableau 4-5).

Fonctions d'oracle		Redondance dans une donnée d'oracle	Redondance dans un ensemble de données d'oracle
o1 (transformation de référence)		C	C
o2 (transformation inverse)		C	C
o3 (modèle attendu)		D	B
o4 (contrat générique)	Complétude totale	A	A
	Complétude partielle	B	A
o5 (assertion OCL)		C	C+
o6 (assertion avec patterns de model snippets)		D	D+

Tableau 4-5 - Redondance des fonctions d'oracle

d- Taille

La taille d'une *transformation* utilisée comme donnée d'oracle est similaire à la taille de la transformation sous test. Nous considérons que cela place ce type de donnée d'oracle au niveau supérieur concernant la taille de donnée d'oracle. Nous affectons alors la valeur « A » à sa propriété de taille.

La taille d'un *modèle attendu* est très variable selon le méta-modèle employé et les vérifications nécessaires. Néanmoins, si des modèles de sortie de la transformation peuvent être extrêmement grands, il n'est pas nécessaire que tous les modèles de sortie produits à partir de modèles de test le soient également. Si nous considérons que toutes les méta-classes du méta-modèle de sortie effectif sont considérées par la transformation alors sa taille sera plus grande que la majorité des modèles attendus. Pour cela, nous fixons sa valeur inférieure à celle des transformations (tableau 4-6). En revanche, la taille de l'ensemble des données d'oracle pourra être plus grande, nous lui attribuons la note maximale pondérée « A+ ».

Les *contrats génériques* analysent à la fois les modèles de test et de sortie, ainsi leur *taille* est supérieure à celle d'un oracle qui serait dédié à la vérification de la transformation d'un modèle

de test déterminé. Quand la complétude du contrat est totale, sa taille est similaire à celle des transformations. C'est la même chose pour la taille des ensembles de contrats de complétude partielle, et totale. Nous leur attribuons donc la valeur « A » pour leur taille. En revanche, la taille des contrats est difficile à évaluer quand sa complétude est partielle. Dans ce cas, la taille de chaque contrat est forcément inférieure puisqu'il est utilisé pour vérifier moins d'éléments. Nous considérons cette taille similaire à celle des modèles attendus. D'un côté, ces contrats sont définis à un niveau méta. Sans considération des modèles existants, ils emploient les concepts définis par ces méta-classes et leur appliquent des opérations (consultation, navigation, conditionnelle, etc.). D'un autre côté, les modèles attendus peuvent être plus grands si les modèles de test contenaient beaucoup d'éléments qui seraient considérés par les mêmes parties d'un contrat. Ainsi la taille des contrats génériques de complétude partielle est globalement équivalente à celle des modèles attendus, nous lui attribuons la valeur « B ».

Comme pour la redondance, la taille des assertions OCL et des assertions avec patterns de model snippets est nettement plus faible que celle des autres types de données. Elles n'imposent pas la vérification de tous les éléments d'un modèle de sortie, ou la mise en correspondance de n'importe quelle méta-classe des méta-modèles. La taille de ces données individuellement ou en ensemble est donc toujours minimale. Cependant, l'utilisation d'assertion OCL (plutôt que des model snippets) augmente cette taille car elle est pénalisée par les opérations de navigation depuis la racine qu'elle contient. De plus, nous avons montré que le langage défini pour la consultation des résultats du pattern matching effectué sur la base de model snippet contient moins de concepts qu'OCL. Cela montre que l'utilisation des model snippets conduit à la création d'oracles et d'ensemble d'oracles de taille minimale.

Fonctions d'oracle		**Taille d'une donnée d'oracle**	**Taille d'un ensemble de données d'oracle**
o1 (transformation de référence)		A	A
o2 (transformation inverse)		A	A
o3 (modèle attendu)		B	A+
o4 (contrat générique)	Complétude totale	A	A
	Complétude partielle	B	A
o5 (assertion OCL)		D+	D+
o6 (assertion avec patterns de model snippets)		D	D

Tableau 4-6 - Taille des données d'oracle des fonctions d'oracle

e- Couplage

Par construction, les transformations couvrent exactement autant les méta-modèles que la transformation sous test. Ainsi en considérant les méta-modèles effectifs (introduit sous-section 2.5.1) de la transformation sous test, nous pouvons dire qu'un oracle exploitant une transformation comme donnée d'oracle couvre à 100% les méta-modèles effectifs de la transformation. Nous attribuons à cet oracle la valeur « A » pour indiquer la *couverture* maximale des méta-modèles effectifs (tableau 4-7).

La couverture du méta-modèle par un *contrat générique* est similaire à la propriété de taille. Quand le contrat utilisé a une complétude totale ou pour un ensemble de contrats de complétude partielle (et totale) alors la totalité des méta-modèles effectifs d'entrée et de sortie est couvert. La valeur de leur couverture est donc « A ». Par contre avec une complétude partielle, les contrats vérifient moins de propriétés et manipulent donc moins de concepts définis dans les méta-modèles. Alors la valeur de son couplage est « B ».

La définition d'un *modèle attendu* ne concerne que le méta-modèle cible. Chaque modèle attendu n'ayant qu'une complétude semi-totale, il ne couvre qu'une partie du méta-modèle cible effectif. Ainsi en moyenne, le couplage d'un oracle utilisant un modèle de sortie attendu est moins important que celle d'un oracle générique. En revanche, l'ensemble des modèles attendus permet de vérifier la transformation complète. Ainsi cet ensemble couvre la totalité du méta-modèle cible effectif. Nous attribuons donc la valeur « C » au couplage de modèles attendus.

Le couplage des *assertions* ne concerne également que le méta-modèle cible. Sa valeur est plus faible que celle des modèles attendus. En effet, le couplage des modèles attendus n'est pas maîtrisé pendant leur définition car il est entièrement dépendant du modèle de test utilisé. En revanche le couplage d'une assertion dépend avant tout de la vérification effectuée. Celle ci peut être modularisée davantage. C'était le cas dans l'exemple utilisé aux points 4.3.1e- et 4.3.1f- où un découpage supplémentaire était fait en quatre assertions. Le couplage des assertions OCL est plus important que celui des model snippets. En effet, la navigation depuis la racine du modèle, impose aux assertions OCL de considérer des éléments inutiles à la vérification mais qui augmentent le couplage. Ainsi le couplage d'un model snippet est inférieur à un modèle attendu (nous lui attribuons la valeur « D »). Alors que le couplage des assertions OCL est supérieur unitairement au modèle snippet mais inférieur à celui d'un modèle attendu (valeur « C- »). Cependant les valeurs du couplage de leurs ensembles sont équivalents (valeur « C »).

Fonctions d'oracle		**couplage d'une donnée d'oracle**	**couplage d'un ensemble de données d'oracle**
o1 (transformation de référence)		A	A
o2 (transformation inverse)		A	A
o3 (modèle attendu)		C	C
o4 (contrat générique)	Complétude totale	A	A
	Complétude partielle	B	A
o5 (assertion OCL)		C-	C
o6 (assertion avec patterns de model snippets)		D	C

Tableau 4-7 - Couplage des données d'oracle des fonctions d'oracle

f- Conclusion

Ainsi nous obtenons le tableau 4-8, dans lequel sont présentées les différentes valeurs des propriétés de chaque fonction d'oracle. Nous les utilisons dans la section suivante pour qualifier chacune des fonctions.

Fonctions d'oracle		Géné-ricité	Complé-tude	Redondance dans		Taille d'		Couplage d'	
				une donnée	un ens.	une donnée	un ens.	une donnée	un ens.
o1 (transformation de référence)		Oui	Totale	C	C	A	A	A	A
o2 (transformation inverse)		Oui	Totale	C	C	A	A	A	A
o3 (modèle attendu)		Non	Semi-totale	D	B	B	A+	C	C
o4 (contrat générique)	Complétude totale	Oui	Totale	A	A	A	A	A	A
	Complétude partielle	Oui	Partielle	A	B	B	A	B	A
o5 (assertion OCL)		Non	Partielle	C	C+	D+	D+	C-	C
o6 (assertion avec patterns utilisant des model snippets)		Non	Partielle	D	D+	D	D	D	C

Tableau 4-8 - Fonctions d'oracle avec les valeurs de leurs propriétés

4.5.2 Qualités des fonctions d'oracle

En fonction des propriétés de chaque fonction d'oracle, leurs qualités varient, comme illustrée dans le tableau 4-9. Nous allons discuter les valeurs de ce tableau dans cette sous-section en considérant successivement chaque fonction d'oracle.

Fonction d'oracle	Complexité et risque d'erreur	Réutilisation avec différentes versions
o1 (transformation de référence)	C	C
o2 (transformation inverse)	C	C
o3 (modèle attendu)	B	C
o4 (contrat générique)	B	B-
o5 (assertion OCL)	B+	A-
o6 (assertion avec patterns utilisant des model snippets)	A-	A

Tableau 4-9 - Fonctions d'oracle avec leurs qualités

a- Qualités de la fonction d'oracle utilisant des transformations :

En employant les fonctions d'oracle o1 et o2 qui utilisent des transformations comme données d'oracle, la complexité de ce type d'oracle est importante et entraîne un risque élevé d'erreur. En analysant les différentes propriétés des oracles qu'elles permettent de définir (tableau 4-8), nous les mettons en rapport avec les qualités souhaitées dans les graphiques de la figure 4-14. Nous y voyons que l'aire représentant l'influence des propriétés sur la complexité et le risque d'erreur (à gauche) est très largement recouverte par l'aire des propriétés de ce type

d'oracle. En revanche l'aire représentant l'influence des propriétés sur la réutilisation (à droite) l'est très peu.

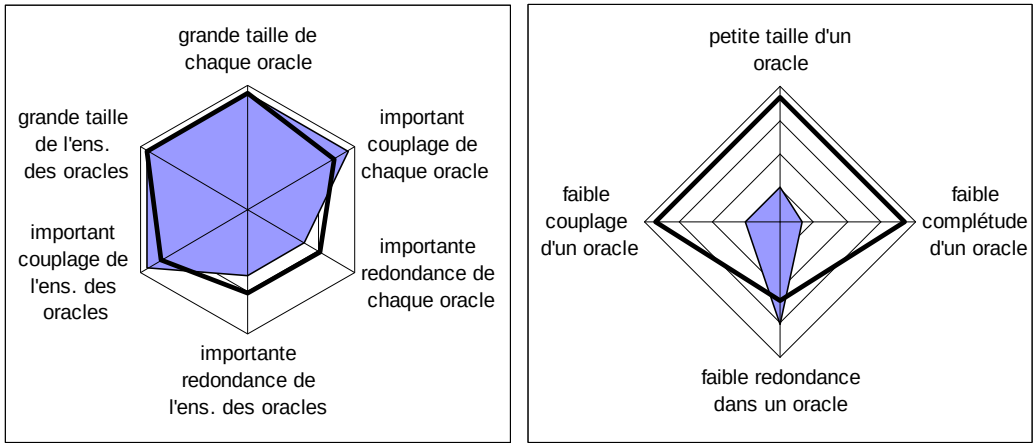


Figure 4-14 - Influence des valeurs des propriétés d'oracles utilisant des *transformations* sur la complexité et le risque d'erreur à gauche et sur la réutilisation à droite

Ainsi nous attribuons la qualité « C » concernant le risque d'erreur et également « C » concernant la réutilisation avec une nouvelle version d'un oracle défini avec les fonctions d'oracle o1 et o2 (tableau 4-9).

Pour cette raison, nous ne recommandons pas d'utiliser ces fonctions d'oracles. Cela n'est envisageable que dans les cas où la donnée d'oracle (la transformation de référence ou inverse) est disponible. Ainsi, l'oracle défini peut être utilisable dans n'importe quel cas de test parce qu'il est générique. Néanmoins, la transformation disponible doit elle-même avoir été vérifiée pour servir de donnée d'oracle.

b- Qualités de la fonction d'oracle utilisant des contrats génériques :

Nous distinguons un contrat de complétude totale et des contrats de complétudes partielles. Le premier cas est entièrement défavorable. Nous constatons dans les graphiques de la figure 4-15 que définir un contrat de complétude totale est particulièrement complexe et donc risqué. De plus, ce contrat est difficilement réutilisable.

En revanche, si l'utilisation de plusieurs contrats de complétude partielle implique aussi une forte complexité (figure 4-16), chacun est plus facilement réutilisable.

Ainsi, nous ne conseillons pas d'utiliser des contrats dans le but de vérifier complètement la transformation, en particulier avec un seul contrat complet. Néanmoins, il est possible de les utiliser en décomposant les vérifications, ce qui permettra de les créer plus et de les réutiliser plus facilement.

Nous ne distinguons pas les qualités de la fonction d'oracle selon la complétude des oracles qu'elle permet de définir. En effet, il n'est pas imaginable vu leurs qualités de définir des

oracles utilisant un contrat de complétude totale. Ainsi nous attribuons les qualités B pour la complexité et B- pour la réutilisation des oracles définis à partir de la fonction d'oracle o4.

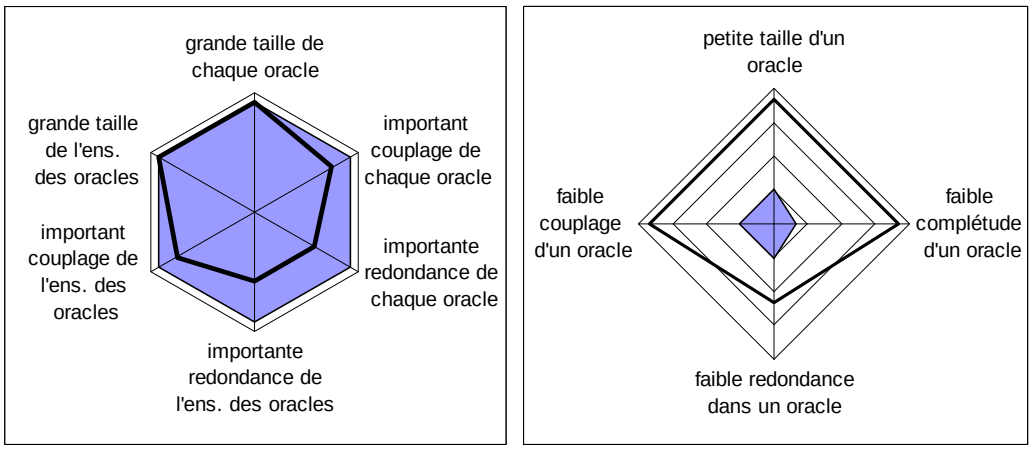


Figure 4-15 - Influence des valeurs des propriétés d'oracles utilisant des *contrats génériques de complétude totale* sur la complexité et le risque d'erreur à gauche et sur la réutilisation à droite

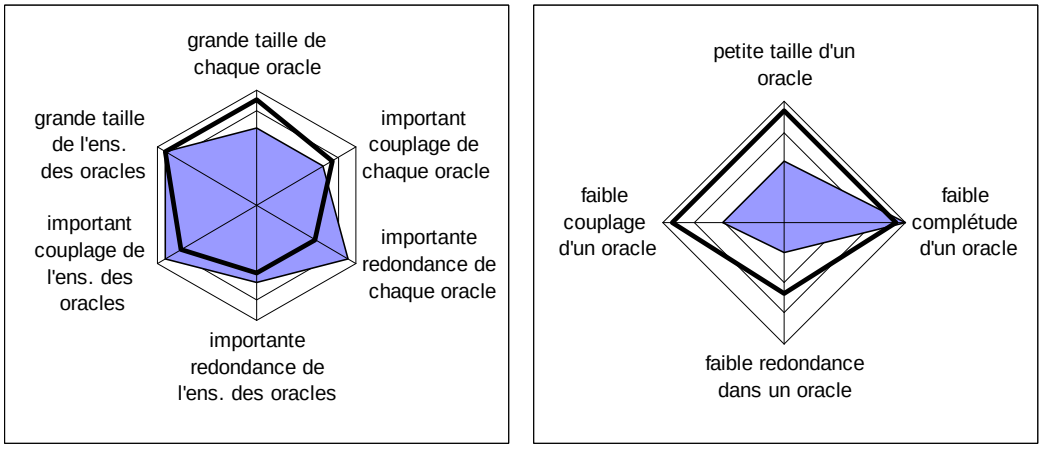


Figure 4-16 - Influence des valeurs des propriétés d'oracles utilisant des *contrats génériques de complétude partielle* sur la complexité et le risque d'erreur à gauche et sur la réutilisation à droite

c- Qualités de la fonction d'oracle utilisant des modèles attendus :

Définir des oracles avec des modèles attendus est une solution classique. Elle permet de modulariser les vérifications en prenant en compte les modèles de test (grâce à la non généricité et la complétude semi-totale). La complexité pour définir ces oracles est moins importante même si le dépassement que nous constatons dans le graphique de la figure 4-17 concernant la taille de l'ensemble des données d'oracle peut être un problème si les modèles de test sont nombreux et grands. En revanche la réutilisation de ce type d'oracle est limitée, comme nous le voyons sur le graphique de droite de la figure 4-17.

Ainsi, nous attribuons la qualité B pour le risque d'erreur et C pour la réutilisation d'oracles définis à partir de la fonction d'oracle o3.

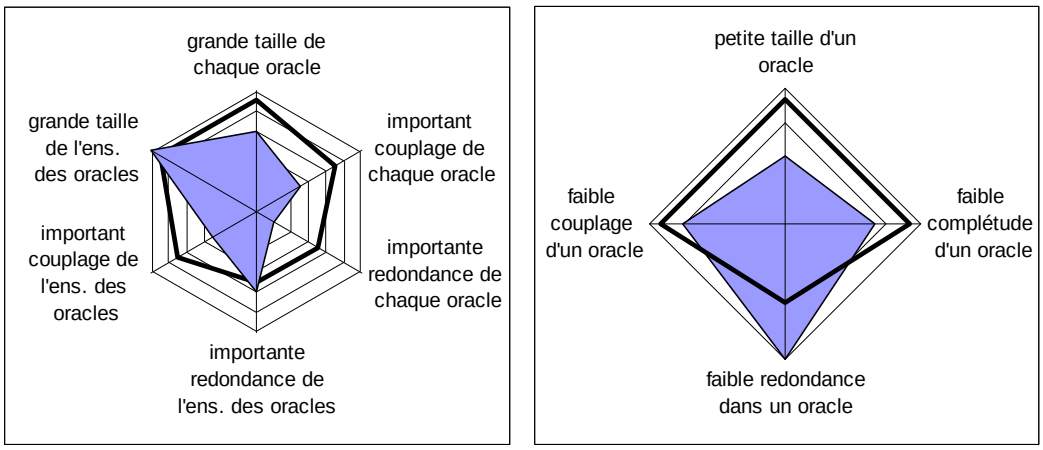


Figure 4-17 - Influence des valeurs des propriétés d'oracles utilisant des *modèles attendus* sur la complexité et le risque d'erreur à gauche et sur la réutilisation à droite

L'utilisation d'un même oracle pour la vérification d'une même version de la transformation peut rarement se produire si la transformation n'est pas injective.

d- Qualités de la fonction d'oracle utilisant des assertions OCL :

L'utilisation de la fonction d'oracle o5 utilisant des assertions OCL a de bonnes qualités. Nous constatons dans les graphiques que l'aire de la complexité est peu couverte alors que l'aire de la réutilisabilité l'est presque entièrement. Les oracles définis avec cette fonction permettent de cibler précisément des vérifications à réaliser. Cela les rend modulaires et leur donne de bonnes qualités. Nous attribuons la qualité B+ pour risque d'erreur et la qualité A- pour la réutilisabilité d'oracles formés à partir de la fonction d'oracle o5.

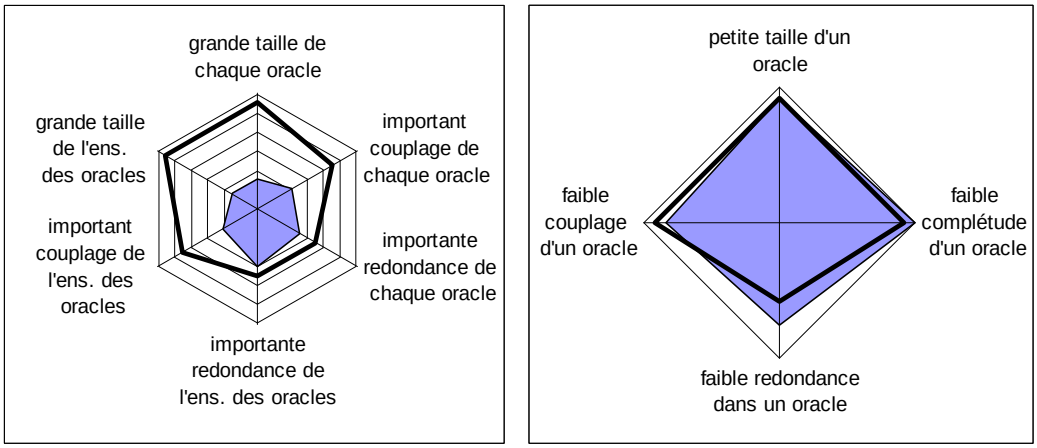


Figure 4-18 - Influence des valeurs des propriétés d'oracles utilisant des *assertions OCL* sur la complexité et le risque d'erreur à gauche et sur la réutilisation à droite

e- Qualités de la fonction d'oracle utilisant des assertions avec patterns de model snippets:

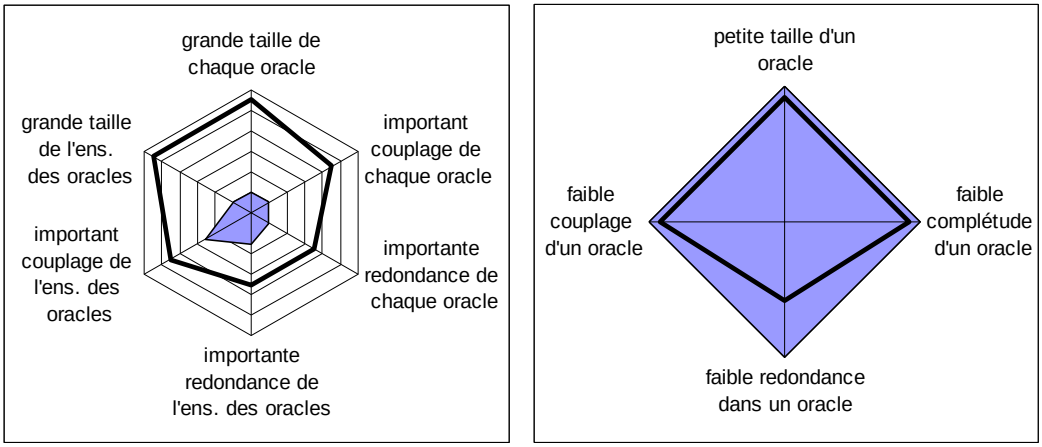


Figure 4-19 - Influence des valeurs des propriétés d'oracles utilisant des *assertions avec patterns de model snippets* sur la complexité et le risque d'erreur à gauche et sur la réutilisation à droite

Enfin, la fonction d'oracle o6 a les meilleures qualités. Nous constatons dans les graphiques que l'aire de la complexité est la plus petite alors que l'aire de la réutilisabilité est entièrement recouverte. Nous attribuons donc la qualité A- au risque d'erreur et A à la réutilisation d'oracles définis à partir de la fonction d'oracle o6.

4.6 Conclusion

Dans ce chapitre, nous avons tout d'abord présenté la problématique de l'oracle pour le test de transformations de modèles. La complexité des transformations de modèles et des modèles qu'elles manipulent rend difficile l'écriture d'oracle. Nous introduisons six fonctions d'oracle qui exploitent les techniques de comparaison de modèles, de contrats et de pattern matching. Ces fonctions d'oracle exploitent différentes données d'oracle et se caractérisent par différentes propriétés qui nous permettent de les qualifier selon leur risque d'erreur et leur réutilisabilité. Toutes les valeurs que nous attribuons aux propriétés des oracles et aux qualités des fonctions ne sont pas des valeurs absolues, nous les avons évaluées par analyse et dans nos expériences. Nous les utilisons pour comparer les fonctions entre elles. Il en ressort que les fonctions qui introduisent de la modularité et permettent de décomposer les vérifications ont de meilleurs qualités. Les fonctions exploitant des données d'oracle de complétude totale sont inappropriées (les fonctions utilisant des transformations de modèles ou des contrats complets). En revanche, les fonctions les plus modulaires qui permettent de considérer des vérifications très précises sur des modèles choisis (pas de généricité) sont les plus intéressantes (les fonctions utilisant des patterns). Finalement les qualités techniques introduites par les model snippets que nous exploitons avantageusement pour la définition de données d'oracle donnent l'avantage à cette fonction d'oracle sur celle exploitant des assertions OCL.

5
Mise en œuvre et qualification de composants de transformation de modèles

Dans les deux précédents chapitres, nous avons étudié et présenté nos deux principales contributions théoriques. Elles concernent l'adaptation de l'analyse de mutation et l'élaboration d'oracles pour le test de transformations de modèles. Dans ces deux contributions, nous avons étudié deux des principales phases du test : une technique de qualification des modèles de test et la vérification des modèles produits. Nous avons considéré des particularités d'une transformation qui rendent son test original : la réutilisation des transformations, la complexité des modèles manipulés, ou encore les opérations qu'elle réalise. Pour valider ces deux contributions qui participent pleinement à la fiabilisation de transformations de modèles, nous présentons dans ce chapitre une méthodologie globale pour le test de transformations de modèles.

Avec cette méthodologie globale, nous étudions la construction de composants de transformations de modèles. Ces composants intègrent à la fois les cas de test de la transformation et une spécification exécutable sous forme de contrats. Cette approche exploite directement l'analyse de mutation pour le développement de composants en mesurant leur niveau de confiance.

Nous utilisons également l'analyse de mutation dans d'autres travaux qui portent sur la définition de critères de test fonctionnels. Nous traitons ce point dans la dernière section de ce chapitre.

Pour la mise au point, l'application, et la validation des deux contributions théoriques, ainsi que pour l'application de la méthodologie étudiée dans ce chapitre, nous avons réalisé plusieurs développements logiciels. Ils se répartissent en trois contributions techniques. Tout d'abord, nous avons développé un système de contraintes dans le langage Kermeta permettant d'appliquer le design-by-contract et d'intégrer le support d'OCL. En plus du bénéfice apporté à tout utilisateur de Kermeta, ce travail nous permet de créer des oracles. Ensuite, nous avons développé un harnais de test qui permet d'exploiter les fonctions d'oracle proposées. Finalement, nous avons réalisé une plateforme expérimentale qui permet de mettre en

application l'analyse de mutation. Ces développements sont un travail important de cette thèse pour plusieurs raisons :

- ils permettent d'illustrer l'application des contributions des précédents chapitres,
- ils facilitent la distribution et le transfert des techniques proposées,
- ils contribuent à des travaux hors du contexte de cette thèse,
- ils permettent de démontrer que nos contributions ont un champ d'application plus étendu que cette thèse et participent directement ou indirectement au test de transformations de modèles.

Dans ce chapitre, nous présentons dans la section 5.1 la méthodologie de construction de composants de transformations de modèles de confiance (ne considérant que ce genre de composant, nous abrègerons souvent cette expression). Dans la section 5.2, nous expliquons les contributions techniques de cette thèse. Dans la section 5.3, la méthodologie est expérimentée dans sa globalité avec la transformation class2rdbms. Finalement, dans la section 5.4, nous montrons comment nos travaux contribuent dans l'étude d'autres problématiques.

5.1 Construction de composants de transformation de modèles de confiance

Cette section définit un modèle et une méthodologie pour la construction de composants de transformations de modèles. L'objectif est d'assurer la confiance d'un tel composant pour permettre son utilisation dans des développements de logiciels fiables. Il s'agit de proposer des unités pour lesquelles sont clairement spécifiées leurs possibilités d'utilisation et les garanties qu'elles présentent.

5.1.1 Formation de composants de transformation de modèles

Le modèle proposé pour former des composants de transformations de modèles de confiance est basé sur l'intégration de la spécification et du test des composants logiciels. Ce modèle est particulièrement adapté à l'approche de design-by-contract [Meyer]. Dans ce cas, la spécification est systématiquement traduite en contrats exécutables (invariant, pré/post conditions). La figure 5-1 représente ce modèle de composant avec un diagramme en triangle. Les trois sommets de ce triangle symbolisent les trois éléments constitutifs d'un composant de confiance :

- L'implémentation de la transformation de modèles.
- Les cas de test.
- La spécification sous forme exécutable.

Ce modèle a été proposé dans des travaux de Le Traon et al. [Le Traon'99, Baudry'00b] qui se concentraient sur les programmes orientés objets. Dans nos travaux de thèse, nous considérons son application pour les transformations de modèles. Dans ce contexte, l'implémentation peut être réalisée avec des langages orientés objet ou dans d'autres langages

dédiés à la transformation de modèles (section 2.2.3). Nous avons étudié dans les deux précédents chapitres les modèles de test et les différentes fonctions d'oracles qui sont employées pour l'élaboration de cas de test. La spécification est transcrite en une version exécutable avec des contrats. Ils peuvent être écrits en OCL ou dans un autre langage de contraintes. Le contexte des transformations de modèles implique que les contrats soient complexes, car ils sont appliqués à des modèles. Les contrats sont exprimés à un niveau sémantique et traduits en propriétés exécutables qui sont vérifiées pendant l'exécution de la transformation. Aujourd'hui il n'y a pas de standard pour exprimer les contrats des transformations de modèles ou pour définir la spécification d'une transformation.

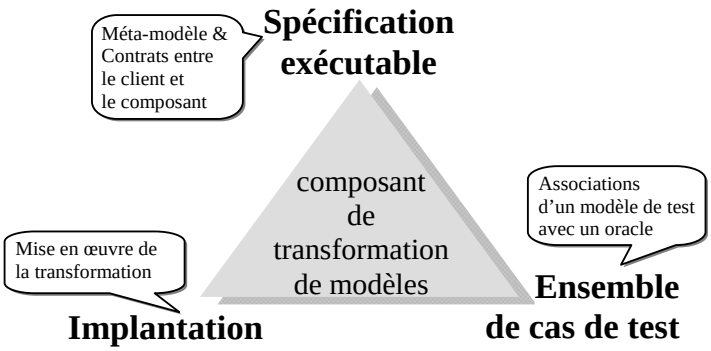


Figure 5-1. Modèle de composant de transformation de modèles

Dans la suite nous distinguons deux niveaux de contrats respectant la classification de Beugnard et al. [Beugnard'99]. Dans la figure 5-2, nous illustrons le positionnement de ces différents contrats dans le processus de la transformation :

- **Contrats basiques :** le premier niveau, basique, ou syntaxique, est requis simplement pour faire fonctionner le système. Des langages de définition d'interface, ainsi que des langages typés ou orientés objet, laissent le concepteur du composant spécifier : les opérations que le composant peut réaliser, les paramètres d'entrée et de sortie que chaque composant requiert, et les exceptions possibles qui peuvent être levées pendant l'opération. Appliqués aux composants de transformations de modèles, les méta-modèles source et cible décrivent la structure des données manipulées. Un méta-modèle est formé d'un modèle, écrit en Ecore par exemple, ainsi que d'invariants. Les méta-modèles font partie de la spécification, les modèles d'entrée/sortie doivent s'y conformer. L'ensemble des méta-modèles source et cible (dont leurs invariants) fait également partie de la spécification exécutable et est donc intégré aux composants. Néanmoins, les domaines d'entrée et de sortie définis par les méta-modèles source et cible peuvent être trop étendus. En particulier, le domaine d'entrée de la transformation a besoin d'être contraint davantage pour que les modèles puissent être transformés sans mettre la transformation en échec. Ces contraintes sont exprimées avec des pré-conditions basiques. Nous distinguons trois catégories de contrats basiques :
    - **contrats du méta-modèle source CMS** (dans des invariants du méta-modèle)
    - **contrats basiques du domaine d'entrée CBDE** (dans des pré-conditions)
    - **contrats du méta-modèle cible CMC** (dans des invariants du méta-modèle)

- **Contrats de sémantique comportementale :** au second niveau, les contrats de sémantique comportementale augmentent le niveau de confiance dans le contexte d'exécution pour une transformation de modèles spécifique. Par exemple, le méta-modèle source de class2rdbms ne définit pas qu'une classe du modèle d'entrée doit inclure au moins un attribut. Ainsi, des contraintes spécifiques peuvent être attachées au domaine d'entrée de la transformation, qui joue le rôle de pré-condition spécifique pour la transformation. Nous distinguons trois catégories de contrats de sémantique comportementale :
  - **contrats sémantiques du domaine d'entrée CSDE** (dans des pré-conditions) expriment des propriétés qui spécifient le domaine source plus précisément que le méta-modèle et les pré-conditions basiques, d'un point de vue sémantique
  - **contrats sémantiques du domaine de sortie CSDS** (dans des post-conditions) expriment des propriétés spécifiques dans les modèles de sortie propres au comportement attendu de la transformation.
  - **contrats sémantiques reliant entrée et sortie CSR** (dans des post-conditions) expriment des propriétés reliant les modèles d'entrée et de sortie. Ces contrats mettent directement en relation le modèle d'entrée avec le résultat de sa transformation pour vérifier qu'elle soit réalisée conformément à la spécification.

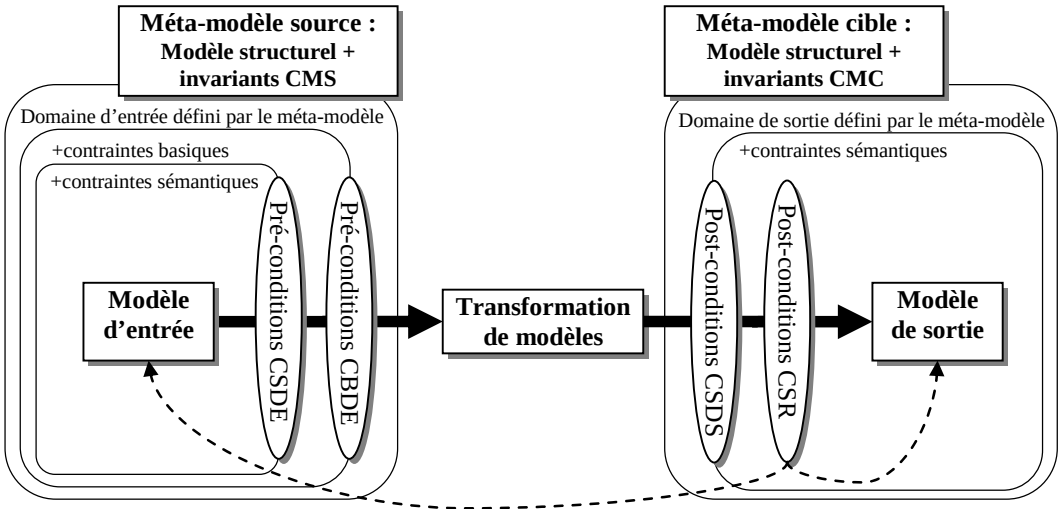


Figure 5-2 - Contrats basiques et sémantiques dans le processus d'une transformation

Formés de cette manière, les trois sommets du composant sont exécutables. Ils permettent de constituer un composant autotestable.

5.1.2 Qualification des composants de transformations de modèles

La qualification d'un composant doit garantir que son niveau de confiance permet son intégration au développement. Le niveau de confiance dans un composant est considéré en deux temps :

1. La qualité de chaque sommet du triangle est évaluée isolement. La transformation embarquée doit être correcte. Elle doit être vérifiée avec un ensemble de cas de test

performant. Et les contrats doivent traduire au plus près la spécification. Ainsi la confiance portée dans un composant est d'abord reliée à l'efficacité des cas de test et à la complétude des contrats.

2. La *cohérence* entre les trois sommets est évaluée. D'abord un ensemble de cas de test est qualifié par analyse de mutation en confrontant les mutants de l'implantation avec les données de test. Puis il est possible d'améliorer l'implantation : l'exécution d'un bon ensemble de cas de test permet de détecter les erreurs et de les corriger. La confiance dans l'implantation est d'autant plus grande que les cas de test sont efficaces. Enfin, l'exactitude des contrats doit être considérée pour les rendre efficaces dans des oracles pour les cas de test. On aura d'autant plus confiance dans les contrats qu'ils sont capables de détecter les erreurs des mutants.

Pour évaluer la cohérence, nous employons l'analyse de mutation. Cette technique que nous avons adaptée au chapitre 3 permet de confronter les sommets et de fournir une mesure de la cohérence. Nous varions l'emploi de l'analyse de mutation pour qualifier l'ensemble de modèles de test et les contrats. Pour qualifier l'ensemble de modèles de test, l'oracle utilisé pour l'analyse est la comparaison avec le résultat produit par l'implantation (c'est l'utilisation nominale de l'analyse de mutation). Pour qualifier les contrats, ils sont utilisés comme oracle pour l'analyse de mutation. En effet, les modèles de test sont efficaces pour tuer les mutants et ils permettent d'assurer que l'implantation est correcte, donc nous évaluons les contrats en les confrontant aux mutants de l'implantation correcte avec ces modèles de test qualifiés. La figure 5-3 représente ces confrontations et la mesure de cohérence qui en résulte.

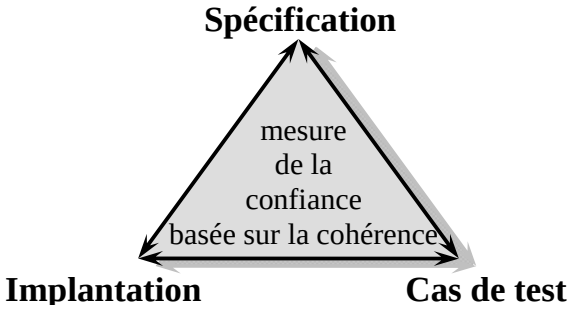


Figure 5-3. Confiance basée sur la cohérence du triangle

5.1.3 Augmentation du niveau de confiance

La confrontation entre ces trois sommets conduit à l'amélioration de chacune d'elle. L'amélioration de chaque sommet est effectuée dans un processus itératif qui fournit comme résultat un composant de confiance ainsi qu'une mesure de la cohérence globale.

Le processus pour construire la confiance dans un composant de transformation de modèles se compose de trois étapes, dont l'illustration de l'application nominale est présentée figure 5-4. Le niveau de confiance dans un sommet est représenté par un cercle de taille d'autant plus grande que le niveau de confiance est élevé. Le processus commence avec un composant initial qui comporte déjà ces trois sommets. Le but est alors d'améliorer chaque sommet et de vérifier

la cohérence globale entre chaque sommet. Sur le côté droit de la figure, l'évolution de la confiance que nous avons dans chaque sommet de la vue en triangle est mise en évidence.

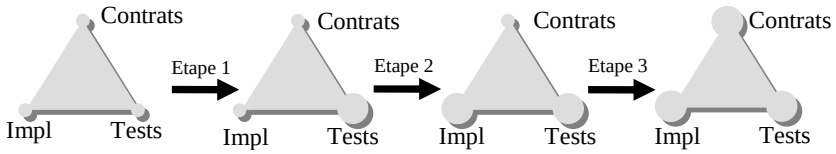


Figure 5-4 – Processus nominal de construction d'un composant de transformation de modèles de confiance

Étape 1- amélioration des cas de test. Le but est d'améliorer l'efficacité de l'ensemble de cas de test en utilisant un critère de test approprié. Dans cette section, nous suggérons d'évaluer cette qualité en utilisant l'analyse de mutation. Nous exploitons l'adaptation que nous avons réalisée au chapitre 3. Elle va produire un ratio qui permet d'estimer la force de détection des erreurs de l'ensemble de modèles de test. Il s'agit du score de mutation que nous notons Q_{mt}.

$$Q_{mt} = Score\ de\ mutation = \frac{nombre\ de\ mutants\ tués}{nombre\ de\ mutants\ non\ équivalents}$$

Si ce ratio est insuffisant, cela signifie que l'ensemble de modèles de test doit être amélioré par la création de nouveaux modèles. Il s'agit d'appliquer le processus de l'analyse de mutation tel que nous l'avons proposé dans la section 3.3. L'étape 1 se termine quand la qualité des modèles de test atteint un niveau satisfaisant qui est fixé par un score de mutation à atteindre. Le niveau de satisfaction est fixé par le testeur en fonction du niveau de criticité de l'utilisation du composant.

Étape 2- Test et amélioration de l'implantation. Dans cette étape, l'implantation de la transformation de modèles est testée grâce à l'ensemble optimisé de cas de test puis corrigée. Les cas de test mis au point à l'étape précédente ont un score de mutation élevé et sont donc capable de détecter des erreurs dans les programmes de transformations de modèles. Ces cas de test sont exécutés et si des erreurs sont détectées, elles doivent être diagnostiquées et corrigées. A la fin de cette seconde étape, la confiance du sommet implantation est satisfaisante puisqu'elle a été validée par des cas de test efficaces.

Étape 3- Amélioration des contrats. Le but est d'embarquer des contrats efficaces [Le Traon'06] et complets, en considérant chacun comme une partie exécutable de la spécification. A cette étape, le composant possède une implantation sûre et des cas de test efficaces. Nous utilisons à nouveau l'analyse de mutation pour estimer la complétude des contrats en termes de taux de détection d'erreur (c'est-à-dire la proportion de fautes actuellement détectée par les contrats connaissant les mutants pour lesquels les cas de test produisaient des modèles de sortie incorrectes). Cela permet d'estimer la complétude des contrats comme oracles embarqués.

Par rapport à la première étape, ici la force de détection des erreurs par l'ensemble de modèles de test est connue avec le score de mutation Q_{mt}. En utilisant les contrats comme oracles embarqués, la nouvelle proportion d'erreurs détectées (%erreursDétectées) est calculée.

La qualité des contrats est contrôlée en fonction de Q_{mt}. La qualité estimée Q_{cont} des contrats est définie par :

$$Q_{cont} = \frac{\%erreursDétectées}{Q_{mt}}$$

Avec $Q_{mt} > 0$ car cette étape 2 suit l'étape 1 qui a forcément permis d'augmenter Q_{mt}.

Cette mesure peut-être vue comme une estimation de la complétude des contrats pour servir à la définition d'oracle pour les cas de test. Si Q_{cont} est égale à 0, aucune erreur n'a été détectée par les contrats, et si Q_{cont} est égale à 1, les contrats sont capables de détecter toutes les erreurs injectées. Après l'étape 3, les contrats sont améliorés et atteignent un niveau de qualité satisfaisant. A la fin du processus, les contrats sont embarqués dans le composant de transformation de modèles qui devient « vigilant ». Ainsi l'emploi du composant permet d'en détecter les états erronés dynamiquement.

Itération du processus. Ces trois étapes se succèdent dans un processus itératif. Le déroulement illustré dans la figure 5-4 illustre une séquence nominale d'une seule itération. En pratique, les étapes peuvent être répétées les unes après les autres. La construction du composant se fait en confrontant les sommets du triangle, donc l'amélioration d'un sommet influence la cohérence globale du composant. Ainsi, la correction de l'implantation conduit à la création d'un nouvel ensemble de mutants. Les modèles de test puis les contrats doivent donc être qualifiés de nouveau. De même, l'amélioration des contrats conduit à tester à nouveau l'implémentation puisque ces contrats peuvent servir à former l'oracle de n'importe quel cas de test. Finalement, le processus suit le déroulement décrit dans le diagramme d'état de la figure 5-5. Ce processus converge quand les cas de test ne détectent plus d'erreur imposant de corriger l'implantation.

Quand le processus est terminé, nous obtenons un composant de transformation de modèles amélioré avec une estimation de son niveau de confiance.

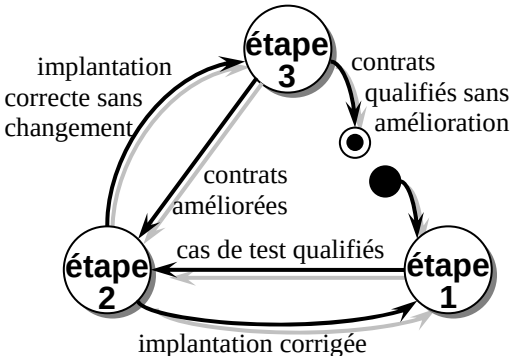


Figure 5-5 - Diagramme d'état lors du processus de construction du composant

5.2 Développements pour la définition, la qualification et l'amélioration de composants de confiance

Pour rendre ce processus exploitable pour les transformations de modèles, il est nécessaire d'une part de doter les transformations d'un langage de contraintes et de développer des outils supportant nos techniques (l'analyse de mutation, les différentes fonctions d'oracle), et d'autre part d'exploiter ces outils pour évaluer de manière empirique le processus. Dans cette section, nous présentons les trois principales contributions techniques de cette thèse. Elles sont directement exploitées dans nos travaux. En particulier, elles permettent de réaliser les expériences présentées dans les deux sections suivantes. Nous présentons d'abord l'intégration d'un système de contraintes dans Kermeta. L'importance de cette contribution dépasse nos travaux. Elle permet l'application du design-by-contract dans Kermeta et a servi de base pour mettre en œuvre un support d'OCL dans Kermeta. Le support des contraintes, en particulier sous une forme OCL standard, est une fonctionnalité majeure du langage Kermeta. Pour nos travaux, les contraintes permettent la mise en œuvre d'oracle et la transcription de la spécification sous forme exécutable. Notre deuxième contribution technique est un harnais de test. Il permet l'élaboration de cas de test exploitant les différentes fonctions d'oracles que nous avons proposées dans le précédent chapitre. La dernière contribution technique est une plate-forme pour l'application de l'analyse de mutation sur des transformations de modèles. Cet outil permet la mise en œuvre du processus du « triangle » que nous venons de présenter.

5.2.1 Utilisation du langage Kermeta

Kermeta [Muller'05a] a été conçu par l'équipe Triskell[1] pour être le langage cœur d'une plateforme orientée modèle. Il peut être considéré comme un dénominateur commun de plusieurs technologies orientées modèle, comme illustré dans la figure 5-6. Il consiste en une extension d'Essential MOF (EMOF) 2.0 pour supporter une sémantique opérationnelle exécutable. Il permet l'ajout de comportement dans les méta-modèles dans le but de rendre possible leur simulation. Le langage d'action de Kermeta est impératif, orienté objet et a un système de type orienté modèle. Il supporte la définition de transformations de modèles.

Lorsque nous avons débuté les travaux de cette thèse, nous ne disposions pas d'outil dédié à l'ingénierie des modèles intégrant nativement un système de contraintes. En revanche, le développement de Kermeta était suffisamment mature pour le développement de méta-modèles exécutables et l'élaboration de transformations de modèles. Ainsi la maitrise complète que nous avions de ce langage et le soutien que nous avons reçu des développeurs de l'équipe Triskell nous ont encouragés à réaliser nos développements avec et pour Kermeta. Les bénéfices ont été de pouvoir réaliser une intégration adaptée à nos besoins et de l'ouvrir aux besoins de l'équipe et de la communauté. Kermeta est un langage répandu qui a été téléchargé plus de six mille cinq cent fois[2], dans une soixantaine d'institutions différentes identifiées, et dans une quarantaine de

[1] A l'initiative de Franck Fleurey
[2] Au 29 juillet 2008

pays différents identifiés. Il est employé dans de nombreux projets de recherche nationaux et internationaux, mais également dans des activités d'enseignement.

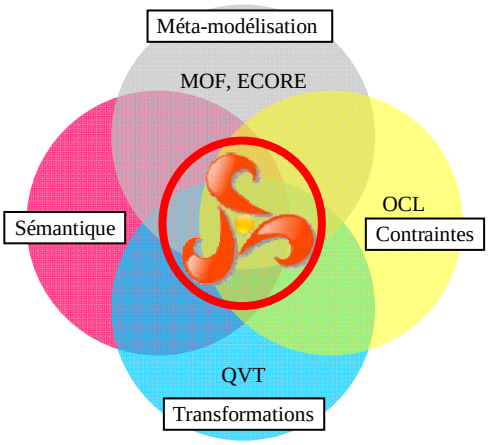


Figure 5-6 - Kermeta: un noyau exécutable de méta-modélisation

Kermeta supporte plusieurs fonctionnalités inspirées du paradigme de programmation par aspect (Aspect-Oriented Paradigm, AOP) [Kiczales'97, Filman'05]. Un mécanisme d'introduction AOP est supporté au travers de la directive « require » (figure 5-7). Il permet au concepteur de définir des fragments de méta-modèle qu'il peut tisser ensemble pour obtenir le méta-modèle complet. Ces fragments peuvent être, par exemple, la structure des données, la sémantique opérationnelle ou de nouvelles fonctionnalités des méta-classes utiles pour une transformation particulière.

D'un point de vue architectural, Kermeta va autoriser la séparation des préoccupations du concepteur. Les concepteurs des méta-modèles vont typiquement travailler avec plusieurs artefacts : la structure est exprimée en un méta-modèle Ecore, la sémantique opérationnelle est définie avec une ressource Kermeta. Le tissage de ces fragments de méta-modèle est réalisé automatiquement en Kermeta.

Dans l'exemple de la figure 5-7, le corps de l'opération `allParents()` est implémenté avec le langage d'action de Kermeta. Cette opération est seulement déclarée dans le modèle ecore décrivant la structure des modèles. Ainsi la sémantique opérationnelle est définie et tissée avec le méta-modèle ecore.

5.2.2 Intégration d'un support des contraintes dans Kermeta

Dans cette sous-section, nous détaillons notre contribution que constitue l'intégration d'un système de contraintes à Kermeta. Il s'agit d'une phase de développement qui a concerné plusieurs parties du langage et de son framework. Cela a nécessité, entre autre, l'ajout de mots clefs dans la syntaxe concrète, de concepts dans le méta-modèle du langage (la syntaxe abstraite), le codage du comportement des contraintes en Java et Kermeta. Ce développement et son test ont duré plus de deux mois et ont nécessité l'écriture d'environ 5000 lignes de code.

Dans un premier temps, nous motivons ce travail. Puis nous détaillons la manière dont nous l'avons réalisé dans Kermeta. Enfin, nous expliquons l'ajout d'un support d'OCL à Kermeta.

```
package simpleUML_MM;
require " simpleClassConstraints.kmt"
using kermeta
using kermeta::standard

@aspect "true"
class Class{
  // Operational semantic of the method allParents
 operation allParents() : Class[0..*] is  do
     result := getParents (
        self.container.asType(ClassModel).classifier
           .select{clr | Class.isInstance(clr)}.size)
 end

 operation getParents(nbmaxRecurs:Integer):Class[0..*]
 is do
   result := OrderedSet<Class>.new
   if(self.parent != void)
   then result.add(self.parent)
        if(nbmax_recursion > 0)
        then nbmax_recursion := nbmaxRecurs-1
             result.addAll(self.parent.getParentsRecursive(nbmax_recursion))
        end
   end
 end
}
```

Figure 5-7 - Implémentation d'une opération du méta-modèle de classe

a- Motivations

Les contraintes sont utilisées pour exprimer des propriétés sur des modèles. Dans le domaine de l'IDM, nous les employons de différentes manières:

- Directement dans les travaux de cette thèse, les contraintes permettent l'élaboration d'oracles de test de transformations. Nous avons montré au précédent chapitre qu'il s'agit d'une solution envisageable, en particulier sous la forme d'assertions dédiées à un modèle de sortie donné.
- Avec un support des contraintes, il est possible de développer des programmes Kermeta (de toutes formes) en suivant une méthodologie design-by-contract. Dans ce chapitre, nous montrons également que transcrire la spécification de la transformation avec des contrats permet de définir des composants autotestables,
- Au-delà des travaux de cette thèse, les contraintes sont utilisées dans les activités de méta-modélisation. Elles peuvent être définies au niveau des méta-modèles pour contraindre leurs modèles. Le Meta-Object Facility (MOF) [OMG'04] est le langage standard de l'Object Management Group (OMG) pour la définition de méta-modèles. Le

MOF permet de définir des méta-modèles qui sont des modèles exprimant des propriétés structurelles. Par exemple ils spécifient les relations entre concepts et leurs cardinalités. Mais des langages comme le MOF, ou les diagrammes de classes d'UML, ont un pouvoir d'expression limité. Ils ne fournissent pas de moyen de spécifier toute la sémantique statique de méta-modèles, ce qui nécessite l'emploi de contraintes.

Les contraintes que nous considérons sont de trois types: les *pré-conditions*, les *post-conditions*, et les *invariants*. Elles sont définies telles que dans le design-by-contract. Leur support est une fonctionnalité majeure de Kermeta, ce qui lui confère un quatrième aspect (le rond de droite de la figure 5-6) déterminant pour en faire une plate-forme complète pour le développement dirigé par les modèles. Les contraintes dans Kermeta sont utilisées dans différents projets de recherche nationaux ou internationaux, comme par exemple : Speeds, DiVA, Mopcom, Domino.

b- Mise en œuvre

La figure 5-8 représente une partie du méta-modèle de Kermeta qui permet de définir les contraintes. Elles sont représentées par la classe `Constraint` et peuvent être de trois types : des invariants, des pré-conditions, ou des post-conditions. Les invariants sont contenus par une définition de classe (`ClassDefinition`). Les pré-conditions et post-conditions sont contenues par des opérations.

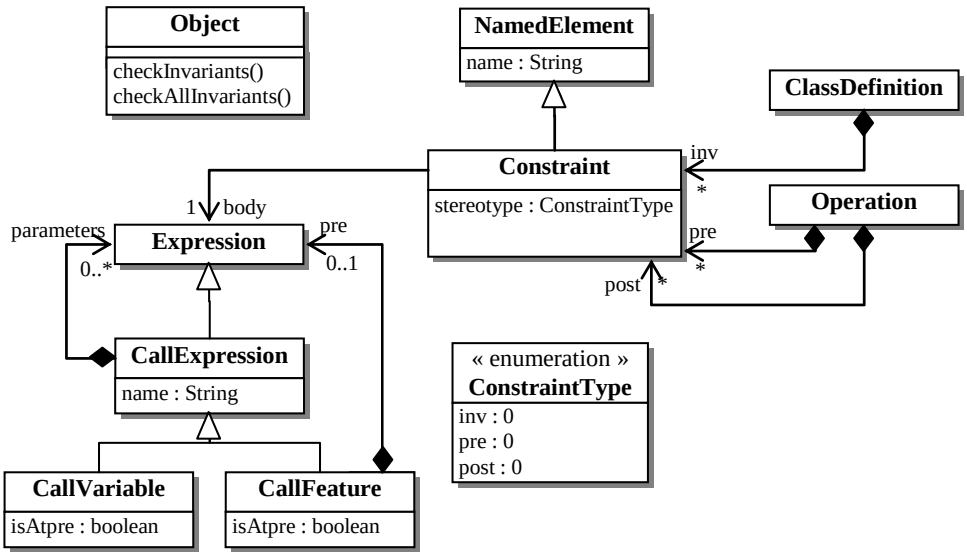


Figure 5-8 - Intégration des contraintes dans le méta-modèle de Kermeta

L'opération `checkInvariant` définie dans la classe Object permet d'appeler la vérification des invariants d'un objet. L'opération `checkAllInvariant` appelle en plus la vérification des invariants de tous les objets contenus. Elle permet de vérifier à partir de la racine d'un modèle les invariants de toutes ses classes. Ces opérations renvoient `true` si tous les invariants sont respectés et lèvent une exception dans le cas contraire. Ces exceptions peuvent être récupérées, elles fournissent des informations sur les invariants violés. Dans l'environnement Kermeta, les invariants ne sont pas systématiquement vérifiés. Les concepteurs réalisent les appels à ces

opérations `check(All)Invariant`. Cela permet de modulariser ces vérifications. Une stratégie respectant la méthodologie de design-by-contract [Meyer'92a] peut être appliquée : les invariants sont appelés avant et après chaque opération ainsi qu'après la création de chaque instance d'une classe. Mais pendant une transformation de modèles, il peut être utile d'employer une autre stratégie. Par exemple, les concepteurs peuvent laisser les modèles prendre des états non-conformes pendant qu'une partie de la transformation est réalisée.

Les pré et post-conditions sont vérifiées automatiquement par l'interpréteur de Kermeta. Elles peuvent toutefois être activées et désactivées au lancement d'un programme Kermeta pour augmenter les performances.

L'ensemble des expressions Kermeta peut être librement utilisé dans les contraintes. Il appartient à l'utilisateur de s'assurer que les expressions contenues dans les contraintes n'ont pas d'effet de bord. Dans le futur, nous envisageons d'implémenter des vérifications statiques pour interdire les constructions avec des effets de bord. Cependant, dans le cas général, ce problème n'est pas décidable car les contraintes Kermeta permettent l'appel à des fonctions arbitraires.

Les contraintes Kermeta peuvent appeler n'importe quelle expression Kermeta. Elles bénéficient du pouvoir d'expressivité du langage. De cette façon, il est possible d'appeler une opération dont seule la signature est définie dans le méta-modèle. Il suffit alors d'implémenter son comportement en Kermeta. De la sorte, il est possible de construire des méta-modèles complets disposant d'une structure (définie en Ecore ou directement en Kermeta), du comportement du méta-modèle (définie en Kermeta), et des contraintes (définies en Kermeta, ou comme nous le verrons au point 5.2.2c- en OCL).

En plus des expressions Kermeta, l'opérateur spécifique @pre peut être utilisé dans les post-conditions afin de faire référence à un état avant l'exécution de l'opération. Cet opérateur correspond à celui d'OCL et à l'opérateur `old` du langage Eiffel. Il peut être appliqué pour exprimer les changements d'états de propriétés (`CallFeature`) ou de variables (`CallVariable`) Kermeta.

Nous avons également considéré l'héritage dans cette intégration. Tout d'abord les invariants d'une classe sont hérités et vérifiés dans une classe fille. Les pré-conditions et les post-conditions d'une opération sont également héritées par la redéfinition de l'opération. Il conviendra d'assurer que les pré-conditions de l'opération fille soient assouplies et que les post-conditions soient renforcées.

Finalement, nous avons également modifié la syntaxe concrète du langage Kermeta pour intégrer tous les mots clefs relatifs aux contraintes et définir la manière d'écrire des contraintes dans un programme. Les exemples de la figure 5-9 et de la figure 5-10 illustrent l'emploi de différentes contraintes. Nous remarquons que l'invariant de la classe `Class` appelle l'opération `allParents` que nous avions implémentée et illustrée dans la figure 5-7.

```
package simpleUML_MM;
require kermeta
require " simpleUML_MM.ecore"
using kermeta::standard

@aspect "true"
class ClassModel{
  inv invariant1 is
  self.classifier.select{clr|Class.instances(clr)}
     .forAllCpl{ c1, c2 | c1.name.equals(c2.name).equals(c1.equals(c2))}
}

@aspect "true"
class Class{
  inv invariant2 is
     self.allParents.includes(self).~not
}
```

Figure 5-9 – Exemple d'invariants dans le fichier simpleClassConstraints.kmt

c- Support d'OCL dans Kermeta

OCL (Object Constraint Language) [Warmer'03, OMG'03] est un standard largement adopté pour l'expression de contraintes dans l'IDM. Il est donc nécessaire de le supporter dans la plate-forme Kermeta et dans la définition des contraintes utilisées dans notre approche du test de transformations de modèles.

```
operation foo(param : String) : String
  pre paramnotempty is param != ""

  post wordisappended is
  do
     if result.size > param.size then result := param + " world"
    end
  end
```

Figure 5-10 -Syntaxe Kermeta pour définir des pré et post-conditions

Avec le support d'OCL par Kermeta, il est possible de définir des contraintes OCL que Kermeta permet d'exploiter automatiquement. Ces contraintes OCL sont transformées en contraintes Kermeta. Cette transformation exogène a pu été réalisée car tous les concepts d'OCL sont dans le langage de contraintes de Kermeta ou peuvent être transcrits en expressions Kermeta. Cette transformation est automatique et a été écrite en Kermeta, elle est appliquée après une phase de parsing du code OCL (figure 5-11).

Le concepteur fournit uniquement son méta-modèle en un fichier « `.ecore` » et ses contraintes OCL dans un fichier « `.ocl` ». Puis les contraintes OCL sont transformées en contraintes Kermeta. Finalement elles sont automatiquement tissées avec les méta-modèles qui les emploient.

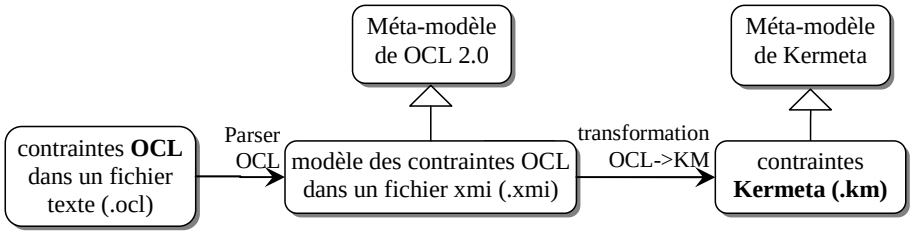


Figure 5-11 - Processus de transformation des contraintes OCL en Kermeta

Par exemple les invariants OCL de la figure 5-12 deviennent les invariants Kermeta de la figure 5-9. Nous remarquons l'instruction « `require simpleUML_MM.ecore` » qui réalise le tissage des contraintes avec le méta-modèle.

La mise en œuvre d'OCL bénéficie directement de l'intégration de notre système de contraintes à Kermeta. Les qualités des contraintes Kermeta bénéficient directement au support d'OCL. Par exemple, il est possible d'étendre la logique du premier ordre d'OCL en appelant n'importe quelle opération déclarée dans le méta-modèle. En implémentant cette opération en Kermeta, cela permet l'évaluation de la contrainte et lui offre une plus grande expressivité. Il devient alors possible d'appeler des fonctions mathématiques, de définir facilement des récursivités et la clôture transitive.

```
package simpleUML_MM
context ClassModel inv:
self.classifier->select(clr| clr.oclIsTypeOf(Class))
      ->forAll(cl1,cl2 | (cl1.name=cl2.name)=(cl1=cl2))
context Class inv:
 not self.allParents()->includes(self)
endpackage
```

Figure 5-12 - Contraintes sur le méta-modèle de classe dans le fichier simpleClass.ocl

Le support d'OCL permet à Kermeta d'être moins invasif dans les contraintes qui ne sont plus nécessairement écrites elles-mêmes en Kermeta. Le concepteur peut publier son méta-modèle et les contraintes associées dans des formats standards : Ecore et OCL sans dépendance avec Kermeta. Dans ce cas, seul le comportement d'opérations nécessaires aux contraintes ou à la simulation est réalisé en Kermeta. Le concepteur des méta-modèles qui utilise déjà Kermeta bénéficie de la compatibilité des standards existants et de la possibilité d'intégrer des contraintes OCL dans ses méta-modèles Ecore ou ses transformations.

5.2.3 Harnais de test supportant les différentes fonctions d'oracle

Nous avons réalisé un support des différentes fonctions d'oracle proposées au précédent chapitre. Pour cela nous avons développé un harnais de test. Il permet de définir différents cas de test regroupant un modèle de test et un oracle. Ces cas de test peuvent être exécutés en produisant le verdict du test.

Nous avons pris en considération la réutilisation des tests avec différentes versions d'une transformation. Il est possible de réutiliser les cas de test, ou seulement les modèles de test, les

oracles, ou même seulement les données d'oracle (les patterns basés sur des model snippets ou les assertions OCL).

Nous avons développé cet outil Kmoth (Kermeta MOdel Transformation Harness) avec Kermeta en exploitant ces différentes fonctionnalités. Nous avons construit un méta-modèle qui permet de réaliser des instances représentant les tests à effectuer, des *modèles de cas de test*. Ce méta-modèle est illustré dans l'Annexe E. Nous avons défini la sémantique de ce méta-modèle en programmant le comportement des différentes méthodes (en Kermeta) qui permettent d'exécuter les tests.

Nous avons défini la structure et les méta-classes de ce méta-modèle pour permettre la réutilisation à deux niveaux :

1. Au premier niveau, nous réutilisons certaines instances d'un modèle de test pour réaliser différents tests d'une même version de la transformation. Les objets `Min` peuvent être associés à plusieurs cas de test (`TestCase`). Un oracle peut également appartenir à plusieurs cas de test. Nous avons vu dans le précédent chapitre que les oracles génériques étaient par exemple associés à plusieurs cas de test. Il est aussi possible de réutiliser les données d'oracle des fonctions utilisant des patterns (avec des model snippets et des assertions OCL).
2. Au deuxième niveau, nous réutilisons certains cas de test pour tester plusieurs versions d'une transformation. Nous avons créé une méta-classe `Intent` qui permet de spécifier l'intention des cas de test. Elle peut contenir plusieurs cas de test car les intentions peuvent nécessiter l'emploi de plusieurs cas de test. Une intention spécifie des objectifs fonctionnels que ces cas de test doivent vérifier. Par exemple, une intention peut être : « Vérifier qu'une classe persistante sans parent persistant correspond à une table ». Cette intention peut contenir plusieurs cas de test, chacun associé à un modèle de test différent (`Min`).

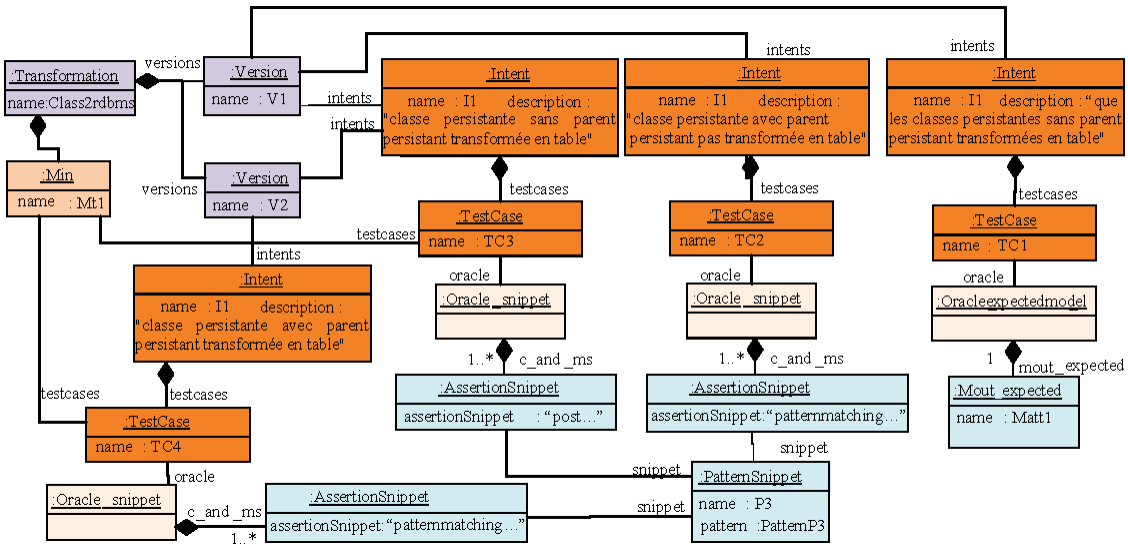


Figure 5-13 - Extrait d'un modèle de cas de test pour la transformation class2rdbms

Par exemple, un extrait d'un modèle de cas de test (simplifié) est illustré dans la figure 5-13. Il permet de tester la transformation class2rdbms dans les deux versions que nous avons étudiées au chapitre 4. Nous constatons que le pattern P3 est utilisé plusieurs fois que ce soit avec une même version et avec plusieurs, que le cas de test TC3 est utilisé avec les deux versions car l'intention qu'il réalise reste valable.

5.2.4 Plate-forme pour l'application de l'analyse de mutation

Nous avons réalisé un ensemble de programmes pour appliquer l'analyse de mutation. Cette réalisation permet la mise en œuvre du processus de construction de composants de confiance. Il permet également d'exploiter directement l'analyse de mutation.

Nous avons décomposé l'implémentation de cette plate-forme en trois parties. Elles fournissent les différentes fonctionnalités que nous avons discutées dans cette thèse (chapitre 3):

- Un programme `KMutationAnalysis` pour la mise en œuvre du processus de l'analyse de mutation : les modèles de test sont exécutés avec la transformation et ses mutants. En sortie, le score de mutation est calculé et la matrice détaillant les mutants que chaque modèle de test tue est retournée. Il est possible d'utiliser la comparaison de modèles ou de contrats comme oracle. La matrice informe de la manière dont un mutant est tué : par comparaison ou par quel contrat. Ce déroulement est la principale fonctionnalité nécessaire pour appliquer l'analyse de mutation. Dans ce programme, nous avons intégré une autre fonctionnalité pour détecter les modèles de test hors du domaine d'entrée : ceux qui font échouer la transformation ou qui sont transformés hors du domaine de sortie (nous discutons ce point dans la sous-section 5.3.6 de ce chapitre).
- Un second programme `KMatrixReduction` supporte la minimisation de la matrice retournée à la fin de l'analyse. L'algorithme présenté au point 3.3.2b- est implanté. En sortie, nous obtenons la liste des réductions possibles et les matrices dont la minimisation est maximale. Ces matrices contiennent les ensembles de test minimaux qui obtiennent le même score de mutation que l'ensemble qualifié par l'analyse de mutation.
- Un autre programme `RDBMSComparator` spécifique à l'étude de la transformation class2rdbms vérifie l'égalité de deux modèles RDBMS. Nous avons écrit un programme Kermeta qui compare deux modèles RDBMS. Nous nous sommes basés sur le méta-modèle de RDBMS pour définir comment comparer les objets et la structure des modèles. Ce comparateur ad-hoc était nécessaire car il n'y avait pas de comparateur intégré à Kermeta au moment de ces travaux.
- Nous avons également développé un programme `KTestRegression` pour contrôler les évolutions du comportement de la transformation. Il compare les modèles de sortie obtenus avec deux versions différentes de la transformation. Quand l'implantation subit des modifications, nous pouvons constater leur influence sur la transformation des modèles de test. Des tests de non-régression peuvent ainsi être appliqués : si un modèle

de test était transformé en un modèle de sortie correct, alors celui-ci ne doit pas être affecté par des corrections effectuées dans l'implantation de la transformation.

Ces programmes sont implantés en Java et interagissent avec des programmes Kermeta. Ainsi le processus de l'analyse appelle des transformations (sous test et mutantes) écrite en Kermeta. De même, le comparateur de modèles RDBMS est écrit en Kermeta. Nous utilisons cette plate-forme dans un programme de lancement Java dont nous illustrons l'emploi dans l'Annexe C.

5.3 Expérimentations pour le développement d'un composant de confiance

Cette section illustre le déroulement du processus itératif proposé pour augmenter le niveau de confiance d'un composant de transformation de modèles. Nous utilisons la transformation CLASS2RDBMS et son implémentation réalisée en Kermeta (www.kermeta.org). Les parties du processus qui ont été illustrées dans les deux précédents chapitres sont rappelées ici.

5.3.1 Étape 1 : Qualification et amélioration des modèles de test

Au cours de la première étape nous considérons le sommet du triangle des cas de test (figure 5-14). L'ensemble de cas de test est qualifié par analyse de mutation. Si le score de mutation est insuffisant, il est amélioré par la création de nouveaux modèles de test.

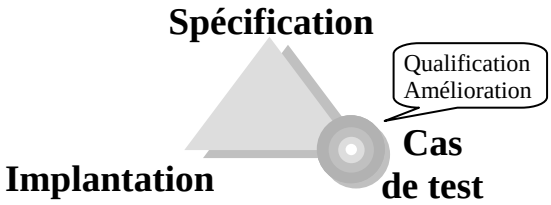


Figure 5-14 - Qualification et amélioration du premier sommet, les cas de test.

L'ensemble initial de modèles de test peut être obtenu de plusieurs façons :

- **Modèles d'entrée existants** : durant la mise au point de l'implantation, le développeur a écrit différents modèles d'entrée.
- **Génération automatique d'un ensemble de modèles de test**
- **Écriture par le testeur d'un premier ensemble de modèles de test**

L'ensemble initial de modèles de test, déjà utilisé au chapitre 3 (section 3.4), provient de la mise au point.

L'ensemble initial de modèles de test est constitué de sept modèles de test et cent soixante six mutants sont générés pour class2rdbms. Cet ensemble a un score de mutation de 78%. Selon le niveau de confiance visé, ce score peut être insuffisant et la qualité de ce sommet d'un niveau trop faible. De plus, dans un but expérimental nous cherchons à atteindre le score de 100%. Nous écrivons donc neuf nouveaux modèles de test pour atteindre le score de 100%. Les détails de ce processus ont déjà été donnés au chapitre 3.

Après l'application de la première étape du processus, nous obtenons un ensemble de seize modèles de test dont le score de mutation est de 100%. En appliquant l'algorithme de réduction de matrice (section 3.3.2b-), nous sélectionnons six modèles de test suffisants pour obtenir le score de 100%. Le tableau 5-1 résume le résultat obtenu après cette première étape.

	avant amélioration des modèles	**après amélioration des modèles**	**après amélioration et réduction**
nombre de modèles de test	7	16	6
score de mutation	78%	100.0%	100%

Tableau 5-1 - Résultat de la première étape à la première itération.

Finalement, des cas de test sont formés à partir des modèles de test. Pour cela, des oracles sont élaborés en fonction de la spécification et des modèles de test disponibles. Ils sont construits avec les fonctions fournies au précédent chapitre et par la construction de données d'oracle. Les contrats disponibles sont également utilisés pour former des oracles.

L'écriture d'oracle pour les six modèles de test sélectionnés a été illustrée dans le précédent chapitre. Les différents cas de test que nous obtenons sont détaillés dans l'Annexe A. Nous illustrons cette section en employant les oracles utilisant la comparaison avec des modèles attendus (Annexe A - A.2).

5.3.2 Étape 2 : Test et correction d'erreur

Dans cette seconde étape, l'implantation est considérée (figure 5-15) et confrontée aux cas de test. Si certains ne passent pas, l'implantation est corrigée après localisation des erreurs. Après la correction, des tests de non-régression doivent être réalisés en exécutant à nouveau les cas de test.

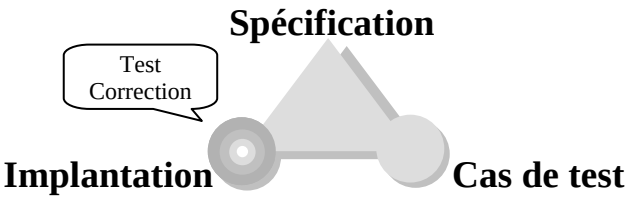


Figure 5-15 - Test et correction de l'implémentation.

Dans notre expérimentation, certains cas de test ne passent pas. Dans le tableau 5-2, nous synthétisons les résultats de ce point en ne listant que les résultats obtenus avec la fonction d'oracle exploitant le modèle attendu.

Le cas de test CTma1 (Annexe A - A.2) échoue avec l'implantation initiale, son étude révèle une erreur concernant la création des noms de colonnes. Les noms d'attributs et d'associations doivent être concaténés autour du caractère « _ » pour former le nom d'une colonne. Or le modèle de sortie de la transformation du modèle de test 7 avec l'implantation initiale contient des colonnes avec les noms "attbpsi" et "iotabpsi" au lieu de "att_b_psi" et "iota_b_psi".

Fonction d'oracle	Nombre de cas de test exécutés	Cas de test échouant avec l'**implantation initiale**	Cas de test échouant après la **première correction**	Cas de test échouant après la **seconde correction**
o3 (modèle attendu)	6	CTma1, CTma2 CTma3, CTma4 CTma6	CTma2 CTma3	CTma3

Tableau 5-2 - Échecs des cas de test pendant le test de l'implantation

Après cette première correction, deux cas de test ne passent toujours pas. L'étude du cas de test CTma2 révèle une seconde erreur : plusieurs tables sont créées à partir de classes persistantes appartenant à la même hiérarchie d'héritage. La spécification interdit cette création multiple, nous avons donc corrigé l'implantation pour considérer cette exigence. C'est l'emploi du modèle de test 8 qui a révélé cette erreur. Les conséquences de cette correction sont plus importantes que celles de la précédente correction.

Cette seconde correction entraine une régression car le modèle de test MT9 ne peut plus être transformé alors qu'il l'était avant la correction. Nous avons constaté cela à partir de la fonctionnalité vérifiant les régressions de la plateforme développée (5.2.3). Une troisième erreur a été corrigée, elle portait sur la création de `FKey` pour des colonnes n'existant pas.

Nous remarquons que ces deux dernières erreurs n'étaient pas mises en évidence par l'ensemble initial de modèles de test. Ce sont deux modèles de test (les MT8 et MT9, utilisés dans les cas de test CTma2 et CTma3 par exemple) créés pour améliorer le score de mutation de l'étape précédente qui révèlent ces erreurs.

5.3.3 Seconde itération

Une fois la transformation corrigée, il faut de nouveau appliquer la première étape puis la seconde (comme illustré figure 5-5) : produire à nouveau les mutants, procéder à l'analyse de mutation, tester. En effet, les mutants ne sont plus valables puisque l'implantation a changé. Ils ont été créés à partir de la transformation initiale et ne correspondent plus à l'implantation corrigée. Il est nécessaire de revenir à la première étape pour produire un nouvel ensemble de mutants et pour obtenir un nouveau score de mutation. En effet, si T est la transformation sous test et $\{Mij\}$ est l'ensemble des mutants de T tels que $M_{ij} = om_{ij}(T)$ la jième application de l'opérateur de mutation om_i, alors la correction c de T produit T' telle que $T' = c(T)$. T' permet de produire $\{M'_{ij}\}$ le nouvel ensemble des mutants de T' tels que $M'_{ij} = om_{ij}(T') = om_{ij}(c(T))$ ce qui n'implique pas à priori que $M'_{ij} = c(M_{ij})$.

Nous appliquons donc à nouveau la première étape. Les corrections faites dans la transformation conduisent à la création de deux cent mutants. Cent soixante six mutants correspondent au précédent ensemble car à posteriori nous constatons qu'ils ont été produits par l'application des mêmes opérateurs sur des parties non corrigées de l'implantation. A ces mutants s'ajoutent seize mutants supplémentaires avec des erreurs de filtrage et dix-huit autres avec des erreurs de navigation portant sur les corrections apportées à la transformation. Nous

appliquons l'analyse de mutation à l'ensemble des seize modèles de test en utilisant ces mutants et la version corrigée de la transformation. Tous les modèles de test sont considérés car la réduction, qui avait permis d'en sélectionner six, se basait sur les précédents mutants. Le nouveau score de mutation est de 99% puisque les modèles de test ne tuent pas deux mutants. Finalement, la création de deux nouveaux modèles de test permet d'atteindre à nouveau le score de 100%.

Puis nous appliquons à nouveau la seconde étape pour vérifier que les deux nouveaux modèles de test et leurs cas de test respectifs ne révèlent pas de nouvelles erreurs de la transformation. Ce n'est pas le cas dans cette expérience.

5.3.4 Étape 3: Améliorer les contrats

Pendant cette troisième étape, nous mettons en œuvre une méthode d'amélioration de la spécification embarquée, sous sa forme de contrats exécutables (figure 5-16). Il s'agit de mettre à l'épreuve ces contrats avec les mutants. Pour cela ils sont utilisés comme oracle de l'analyse de mutation. Comme nous savons que les modèles de test sont capables de déclencher les erreurs injectées dans les mutants (score de 100% à l'étape précédente), nous pouvons mesurer la capacité des contrats à détecter ces erreurs. L'amélioration des contrats consiste à créer des contrats qui traduisent la spécification le plus complètement possible.

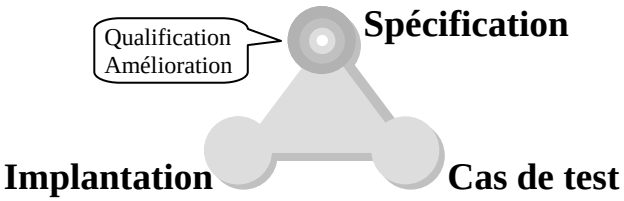


Figure 5-16 - Qualification et amélioration du troisième sommet.

Les mutants qui étaient tués avec les modèles de test doivent être tués en utilisant les contrats. En revanche, les mutants équivalents ne doivent pas être tués par les contrats. Leur utilisation permet de contrôler l'exactitude des contrats qui sont complexes à écrire. Si un nouveau contrat est violé en utilisant les mutants équivalents, une analyse est nécessaire. Le contrat peut être erroné, mais il peut aussi mettre en évidence une erreur de la transformation. Dans ce cas une nouvelle itération est réalisée en utilisant ce(s) nouveau(x) contrat(s) à l'étape 2.

a- Un ensemble initial de contrats

Initialement, les contrats peuvent provenir de deux sources:

- Des contrats compris dans la spécification
- Les contrats disponibles dans l'implantation quand le développement a été réalisé avec une méthodologie design-by-contract

Dans notre expérience, la spécification du workshop [Bézivin'05] fournit un contrat (en OCL) qui précise le domaine d'entrée de la transformation :

```
context Class inv:
allAttributes()->size > 0 and
allAttributes()->exists(attr | attr.is_primary = true)
```

Nous pouvons utiliser ce contrat directement en Kermeta. Il suffit d'implémenter la méthode `allAttributes()` en Kermeta.

Les opérations de l'implantation comportent également huit contrats (deux pré-conditions et six post-conditions). A ce moment, nous considérons que nous avons des contrats basiques (niveau 1) dans notre composant. La qualité (Q_{cont}, mesurée comme défini au 5.1.3) de ces neuf contrats est de 59%. Ce résultat est représenté dans la première colonne de la figure 5-17.

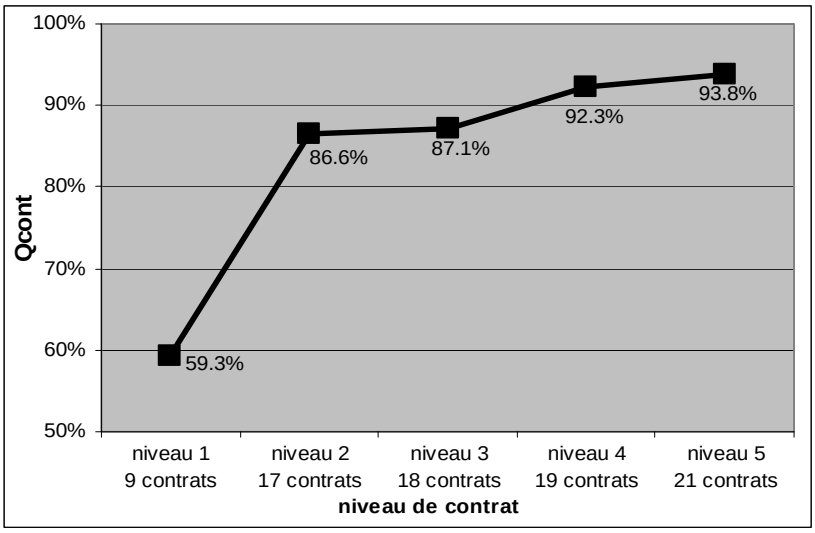


Figure 5-17 – Progression du score de mutation pendant l'amélioration des contrats.

b- Amélioration des contrats: introduction de contrats de sémantique comportementale

Nous procédons à l'amélioration des contrats en deux étapes. Dans ce point b-, nous traduisons la spécification en contrats et nous écrivons des contrats de sémantique comportementale (comme définis dans la section 5.1.1). Ensuite dans le point c-, nous améliorons les contrats à l'aide des mutants.

Dans un premier temps, nous avons défini deux *contrats du domaine de sortie* (des post-conditions). Le méta-modèle cible (figure 2-7) n'est pas assez restrictif concernant les clés des tables: il n'impose pas que les clés primaires d'une table soient parmi ses colonnes et qu'une clé étrangère ait ses colonnes parmi celles de sa table. Nous écrivons deux contrats qui prennent en compte ces contraintes (en Kermeta) :

```
post pkey_is_in_the_column_of_its_table is
     result.table.forAll{t|t.pkey.forAll  {pk|pk.container == t}}

post colunms_of_fkey_are_among_the_ones_of_its_table is
result.table.forAll{t|t.fkeys.forAll{fk|fk.cols.forAll{c|t.cols.contains(c)}}}
```

Nous avons également défini six *contrats reliant entrée et sortie* (des post-conditions). Par exemple, un de ces contrats écrit en Kermeta est donné dans la figure 5-18. Il permet de vérifier que : dans une table correspondant à une classe cl, il y a des colonnes qui correspondent aux attributs cl dont les types t sont des classes persistantes (sans parent persistant) ayant des attributs primaires de type primitif.

Ces contrats de second niveau s'ajoutent aux contrats du premier niveau. En évaluant la qualité de cet ensemble de vingt-et-un contrats, nous mesurons un Q_{cont} de 86% (comme illustré figure 5-17), soit un gain de 27%.

c- Amélioration des contrats grâce aux opérateurs de mutation

Dans un deuxième temps, nous améliorons les contrats pour renforcer leur pouvoir de détection d'erreurs. L'ensemble amélioré des contrats doit traduire au maximum la spécification pour considérer davantage d'exigences, ou de manière plus précise : soit des contrats entièrement nouveaux sont élaborés, soit les existants sont modifiés. Dans ce dernier cas, un nouveau contrat est systématiquement créé pour deux raisons :

- La complexité d'une modification ne doit pas corrompre un contrat existant. Il est plus facile de créer de nouveaux contrats, comme nous avons expliqué qu'il est plus facile de créer de nouveaux modèles de test (3.3.1a-).
- Chaque contrat considère une partie donnée de la spécification. De cette façon (comme expliqué au chapitre précédent), il peut être possible de réutiliser certains contrats quand la spécification est modifiée.

Comme dans la première étape (section 5.3.1), nous utilisons notre connaissance des erreurs injectées dans les mutants vivants. Exploiter ces erreurs permet de faciliter l'amélioration des contrats. En effet, nous pouvons constater avec l'exemple de la figure 5-18 (mais également dans l'Annexe A - A.3) la complexité des contrats.

```
1 post attribute_persistent_classes is
2     // Création des colonnes correspondant aux attributs dont le type est une
3     // classe persistante transformée en table
4  inputModel.classifier
5  .select{cr| Class.isInstance(cr)}
6  .select{cs | cs.asType(Class).is_persistent}
7  .select{csp | not csp.asType(Class).allParents.exists{p | p.is_persistent}}
8  .forAll
9     {csp | // pour toute classe persistante sans parent persistant,
10       result.table.select{t|t.name == csp.name} //dans la table correspondante,
11       .exists{tn |   csp.asType(Class).attrs
12               .select{at | Class.isInstance(at.type)}
13                  // les attributs dont le type est une classe
14               .select{atc|atc.type.asType(Class).is_persistent}
15                  //qui est persistante
16               .select{atcp |    not atcp.type.asType(Class).allParents
17                                 .exists{p | p.is_persistent}}
18                                 // mais sans parent persistant
19               .forAll
20                  {atcpwp|    // (atcpwp est un attribut don't le type est une classe
21                              // persistante sans parent persistant
22                     atcpwp.type.asType(Class).attrs
23                     .forAll
24                        {atcpta| //pour tous les attributs de ce type
25                           if    PrimitiveDataType.isInstance(atcpta.type)
26                                    //qui sont de type primitif
27                                 and atcpta.is_primary   // et qui sont primaires
28                           then
29                              tn.cols.select{ctn |
30                                    ctn.name == atcpwp.name+"_"+atcpta.name
31                                       // alors il y a une colonne du nom de cette
32                                       // attribut + « _ » + le nom de l'attribut
33                                       // primaire de type primitif
34                                    and ctn.type == atcpta.type.name
35                                       // et du type de ce dernier attribut,
36            //to be improved
37                                 }.size==1 //cette colonne est l'unique correpondante
38                           else true end
39                        }
40                  }
41          }
42   }
```

Figure 5-18 - Un exemple de contrat de sémantique comportementale

Dans l'expérimentation, nous avons, par exemple, essayé de tuer un mutant qui n'associe pas les colonnes à une clé étrangère (`FKey`). Nous avons créé un contrat à partir de celui de la figure 5-18 en vérifiant que la colonne considérée `ctn` est une des colonnes d'une `fkey` de la table `tn`. C'est une amélioration qui se traduit par le code de la figure 5-19. Il remplace le commentaire « `//to be improved` » à la ligne 36 de la figure 5-18 pour créer un nouveau

contrat. L'opérateur de mutation supprime la création d'une relation entre une `fkey` et ses colonnes, donc le contrat amélioré vérifie l'existence de cette relation.

```
and tn.fkeys.exists{ftn | ftn.cols.contains(ctn)}
```

Figure 5-19 - Amélioration d'un contrat

En appliquant cette méthode, nous améliorons la valeur Q_{cont} de l'ensemble des contrats du composant. Nous l'illustrons dans les troisième, quatrième, cinquième colonnes de la figure 5-17. Cette fois, nous obtenons un gain pour Q_{cont} de 7%.

5.3.5 Conclusion des expérimentations

Pour conclure cette expérimentation, il ne reste plus qu'à employer ces nouveaux contrats améliorés pour le test à la seconde étape du processus. Cela ne révèle pas de nouvelle erreur.

Dans cette section, nous avons illustré l'application du processus itératif (figure 5-5) qui permet de renforcer chacun des sommets du triangle et la cohérence entre ses sommets. La figure 5-20 synthétise le résultat de l'expérimentation.

Dans la première étape (appliquée deux fois), nous avons augmenté le score de mutation de l'ensemble de modèles de test de 22% avec huit nouveaux modèles de test (voir les courbes dans la partie droite de la figure 5-20). Le nombre de modèles a plus que doublé passant de sept à dix-huit. L'amélioration des tests est facilitée par l'exploitation des erreurs injectées par les opérateurs de mutation.

Dans la deuxième étape (appliquée deux fois), nous avons testé l'implantation. Trois erreurs ont été détectées et corrigées. Le nombre de mutants générés après correction est passé de cent soixante six à deux cents (partie gauche de la figure).

Dans la troisième étape, nous avons augmenté la qualité des contrats faisant augmenter la mesure Q_{cont} de 35%. Douze nouveaux contrats ont été créés. Ils permettent de traduire davantage la spécification dans une forme exécutable.

Par cette expérimentation, nous montrons que la méthodologie permet d'exploiter directement les contributions de cette thèse pour assurer la qualité d'un composant de transformation de modèles :

- La confiance dans le composant est assurée par les tests réalisés avec des modèles de test qualifiés.
- Les modèles de test permettent de tester le composant dans un nouveau contexte d'utilisation
- Une spécification exécutable est embarquée et sa précision a été mesurée.

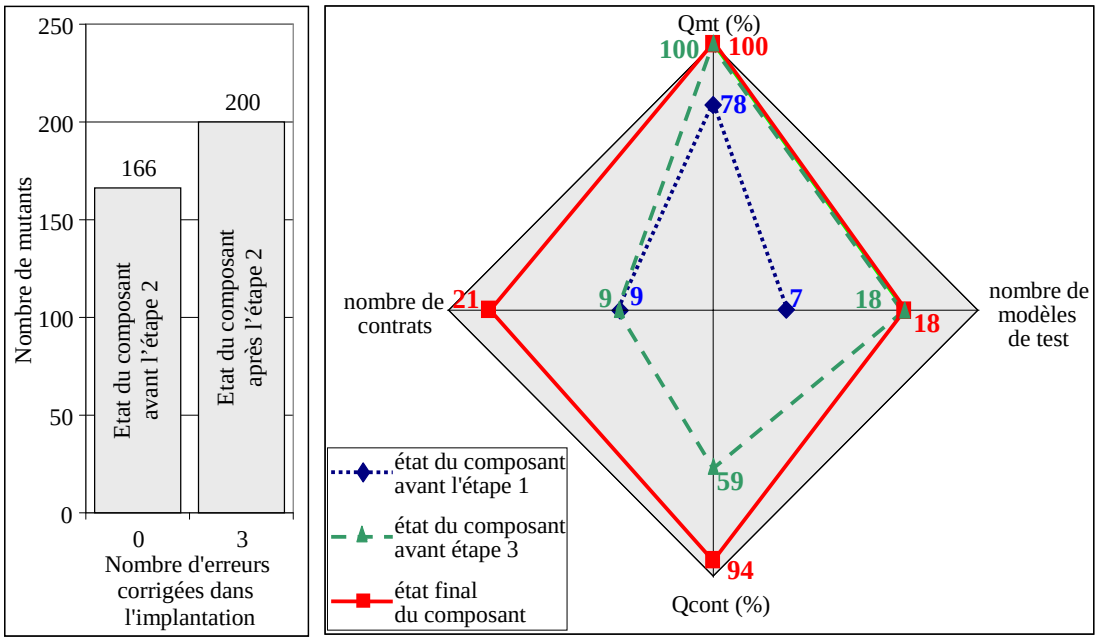


Figure 5-20 – Évolution des éléments du composant en appliquant la méthodologie

5.3.6 Une adaptation de la méthodologie pour l'utilisation d'un générateur de modèles

Le contexte des transformations de modèles impose de considérer avec précision l'entrée de l'analyse de mutation. En effet, le domaine d'entrée de la transformation est vaste mais peu spécifié par des éléments exécutables : uniquement le méta-modèle source (avec ses invariants) et des pré-conditions disponibles dans la spécification.

Si le développeur a écrit les modèles de test utilisés dans l'ensemble initial, alors il aura pris soin de vérifier que ces modèles peuvent être transformés. Ce n'est pas le cas quand les modèles sont générés automatiquement. En effet, l'analyse de mutation ne peut pas être appliquée à certains modèles générés pour trois raisons :

- soit la transformation sous test ne peut pas les transformer,
- soit la transformation sous test les transforme en modèles n'appartenant pas au domaine de sortie,
- soit la transformation sous test les transforme en modèles n'appartenant pas au domaine d'entrée du comparateur de modèles, violant ses pré-conditions.

Chacune de ces raisons peut être occasionnée par une de ces deux causes :

- Soit la transformation est erronée. Elle devrait produire des modèles corrects. Elle doit donc être corrigée.
- Soit le domaine d'entrée de la transformation n'est pas suffisamment défini. Les pré-conditions fournies dans la spécification sont insuffisantes. Dans ce cas, ces modèles

n'appartiennent pas au domaine d'entrée de la transformation et ne peuvent servir de modèles de test.

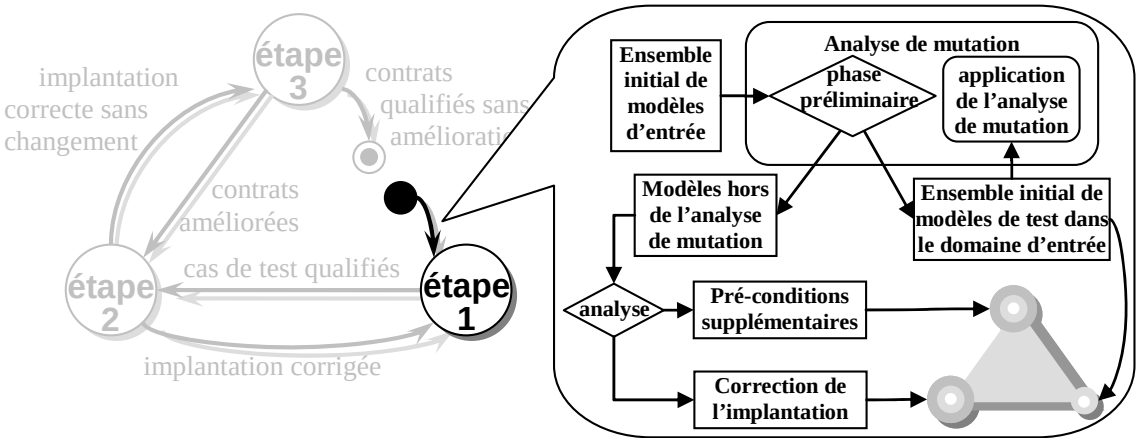


Figure 5-21 - Étape préliminaire à l'analyse de mutation pour l'étude de l'ensemble initial de modèles de test

Ainsi en amont de la méthodologie, en préliminaire de l'étape 1, nous étudions l'ensemble initial de modèles de test. Cet ensemble est représenté dans la figure 5-21. L'implantation que nous avons réalisée pour l'expérimentation (5.2.3) met en évidence les modèles que l'analyse de mutation ne peut pas exploiter. Chacun est étudié pour déterminer s'il révèle une erreur ou s'il n'appartient pas au domaine d'entrée. Dans le premier cas, l'implantation est corrigée et l'analyse de mutation peut ensuite être appliquée avec cette nouvelle implantation (impliquant de nouveaux mutants). Dans le deuxième cas, de nouvelles pré-conditions sont créées pour préciser le domaine d'entrée de la spécification. Ces nouvelles pré-conditions permettent d'ajuster un générateur de modèles. Il produit alors des modèles appartenant au domaine d'entrée plus précis de la transformation et sur lesquels l'analyse de mutation peut être appliquée.

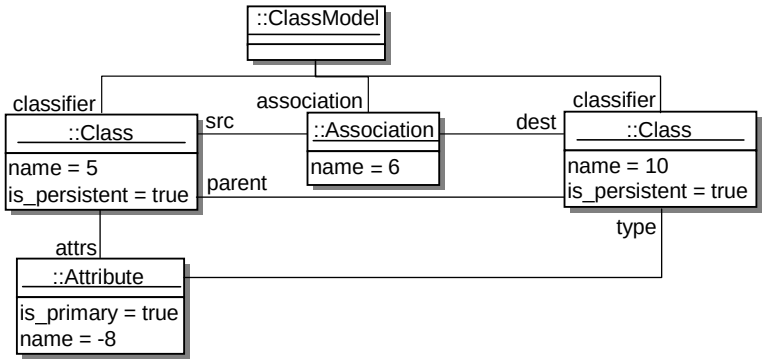


Figure 5-22 – Un modèle généré en entrée de la transformation

Nous avons expérimenté cette phase préliminaire. Le générateur de modèles Cartier[1] [Sen'08] produit des modèles tels que celui de la figure 5-22. Ce modèle appartient au domaine

[1] Développé par Sagar Sen dans ses travaux de thèse.

d’entrée initial de la transformation mais il est transformé en un modèle qui n’est pas conforme au méta-modèle cible. Ce modèle d’entrée ne comporte pas de type primitif, donc aucune colonne n’est créée car elle ne pourrait pas être typée. Ce modèle ne révèle pas d’erreur de la transformation mais impose la création d’une pré-condition.

Les pré-conditions que nous avons écrites grâce à cette adaptation du processus sont données dans l’Annexe B.

5.4 L’analyse de mutation pour l’expérimentation de critères fonctionnels

Les travaux de cette thèse s’intègrent dans une recherche à longue échéance pour la validation des transformations de modèles. Les différentes contributions que nous proposons participent directement au test de transformations de modèles. D’autres aspects de la problématique sont le sujet de différentes études dans l’équipe Triskell. En particulier, Franck Fleurey et al. [Fleurey'07a] ont défini des critères fonctionnels pour le test de transformations de modèles. L'approche proposée a besoin d'une validation expérimentale. Il s'agit de démontrer que les modèles qui satisfont ces critères sont efficaces pour détecter des erreurs. Notre adaptation de l'analyse de mutation (chapitre 3) et son implémentation dans une plateforme expérimentale permettent de réaliser cette évaluation.

De cette manière, nous montrons que l'adaptation de l'analyse de mutation va au delà de son utilisation directe pour tester une transformation. Elle est aussi utile pour développer d'autres techniques de test qui seront soit complémentaires, soit adaptées à d'autres contextes d'utilisation.

5.4.1 Critères de couverture du domaine d'entrée

La technique se base sur un partitionnement du domaine d'entrée. Les propriétés du méta-modèle source de la transformation sont considérées pour déterminer des partitions de leur plage de valeur. La figure 5-23 illustre le partitionnement réalisé pour le méta-modèle source de la transformation class2rdbms. Par exemple, l’ensemble des valeurs de « `attrs` » est partitionné en trois intervalles [0..0], [1..1], et [2..+∞ [.

Fleurey et al ont défini différentes stratégies qui exploitent ce partitionnement. Une stratégie consiste à déterminer des combinaisons de partitions dans des fragments de modèle. Parmi les dix stratégies existantes, `AllRanges` crée un fragment de modèle par partition, alors que `AllPartitions` crée un fragment de modèle par propriété en combinant les partitions de celle-ci. Cela donne par exemple:

- Un fragment de modèle avec le critère `AllRanges` : Un objet de type `Classifier c` avec `c.name = « »`
- Un fragment de modèle avec le critère `AllPartitions` : Deux objets de type `Class c1` et `c2` avec `c1.is_persistent = True` et `c2.is_persistent=False`

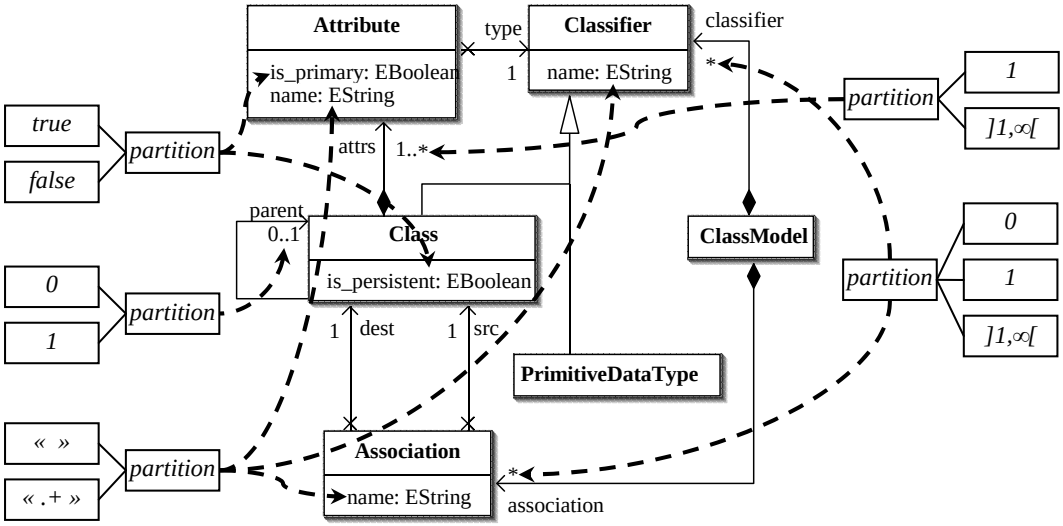


Figure 5-23 - Partitions pour le méta-modèle source de la transformation class2rdbms

La sélection des modèles en fonction d'une stratégie se base sur les fragments de modèle générés. Pour qu'un ensemble de modèles satisfasse une stratégie, chaque fragment de modèle doit être contenu par au moins un modèle.

5.4.2 Mise à l'épreuve de modèles satisfaisant les critères

Pour évaluer la performance d'une stratégie, nous sélectionnons différents ensembles de modèles qui la satisfont et nous mesurons leur score de mutation. Ces ensembles diffèrent entre eux sur les critères qu'ils respectent et sur leur taille. Nous comparons plusieurs ensembles parce que la génération est en partie aléatoire sur le choix des attributs par exemple, et que nous devons établir des statistiques avec des modèles de différentes tailles. Nous analysons les résultats obtenus pour montrer que les modèles de test sélectionnés en utilisant une stratégie détectent davantage d'erreurs que des modèles sélectionnés aléatoirement.

Dans cette expérience, nous étudions deux stratégies différentes `AllRanges` et `AllPartitions`. En appliquant l'outil MMCC [Fleurey'07a] au méta-modèle source (figure 2-7) de la transformation class2rdbms, quinze fragments de modèles sont produits avec la stratégie `AllRanges` et cinq avec `AllPartitions`. Nous générons automatiquement avec Cartier des ensembles dont chaque modèle contient un fragment de modèle différent. Nous obtenons donc des ensembles de quinze modèles satisfaisant `AllRanges` et de cinq modèles satisfaisant `AllPartitions`. Nous décidons enfin de réaliser huit ensembles par stratégies. Chaque ensemble varie selon la taille de ses modèles.

De plus, nous générons de manière aléatoire des ensembles de modèles parmi deux cent modèles de différentes tailles. Huit ensembles de quinze modèles et huit ensembles de cinq modèles sont sélectionnés aléatoirement. Ils permettent de réaliser une comparaison des performances d'une sélection réalisée avec et sans stratégie.

Tous ces modèles ont été produits grâce à Cartier [Sen'08], un outil développé par Sagar Sen pour la synthèse automatique de modèles.

De cette manière, l’expérimentation se compose de trente-deux ensembles contenant un total de trois cent vingt modèles (la répartition est rappelée en abscisse de la figure 5-24).

5.4.3 Validation des critères proposés

Nous appliquons l’analyse de mutation pour chacun des ensembles et obtenons leurs scores de mutation. Pour comparer les différentes stratégies de sélection, nous présentons ces mesures dans le diagramme « boîte à moustaches » de la figure 5-24. Chaque boîte correspond à huit ensembles classés selon la stratégie employée et le nombre de modèles des ensembles.

En premier lieu, nous observons les résultats des ensembles générés sans stratégie. Nous constatons que les ensembles qui comportent plus de modèles (troisième colonne contre première colonne) ont de meilleurs scores de mutation : le plus faible ensemble de quinze modèles a un score supérieur au plus fort ensemble de cinq modèles. Ensuite, nous comparons l’utilisation ou non d’une stratégie. Dans ce cas, tous les ensembles générés sans stratégie ont un score de mutation inférieur au plus faible premier quartile des ensembles générés avec stratégie. Finalement, la comparaison des deux stratégies montre que `AllPartitions` obtient de meilleurs scores de mutation et que la taille de sa « boîte » est plus réduite. En analysant ces mesures, nous déduisons que l’utilisation de plus d’informations pour la sélection de modèles de test fait converger les scores de mutation des ensembles de modèles de test vers des valeurs plus élevées. L’utilisation d’une stratégie plus fine permet de sélectionner des modèles plus performants pour détecter des erreurs.

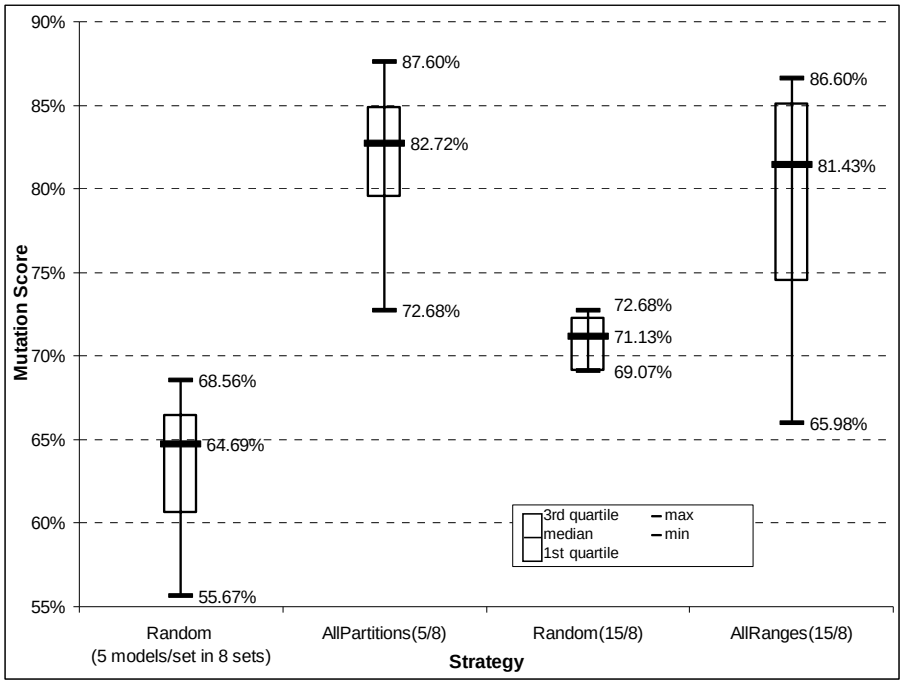


Figure 5-24 - Diagramme "boîte à moustaches" pour la comparaison des stratégies de sélection de modèles

Nous montrons que la sélection de modèles de test est meilleure avec une stratégie que de manière aléatoire. D’un point de vue applicatif, il n’est pas coûteux d’utiliser une stratégie car

MMCC [Fleurey'07a] automatise cette sélection : il génère les fragments de modèle à partir d'un méta-modèle et vérifie la satisfaction d'une stratégie par un ensemble de modèles. Ainsi, en exploitant notre adaptation de l'analyse de mutation et son implémentation, nous contribuons à la validation d'une technique pour le test de transformation de modèles.

5.5 Conclusion

Dans ce chapitre nous avons réalisé plusieurs études. Nous avons tout d'abord expérimenté les contributions des précédents chapitres : l'analyse de mutation et les fonctions d'oracle adaptées au test de transformations de modèles. Nous avons utilisé une approche globale des composants de transformation de modèles en considérant l'implantation, les tests, et la spécification embarqués dans le composant. Cette approche s'appuie sur une méthodologie globale pour la qualification et l'amélioration du niveau de confiance du composant. Les expériences menées ont montré que nos propositions permettent tout d'abord de vérifier et de détecter les erreurs d'une transformation de modèles, ensuite de qualifier ces tests et les contrats qui traduisent la spécification sous une forme exécutable.

Pour réaliser ces expériences, valider l'application des techniques étudiées, nous avons développé plusieurs outils. Le support des contraintes dans Kermeta est un apport à la fois pour nos travaux mais également pour les travaux exploitant la plate-forme Kermeta (en particulier avec le support d'OCL). Le harnais de test est un moyen pour réaliser directement le test de transformations de modèles. Finalement la plate-forme pour le support de l'analyse de mutation, nous permet d'appliquer la méthodologie globale de qualification de composants de transformation de modèles.

L'adaptation de l'analyse de mutation qui est le fondement de ce dernier outil, nous permet d'étendre nos travaux à d'autres problématiques du test de transformations de modèles En particulier, nous avons étudié différents critères de test fonctionnels. En confrontant des modèles satisfaisant ces critères, nous avons montré que ces modèles ont un meilleur pouvoir de détection des erreurs quand la quantité d'information utilisée pour la sélection de modèles de test (et en conséquence pour la génération) est plus importante.

6
Conclusions et perspectives

Les travaux de cette thèse se placent dans un contexte où la complexité grandissante des systèmes logiciels conduit à l'émergence de l'Ingénierie Dirigée par les Modèles (IDM). Cette nouvelle approche du génie logiciel propose d'élever le niveau d'abstraction du développement pour traiter au maximum le système sous forme de modèles. Cette abstraction permet de s'affranchir de contraintes techniques d'une plate-forme, d'un langage. Le développement de l'IDM a déjà fourni des techniques et des outils pour la méta-modélisation, la transformation de modèles, la simulation de modèles, etc. Il est possible de produire des modèles et de les traiter automatiquement pour les améliorer, les analyser, leur ajouter automatiquement des contraintes fonctionnelles ou non. Cependant le succès d'une approche de développement ne réside pas seulement dans les techniques de développement mais également dans les techniques de test qui permettent d'assurer que les différents résultats obtenus soient correct.

Au cours de cette thèse, nous nous sommes intéressés à plusieurs problématiques du test de transformations de modèles. Nous les avons traités et avons obtenu différents résultats que nous reprenons dans la section 6.1. Nous envisageons plusieurs perspectives à nos travaux que nous présentons dans la section 6.2

6.1 Contributions

Les trois contributions majeures de cette thèse s'articulent autour de deux axes : la mise au point de deux techniques de test adaptées aux transformations de modèles, l'intégration de ces techniques dans une approche globale d'un composant de transformation de modèles. Ce deuxième axe a compris différentes expériences et développements d'outils.

La difficulté de l'élaboration de techniques pour le test de transformations de modèles réside essentiellement dans la nature et la complexité des données manipulées : des modèles. Nous avons également pris en compte l'absence de langage de transformation unanimement reconnu et le fait que les traitements d'une transformation sont essentiellement basés sur des méta-modèles définis au niveau méta.

6.1.1 Analyse de mutation pour le test de transformations de modèles

Dans le chapitre 3, nous avons proposé une adaptation de l'analyse de mutation pour le test de transformations de modèles. L'analyse de mutation est impliquée dans plusieurs activités du test. Elle permet de qualifier un ensemble de données de test ce qui est une forme de *critère d'arrêt*. Ce critère permet de guider la génération de nouvelles données si la qualité des données est insuffisante. Cette technique permet également de mettre à l'épreuve les oracles et de vérifier l'efficacité de techniques de génération de données de test.

Il était nécessaire d'adapter l'analyse de mutation au contexte des transformations de modèles pour avoir une technique indépendante de l'implantation des transformations, en particulier du langage utilisé, et pour refléter les erreurs particulières aux transformations.

Nous avons proposé de baser les opérateurs spécifiques aux transformations, non pas au niveau syntaxique du code de l'implantation mais à un niveau sémantique. Les dix opérateurs que nous proposons définissent des erreurs qui altèrent les opérations élémentaires d'une transformation : la navigation d'un modèle, la sélection d'objets par filtrage, et la création d'objets.

Notre adaptation de l'analyse de mutation permet d'évaluer différentes techniques et critères pour le test de transformations. Par exemple, nous travaillons à l'évaluation de critères de test fonctionnels, comme nous l'avons expliqué dans la section 5.4. De plus, la définition des opérateurs au niveau sémantique permet de comparer ces techniques dans différents langages de transformation (en introduisant la même erreur au niveau sémantique dans une transformation implantée dans deux langages différents). Il est également possible d'exploiter les mutants pour mettre à l'épreuve des oracles.

6.1.2 Oracles pour le test de transformations de modèles

Dans le chapitre 4, nous avons proposé six fonctions d'oracle pour le test de transformations de modèles. Ces fonctions diffèrent sur la nature des données d'oracle qu'elles nécessitent : des transformations de référence ou inverse, des modèles, des contraintes OCL (sous forme de contrat ou d'assertion), des assertions avec des patterns de model snippets.

Nous avons étudié différentes propriétés des fonctions pour évaluer quelle fonction permet de réaliser des oracles *réutilisables* sans *risque d'erreur*. Nous avons évalué ces deux qualités en fonction de la généricité et la complétude des fonctions, les redondances et la taille des données, le couplage entre les données et les méta-modèles. Nous avons déduit que les fonctions qui modularisent le plus leur vérification permettent de réaliser des oracles avec moins de risque d'erreur tout en permettant une plus grande réutilisabilité.

6.1.3 Qualification de composants de transformations de modèles

Dans le chapitre 5, nous avons exploité nos premières contributions pour adapter et expérimenter une méthodologie globale pour la construction de composants de transformation de modèles. Nous intégrons dans ces composants à la fois : l'implantation, les cas de test, et la spécification sous forme de contrat. Nous avons expérimenté cette méthodologie grâce à

plusieurs développements : un support des contraintes dans Kermeta, une plate-forme expérimentale supportant l'analyse de mutation pour les transformations, et un harnais de test pour l'exécution des tests qui supporte les fonctions d'oracle proposées.

Les développements réalisés sont importants pour nos recherches mais ils bénéficient également à d'autres travaux. Le principal est notre intégration du support des contrats dans Kermeta. Il sert de base au support du standard OCL dans Kermeta. Cette fonctionnalité apporte à la plate-forme Kermeta le quatrième aspect essentiel pour en faire une plate-forme complète pour l'Ingénierie Dirigée par les Modèles (comme illustrée dans la figure 5-6 : avec la méta-modélisation, la sémantique, les transformations, et les contraintes).

La plate-forme développée pour supporter l'analyse de mutation pour les transformations de modèles nous sert à exploiter cette technique directement pour le test, mais pas seulement. Elle permet également d'utiliser des contrats comme oracle, ce que nous exploitons pour les qualifier quand ils sont embarqués dans un composant de confiance. Par ailleurs, nous l'utilisons pour expérimenter d'autres techniques de test comme la génération automatique de modèles de test. Nous avons montré cet apport pour nos travaux de validation de critères de test fonctionnels dans la section 5.4.

A la fin de cette thèse, les résultats obtenus sont variés. En plus de proposer deux techniques pour le test des transformations de modèles, nous avons apporté une contribution pour l'avancement de la recherche sur les problématiques du test de transformation. L'outillage développé et les techniques que nous avons étudiées dans nos études participent concrètement aux recherches en cours dans l'équipe Triskell, et permettent d'envisager plusieurs perspectives que nous présentons dans la section suivante. Nous avons fourni dans ce travail à la fois des techniques exploitables mais également des moyens pour expérimenter et valider les travaux qui restent à faire. Nous exploitons déjà concrètement dans nos travaux l'analyse de mutation pour mettre à l'épreuve des oracles.

6.2 Perspectives

La problématique du test de transformations de modèles est vaste et comporte encore de nombreux problèmes à résoudre. Nos travaux ouvrent certaines perspectives pour traiter certains de ces problèmes.

La principale activité du test que nous n'avons pas traitée est la *localisation d'erreur*. Il s'agit de la phase du processus qui suit l'application de l'oracle et la production du verdict. L'évolution logique à nos travaux consiste à perfectionner les fonctions d'oracles proposées. Il est possible d'envisager que l'oracle ne renvoie pas uniquement le verdict mais qu'il soit capable de fournir des informations sur la défaillance. En décrivant précisément la nature de la défaillance et les éléments incorrects du modèle de sortie, la localisation de l'erreur serait facilitée. Les techniques de traçabilité [Glitia'08, Bondé'05, Amar'08] apportent un gain puisqu'elles permettent de maintenir un lien entre les éléments du modèle d'entrée et le modèle de sortie. Une évolution à ces travaux de traçabilité consiste à produire une trace des opérations

réalisées par la transformation sur les modèles. La localisation de l'erreur pourrait être faite en déterminant quelles opérations de la transformation ont :

- modifié les éléments erronés du modèle de sortie,
- consulté les éléments du modèles d'entrée correspondant aux éléments erronés du modèle de sortie (cette correspondance étant déjà fournie par les outils existants comme ETraceTool[1]).

L'exploitation des travaux de traçabilité pour le test de transformations de modèles est une première perspective qui nous intéresse.

Nous pouvons également envisager de spécialiser les techniques de test développées au test de transformations particulières. En particulier, les transformations iso-fonctionnelles introduisent une problématique intéressante. Les transformations de modèles ont parfois une propriété : l'entrée et la sortie sont sémantiquement équivalentes, les modèles d'entrée et de sortie ont le même sens et le même but. Dans ce cas, elles sont iso-fonctionnelles. Les oracles pour tester ces transformations sont complexes. Ils doivent vérifier principalement deux choses :

- que le traitement que souhaite réaliser la transformation a été correctement effectué,
- que la transformation n'a pas affecté la sémantique du modèle.

Nous avons expérimenté l'utilisation de l'oracle pour le test d'une de ces transformations. Elle « aplatit » un diagramme d'état composite. Un exemple est donné dans la figure 2-5, avec à gauche un diagramme d'état avec un état composite, et à droite une version aplatie de ce diagramme. Pour vérifier que le diagramme d'état produit n'a plus d'état composite, nous pouvons utiliser un oracle avec un contrat générique. Il est possible de vérifier que deux diagrammes correspondent mais il est plus difficile de vérifier qu'ils ne correspondent pas.

La première raison est la définition même de l'équivalence fonctionnelle de modèles en particulier parce que les spécifications sont rarement formelles. La spécification de la sémantique des diagrammes d'états d'UML par exemple laisse trop de liberté. La seconde raison est que cette transformation n'est pas déterministe et que parfois, de nombreux diagrammes d'état aplatis peuvent être générés. En effet la question de l'iso-fonctionnalité entre deux modèles est posée parce que le lien entre les modèles d'entrée et de sortie n'est pas direct et que plusieurs modèles peuvent potentiellement avoir la même sémantique. Dans notre exemple (de la figure 2-5), il est par exemple possible de produire un autre modèle de sortie qui comprend deux états finaux (un pour l'état 3 et l'autre pour l'état 4). Les model snippets permettent de définir des alternatives et de proposer plusieurs solutions simultanées pour l'oracle. Mais dans les cas où les possibilités sont trop nombreuses, toutes nos approches sont limitées. La conservation de la sémantique des modèles au cours d'une transformation est un sujet d'étude encore ouvert.

[1] http://sourceforge.net/projects/etracetool/

La dernière perspective que nous envisageons concerne la gestion du test après l'évolution des transformations. Nous avons proposé différentes fonctions d'oracle et analysé que les fonctions les plus intéressantes pour le test de transformations sont les deux utilisant des assertions OCL et des model snippets. Elles permettent en particulier de modulariser les vérifications de l'oracle. L'avantage est de pouvoir les réutiliser si la transformation évolue. En effet, la réutilisation des transformations est courante dans l'IDM, elle justifie d'ailleurs l'effort consacré au développement de ces transformations complexes. En revanche, une transformation n'est pas forcément réutilisée en l'état [Siikarla'08b, Siikarla'08a, Fleurey'07b]. Sa spécification peut varier sur le plan fonctionnel, en fonction de contraintes de plate-forme qui évoluent, ou en fonction du domaine d'application. Ainsi si le méta-modèle d'entrée ou de sortie varie, les modèles de test ne seront plus systématiquement valides. Nous avons aussi expliqué que certains oracles peuvent être rejetés parce qu'ils sont affectés par les changements d'un méta-modèle (quand leur couplage est trop important). Mais il est plus complexe d'envisager et d'automatiser la sélection d'artéfact de test quand les changements sont fonctionnels. De la même façon que pour les problèmes d'iso-fonctionnalité, il est difficile de décider quels éléments sont affectés par un changement s'il n'est pas formalisé. Des recherches sont nécessaires pour étudier la sélection automatique des modèles de test et des oracles qui sont valides après une évolution de la transformation.

Annexe A

Dans cette annexe, nous illustrons ce chapitre avec la présentation des différentes données d'oracle utilisées pour vérifier la transformation de cinq modèles de test. Sont regroupés :

- Les six modèles de test permettant d'atteindre le score de mutation de 100%. Nous les avons obtenus au chapitre 3.
- Les six modèles attendus en sortie par la transformation de ces cinq modèles de test.
- Les contrats génériques développés.
- Les assertions OCL.
- Les assertions de patterns de model snippets.
- Tous les cas de test que ces différentes données d'oracle permettent de définir.

La transformation utilisée ne permet pas d'implémenter de transformation inverse. En revanche il est possible d'utiliser une transformation de référence comme celle développée par Lawley et Steel [Lawley'05].

A.1 Six modèles de test

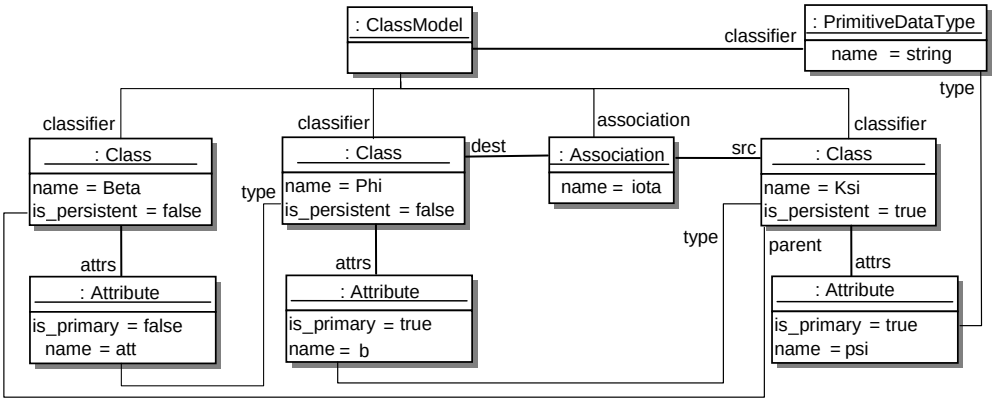


Figure 6-1 - Modèle de test MT7

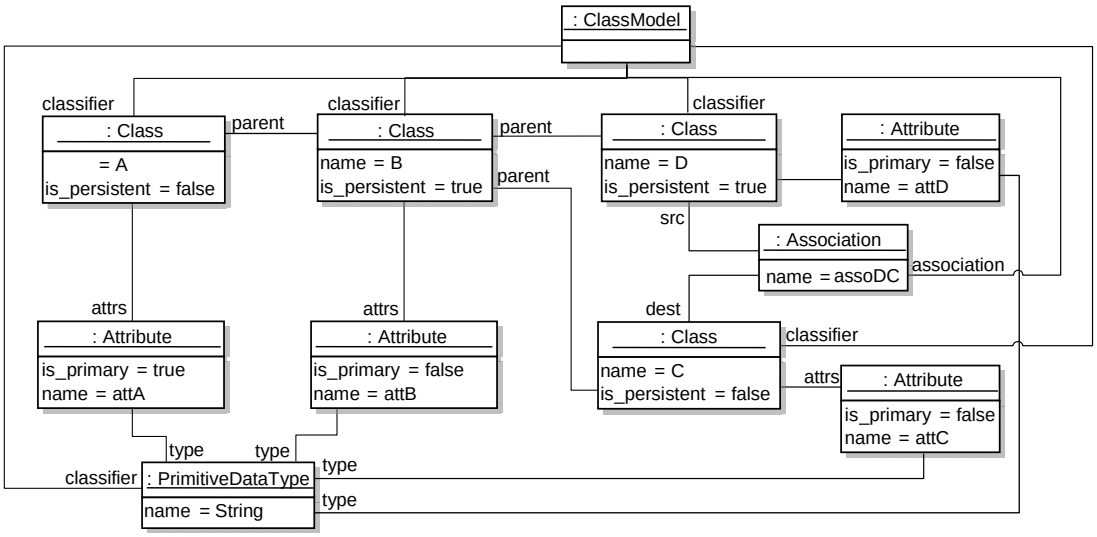


Figure 6-2 - Modèle de test MT8

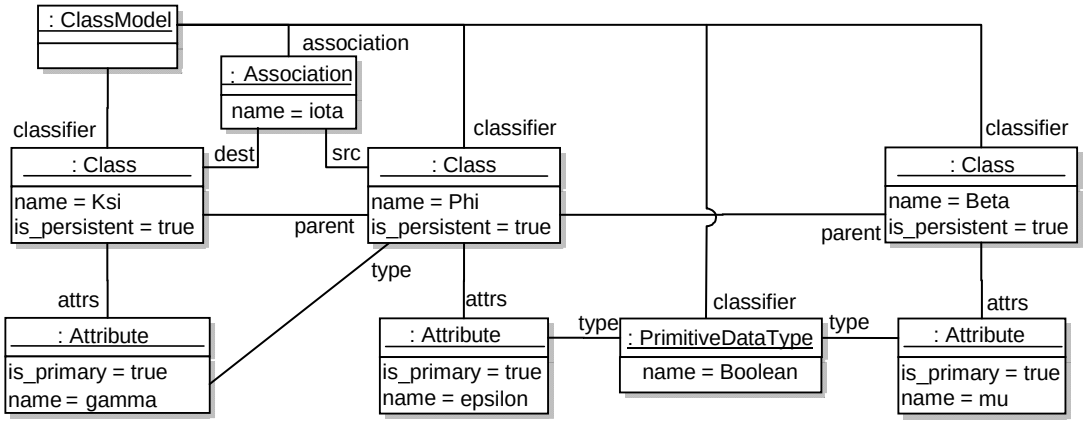


Figure 6-3 - Modèle de test MT9

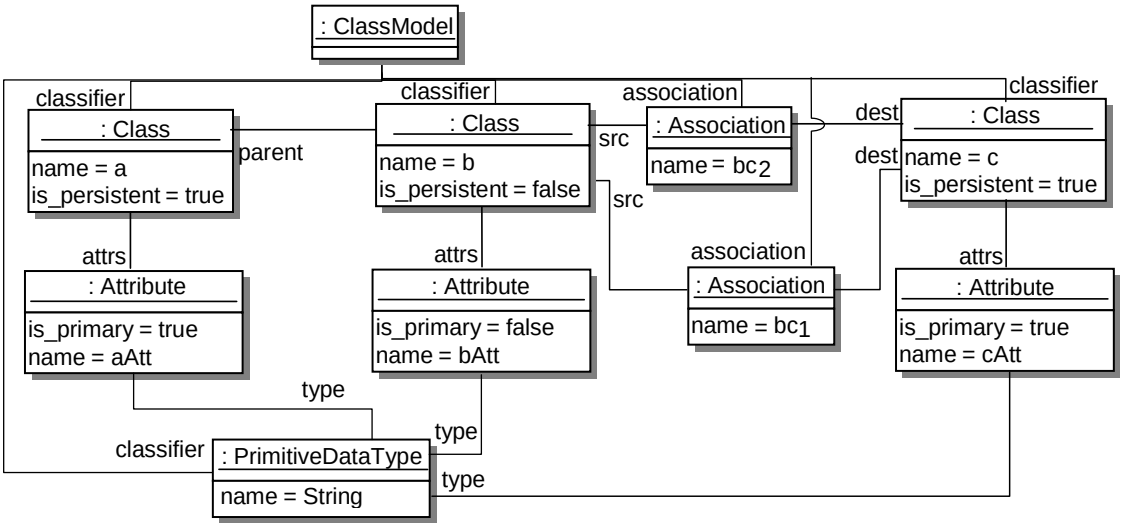


Figure 6-4 - Modèle de test MT14

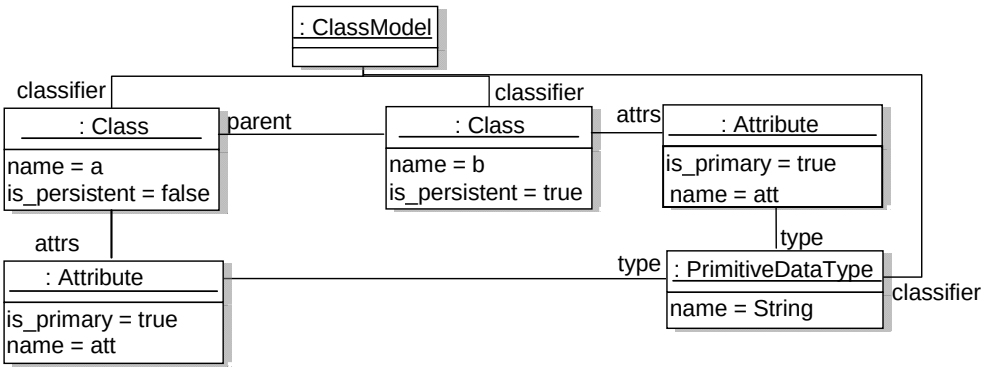


Figure 6-5 - Modèle de test MT15

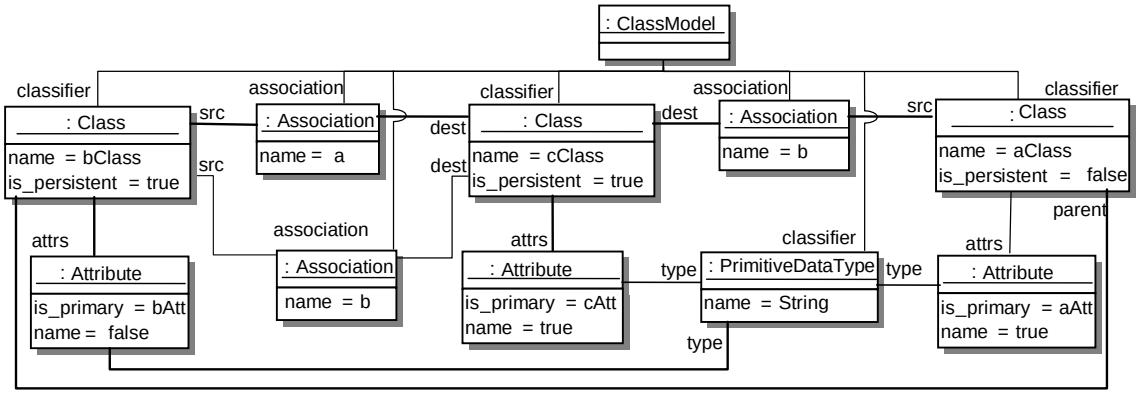


Figure 6-6 - Modèle de test MT16

A.2 Six modèles attendus

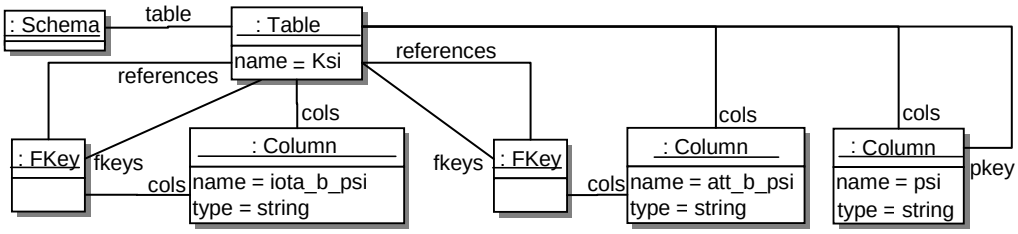


Figure 6-7 - Modèle attendu MA7

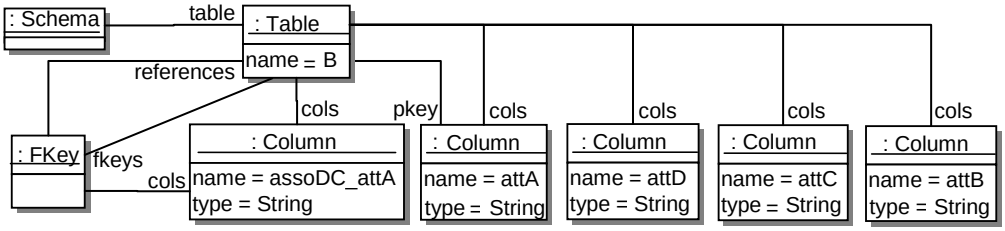


Figure 6-8 - Modèle attendu MA8

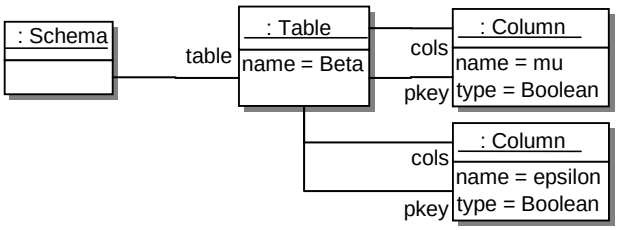


Figure 6-9 - Modèle attendu MA9

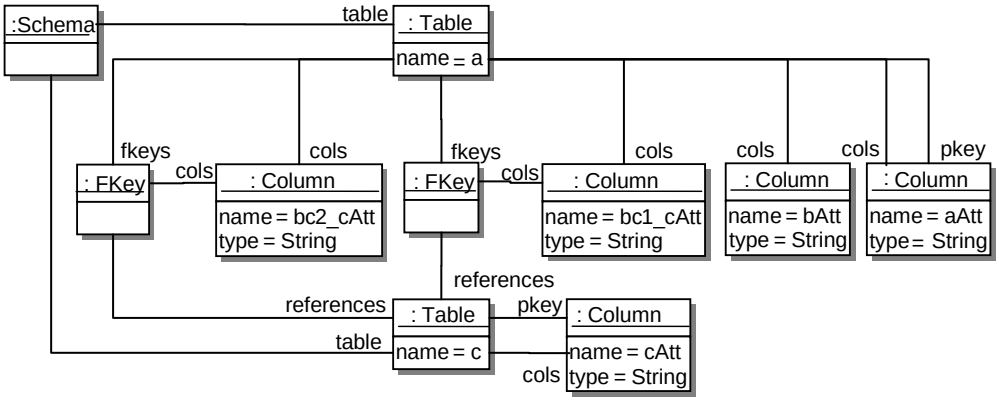


Figure 6-10 - Modèle attendu MA14

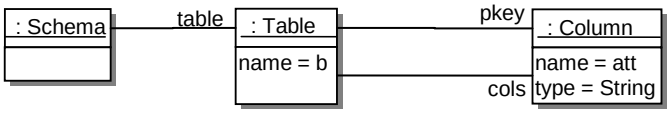


Figure 6-11 - Modèle attendu MA15

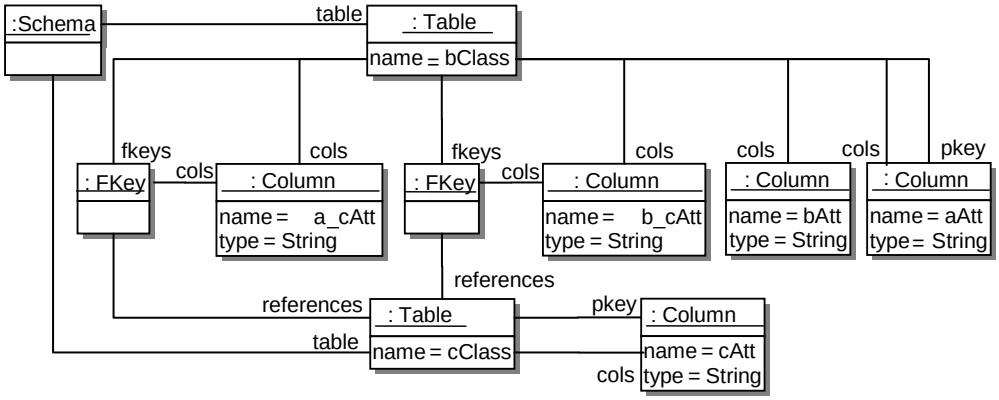


Figure 6-12 - Modèle attendu MA16

Ces six modèles attendus permettent de construire six oracles pour former six cas de test :

CTma1 : (MT7, o3, MA7)

CTma2 : (MT8, o3, MA8)

CTma3 : (MT9, o3, MA9)

CTma4 : (MT14, o3, MA10)

CTma5 : (MT15, o3, MA11)

CTma6 : (MT16, o3, MA12)

A.3 Contrats

Nous avons écrit des contrats de différents niveaux en Kermeta. Ici sont regroupés les contrats de niveau 2 à 5 :

Niveau 2 :

```
post pkey_is_in_the_column_of_its_table is
     result.table.forAll{t|t.pkey.forAll{pk|pk.container == t}}


post columns_of_fkey_are_among_the_ones_of_its_table is
result.table.forAll{t|t.fkeys.forAll{fk|fk.cols.forAll{c|t.cols.contains(c)}}}


post table_number_equal_persistent_class_number is
     // partie des règles 1 et 2
     // Il y a autant de table que de classes persistante moins le nombre de
     // classes persistante ayant un parent persistant
result.table.size ==
   inputModel.classifier
   .select{cr|Class.isInstance(cr)}
   .select{cs | cs.asType(Class).is_persistent}
   .select{csp | not csp.asType(Class).allParents.exists{p | p.is_persistent}}
   .size


post table_name_correspond_class_name is
     // partie des règles 1 et 2
     // Il y a au moins une classe persistante de même nom que chaque table,
   result.table.forAll{ t |  inputModel.classifier
                           .select{cr | Class.isInstance(cr)}
                           .select{cs | cs.asType(Class).is_persistent}
                           .exists{cs | cs.name == t.name}}


post class_name_correspond_table_name is
     // partie des règles 1 et 2
     // Il y a au moins une table de même nom que chaque classe persistante
     // sans parent persistant,
   inputModel.classifier
   .select{cr| Class.isInstance(cr)}
   .select{cs | cs.asType(Class).is_persistent}
   .select{csp | not csp.asType(Class).allParents.exists{p | p.is_persistent}}
   .forAll{csp | result.table.exists{t |t.name == csp.name}}
```

```
post attribute_primitive_type is
     // Création des colonnes correspondant aux attributs de type primitif
inputModel.classifier
.select{cr| Class.isInstance(cr)}
.select{cs | cs.asType(Class).is_persistent}
.select{csp | not csp.asType(Class).allParents.exists{p | p.is_persistent}}
.forAll
  {csp | //pour toute classe persistante sans parent persistant
     result.table.select{t|t.name == csp.name} //dans la table correspondante,
     .exists
       {tn | csp.asType(Class).attrs
               .select{at | PrimitiveDataType.isInstance(at.type)}
               // les attributs de type primitif
               .forAll
                 {atp|tn.cols
                    .select
                       {ctn|ctn.name==atp.name and ctn.type == atp.type.name
                       }.size==1
                 }// ont 1 et 1 seule colonne correspondante dans la table
       }
  }
```

```
post attribute_persistent_classes is
     // Création des colonnes correspondant aux attributs dont le type est une
     // classe persistante transformée en table
inputModel.classifier
.select{cr| Class.isInstance(cr)}
.select{cs | cs.asType(Class).is_persistent}
.select{csp | not csp.asType(Class).allParents.exists{p | p.is_persistent}}
.forAll
  {csp | // pour toute classe persistante sans parent persistant,
     result.table.select{t|t.name == csp.name} //dans la table correspondante,
     .exists
       {tn |csp.asType(Class).attrs
             .select{at | Class.isInstance(at.type)}
               // les attributs dont le type est une classe
             .select{atc|atc.type.asType(Class).is_persistent}
               //qui est persistante
             .select{atcp |  not atcp.type.asType(Class).allParents
                             .exists{p | p.is_persistent}}
                             // mais sans parent persistant
             .forAll
               {atcpwp|//(atcpwp est un attribut don't le type est une classe
                       //persistante sans parent persistant
                  atcpwp.type.asType(Class).attrs
                  .forAll
                     {atcpta|   //pour tous les attributs de ce type
                        if   PrimitiveDataType.isInstance(atcpta.type)
                                //qui sont de type primitif
                             and atcpta.is_primary // et qui sont primaires
                        then
                           tn.cols
                           .select
                              {ctn |
                                 ctn.name == atcpwp.name+"_"+atcpta.name
                                    // alors il y a une colonne du nom de cette
                                    // attribut + « _ » + le nom de l'attribut
                                    // primaire de type primitif
                                 and ctn.type == atcpta.type.name
                                    // et du type de ce dernier attribut,
                              }.size==1
                                 //cette colonne est l'unique correpondante
                        else true
                        end
                     }
               }
       }
  }
```

```
post association_persistent_classes is
   inputModel.classifier
   .select{cr| Class.isInstance(cr)}
   .select{cs | cs.asType(Class).is_persistent}
   .select{csp | not csp.asType(Class).allParents.exists{p | p.is_persistent}}
   .forAll
      {csp | // pour toute classe persistante sans parent persistant,
         result.table
         .select{t | t.name == csp.name} //dans la table correspondante,
         .exists
            {tn |inputModel.association.select{ass|ass.src == csp}
                  .select{atc | atc.dest.asType(Class).is_persistent}
                     // qui est persistante
                  .select{atcp |  not atcp.dest.asType(Class).allParents
                                  .exists{p | p.is_persistent}}
                                  // mais sans parent persistant
                  .forAll
                     {atcpwp |  // (atcpwp is une association ciblant une classe
                                // persistante sans parent persistant)
                        atcpwp.dest.attrs
                        .forAll //pour tous les attributs de cette classe cible
                           {atcpta|
                              if   PrimitiveDataType.isInstance(atcpta.type)
                                      //qui sont de type primitif
                                   and atcpta.is_primary // et qui sont primaires
                              then
                                 tn.cols
                                 .select
                                    {ctn |
                                       ctn.name == atcpwp.name+"_"+atcpta.name
                                          // alors il y a une colonne du nom de
                                          // cette association + « _ » + le nom de
                                          // l'attribut primaire de type primitif
                                       and ctn.type == atcpta.type.name
                                          // et du type de ce dernier attribut
                                    }.size==1
                                       //cette colonne est l'unique correpondante
                              else true
                              end
                           }
                     }
            }
      }
```

Niveau 3 :

```
post attribute_primitive_type_and_primary_key is
     // Ce contrat améliore le contrat « attribute_primitive_type »
     // Il considère la création des clés primaires (pkey)
inputModel.classifier
.select{cr| Class.isInstance(cr)}
.select{cs | cs.asType(Class).is_persistent}
.select{csp | not csp.asType(Class).allParents.exists{p | p.is_persistent}}
.forAll
  {csp | //pour toute classe persistante sans parent persistant
    result.table.select{t| t.name == csp.name}
      //dans la table correspondante,
    .exists
      {tn | csp.asType(Class).attrs
              .select{at | PrimitiveDataType.isInstance(at.type)}
              // les attributs de type primitif
              .forAll
                {atp|
                  tn.cols
                    .select
                    {ctn|ctn.name==atp.name and ctn.type == atp.type.name
                    }.size==1
                    //ont 1 et 1 seule col. correspondante dans la table
                  and
                    if atp.is_primary
                    then // si cette attribut est primaire
                      tn.pkey
                      .contains
                        (tn.cols.select{ctn|ctn.name==atp.name
                                        and ctn.type == atp.type.name
                                        }.one
                          //la table doit avoir cette colonne comme pkey
                        )
                  else true
                  end
                }
      }
  }
```

```
post attribute_persistent_classes_considering_fkey is
   // Ce contrat améliore le contrat attribute_persistent_classes
   // Ce contrat vérifie les colonnes d'une clé étrangère venant d'attributs
inputModel.classifier
.select{cr| Class.isInstance(cr)}
.select{cs | cs.asType(Class).is_persistent}
.select{csp | not csp.asType(Class).allParents.exists{p | p.is_persistent}}
.forAll
   {csp | // pour toute classe persistante sans parent persistant,
      result.table.select{t|t.name == csp.name} //dans la table correspondante,
      .exists
         {tn |csp.asType(Class).attrs
               .select{at | Class.isInstance(at.type)}
                 // les attributs dont le type est une classe
               .select{atc|atc.type.asType(Class).is_persistent}
                 //qui est persistante
               .select{atcp |  not atcp.type.asType(Class).allParents
                               .exists{p | p.is_persistent}}
                               // mais sans parent persistant
               .forAll
                 {atcpwp| //(atcpwp est un attribut dont le type est une classe
                          // persistante sans parent persistant)
                    atcpwp.type.asType(Class).attrs
                    .forAll
                      {atcpta|  //pour tous les attributs de ce type
                         if   PrimitiveDataType.isInstance(atcpta.type)
                                //qui sont de type primitif
                              and atcpta.is_primary // et qui sont primaires
                         then
                           tn.cols.select
                              {ctn |
                                ctn.name == atcpwp.name+"_"+atcpta.name
                                  // alors il y a une colonne du nom de cette
                                  // attribut + « _ » + le nom de l'attribut
                                  // primaire de type primitif
                                and ctn.type == atcpta.type.name
                                  // et du type de ce dernier attribut,
                                and tn.fkeys.exists{ftn|ftn.cols.contains(ctn)}
                                  //et la colonne est fait partie des colonnes
                                  // d'une clé étrangère (FKey) de la table
                              }.size == 1
                                //cette colonne est l'unique correspondante
                         else true  end
                      }
                 }
         }
   }
```

```
post association_persistent_classes_considering_fkeys is do
//Ce contrat améliore le contrat attribute_persistent_classes
//Ce contrat vérifie les colonnes d'une clé étrangère venant d'associations
inputModel.classifier
.select{cr| Class.isInstance(cr)}.select{cs | cs.asType(Class).is_persistent}
.select{csp | not csp.asType(Class).allParents.exists{p | p.is_persistent}}
.forAll
  {csp |  // pour toute classe persistante sans parent persistant,
    result.table.select{t | t.name == csp.name}
         //dans la table correspondante,
    .exists
      {tn |inputModel.association.select{ass|ass.src == csp}
              // les associations de cette classe
            .select{atc | atc.dest.asType(Class).is_persistent}
              // persistante
            .select{atcp |  not atcp.dest.asType(Class).allParents
                    .exists{p|p.is_persistent}}
              // mais sans parent persistant
            .forAll         // (atcpwp est une association ciblant une classe
              {atcpwp |    // persistante sans parent persistant)
                atcpwp.dest.attrs.forAll
                  {atcpta|//pour tous les attributs de cette classe cible
                    if   PrimitiveDataType.isInstance(atcpta.type)
                            //qui sont de type primitif
                         and atcpta.is_primary // et qui sont primaires
                    then
                      tn.cols.select{ctn |
                            ctn.name == atcpwp.name+"_"+atcpta.name
                              // alors il y a une colonne du nom de
                              // cette association + « _ » + le nom de
                              // l'attribut primaire de type primitif
                            and ctn.type == atcpta.type.name
                              // et du type de ce dernier attribut
                            and tn.fkeys.exists
                                 {fk|fk.references ==
                                      result.table
                                      .select{tr|tr.name==atcp.dest.name}
                                      .one
                                 }  //et la table a une clé étrangère qui
                            //référence la table correspondant à la
                            //classe persistante ciblée par l'association
                         }.size==1
                  else true  end
                }
            }
      }
  }
```

Niveau 4 :

```
post   association_persistent_classes_considering_fkeys_and_parent_child_assocs
is // Ce contrat améliore association_persistent_classes_considering_fkeys
   // Il considère les associations des classes de l'arbre d'héritage
inputModel.classifier
.select{cr| Class.isInstance(cr)}.select{cs | cs.asType(Class).is_persistent}
.select{csp | not csp.asType(Class).allParents.exists{p | p.is_persistent}}
.forAll
   {csp | // pour toute classe persistante sans parent persistant,
      result.table.select{t|t.name == csp.name} //dans la table correspondante,
      .exists{tn | inputModel.association.select
         {ass|ass.src==csp.asType(Class) // les associations de cette classe
            or ass.src.allParents.contains(csp.asType(Class))//ou de ses filles
            or csp.asType(Class).allParents.contains(ass.src)
                                                          //ou de ses parents
         }.select{atc | atc.dest.asType(Class).is_persistent} // persistante
            .select{atcp |  not atcp.dest.asType(Class).allParents
                    .exists{p|p.is_persistent}} // mais sans parent persistant
            .forAll{atcpwp |   // (atcpwp est une association ciblant une classe
                               // persistante sans parent persistant)
                 atcpwp.dest.attrs.forAll
                    {atcpta| //pour tous les attributs de cette classe cible
                       if   PrimitiveDataType.isInstance(atcpta.type)
                               //qui sont de type primitif
                            and atcpta.is_primary // et qui sont primaires
                       then tn.cols.select
                            {ctn |
                               ctn.name == atcpwp.name+"_"+atcpta.name
                                  // alors il y a une colonne du nom de
                                  // cette association + « _ » + le nom de
                                  // l'attribut primaire de type primitif
                               and ctn.type == atcpta.type.name
                                  // et du type de ce dernier attribut
                               and tn.fkeys.exists
                                     {fk|fk.references == result.table
                                        .select{tr|tr.name==atcp.dest.name}.one
                                     }  // et la table a une clé étrangère qui
                                        // référence la table correspondant à la
                                        // classe persistante ciblée par
                                        // l'association
                            }.size==1
                       else true  end
                 }
            }
      }
   }
```

Niveau 5 :

```
post attribute_primitive_type_and_child_attributes is
   // Ce contrat améliore le contrat attribute_primitive_type
   // Il vérifie les colonnes d'attributs primitifs en considérant l'héritage
inputModel.classifier
.select{cr| Class.isInstance(cr)}
.select{cs | cs.asType(Class).is_persistent}
.select{csp | not csp.asType(Class).allParents.exists{p | p.is_persistent}}
.forAll
   {csp | // pour toute classe persistante sans parent persistant,
      result.table
      .select{t | t.name == csp.name} //dans la table correspondante,
      .exists
        {tn|var list_att: OrderedSet<Attribute> init OrderedSet<Attribute>.new
              list_att.addAll(csp.asType(Class).attrs)
              inputModel.classifier
                .select{cr|Class.isInstance(cr)}
                .select {crc|crc.asType(Class)
                           .allParents.contains(csp.asType(Class))}
                     //pour tout enfant de la classe
                .collect{cc|cc.asType(Class).attrs} //collecter ses attributs
                .each{atts|list_att.addAll(atts)}
                  //et ajouter les à sa liste d'attribut
              list_att.select{at | PrimitiveDataType.isInstance(at.type)}
                .forAll // pour tout attribut de type primitif
                  {atp|
                     tn.cols
                     .select {ctn|ctn.name==atp.name and
                                  ctn.type == atp.type.name
                           }.size==1
                  } il y a une seule colonne correspondante dans la table
        }
   }
```

```
post attribute_persistent_classes_considering_fkey_and_child_attributes is
// Ce contrat améliore attribute_persistent_classes_considering_fkey
// Il vérifie les colonnes d'attributs non primitifs en considérant l'héritage
inputModel.classifier.select{cr| Class.isInstance(cr)}
.select{cs | cs.asType(Class).is_persistent}
.select{csp | not csp.asType(Class).allParents.exists{p | p.is_persistent}}
.forAll {csp | // pour toute classe persistante sans parent persistant,
   result.table.select{t | t.name == csp.name} //dans la table correspondante,
   .exists
      {tn| var list_att : OrderedSet<Attribute> init OrderedSet<Attribute>.new
            list_att.addAll(csp.asType(Class).attrs)
            inputModel.classifier.select{cr|Class.isInstance(cr)}
               .select
                  {crc|crc.asType(Class).allParents.contains(csp.asType(Class))}
                  //pour tout enfant de cette classe
               .collect{cc|cc.asType(Class).attrs}//collecter ses attributs
               .each{atts|list_att.addAll(atts)}
                  //et les ajouter à sa liste d'attribut
            list_att.select{at | Class.isInstance(at.type)}
                  // ces attributs dont le type est une classe
               .select{atc| atc.type.asType(Class).is_persistent} // persistante
               .select{atcp|not atcp.type.asType(Class).allParents
                              .exists{p|p.is_persistent}}//sans parent persistant
               .forAll{atcpwp |  // (atcpwp est un attribut dont le type est
                                 //une classe persistante sans parent persistant
                     atcpwp.type.asType(Class).attrs
                     .forAll {atcpta | //pour tous les attributs de ce type
                           if PrimitiveDataType.isInstance(atcpta.type)
                                 //si c'est un attribut d'un type primitif
                              and atcpta.is_primary   // et qui est primaire
                           then tn.cols.select
                                 {ctn | ctn.name == atcpwp.name+"_"+atcpta.name
                                       // alors il y a une colonne du nom de cette
                                       // attribut + « _ » + le nom de l'attribut
                                       // primaire de type primitif
                                    and ctn.type == atcpta.type.name
                                       // et du type de ce dernier attribut,
                                    and tn.fkeys.exists{ftn|ftn.cols.contains(ctn)}
                                       //et la colonne est fait partie des colonnes
                                       // d'une clé étrangère (FKey) de la table
                                 }.size == 1
                                       //cette colonne est l'unique correspondante
                           else true end
                        }
                  }
      }
}
```

A partir de ces contrats, nous obtenons quatre-vingt-quatre cas de test (6 modèles fois quatorze contrats) :

CTcg1 : (MT7, o4, pkey_is_in_the_column_of_its_table)

CTcg2 : (MT8, o4, pkey_is_in_the_column_of_its_table)

CTcg3 : (MT9, o4, pkey_is_in_the_column_of_its_table)

CTcg4 : (MT14, o4, pkey_is_in_the_column_of_its_table)

CTcg5 : (MT15, o4, pkey_is_in_the_column_of_its_table)

CTcg6 : (MT16, o4, pkey_is_in_the_column_of_its_table)

CTcg7 : (MT7, o4, columns_of_fkey_are_among_the_ones_of_its_table)

CTcg8 : (MT8, o4, columns_of_fkey_are_among_the_ones_of_its_table)

CTcg9 : (MT9, o4, columns_of_fkey_are_among_the_ones_of_its_table)

CTcg10 : (MT14, o4, columns_of_fkey_are_among_the_ones_of_its_table)

CTcg11 : (MT15, o4, columns_of_fkey_are_among_the_ones_of_its_table)

CTcg12 : (MT16, o4, columns_of_fkey_are_among_the_ones_of_its_table)

CTcg13 : (MT7, o4, table_number_equal_persistent_class_number)

CTcg14 : (MT8, o4, table_number_equal_persistent_class_number)

CTcg15 : (MT9, o4, table_number_equal_persistent_class_number)

CTcg16 : (MT14, o4, table_number_equal_persistent_class_number)

CTcg17 : (MT15, o4, table_number_equal_persistent_class_number)

CTcg18 : (MT16, o4, table_number_equal_persistent_class_number)

CTcg19 : (MT7, o4, table_name_correspond_class_nam)

CTcg20 : (MT8, o4, table_name_correspond_class_nam)

CTcg21 : (MT9, o4, table_name_correspond_class_nam)

CTcg22 : (MT14, o4, table_name_correspond_class_nam)

CTcg23 : (MT15, o4, table_name_correspond_class_nam)

CTcg24 : (MT16, o4, table_name_correspond_class_nam)

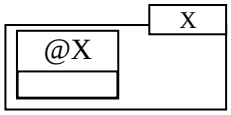
, etc.

A.4 Assertions de patterns de model snippets

En utilisant des assertions de patterns de model snippets, chaque cas de test peut être dédié à une vérification précise. Nous associons à chaque cas de test une intention. Puis nous définissons un oracle qui répond à un objectif précis. Chaque oracle est mis en œuvre avec une assertion et un pattern. Ces derniers pouvant être réutilisés dans différents patterns et donc différents oracles de différents cas de test.

Intention a : Chaque classe persistante sans parent est transformée en une table de même nom.

Objectif a1 : Le modèle MS7 doit contenir une table nommée "Ksi" :

Pab1 : 

(le model snippet est une seule table sans nom)

```
APa1 : patternmatching(Pab1, Mout).contains(x|x.name = “Ksi”)
```

CTms1 : (MT7 , o6 , APa1)

Intention b : Chaque classe persistante sans parent persistant est transformée en table de même nom.

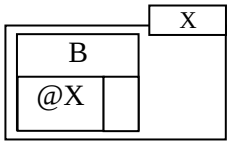
Objectif b1 : Le modèle MS8 doit contenir une table nommée "B" mais pas de table nommée "D"

```
APb1 :  patternmatching(Pab1, Mout).contains(x|x.name = “B”) and
```

CTms2 : (MT8 , o6 , APb1)

Intention c : Quand une classe est transformée en table, ses propres attributs de type primitif sont changés en colonnes de même nom.

Objectif c1 : Le modèle MS8 doit contenir une table nommée "B" dont une colonne s’appelle "attB"



(le model snippet est une seule table nommée B avec une colonne sans nom précisé)

Pcdei1 :

```
APc1 :  patternmatching(Pcdei1, Mout).contains(x|x.name= “attB”)
```

CTms3 : (MT8 , o6 , APc1)

Intention d : Quand une classe est transformée en table, ses attributs hérités qui sont de types primitifs doivent être changés en colonnes de même nom.

Objectif d1 : Le modèle MS8 doit contenir une table nommée "B" dont une colonne s'appelle "attA"

```
APd1 :  patternmatching(Pcdei1, Mout).contains(x|x.name= “attA”)
```

CTms4 : (MT8 , o6 , APd1)

Intention e : Quand une classe est transformée en table, les attributs de ses fils de types primitifs sont changés en colonnes de même nom.

Objectif e1 : Le modèle MS8 contient une table nommée "B" dont une colonne s'appelle "attC"

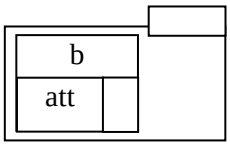
```
APe1 :  patternmatching(Pcdei1, Mout).contains(x|x.name= “attC”)
```

CTms5 : (MT8 , o6 , APe1)

Intention f : Quand une classe est transformée en table, si elle hérite d'un attribut qu'elle redéfinit alors un seul de ces deux attributs peut être changé en colonne.

Objectif f1 : le modèle MS15 contient une table b avec une seule colonne nommée att

Pf1 :



(le model snippet est une table nommée b avec une colonne nommée att)

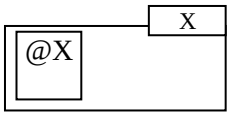
```
APf1 :  patternmatching(Pf1, Mout).size = 1
```

CTms6 : (MT15 , o6 , APf1)

Intention g : Quand une classe est transformée en table, elle ne peut pas avoir de colonne créée avec le nom d'un attribut qui n'était pas de type primitif.

Objectif g1 : Le modèle MS7 ne doit pas contenir de colonne nommée "att"

Pgn1 :



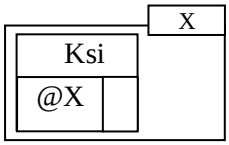
(le model snippet est une seule colonne sans nom précisé)

```
APg1 :  not patternmatching(Pgn1, Mout).contains(x|x.name= “att”)
```

CTms7 : (MT7 , o6 , APg1)

Intention h : Quand une classe est transformée en table, si elle a un attribut x dont le type est une classe non persistante qui a un attribut z dont le type est une classe persistante avec un attribut z de type primitif, alors une colonne est créée et nommée x_y_z.

Objectif h1 : Le modèle MS7 contient une colonne nommée "att_b_psi" dans une table nommée "Ksi"

Phj1 : 

(le model snippet est une seule table nommée Ksi avec une colonne sans nom précisé)

```
APh1 :  patternmatching(Phj1, Mout).contains(x|x.name= “att_b_psi”)
```

CTms8 : (MT7 , o6 , APh1)

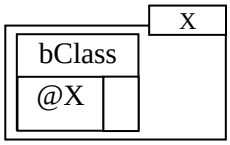
Intention j : Quand une classe est transformée en table, quand une de ses associations x pointe vers une classe non persistante ayant un attribut y dont le type est une classe persistante sans parent persistant et ayant un attribut primaire z de type primitif, alors une colonne correspondante est créée avec le nom x_y_z.

Objectif j1 : Le modèle MS7 contient une colonne nommée " iota_b_psi "

```
APj1 :  patternmatching(Phj1, Mout).contains(x|x.name= “iota_b_psi”)
```

CTms9 : (MT7 , o6 , APj1)

Intention m : Quand une classe est transformée en table, quand une de ces association x pointe vers une classe persistante avec un attribut primaire y

Pm1 : 

(le model snippet est une seule table nommée bClass avec une colonne sans nom précisé)

Objectif m1 : Le modèle MS16 doit contenir une colonne nommée "a_cAtt" dans une table nommée "bClass"

```
APm1 :  patternmatching(Pm1, Mout).contains(x|x.name= “a_cAtt”)
```

CTms10 : (MT16 , o6 , APm1)

Intention n : Quand une classe est transformée en table, quand un de ces attributs est typé avec une classe persistante qui a un parent persistant, alors il n’y a pas de colonne correspondante créée.

Objectif n1 : Le modèle MS9 ne doit pas contenir une colonne nommée "gamma_epsilon"

```
APn1 :  not patternmatching(Pgn1, Mout).contains(x|x.name= “gamma_epsilon”)
```

CTms11 : (MT9 , o6 , APn1)

Intention n’ : Quand une classe est transformée en table, quand une de ces associations pointe vers une classe persistante dont un attribut est typé avec une classe persistante qui a un parent persistant, alors il n’y a pas de colonne correspondante créée.

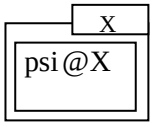
Objectif n’1 : Le modèle MS9 ne doit pas contenir une colonne nommée "iota_gamma_epsilon"

```
APn'1 :   not patternmatching(Pgn1, Mout)
                          .contains(x|x.name="iota_gamma_epsilon")
```

CTms12 : (MT9 , o6 , APan'1)

Intention o : Quand un attribut de type primitif est changé en colonne, alors le type de la colonne est le type de cet attribut.

Objectif o1 : Dans le modèle MS7, la colonne nommée "psi" doit être de type "string"

Po1 : 

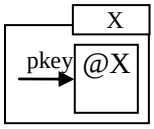
(le model snippet est une seule colonne nommée "psi" sans type précisé)

```
APo1 :  patternmatching(Po1, Mout).contains(x|x.name= "string")
```

CTms13 : (MT7 , o6 , APo1)

Intention p : Quand une classe est changée en table, quand un de ces attributs à la fois primaire et de type primitif est transformé en colonne alors il devient une pkey de la table.

Objectif p1 : Dans le modèle MS8, il n'y a qu'une pkey, elle se nomme "attA"

Pqp1 : 

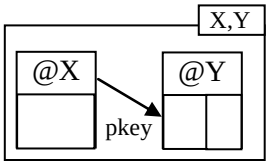
(le model snippet est une seule colonne qui est une pkey sans nom précisé)

```
APp1 :  patternmatching(Pqp1, Mout).size = 1 and
        patternmatching(Pqp1, Mout).contains(x|x.name= "attA")
```

CTms14 : (MT8 , o6 , APp1)

Intention r : Les clés primaires pkey d'une table pointent uniquement vers les colonnes de cette table.

Objectif r1 : Dans le modèle MS14, il n'y a pas de pkey d'une table pointant vers une colonne de d'une table différente

Pr1 : 

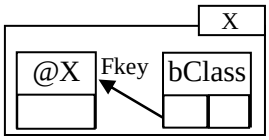
(le model snippet est une table sans nom précisé qui a une pkey pointant vers une colonne d'une table sans nom précisé)

```
APr1 :  not patternmatching(Pr1, Mout).contains((x,y)|not x.name=y.name)
```

CTms15 : (MT14 , o6 , APr1)

Intention s : Quand une classe est transformée en table, quand une de ses associations pointe vers une classe persistante (ou héritant d'une classe persistante) qui a été transformée en table alors une clé étrangère est créée entre les deux tables.

Ps1 :



(le model snippet est une table nommée "bClass" qui a une Fkey associée à une de ses colonnes et qui pointe vers une table sans nom précisé)

Objectif s1 : Dans le modèle MS16, la table nommée "bClass" possède deux Fkeys vers la table "cClass"

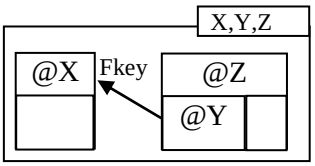
```
APs1 :  patternmatching(Ps1, Mout).size = 2 and
           patternmatching(Ps1, Mout).contains(x | x.name = “cClass”)
```

CTms16 : (MT16 , o6 , APs1)

<u>Intention t :</u> Quand une classe est transformée en table, quand une colonne est créée à partir d’une de ses associations pointant vers une classe qui a un attribut typé avec une classe persistante transformée en table alors une clé étrangère est créée entre les deux tables et est associée à cette colonne.

Objectif t1 : Dans le modèle MS7, la table nommée "Ksi" possède une Fkey associée à sa colonne nommée "iota_b_psi" vers la table nommée "Ksi"

Ptuv1 :



(le model snippet est une table sans nom précisé qui a une Fkey associé à une de ses colonnes sans nom précisé et qui pointe vers une table sans nom précisé)

```
APt1 :  patternmatching(Ptuv1, Mout).contains(x | x.name = “Ksi” and
        y.name = “iota_b_psi” and z.name = “Ksi”)
```

CTms17 : (MT7 , o6 , APt1)

<u>Intention u :</u> Quand une classe est transformée en table, quand une colonne est créée à partir d’un de ses attributs dont le type vient de la navigation vers une table persistante alors une clé étrangère est créée entre les deux tables et est associée à cette colonne.

Objectif u1 : Dans le modèle MS7, la table nommée "Ksi" possède une Fkey associée à sa colonne nommée "att_b_psi" vers la table nommée "Ksi"

```
APu1 :  patternmatching(Ptuv1, Mout).contains(x | x.name = “Ksi” and
        y.name = “att_b_psi” and z.name = “Ksi”)
```

CTms18 : (MT7 , o6 , APu1)

<u>Intention v :</u> Quand une classe est transformée en table, quand une colonne est créée à partir d’une de ses associations pointant vers une classe persistante ou ayant un parent persistant transformée en table alors une clé étrangère est créée entre les deux tables et est associée à cette colonne.

Objectif v1 : Dans le modèle MS8, la table nommée "B" possède une clé étrangère associée à sa colonne nommée "assoDC_attA" vers la table nommée "B"

```
APv1 :  patternmatching(Ptuv1, Mout).contains(x | x.name = “B” and
        y.name = “assoDC_attA” and z.name = “B”)
```

CTms19 : (MT7 , o6 , APav1)

Dans le tableau 6-1, nous listons les différents cas de test en fonction du modèle de test et du pattern qu’ils utilisent. De cette manière, nous observons que les patterns et également les modèles de test sont utilisés dans différents cas de test. Nous précisons que dans certains cas, plusieurs cas de test auraient pu satisfaire une intention. Nous avons considéré les modèles de test dans l’ordre pour construire un oracle correspondant. C’est pourquoi les premiers modèles de test sont utilisés dans davantage de cas de test. Nous remarquons que les derniers cas de test sont utilisés dans certains cas de test. Cela montre qu’ils étaient indispensables pour satisfaire certaines intentions. Cette observation participe à justifier notre adaptation de l’analyse de mutation aux transformations de modèles.

	Modèle de test 7	Modèle de test 8	Modèle de test 9	Modèle de test 14	Modèle de test 15	Modèle de test 16
Pab1	CTms1	CTms2				
Pcdei1		CTms3, CTms4, CTms5				
Pf1					CTms6	
Pgn1	CTms7		CTms11, CTms12			
Phj1	CTms8, CTms9					
Pm1						CTms10
Po1	CTms13					
Pqp1		CTms14				
Pr1				CTms15		
Ps1						CTms16
Ptuv1	CTms17, CTms18	CTms19				

Tableau 6-1 - Répartition des cas de test en fonction du modèle de test et du pattern utilisés

A.5 Assertions OCL

Avec des assertions OCL, il est possible d'écrire les mêmes patterns qu'avec des assertions de patterns de model snippets. Ainsi, nous obtenons (nous ne rappelons pas les intentions et les objectifs qui sont les mêmes que dans la précédente section) :

```
AOCLa : Mout.table.exist(t|t.name = Ksi)
CTaocl1 : (MT7 , o5 , AOCLa )

AOCLb : Mout.table.exist(t|t.name = B)
CTaocl2 : (MT8 , o5 , AOCLb  )

AOCLc : Mout.table.select(t|t.name=B).collect(t|t.cols).exist(c|c.name=attD)
CTaocl3 : (MT8 , o5 , AOCLc )

AOCLd : Mout.table.select(t|t.name=B).collect(t|t.cols).exist(c|c.name=attA)
CTaocl4 : (MT8 , o5 , AOCLd  )

AOCLe : Mout.table.select(t|t.name=B).collect(t|t.cols).exist(c|c.name=attC)
CTaocl5 : (MT8 , o5 , AOCLe  )

AOCLf : Mout.table.select(t|t.name = b and
                            t.cols.select(c|c.name=att).size()=1)
                      .size() = 1
CTaocl6 : (MT15 , o5 , AOCLf  )

AOCLg : Mout.table.collect(t | t.cols).select(c | c.name = att).size() = 0
CTaocl7 : (MT7 , o5 , AOCLg  )

AOCLh : Mout.table.collect(t | t.cols).exist(c | c.name = att_b_psi)
CTaocl8 : (MT7 , o5 , AOCLh  )

AOCLj : Mout.table.collect(t | t.cols).exist(c | c.name = iota_b_psi)
CTaocl9 : (MT7 , o5 , AOCLj )

AOCLm :
   Mout.table.select(t|t.name=bClass).collect(t|t.cols).exist(c|c.name=a_cAtt)
CTaocl10 : (MT16 , o5 , AOCLm )

AOCLn : Mout.table .collect(t | t.cols)
                  .select(c | c.name = gamma_epsilon).size() = 0
CTaocl11 : (MT9 , o5 , AOCLn )

AOCLn' : Mout.table.collect(t | t.cols)
                      .select(c | c.name = iota_gamma_epsilon).size() = 0
CTaocl12 : (MT9 , o5 , AOCLn' )
```

```
AOCLo : Mout.table.collect(t | t.cols)
                  .select(c | c.name = psi).forAll( ct | ct.type = string)
CTaocl13 : (MT7 , o5 , AOCLo  )

AOCLp : Mout.table.collect(t | t.pkey).size()=1 and
        Mout.table.collect(t | t.pkey).forAll(p | p.name = attA)
CTaocl14 : (MT8 , o5 , AOCLp  )

AOCLr : result.table.collect(t|t.pkey)
                     .select(c|c.container.pkey.exist(p|p != c)).size()=0
CTaocl15 : (MT14 , o5 , AOCLr  )

AOCLs : result.table.select(t | t.name = bClass).collect(t|t.fkeys)
                     .collect(f|f.references)
                     .select(t|t.name = cClass).size()=2
CTaocl16 : (MT16 , o5 , AOCLs  )

AOCLt :
result.table.exist(t|t.name = Ksi and
                     t.fkeys
                     .exist(f|f.cols.exist(cf |
                               t.cols.exist(ct|ct=cf and ct.name=iota_b_psi))
                               and f.references.exist(r | r.name = Ksi)
                          )
                )
CTaocl17 : (MT7 , o5 , AOCLt )

AOCLu :
result.table.exist(t|t.name = Ksi and
                     t.fkeys
                     .exist(f|f.cols.exist(cf |
                               t.cols.exist(ct|ct=cf and ct.name=att_b_psi))
                               and f.references.exist(r | r.name = Ksi)
                          )
                )
CTaocl18 : (MT18 , o5 , AOCLu )

AOCLv : result.table.exist(t|t.name = B and
                     t.fkeys
                     .exist(f|f.cols.exist(cf |
                               t.cols.exist(ct|ct=cf and ct.name=assoDC_attA))
                               and f.references.exist(r | r.name = Ksi)
                          ))
CTaocl19 : (MT19 , o5 , AOCLv  )
```

Annexe B

Dans cette annexe, nous regroupons les différentes pré-conditions nous avons écrites pour la transformation class2rdbms.

```
context Class2RDBMSk::transform(inputModel : ClassModel) : Schema
//pas de typage circulaire :
pre :   inputModel.classifier
        .select(cr| cr.oclIsTypeOf(Class))
        .forAll(cs| not cs.allAttribute().exist(a|a.type = self))

//pas d'héritage circulaire :
pre :   inputModel.classifier
        .select(cr| cr.oclIsTypeOf(Class))
        .forAll(cs| not cs.allParents().exist(a|a = self))

//pas de réécriture d'une association d'une classe persistente :
pre :   inputModel.classifier
        .select(cr| cr.oclIsTypeOf(Class))
        .forAll(cs| not cs.association.name
                        .exist( n| cs.allParent()
                                  .select(p|p.is_persistent)
                                  .association.name.exist(np=n)))

// pas de réécriture d'un attribut primaire d'une classe persistente par un
// attribut non primaire :
pre :   inputModel.classifier
        .select(cr | cr.oclIsTypeOf(Class))
        .forAll(cs | not cs.attrs.select(a|not a.is_primary).name
                        .exist(n | cs.allParent().select(p|p.is_persistent)
                                  .attrs.select(ap | ap.is_primary).name
                                  .exist(np |  np=n)))
```

```
// il y a au moins une classe à transformer :
pre : inputModel.classifier
      .select(cr| cr.oclIsTypeOf(Class))
      .select(cs | cs.is_persistent)
      .select(csp | not csp. oclAsType (Class).allParents()
                        .exists(p | p.is_persistent)
              ).size()>=0

//pas de classifiers de même nom :
pre : inputModel.classifier
      .forAll(cl1, cl2| cl1.name = cl2.name implies cl1=cl2)

//pas d'associations de même nom sortant d'une même classe :
pre : inputModel.classifier
      .select(cr| cr.oclIsTypeOf(Class))
      .forAll(cs| cs.association
                   .forAll(a1, a2 | a1.name = a2.name implies
                           a1 = a2 or a1.src != a2.src))


//un seul attribut de même nom par classe :
pre : inputModel.classifier
      .select(cr| cr.oclIsTypeOf(Class))
      .forAll(cs| cs.attrs
                   .forAll(a1, a2 | a1.name = a2.name implies a1 = a2))

//il existe des attributs typés avec des PrimitiveDataType :
pre :   inputModel.classifier
        .select(cr| cr.oclIsTypeOf(Class))
        .attrs
        .exists(at|at.type.oclIsTypeOf(PrimitiveDataType))
```

Annexe C

Dans cette annexe, nous illustrons le programme de lancement de la plateforme expérimentale développée pour l'application de l'analyse de mutation. Le code de la page suivante donne un exemple de code de lancement de l'analyse de mutation avec minimisation du nombre de modèles de test. Une instance de la classe `MutationAnalysis_full` (ligne 2) est paramétrée avec la méthode `set_param_mutation_analysis() (ligne 15)` et l'analyse est lancée en appelant la méthode `apply_mutation()` (ligne 24). Le programme de réduction du nombre de modèles de test est lancé en appelant la méthode `apply_MatrixReduction()` (ligne 28).

```
1  public class Launch_Mutations {
2     private static KMutationAnalysis lch;
3     private static KMatrixReduction tr;
4
5     public static void main(String[] args) throws Exception {
6        lch = new KMutationAnalysis();
7        tr = new KMatrixReduction();
8
9        ArrayList list_type_mutant = new ArrayList<String>();
10       list_type_mutant.add("creation");
11       list_type_mutant.add("navigation");
12       list_type_mutant.add("filtering");
13
14       try {
15          lch.set_param_mutation_analysis( //paramétrage de l'analyse
16             "transformation/Class2RDBMSk.kmt",// transformation sous test
17             "mutants/",                  //répertoire des mutants
18             list_type_mutant,            //sous-répertoires des mutants
19             "MM/simpleUML_MM.ecore",     // méta-modèle source
20             "MM/simpleRDBMS_MM.ecore",   // méta-modèle cible
21             "Min/",                      // ensemble des modèles de test
22             "Mout",                      // destination des modèles de sortie
23             true);  // utilisation de la comparaison de modèles comme oracle
24          lch.apply_mutation(           //lancement de l'analyse
25             "Result_temp/step1");     //destination de la matrice rétournée
26       } catch (Exception e) {e.printStackTrace();}
27
28       tr.apply_MatrixReduction(
29                  //lancement de la minimisation des modèles de test
30           "Result_temp/step1/result_matrix_mutation_simplified.csv",
31                  //matrice à minimiser
32           "Result_temp/step1/reduction_all_dt_considered",
33                  //destination des matrices minimiséed
34             ,1,0, null);
35    }
```

Annexe D

Cette méthode `find_minimizations_recursive()` est récursivement appelée pour permettre un parcours en largeur de l'arbre. Elle est initialement paramétrée avec la liste des index des colonnes de la matrice obtenue en sortie de l'analyse de mutation et avec deux vecteurs vides. Elle appelle deux méthodes :

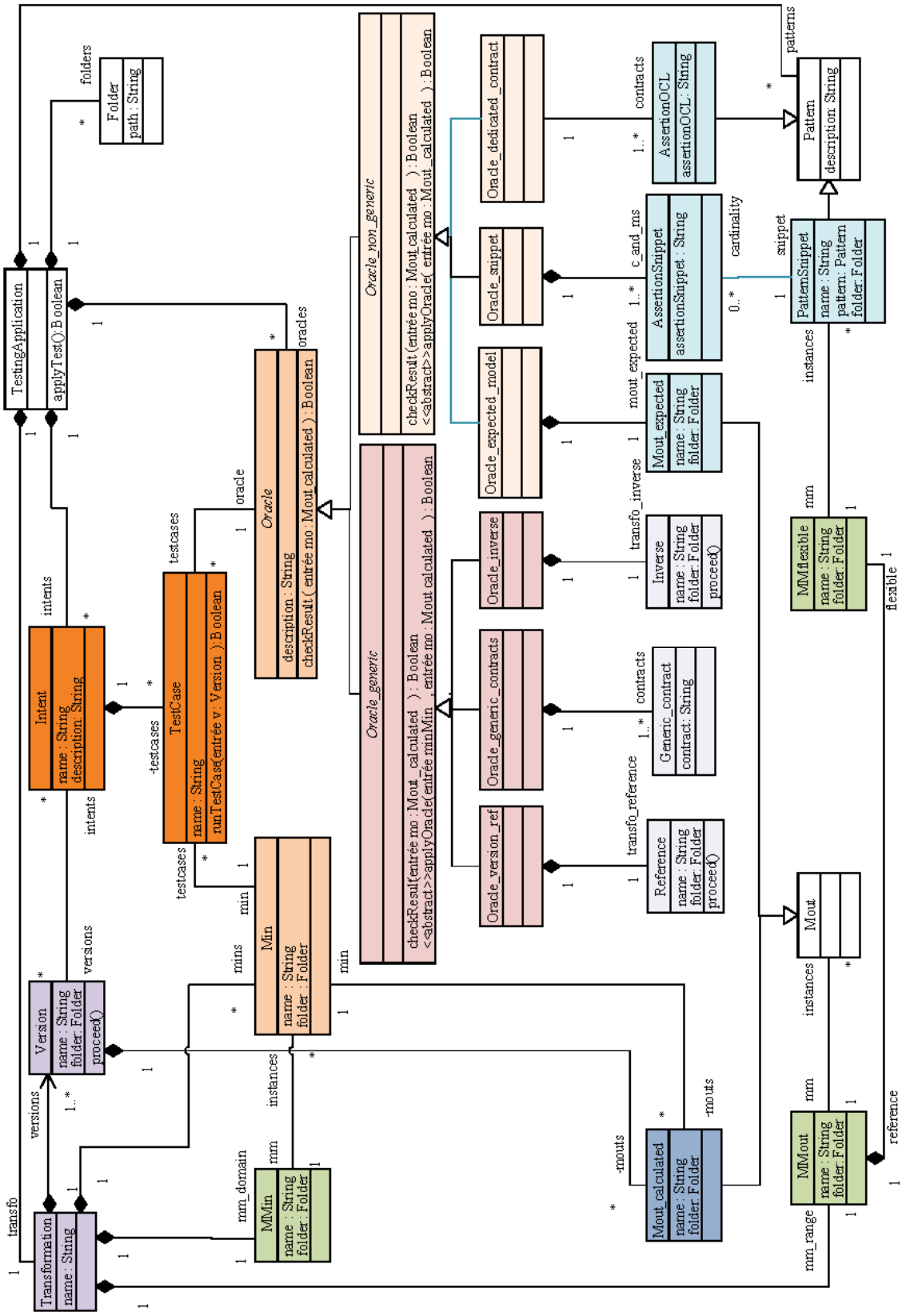
- `minimization_already_studied()` qui vérifie si la minimisation en cours n'a pas déjà été trouvée, ce qui permet de réaliser l'optimisation 1,
- `contained`() qui vérifie qu'une colonne c est incluse dans un ensemble de colonne.

```
private void find_minimizations_recursive(
                              Vector<Integer> liste_index_to_study,
                              Vector<Integer> liste_index_not_containable,
                              Vector<Integer> liste_index_current_branch){

   Vector<Vector> liste_param_recursive = new Vector<Vector>();
   Vector<Integer> liste_index_not_containable_recursive =
                        (Vector<Integer>) liste_index_not_containable.clone();
   for(int i=0; i<liste_index_to_study.size();i++){
      Integer current_index = liste_index_to_study.remove(i);
      if (! liste_index_not_containable.contains(current_index)){
         if(contained(current_index, liste_index_to_study)){
            liste_index_current_branch.add(current_index);
            if (!minimization_already_studied(liste_index_current_branch)){
               Vector<Integer> index_to_remove = new Vector<Integer>();
               index_to_remove.add(current_index);
               available_minimizations .get(liste_index_current_branch.size())
                  .add((Vector<Integer>) liste_index_current_branch.clone());
               if (liste_index_to_study.size()>2){
                  Vector<Vector<Integer>> param_recursive_current =
                                                 new Vector<Vector<Integer>>();
                  param_recursive_current
                     .add((Vector<Integer>) liste_index_to_study.clone());
                  param_recursive_current
                     .add((Vector<Integer>) liste_index_current_branch.clone());
                  liste_param_recursive.add(param_recursive_current);
               }
            }
            liste_index_current_branch
                              .remove(liste_index_current_branch.size()-1);
         }else{
            liste_index_not_containable_recursive.add(current_index);
         }
      }
      liste_index_to_study.add(i,current_index);
   }
   for( Vector param_recursive_current : liste_param_recursive){
      find_minimizations_recursive(
               (Vector<Integer>)param_recursive_current.get(0),
                     liste_index_not_containable_recursive,
                     (Vector<Integer>)param_recursive_current.get(1));
   }
}
```

Annexe E

Dans cette annexe, nous illustrons une version (simplifiée) du méta-modèle de l'outil de « harnais de test » que nous avons développé. Il permet de mettre en œuvre les tests d'une transformation de modèles et de réutiliser différents éléments pour le test de versions différentes de la transformation. Les cas de test peuvent être réutilisés. De nouveaux cas de test peuvent être formés à partir des modèles de test existants ou les différents oracles. Un pattern basé sur un model snippet peut également être utilisé plusieurs fois aussi bien avec une même version de la transformation qu'avec plusieurs versions.


TestingApplication
applyTest():Boolean
Folder
path : String
folders
Intent
name : String
description: String
intents
TestCase
name : String
runTestCase(entrée v : Version):Boolean
testcases
Oracle
description : String
checkResult (entrée mo : Mout_calculated) : Boolean
oracle
oracles
Oracle_non_generic
checkResult (entrée mo : Mout_calculated) : Boolean
<<abstract>>applyOracle(entrée mo : Mout_calculated) : Boolean
Oracle_generic
checkResult(entrée mo : Mout_calculated) : Boolean
<<abstract>>applyOracle(entrée minMin , entrée mo : Mout calculated) : Boolean
Oracle_dedicated_contract
Oracle_snippet
Oracle_expected_model
Oracle_inverse
Oracle_generic_contracts
Oracle_version_ref
AssertionOCL
assertionOCL: String
contracts
AssertionSnippet
assertionSnippet : String
c_and_ms
cardinality
Mout_expected
name : String
folder: Folder
mout_expected
Inverse
name : String
folder: Folder
proceed()
transfo_inverse
Generic_contract
contract: String
contracts
Reference
name : String
folder: Folder
proceed()
transfo_reference
Pattern
description: String
patterns
PatternSnippet
name : String
pattern: Pattern
folder: Folder
snippet
instances
MMflexible
name : String
folder: Folder
mm
flexible
Mout
instances
MMout
name : String
folder: Folder
mm
reference
mm_range
Mout_calculated
name : String
folder: Folder
-mouts
Min
name : String
folder : Folder
min
mins
Version
name : String
folder: Folder
proceed()
versions
Transformation
name : String
transfo
MMin
name : String
folder : Folder
mm_domain
mm
instances

Glossaire

Assertion : prédicat ou conjonction de prédicats qui applique des vérifications sur un état donné d'un système ou d'une donnée.

Cas de test : Ensemble regroupant une donnée de test, l'oracle correspondant, et d'autres informations comme l'état du système avant l'exécution du test. L'exécution de la donnée de test avec le programme sous test dans l'état défini par le cas de test, est analysée par l'oracle pour savoir si le test passe ou échoue.

Composant autotestable : composant logiciel qui embarque son implantation, sa spécification sous forme de contrats exécutables, et un ensemble de cas de test.

Conception par contrat : méthodologie de conception pour des systèmes orientés objet consistant à définir les devoirs et obligations des éléments du système. Les contrats sont de trois types : des pré et post-conditions pour les méthodes, et des invariants pour les classes. La pré-condition définit la condition pour utiliser une méthode. La post-condition définit les propriétés du résultat produit par la méthode. Un invariant définit des propriétés qui doivent vérifiées l'état de l'objet à la création de chaque objet, avant et après chaque appel de méthode.

Couplage : mesure la force de la relation entre deux entités, i.e. proportion des concepts des méta-modèles source et cible d'une transformation sous test, qui sont utilisés dans un modèle de test.

Critère de test : critère qui doit être satisfait par l'ensemble des données de test d'un programme sous test.

Défaillance : événement survenant lorsque le comportement d'un programme ne correspond pas au comportement spécifié (la donnée de sortie est assimilé au comportement du programme).

Donnée d'oracle : paramètre d'une fonction d'oracle qui fournit des informations nécessaires à l'analyse de la donnée de sortie ou de l'exécution d'un test. Elle définit les vérifications effectuées sur l'exécution d'une donnée de test.

Donnée de test : donnée d'entrée d'un programme sous test.

Erreur : partie altérée de l'implantation d'un programme qui ne correspond pas à la spécification et qui peut entraîner des défaillances.

Faute : cause d'une erreur, généralement commise par un développeur.

Fonction d'oracle : fonction qui analyse les sorties ou l'exécution d'un test et fournit un verdict. Elle est paramétrée par le modèle de sortie à vérifier et par une donnée d'oracle.

Framework : ensemble de classes et de collaborations entre leurs instances.

Ingénierie Dirigée par les Modèles : ensembles des techniques de génie logiciel qui s'appuient sur l'utilisation de modèles pour la production de systèmes.

Méta-modèle : modèle représentant un langage de modélisation, il définit les éléments et la structure d'un modèle du langage ainsi que sa sémantique.

Modèle : abstraction d'un système construite dans un but donné. Dans l'Ingénirie Dirigé par les Modèles, un modèle est un graphe d'objets conforme à un méta-modèle.

Model snippet : modèle partielle conforme à une version relâché d'un méta-modèle. Il permet de vérifier la présence d'éléments dans un modèle.

OCL : Object Constraint Language. Langage pour l'expression de contraintes associées aux entités d'un méta-modèle.

Oracle : un oracle est formé avec une fonction d'oracle dont les paramètres sont instanciés par un modèle de sortie donné et une donnée d'oracle donnée.

Spécification : description d'un système logiciel qui en précise les fonctionnalités, les conditions d'utilisation, le comportement.

Test de logiciel dynamique : ensemble d'activité qui consiste à exécuter des données de test sur un programme et appliquer un oracle pour produire le verdict du test.

Test de logiciel statique : activité qui consiste à oserver le programme sous test pour détecter des erreurs.

Transformation de modèles : programme qui produit un modèle à partir d'un modèle.

Verdict de test : un cas de test *passe* si l'exécution de la donnée de test est conforme à l'oracle, sinon il *échoue*.

Bibliographie

[ADONIS'05] ADONIS. *Homepage*. 2005; Available from: www.boc-eu.com.

[Agrawal'03] Agrawal, A., G. Karsai, and A. Ledeczi, *An end-to-end domain-driven software development framework*, in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 2003, ACM: Anaheim, CA, USA.

[Agrawal'89] Agrawal, H., R.A. Demillo, B. Hathaway, W. Hsu, W. Hsu, E.W. Krauser, R.J. Martin, A.P. Mathur, and E. Spafford, *Design of mutant operators for the C programming language.* Report SERC-TR41-P, S.E. Research Center, Purdue University, West Lafayette-USA, 1989.

[Agrawal'95] Agrawal, H., J. Horgan, S. London, and W. Wong. *Fault Localization using Execution Slices and Dataflow Tests*. in *ISSRE'95 (Int. Symposium on Software Reliability Engineering)*. 1995. Toulouse, France.

[Akehurst'06] Akehurst, D.H., B. Bordbar, M.J. Evans, W.G.J. Howells, and K.D. McDonald-Maier. *SiTra: Simple Transformations in Java*. in *MoDELS 2006*. 2006. Genova, Italy.

[Akehurst'05] Akehurst, D.H., W.G. Howells, and K.D. McDonald-Maier. *Kent Model Transformation Language*. in *Model Transformations in Practice, Workshop, part of MoDELS 2005*. 2005. Montego Bay,Jamaica.

[Alanen'03] Alanen, M. and I. Porres. *Difference and Union of Models*. in *UML'03 (Unified Modeling Language)*. 2003. San Francisco, CA, USA.

[Alanen'04] Alanen, M. and I. Porres. *Coral: A Metamodel Kernel for Transformation Engines*. in *Proceedings of the Second European Workshop on Model Driven Architecture (MDA)*. 2004. Kent, United Kingdom.

[Alexander'02] Alexander, R.T., J.M. Bieman, G. Sudipto, and J. Bixia. *Mutation of Java Objects*. in *ISSRE'02 (Int. Symposium on Software Reliability Engineering)*. 2002. Annapolis, MD, USA.

[Amar'08] Amar, B., H. Leblanc, and B. Coulette. *A Traceability Engine Dedicated to Model Transformation for Software Engineering*. in *ECMDA Traceability Workshop'08*. 2008. Berlin, Germany.

[Anastasakis'07] Anastasakis, K., B. Bordbar, and J.M. Kuster. *Analysis of Model Transformations via Alloy*. in *4th International Workshop on Model Driven Engineering, Verification, and Validation: Integrating Verification and Validation in MDE, MoDeVVa'07*. 2007. Nashville, USA.

[Andrews'03] Andrews, A., R. France, S. Ghosh, and G. Craig, *Test adequacy criteria for UML design models.* Software Testing, Verification and Reliability, 2003. **13**(2): p. 95 -127.

[Andrews'05] Andrews, J.H., L.C. Briand, and Y. Labiche, *Is mutation an appropriate tool for testing experiments?*, in *Proceedings of the 27th international conference on Software engineering*. 2005, ACM: St. Louis, MO, USA.

[AndroMDA] AndroMDA. *AndroMDA*. Available from: http://www.andromda.org/.

[Barbosa'01] Barbosa, E.F., J.C. Maldonado, and A.M.R. Vincenzi., *Toward the determination of sufficient mutant operators for C*. Software Testing, Verification and Reliability, 2001. **11**(2): p. 113-136.

[Barnes'03] Barnes, J., *High integrity software: the SPARK approach to safety and security*. . 2003: Addison-Wesley.

[Barnett'05] Barnett, M., K.R.M. Leino, and W. Schulte, *The Spec# Programming System: An Overview* in *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. 2005, Springer Berlin / Heidelberg. p. 49-69.

[Baudry'03] Baudry, B., *Assemblage testable et validation de composants*. PhD thesis, 2003.

[Baudry'06a] Baudry, B., T. Dinh-Trong, J.-M. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. Le Traon. *Model Transformation Testing Challenges*. in *IMDT workshop in conjunction with ECMDA-FA 06*. 2006a. Bilbao, Spain.

[Baudry'06b] Baudry, B., F. Fleurey, and Y. Le Traon. *Improving Test Suites for Efficient Fault Localization*. in *ICSE'06 (International Conference on Software Engineering)*. 2006b. Shanghai, China.

[Baudry'05] Baudry, B., F. Fleurey, Y. Le Traon, and J.-M. Jézéquel, *An Original Approach for Automatic Test Cases Optimization: a Bacteriologic Algorithm*. IEEE Software, 2005. **22**(2): p. 76-82.

[Baudry'00a] Baudry, B., Y. Le Traon, V.L. Hanh, and J.-M. Jézéquel. *Building Trust into OO Components using a Genetic Analogy*. in *ISSRE'00 (Int. Symposium on Software Reliability Engineering)*. 2000a. San Jose, CA, USA.

[Baudry'00b] Baudry, B., Y. Le Traon, J.-M. Jézéquel, and V.L. Hanh. *Trustable Components: Yet Another Mutation-Based Approach*. in *1st Symposium on Mutation Testing*. 2000b. San Jose, CA: Kluwer Academic Publishers, Dordrecht, NL.

[Beizer'90] Beizer, B., *Software Testing Techniques*. 1990: Van Norstrand Reinhold.

[Belaunde'06] Belaunde, M. and G. Dupé. *SmartQVT*. 2006; Available from: http://smartqvt.elibel.tm.fr.

[Bendraou'08] Bendraou, R., P. Desfray, M.P. Gervais, and A. Muller, *MDA Tool Components: A Proposal for Packaging Know-how in Model Driven Development*. SoSyM: Journal on Software & System Modeling, 2008. **7**(3): p. 329-343.

[Beugnard'99] Beugnard, A., J.-M. Jézéquel, N. Plouzeau, and D. Watkins, *Making components contract aware*. IEEE Computer, 1999. **13**(7).

[Bézivin'03a] Bézivin, J., G. Dupé, F. Jouault, G. Pitette, and J.E. Rougui. *First experiments with the ATL model transformation language: Transforming XSLT into XQuery*. in *Generative Techniques in the context of Model Driven Architecture (in conjunction with OOPSLA'03)*. 2003a. Anaheim, CA, USA.

[Bézivin'03b] Bézivin, J., N. Farcet, J.-M. Jézéquel, B. Langlois, and D. Pollet. *Reflective model driven engineering*. in *UML'03 (Unified Modeling Language)*. 2003b. San Francisco, CA, USA.

[Bézivin'03c] Bézivin, J., N. Farcet, J.-M. Jézéquel, B. Langlois, and D. Pollet. *Reflective model driven engineering*. in *UML'03*. 2003c. San Francisco, CA, USA.

[Bézivin'03d] Bézivin, J., S. Gérard, P.-A. Muller, and L. Rioux. *MDA Components: Challenges and Opportunities*. in *Metamodelling for MDA*. 2003d. York, England.

[Bézivin'05] Bézivin, J., B. Rumpe, A. Schürr, and L. Tratt. *MTIP workshop CFP*. 2005; Available from: http://sosym.dcs.kcl.ac.uk/events/mtip05/long_cfp.pdf.

[Binder'99] Binder, R.V., *Testing Object-Oriented Systems: Models, Patterns and Tools*. 1999: Addison-Wesley.

[Bondé'06] Bondé, L., *Transformations de Modèles et Interopérabilité dans la Conception de Systèmes Hétérogènes sur Puce à Base d'IP*. PhD, 2006.

[Bondé'05] Bondé, L., P. Boulet, and J.-L. Dekeyser. *Traceability and interoperability at different levels of abstraction in model transformations*. in *Forum on Specification and Design Languages, FDL'05*. 2005. Lausanne, Switzerland.

[Bondé'04] Bondé, L., C. Dumoulin, and J.-L. Dekeyser. *Metamodels and MDA transformations for embedded systems*. in *FDL'04*. 2004. Lille, France.

[Braun'03] Braun, P. *Metamodel-based Integration of Tools*. in *Proceedings of Workshop on Tool Integration in System Development, with ESEC/FSE'2003*. 2003.

[Briand'03] Briand, L.C., Y. Labiche, and H. Sun, *Investigating the Use of Analysis Contracts to Improve the Testability of Object Oriented Code*. Software Practice and Experience, 2003. **33**(7).

[Brottier'06] Brottier, E., F. Fleurey, J. Steel, B. Baudry, and Y. Le Traon. *Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool*. in *ISSRE'06*. 2006. Raleigh, USA.

[Burmester'04] Burmester, S., H. Giese, J. Niere, M. Tichy, J.P. Wadsack, R. Wagner, L. Wendehals, and A. Zündorf, *Tool integration at the meta-model level within the fujaba tool suite*. International Journal on Software Tools for Technology Transfer (STTT), 2004. **6**(3): p. 203-218.

[Cariou'04] Cariou, E., R. Marvie, L. Seinturier, and L. Duchien. *OCL for the Specification of Model Transformation Contracts*. in *Workshop OCL and Model Driven Engineering of the Seventh International Conference on UML Modeling Languages and Applications (UML 2004)*. 2004. Lisbon, Portugal.

[Chapman'82] Chapman, D., *A program testing assistant*. Commun. ACM, 1982. **25**(9): p. 625-634.

[Chen'03] Chen, T.Y., T.H. Tse, and Z. Zhou, *Fault-based testing without the need of oracles*. Information and Software Technology, 2003. **44**(15): p. 923-931.

[Cheon'02] Cheon, Y. and G.T. Leavens. *A Simple and Practical Approach to Unit Testing: The JML and JUnit Way*. in *ECOOP'02 (European Conference for Object-Oriented Programming)*. 2002. Malaga, Spain.

[Chevalley'01] Chevalley, P. *Applying Mutation Analysis for Object-Oriented Programs Using a Reflective Approach*. in *Eighth Asia-Pacific Software Engineering Conference*. 2001. Macao, China.

[Cicchetti'08] Cicchetti, A., D. Di Ruscio, R. Eramo, and A. Pierantonio. *Meta-model Differences for Supporting Model Co-evolution*. in *2nd Workshop on Model-Driven Software Evolution, MoDSE'2008*. 2008. Athens, Greece.

[Cicchetti'07] Cicchetti, A., D. Di Ruscio, and A. Pierantonio, *A Metamodel Independent Approach to Difference Representation*. Journal of Object Technology, 2007(Special Issue from TOOLS Europe 2007).

[Ciupa'05] Ciupa, I. and A. Leitner. *Automatic Testing Based on Design by Contract™*. in *Net.ObjectDays 2005 (6th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World)*. 2005.

[Ciupa'08] Ciupa, I., A. Pretschner, A. Leitner, M. Oriol, and B. Meyer. *On the Predictability of Random Tests for Object-Oriented Software*. in *ICST'08*. 2008. Lillehammer, Norway.

[Clark'04] Clark, T., A. Evans, P. Sammut, and J. Willans, *Applied Metamodelling: A Foundation for Language Driven Development*. 2004. 189.

[Compuware] Compuware. *Optimalj*. Available from: http ://www.compuware.com/products/optimalj/.

[Cuccuru'05] Cuccuru, A., J.-L. Dekeyser, P. Marquet, and P. Boulet. *Towards UML2 Extensions for Compact Modeling of Regular Complex Topologies*. in *MoDELS 2005*. 2005. Montego Bay, Jamaica.

[Czarnecki'03] Czarnecki, K. and S. Helsen. *Classification of Model Transformation Approaches*. in *Generative Techniques in the context of Model Driven Architecture (in conjunction with OOPSLA'03)*. 2003. Anaheim, CA, USA.

[Czarnecki'06] Czarnecki, K. and S. Helsen, *Feature-Based Survey of Model Transformation Approaches.* IBM Systems Journal, special issue on Model-Driven Software Development, 2006. **45**(3): p. 621-645.

[Darabos'06] Darabos, A., A. Pataricza, and D. Varro. *Towards Testing the Implementation of Graph Transformations*. in *GT-VMT workshop associated to ETAPS'06*. 2006. Vienna, Austria.

[Davis'03] Davis, J., *GME: the generic modeling environment*, in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 2003, ACM: Anaheim, CA, USA.

[de Lara'02] de Lara, J. and H. Vangheluwe. *AToM3: A Tool for Multi-formalism and Meta-modelling*. in *FASE '02*. 2002. Grenoble, France.

[Delamare'06] Delamare, R., B. Baudry, and Y. Le Traon. *Reverse-engineering of UML 2.0 Sequence Diagrams from Execution Traces*. in *Workshop on Object-Oriented Reengineering in conjunction with ECOOP'06*. 2006. Nantes, France.

[Delamaro'01] Delamaro, M.E., J.C. Maldonado, and A.P. Mathur, *Interface Mutation: An approach for integration testing.* IEEE Transactions on Software Engineering, 2001. **27**(3): p. 228-247.

[DeMillo'78] DeMillo, R., R. Lipton, and F. Sayward, *Hints on Test Data Selection : Help For The Practicing Programmer.* IEEE Computer, 1978. **11**(4): p. 34 - 41.

[DeMillo'91] DeMillo, R. and A.J. Offutt, *Constraint-Based Automatic Test Data Generation.* IEEE Transactions on Software Enginnering, 1991. **17**(9): p. 900 - 910.

[DeMillo'93] DeMillo, R. and A.J. Offutt, *Experimental results from an automatic test case generator.* ACM Transactions on Software Engineering and Methodology, 1993. **2**(2): p. 109 - 27.

[Derezinska'08] Derezinska, A. and A. Szustek. *Tool-Supported Advanced Mutation Approach for Verification of C# Programs*. in *Third International Conference onDependability of Computer Systems, DepCos-RELCOMEX '08.* . 2008. Szklarska Poręba, Poland.

[Dillon'94] Dillon, L.K. and Q. Yu. *Oracles for checking temporal properties of concurrent systems*. in *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*. 1994. New Orleans, Louisiana, United States: ACM.

[Eagan'01] Eagan, J., M.J. Harrold, J. Jones, and J. Stasko, *Visually encoding program test information to find faults in software.* Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, 2001.

[Eclipse Foundation'07] Eclipse Foundation. *EMF Compare*. 2007; Available from: www.eclipse.org/emft/projects/compare.

[Ecore] Ecore. *Eclipse Modeling Framework Project (EMF)*. Eclipse Foundation; Available from: http://www.eclipse.org/emf/.

[Ehrig'06] Ehrig, K., J.M. Küster, G. Taentzer, and J. Winkelmann. *Generating Instance Models from Meta Models* in *8th International Conference on Formal Methods*

for Open Object-Based Distributed Systems, FMOODS 2006. 2006. Bologna, Italy.

[EMF] EMF. *Eclipse Modeling Framework Project (EMF)*. Eclipse Foundation; Available from: http://www.eclipse.org/emf/.

[Estublier'05] Estublier, J., J.-M. Favre, J. Bézivin, L. Duchien, R. Marvie, S. Gérard, B. Baudry, M. Bouzhegoud, J.-M. Jézéquel, M. Blay, and M. Riveil, *Rapport de synthèse 1.1.2, AS MDA*. CNRS, 2005.

[Etien'07] Etien, A., C. Dumoulin, and E. Renaux, *Towards a unified notation to represent model transformation*. Research Report RR-6187, INRIA, 2007.

[Favre'06] Favre, J.-M., J. Estublier, and M. Blay, eds. *L'ingénierie dirigée par les modèles au-delà du MDA*. 2006, Lavoisier.

[Ferrari'08] Ferrari, F.C., J.C. Maldonado, and A. Rashid. *Mutation Testing for Aspect-Oriented Programs*. in *International Conference on Software Testing (ICST0'8)*. 2008. Lillehammer, Norway.

[Filman'05] Filman, R.E., T. Elrad, S. Clarke, and M. Ak¸sit, *Aspect-Oriented Software Development*. 2005, Boston Addison-Wesley.

[Fleurey'06] Fleurey, F., *Langage et méthode pour une ingénierie des modèles fiable*. PhD thesis, 2006.

[Fleurey'07a] Fleurey, F., B. Baudry, P.-A. Muller, and Y. Le Traon, *Towards Dependable Model Transformations: Qualifying Input Test Data*. Journal of Software and Systems Modeling, 2007a.

[Fleurey'07b] Fleurey, F., E. Breton, B. Baudry, A. Nicolas, and J.-M. Jézéquel. *Model-Driven Engineering for Software Migration in a Large Industrial Context*. in *MoDELS/UML 2007 conference*. 2007b. Nashville, USA.

[Fleurey'04] Fleurey, F., J. Steel, and B. Baudry. *Validation in Model-Driven Engineering: Testing Model Transformations*. in *MoDeVa'04 (Model Design and Validation Workshop associated to ISSRE'04)*. 2004. Rennes, France.

[Fondement'04] Fondement, F. and R. Silaghi. *Defining Model Driven Engineering Processes*. in *WISME*. 2004. Lisbon, Portugal.

[France'07] France, R., F. Fleurey, R. Reddy, B. Baudry, and S. Ghosh. *Providing Support for Model Composition in Metamodels*. in *11th IEEE International EDOC Conference (EDOC 2007)*. 2007. Annapolis, Maryland, U.S.A.

[France'03] France, R., S. Ghosh, E. Song, and K. Dae-Kyoo, *A Metamodeling Approach to Pattern-Based Model Refactoring*. IEEE Software, 2003. **20**(5): p. 52–58.

[Frankl'97] Frankl, P., S. Weiss, and C. Hu, *All-uses versus mutation testing: An experimental comparison of effectiveness*. The Journal of Systems and Software, 1997. **38**(3): p. 235-253.

[Gerber'02] Gerber, A., M.J. Lawley, K. Raymond, J. Steel, and A. Wood. *Transformation: The Missing Link of MDA*. in *First International Conference on Graph Transformation, ICGT'02*. 2002: Springer Verlag.

[Ghosh'01] Ghosh, S. and A. Mathur, *Interface mutation*. Software Testing, Verification and Reliability, 2001. **11**(4): p. 227-247.

[Giese'06] Giese, H., S. Glesner, J. Leitner, W. Schäfer, and R. Wagner. *Towards Verified Model Transformations*. in *Proceedings of MoDeVa workshop associated to MoDELS'06*. 2006. Genova, Italy.

[Glitia'08] Glitia, F., A. Etien, and C. Dumoulin. *Fine Grained Traceability for an MDE Approach of Embedded System Conception*. in *Fourth ECMDA Traceability Workshop*. 2008. Berlin, Germany.

[Greenfield'03] Greenfield, J. and K. Short. *Software Factories Assembling Applications with Patterns, Models, Frameworks and Tools*. in *International Conference OOPSLA'03*. 2003. Anaheim, USA: ACM.

[Hamlet'77] Hamlet, R.G., *Testing Programs with the Aid of a Compiler*. IEEE Trans. Softw. Eng., 1977. **3**(4): p. 279-290.

[Harrold'94] Harrold, M.J. and G. Rothermel. *Performing Data Flow Testing on Classes*. in *FSE'94 (Foundation on Software Engineering)*. 1994. New Orleans, US.

[Heckel'03] Heckel, R. and M. Lohmann, *Towards Model-Driven Testing*. Electronic Notes in Theoretical Computer Science, 2003. **82**(6).

[Ho'99] Ho, W.-M., J.-M. Jézéquel, A. Le Guennec, and F. Pennaneac'h. *UMLAUT: an Extendible UML Transformation Framework*. in *ASE'99 (Automated Software Engineering)*. 1999. Cocoa Beach, Florida, USA.

[Hoffman'99] Hoffman, D., *Heuristic Test Oracles. The balance between exhaustive comparison and no comparison at all*. Software Testing and Quality Engineering, 1999. **1**(2).

[Hoijin'98] Hoijin, Y., C. Byoungju, and J. Jin-Ok. *Mutation-based inter-class testing*. in *Asia Pacific Software Engineering Conference*. 1998. Taipei, Taiwan.

[Hovsepyan'07] Hovsepyan, A., S.V. Baelen, K. Yskout, Y. Berbes, and W. Joosen. *Composing Application Models and Security Models: On the Value of Aspect-Oriented Technologies*. in *Aspect Oriented Modeling (AOM) Workshop colocated with MoDELS'07*. 2007. Nashville, USA.

[Hubert] Hubert, R. *Arcstyler - the architectural ide for mda*. Available from: http ://www.io-software.com/.

[IEEE'04] IEEE, *IEEE Standard for Software Verification and Validation*. IEEE Computer Society, 2004.

[Jackson'08] Jackson, D. *http://alloy.mit.edu/community/*. 2008.

[Jéron'98] Jéron, T. and P. Morel. *Test Generation Derived from Model-checking*. in *CAV'99*. 1998. Kyoto, Japan.

[Jézéquel'01] Jézéquel, J.-M., D. Deveaux, and Y. Le Traon, *Reliable Objects: a Lightweight Approach Applied to Java*. IEEE Software, 2001. **18**(4): p. 76 - 83.

[Jézéquel'97] Jézéquel, J.-M. and B. Meyer, *Design by Contract: The lessons of Ariane*. IEEE Computer, 1997. **30**(1): p. 129 - 130.

[Jones'02a] Jones, J.A., M.J. Harrold, and J. Stasko. *Visualization of Test Information to Assist Fault Localization*. in *ICSE'02 (Int. Conference in Software Engineering)*. 2002a. Orlando, FL, USA.

[Jones'02b] Jones, J.A., M.J. Harrold, and J. Stasko. *Visualization of Test Information to Assist Fault Localization*. in *ICSE'02*. 2002b. Orlando, FL, USA.

[Jouault'05] Jouault, F. and I. Kurtev. *Transforming Models with ATL*. in *Model Transformations in Practice Workshop at MoDELS 2005*. 2005. Montego Bay, Jamaica.

[Jouault'06] Jouault, F. and I. Kurtev. *On the Architectural Alignment of ATL and QVT*. in *ACM Symposium on Applied Computing - SAC06*. 2006. Dijon, France.

[Judson'03] Judson, S.R., R. France, and D.L. Carver. *Model Transformations at the Metamodel Level*. in *Workshop in Software Model Engineering associated to UML'03*. 2003. San Francisco, CA, USA.

[Kalnins'04] Kalnins, A., J. Barzdins, and E. Celms. *Model Transformation Language MOLA: Extended Patterns*. in *6th International Baltic Conference DB&IS'2004*. 2004.

[Kalnins'05] Kalnins, A., E. Celms, and A. Sostaks. *Model Transformation Approach Based on MOLA*. in *Model Transformations in Practice Workshop at MoDELS 2005*. 2005. Montego Bay, Jamaica.

[Kiczales'97] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. *Aspect-Oriented Programming*. in *ECOOP'97 (European Conference for Object-Oriented Programming)*. 1997.

[Kim'01] Kim, S.-W., J.A. Clark, and J.A. McDermid, *Investigating the effectiveness of object-oriented testing strategies using the mutation method*. Software Testing, Verification and Reliability, 2001. **11**(4): p. 207 - 225.

[Kim'06] Kim, S.-W., M.J. Harrold, and Y.-R. Kwon. *MUGAMMA: Mutation Analysis of Deployed Software to Increase Confidence and Assist Evolution*. in *Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)*. 2006. Raleigh, USA.

[Kolovos'06] Kolovos, D.S., R.F. Paige, and F.a.C. Polack. *Model Comparison: A Foundation for Model Composition and Model Transformation Testing*. in *workshop GaMMa'06*. 2006. Shangaï, China.

[Kolovos'08] Kolovos, D.S., R.F. Paige, and F.A.C. Polack. *The Epsilon Transformation Language*. in *First International Conference on Model Transformation, ICMT 2008*. 2008. Zürich, Zwitzerland.

[Konigs'05] Konigs, A. *Model Transformations with Triple Graph Grammars*. in *Model Transformations in Practice Workshop at MoDELS 2005*. 2005. Montego Bay, Jamaica.

[Kruchten'99] Kruchten, P., *The Rational Unified Process: an introduction*. 1999: Addison-Wesley Longman Publishing Co., Inc. 255.

[Küster'04] Küster, J.M. *Systematic Validation of Model Transformations*. in *Workshop in Software Model Engineering associated to UML'04*. 2004. Lisbon, Portugal.

[Küster'06a] Küster, J.M., *Definition and Validation of Model Transformations*. Software and Systems Modeling, 2006a. **5**(3): p. 233-259.

[Küster'06b] Küster, J.M. and M. Abd-El-Razik. *Validation of Model Transformations - First Experiences using a White Box Approach*. in *workshop MoDeV²a 2006, part of MoDELS'06*. 2006b. Genova, Italy.

[Küster'03] Küster, J.M., R. Heckel, and G. Engels. *Defining and Validating Transformations of UML*. in *IEEE Symposia on Human Centric Computing Languages and Environment*. 2003. Auckland, New Zealand.

[Lamari'07] Lamari, M. *Towards an Automated Test Generation for the Verification of Model Transformations*. in *Symposium on Applied Computing SAC'07*. 2007. Seoul, Korea.

[Laprie'95] Laprie, J.-C., *Guide de la surete de fonctionnement*. 1995: Cepadues.

[Lawley'05] Lawley, M. and J. Steel. *Practical Declarative Model Transformation With Tefkat*. in *Model Transformation in Practice Workshop, part of MoDELS'05*. 2005. Montego Bay, Jamaica.

[Le Traon'06] Le Traon, Y., B. Baudry, and J.-M. Jézéquel, *Design by Contract to Improve Software Vigilance*. IEEE Transactions on Software Engineering, 2006. **32**(8).

[Le Traon'99] Le Traon, Y., D. Deveaux, and J.-M. Jézéquel. *Self-testable Components: from Pragmatic Tests to a Design-for-Testability Methodology*. in *TOOLS'99 (Technology of Object Oriented Languages and Systems)*. 1999. Nancy, France: IEEE Computer Society Press, Los Alamitos, CA, USA.

[Leitner'07] Leitner, A., I. Ciupa, M. Oriol, B. Meyer, and A. Fiva. *Contract Driven Development = Test Driven Development - Writing Test-Cases*. in *ESEC/FSE'07*. 2007. Dubrovnik, Croatia.

[Levendovszky'04] Levendovszky, T., L. Lengyel, G. Mezei, and H. Charaf. *A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS*. in *2nd International Workshop on Graph Based Tools (GraBaTs) with ICGT'2004*. 2004. Rome, Italy.

[Lin'07] Lin, Y., J. Gray, and F. Jouault, *DSMDiff: A Differentiation Tool for Domain-Specific Models*. European Journal of Information Systems, Special Issue on Model-Driven Systems Development, 2007.

[Lin'04] Lin, Y., J. Zhang, and J. Gray. *Model Comparison: A Key Challenge for Transformation Testing and Version Control in Model Driven Software Development*. in *OOPSLA'04*. 2004.

[Lin'05] Lin, Y., J. Zhang, and J. Gray, *A Testing Framework for Model Transformations*, in *Model-driven Software Development - Research and Practice in Software Engineering*. 2005, Springer.

[Luckman'85] Luckman, D.C. and F.W. von Henke, *Overview of ANNA, a specification Language for Ada*. IEEE Software, 1985. **2**(2): p. 9-22.

[Ma'02] Ma, Y.-S., Y.-R. Kwon, and A.J. Offutt. *Inter-Class Mutation Operators for Java*. in *ISSRE'02 (Int. Symposium on Software Reliability Engineering)*. 2002. Annapolis, MD, USA: IEEE Computer Society Press, Los Alamitos, CA, USA.

[Ma'05] Ma, Y.-S., A.J. Offutt, and Y.-R. Kwon, *MuJava : An Automated Class Mutation System*. Software Testing, Verification and Reliability, 2005.

[Mathur'94] Mathur, A.P. and W.E. Wong, *An empirical comparison of data flow and mutation based test adequacy criteria*. Software Testing, Verification and Reliability, 1994. **4**(1): p. 9-31.

[McGregor'01] McGregor, J.D., *A Practical Guide To Testing Object Oriented Testing*. Object Technology Series. 2001: Addison Wesley.

[Mens'05] Mens, K. and P. Van Gorp. *A Taxonomy of Model Transformation*. in *Workshop on Graph and Model Transformation (GraMoT 2005)*. 2005.

[Meyer'92a] Meyer, B., *Applying Design by Contract*. IEEE Computer, 1992a. **25**(10): p. 40 - 51.

[Meyer'92b] Meyer, B., *Object-oriented software construction*. 1992b: Prentice Hall. 1254.

[Meyer'07] Meyer, B., I. Ciupa, A. Leitner, and L.L. Liu, *Automatic Testing of Object-Oriented Software*, in *SOFSEM 2007 (Current Trends in Theory and Practice of Computer Science)*. 2007, Springer-Verlag.

[Mia-software] Mia-software. *Mia-transformation*. Available from: http://www.mia-software.com/.

[Millan'08] Millan, T., L. Sabatier, P. Bazex, and C. Percebois, *NEPTUNE II Une plate-forme pour la vérification et la transformation de modèles*. Génie Logiciel, GL & IS, Numéro spécial Une plate-forme pour la vérification et la transformation de modèles, 2008. **85**: p. 30-34.

[Moore'01] Moore, I. *Jester - a JUnit test tester*. in *XP 2001*. 2001. Villasimius, Sardinia.

[Mottu'07] Mottu, J.-M., O. Barais, M. Skipper, D. Vojtisek, and J.-M. Jézéquel. *Intégration du support OCL dans Kermeta. Spécifiez la sémantique statique de vos méta-modèles*. in *Session de présentation d'outils d'AFADL'07*. 2007. Namur, Belgique.

[Mottu'06a] Mottu, J.-M., B. Baudry, and Y. Le Traon. *Mutation Analysis Testing for Model Transformations*. in *ECMDA'06 (European Conference on Model Driven Architecture)*. 2006a. Bilbao, Spain.

[Mottu'06b] Mottu, J.-M., B. Baudry, and Y. Le Traon. *Reusable MDA Components: A Testing-for-Trust Approach*. in *MoDELS'06*. 2006b. Genova, Italy.

[Mottu'08a] Mottu, J.-M., B. Baudry, and Y. Le Traon. *Model transformation testing : oracle issue*. in *MoDeVVa workshop colocated with ICST08*. 2008a. Lillehammer, Norway.

[Mottu'05] Mottu, J.-M., B. Baudry, Y. Le Traon, and E. Brottier. *Génération automatique de tests pour les transformations de modèles*. in *Ingénierie Des Modèles*. 2005. Paris, France.

[Mottu'08b] Mottu, J.-M., B. Baudry, and Y.L. Traon. *Test de Transformation de Modèles : Expression d'Oracles*. in *4ièmes Journées sur l'Ingénierie Dirigée par les Modèles*. 2008b. Mulhouse, France.

[Muller'05a] Muller, P.-A., F. Fleurey, and J.-M. Jézéquel. *Weaving executability into object-oriented meta-languages*. in *MoDELS'05*. 2005a. Montego Bay, Jamaica: LNCS.

[Muller'05b] Muller, P.-A., F. Fleurey, D. Vojtisek, Z. Drey, D. Pollet, F. Fondement, and P. Studer. *On Executable Meta-Languages applied to Model Transformations*. in *Model Transformation in Practice Workshop, part of MoDELS'05*. 2005b. Montego Bay, Jamaica.

[Murzek'05] Murzek, M., G. Kappel, and G. Kramler. *Model Transformation in Practice Using the BOC Model Transformer*. in *Model Transformations in Practice Workshop at MoDELS 2005*. 2005. Montego Bay, Jamaica.

[Myers'79] Myers, G.J., *The art of Software Testing*. 1979, New York, NY, USA: John Wiley & Sons, Inc. 177.

[Narayanan'08] Narayanan, A. and G. Karsai, *Towards Verifying Model Transformations.* Electron. Notes Theor. Comput. Sci., 2008. **211**: p. 191-200.

[Obeo] Obeo. *Eclipse Foundation, QVT Relational*. Available from: http://wiki.eclipse.org/M2M/QVTR.

[Objecteering Software] Objecteering Software. *Objecteering*. Available from: http://www.objecteering.com.

[Offutt'92] Offutt, A.J., *Investigations of the software testing coupling effect.* ACM Transactions on Software Engineering and Methodology, 1992. **1**(1): p. 5 - 20.

[Offutt'96a] Offutt, A.J., A. Lee, G. Rothermel, R.H. Untch, and C. Zapf, *An Experimental Determination of Sufficient Mutant Operators.* ACM Transactions on Software Engineering and Methodology, 1996a. **5**(2): p. 99 - 118.

[Offutt'97] Offutt, A.J. and J. Pan, *Automatically Detecting Equivalent Mutants and Infeasible Paths.* Software Testing, Verification and Reliability, 1997. **7**(3): p. 165 - 192.

[Offutt'96b] Offutt, A.J., J. Pan, K. Tewary, and T. Zhang, *An experimental evaluation of data flow and mutation testing.* Software Practice and Experience, 1996b. **26**(2).

[Offutt'96c] Offutt, A.J., J. Voas, and J. Payne, *Mutation operators for Ada.* Report ISSE-TR-96-06, Dept. Inf. and Soft. Systems Eng., George Mason University, Fairfax, USA, 1996c.

[Oldevik'07] Oldevik, J. and Ø. Haugen. *Higher-Order Transformations for Product Lines*. in *International Software Product Line Conference SPLC'07*. 2007. Kyoto, Japan.

[OMG'97a] OMG, *Meta Object Facility (MOF) Specification.* AD/97-08-14, 1997a.

[OMG'97b] OMG. *Object Constraint Language Specification*. 1997b; Available from: http://www.omg.org/docs/ad/97-08-08.pdf.

[OMG'03] OMG. *UML 2.0 Object Constraint Language (OCL) Final Adopted specification*. 2003 2005; Available from: http://www.omg.org/docs/formal/06-05-01.pdf.

[OMG'04] OMG. *MOF 2.0 Core Final Adopted Specification.* 2004 2005; Available from: http://www.omg.org/docs/html/06-01-01/Output/06-01-01.htm.

[OMG'05] OMG, *MOF 2.0 Q/V/T OMG Revised Submission*. 2005.

[OMG'07] OMG. *UML 2.1.2 Superstructure and Infrastructure*. 2007 November 2007; Available from: http://www.uml.org/.

[OMG'08] OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Specification, version 1.0*. 2008; Available from: http://www.omg.org/spec/QVT/1.0/.

[Ostrand'88] Ostrand, T.J. and M.J. Balcer, *The category-partition method for specifying and generating functional tests*. Communications of the ACM, 1988. **31**(6): p. 676 - 686.

[Panzl'78] Panzl, D.J., *Automatic Software Test Drivers*. Computer, 1978. **11**(4): p. 44-50.

[Pathfinder Solutions] Pathfinder Solutions. *Pathmate : Transformation engine*. Available from: http:/www.pathfindermda.com.

[Peters'98] Peters, D. and D.L. Parnas, *Using Test Oracles Generated from Program Documentation*. IEEE Transactions on Software Engineering, 1998. **24**(3).

[Pickin'02] Pickin, S., C. Jard, Y. Le Traon, T. Jéron, J.-M. Jézéquel, and A. Le Guennec. *System Test Synthesis from UML Models of Distributed Software*. in *FORTE*. 2002.

[Podnieks'05] Podnieks, K., *MDA: correctness of model transformations. Which models are schemas?* Frontiers in Artificial Intelligence and Applications, 2005. **118**: p. pp. 185-197.

[Poernomo'08] Poernomo, I. *Proofs-as-Model-Transformations*. in *First International Conference on Theory and Practice of Model Transformations ICMT'08*. 2008.

[Pollet'05] Pollet, D., *Une architecture pour les transformations de modèles et la restructuration de modèles UML*. PhD thesis, 2005.

[Pollet'02] Pollet, D., D. Vojtisek, and J.-M. Jézéquel. *OCL as a core UML transformation language*. in *WITUML 2002*. 2002. Malaga, Spain.

[Ramos'07] Ramos, R., O. Barais, and J.-M. Jézéquel. *Matching Model-Snippets*. in *MoDELS'07*. 2007. Nashville, USA.

[Reiter'07] Reiter, T., K. Altmanninger, A. Bergmayr, W. Schwinger, and G. Kotsis. *Models in Conflict - Detection of Semantic Conflicts in Model-based Development*. in *proceedings of 3rd international workshop on model-driven enterprise information systems (MDAIS-2007), in conjunction with ICEIS'07*. 2007. Madeira, Portugal.

[Revault'96] Revault, N., *Principes de méta-modélisation pour l'utilisation de canevas d'applications à objets (MétaGen et les frameworks)*. PhD thesis, 1996.

[Richardson'92] Richardson, D.J., S.L. Aha, and T.O. O'Malley. *Specification-based test oracles for reactive systems*. in *International conference on Software engineering ICSE'92*. 1992. Melbourne, Australia.

[Rosenblum'95] Rosenblum, D.S., *A Practical Approach to Programming With Assertions*. IEEE Transactions on Software Engineering, 1995. **21**(1): p. 19 - 31.

[Sen'06] Sen, S. and B. Baudry. *Mutation-based Model Synthesis in Model Driven Engineering*. in *Proceedings of Mutation'06 workshop associated to ISSRE'06*. 2006. Raleigh, USA.

[Sen'08] Sen, S., B. Baudry, and J.-M. Mottu. *On Combining Multi-formalism Knowledge to Select Models for Model Transformation Testing*. in *Proceedings of ICST'08*. 2008. Lillehammer, Norway.

[Sen'07] Sen, S., B. Baudry, and D. Precup. *Partial Model Completion in Model Driven Engineering using Constraint Logic Programming*. in *Proceedings of INAP'07 (International Conference on Applications of Declarative Programming and Knowledge Management)*. 2007. Würzburg, Germany.

[Sendall'03] Sendall, S. and W. Kozaczynski, *Model Transformation: The Heart and Soul of Model-Driven Software Development.* IEEE Software, 2003. **20**(5): p. 42-45.

[Siikarla'08a] Siikarla, M. *Transformation-Driven Software Evolution Hinders Software Evolution*. in *2nd Workshop on Model-Driven Software Evolution MoDSE'2008*. 2008a. Athens, Greece.

[Siikarla'08b] Siikarla, M. and T. Systa. *Decision Reuse in an Interactive Model Transformation*. in *12th European Conference on Software Maintenance and Reengineering, 2008. CSMR 2008.* . 2008b. Tampere.

[Smolander'91] Smolander, K., K. Lyytinen, V.-P. Tahvanainen, and P. Marttiin. *MetaEdit: a flexible graphical environment for methodology modelling*. in *Proceedings of the third international conference on Advanced information systems engineering*. 1991. Trondheim, Norway: Springer-Verlag New York, Inc.

[Sodius] Sodius. *MDWorkbench*. Available from: http://www.mdworkbench.com/.

[Solberg'06] Solberg, A., R. Reddy, D. Simmonds, R. France, and S. Ghosh, *Developing Service Oriented Systems Using an Aspect-Oriented Model Driven Framework.* International Journal of Cooperative Information Systems, 2006.

[Soley'00] Soley, R. and O.S.S. Group, *Model Driven Architecture.* 2000.

[Steel'04] Steel, J. and M. Lawley. *Model-Based Test Driven Development of the Tefkat Model-Transformation Engine*. in *ISSRE'04 (Int. Symposium on Software Reliability Engineering)*. 2004. Saint-Malo, France.

[Sztipanovits'95] Sztipanovits, J., G. Karsai, C. Biegl, T. Bapty, A. Ledeczi, and A. Misra. *MULTIGRAPH: an architecture for model-integrated computing*. in *First IEEE International Conference on Engineering of Complex Computer Systems*. 1995. Ft. Lauderdale, USA.

[Taentzer'04] Taentzer, G. *AGG: A Graph Transformation Environment for Modeling and Validation of Software*. in *Workshop AGTIVE'2003*. 2004.

[Taentzer'05] Taentzer, G., K. Ehrig, E. Guerra, J. Lara, L. Lengyel, T. Levendovszky, U. Prange, D. Varro, and S. Varro-Gyapay. *Model Transformations by Graph Transformations: A Comparative Study*. in *Model Transformations in Practice Workshop at MoDELS 2005*. 2005. Montego Bay, Jamaica.

[Tolvanen'03] Tolvanen, J.-P. and M. Rossi. *MetaEdit+: defining and using domain-specific modeling languages and code generators*. in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 2003. Anaheim, CA, USA: ACM.

[Tratt'07] Tratt, L., *Model transformations in MT.* Science of Computer Programming, 2007. **68**(3): p. 196-213.

[Trung'05] Trung, D.-T., S. Ghosh, F. Robert, B. Baudry, and F. Fleurey. *A Taxonomy of Faults for UML Designs*. in *2nd MoDeVa workshop - Model design and Validation, in conjunction with MoDELS05*. 2005. Montego Bay, Jamaica.

[Varro'02] Varro, D. *Towards Formal Verification of Model Transformations*. in *PhD Student Workshop of FMOODS 2002, Formal Methods for Open Object-Based Distributed Systems*. 2002. Enschede, The Netherlands.

[Varro'07] Varro, D. and A. Balogh, *Advanced model transformation language constructs in the VIATRA2 framework.* Science of Computer Programming, 2007. **68**(3): p. 214-234.

[Varró'03] Varró, D. and A. Pataricza. *Automated Formal Verification of Model Transformations*. in *Proceedings CSDUML 2003: Critical Systems Development in UML Workshop of UML'03*. 2003: Technische Universität München.

[Vernadat'06] Vernadat, F., C. Percebois, P. Farail, R. Vingerhoeds, A. Rossignol, J.-P. Talpin, and D. Chemouil. *The TOPCASED Project - A Toolkit in OPen-source for Critical Applications and SystEm Development*. in *conference in Data Systems In Aerospace (DASIA)*. 2006. Berlin, Germany.

[Voas'92a] Voas, J.M., *PIE : A Dynamic Failure-Based Technique*. IEEE Transactions on Software Engineering, 1992a. **18**(8): p. 717 - 727.

[Voas'92b] Voas, J.M. and K. Miller, *The Revealing Power of a Test Case*. Software Testing, Verification and Reliability, 1992b. **2**(1): p. 25 - 42.

[W3C'07a] W3C. *XML Query Language, january 2007*. 2007a; Available from: http://www.w3.org/TR/xquery.

[W3C'07b] W3C. *XSL Transformations (XSLT) version 2.0, january 2007*. 2007b; Available from: http://www.w3.org/TR/xslt20/.

[Wang'05] Wang, X., Z.-C. Qi, and S. Li. *An Optimized Method for Automatic Test Oracle Generation from Real-Time Specification*. in *10th International Confenrece on Engineering of Complex Computer Systems ICECCS'05*. 2005. Los Alamitos, USA: IEEE Computer Society.

[Warmer'03] Warmer, J. and A. Kleppe, *The Object Constraint Language - Second Edition, Getting Your Models Ready for MDA*. 2003: Addison-Wesley.

[West Team] West Team. *Modtransf*. Available from: http://www2.lifl.fr/west/modtransf.

[Weyuker'82] Weyuker, E.J., *On testing non-testable programs*. Computer, 1982. **25**(4): p. 465-470.

[Wong'93] Wong, W.E., *On Mutation and Data Flow*. PhD thesis, 1993.

[Xactium] Xactium. *Xmf-mosaic*. Available from: http://www.xactium.com.

[Xanthakis'92] Xanthakis, S., C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios. *Genetic Algorithms Applications to Software Testing*. in *Fifth International Conference. Software Engineering and Its Applications*. 1992. Toulouse, France.

[Xanthakis'00] Xanthakis, S., P. Régnier, and K. Karapoulios, *Le test des logiciels*. 2000: Hermès.

[Xing'05] Xing, Z. and E. Stroulia. *UMLDiff: An Algorithm for Object-Oriented Design Differencing*. in *Automated Software Engineering (ASE'05)*. 2005. USA.

[Yang'94] Yang, L.H.Y.X. *Constructing an automated testing oracle: an effort to produce reliable software*. in *Computer Software and Applications Conference,COMPSAC'94*. 1994. Taipei, Taiwan.

[Yu'07] Yu, H., A. Gamatié, É. Rutten, and J.-L. Dekeyser. *Model Transformations from a Data Parallel Formalism towards Synchronous Languages*. in *Forum on specification and design languages (FDL'07)*. 2007. Barcelona, Spain.