Le test de transformations de modèles : automatisation de l'oracle

Jean-Marie Mottu

sous la direction de MM. Benoît Baudry et Yves Le Traon

le 22 juin 2005

1 INTRODUCTION....4
2 CONTEXTE DU SUJET....5
2.1 INGENIERIE DES MODELES....5
2.1.1 Rappels sur UML et OCL....6
2.1.1.1 Exemple du diagramme de classes UML....6
2.1.1.2 OCL....8
2.1.2 Transformation de modèles....8
2.1.2.1 Principe....8
2.1.2.2 Mise en œuvre des transformations....9
2.2 LE TEST DE LOGICIEL....10
2.2.1 Le processus de test :....10
2.2.2 Génération de données de test....10
2.2.3 L'oracle....11
2.2.4 Critère d'arrêt....12
2.3 LE TEST DE TRANSFORMATION DE MODELES....12
3 ETAT DE L'ART....13
3.1 LES DIFFERENTES PHASES DE TEST DANS UN CYCLE DIRIGE PAR LES MODELES....13
3.1.1 Test de modèles UML....13
3.1.2 Test d'intégration de composants....14
3.1.3 Test de programmes issus de transformations....14
3.2 LE TEST DE TRANSFORMATIONS DE MODELES....14
3.3 ORACLE....15
3.4 CONCLUSION....15
4 PROCEDURE DE CREATION D'UN ORACLE AUTOMATIQUE....15
4.1 BESOIN D'ERREURS....16
4.2 TRAVAUX CONNEXES UTILISES : GENERATION DE DONNEES DE TEST....16
4.2.1 Métamodèle effectif....16
4.2.2 Partitions....17
4.2.3 Objectifs de données de test et critères de test....18
4.2.4 Génération de modèles....19
5 ANALYSE DE MUTATION....19
5.1 PRESENTATION....20
5.2 CREATION DES MUTANTS SPECIFIQUES AUX TRANSFORMATIONS DE MODELES....21
5.3 OPERATEURS DE MUTATIONS POUR LES TRANSFORMATIONS DE MODELES....22
5.4 JUSTIFICATION DE CES OPERATEURS....23
5.5 COMPLEXITE DE L'IMPLANTATION DE CES OPERATEURS....25

5.6 BILAN SUR LA MUTATION 26
6 ORACLE 26
6.1 UTILISATION DU METAMODELE 26
6.2 ORACLE PAR CONTRATS 26
6.2.1 Evaluation de l'utilisation des contrats pour la robustesse 26
6.2.2 Limites d'OCL 27
6.3 VALIDATION DE L'ORACLE 28
7 COMPARAISON DE MODELES 28
7.1 UTILISATION DANS L'ANALYSE DE MUTATION 29
7.2 UTILISATION COMME FONCTION D'ORACLE 29
7.3 SPECIFICITES DE LA TRANSFORMATION DE MODELES 29
7.3.1 Propriétés d'une comparaison de modèles 29
7.3.2 Éléments de solution basé sur les métamodèles et la sémantique 29
8 MISE EN APPLICATION 30
8.1 PRESENTATION DE L'ETUDE DE CAS 31
8.2 UTILISATION DE JAVA, EMF, KMF, MUJAVA 31
8.3 OUTILS CREES 31
8.3.1 Mutation 31
8.3.2 Comparaison ad hoc 32
8.4 EXPERIMENTATIONS 33
8.4.1 Etude des résultats 33
9 CONCLUSION 34
10 ANNEXES 36
11 RÉFÉRENCES 39

1 Introduction

Pour faire face à la complexité et à l'évolution rapide et croissante des applications logicielles, l'ingénierie des modèles (IDM) ouvre de nouvelles voies d'investigation. Dans ce cadre, un modèle n'est plus une simple image ou un élément de documentation, mais bien un élément productif qui doit pouvoir être traité automatiquement. Les modèles deviennent ainsi des entités centrales dans le développement logiciel. Ce nouveau processus de développement procède par transformations de modèles successives pour compléter, enrichir, préciser un modèle métier initial avec les nombreux aspects plus directement liés à l'environnement, la plateforme, les performances…. Ainsi les modèles simplifient la vision d'un système en permettant de représenter, de manière abstraite, un aspect du logiciel, et une transformation permet ensuite d'intégrer cet aspect dans le modèle global du logiciel en développement. La combinaison de ces 2 entités permet le développement simplifié de systèmes fiables, mêmes complexes. Cependant, la fiabilité et l'augmentation de la modularité apportée par l'ingénierie des modèles ne doivent pas être dégradés par des transformations erronées. Ainsi comme tout programme elles doivent être spécifiées, modélisées, implantées et validées. Pour cette étude, nous nous intéressons au test comme principale technique de validation.

Le but du stage que j'effectue dans l'équipe Triskell (IRISA/INRIA) est d'automatiser la fonction d'oracle dans le cadre du test de transformations de modèles. La fonction d'oracle permet de valider ou non les résultats obtenus par l'exécution des données de test avec le programme. L'automatisation du test de logiciel est une nécessité pour pouvoir réaliser un maximum de tests, les plus pertinents possibles et ainsi augmenter le niveau de confiance du système testé. Cette automatisation concerne la génération des données de test et l'oracle pour les cas de test. Ce dernier point constitue l'objet principal de mon stage. La génération automatique de données de test est un sujet d'étude de l'équipe Triskell, sur lequel j'ai également contribué au cours de mon stage.

L'étude de l'oracle ne peut se limiter à l'analyse des résultats des données de test exécutés par différents programmes sous test, il faut pouvoir contrôler que l'oracle détecte efficacement les fautes et pour cela il faut disposer de programmes comportant des erreurs dont nous avons la maîtrise. C'est pourquoi l'étude de l'oracle nécessite la création de programmes volontairement erronés. L'analyse de mutation est une technique qui consiste à créer des programmes comportant des erreurs injectées automatiquement selon des modèles de faute. Ces modèles doivent être adaptés au type de programmes étudiés. La première contribution de ce travail consiste à proposer des modèles de fautes pertinents pour valider des techniques de test de transformations de modèles.

L'automatisation de l'oracle est un sujet qui n'a pas été traité dans le cadre des transformations de modèles mais qui ne l'ai pas beaucoup plus pour le test de programmes plus classiques. Généralement l'hypothèse est faite de disposer du résultat attendu qui peut alors être comparé aux résultats obtenus par une fonction automatique. Cette hypothèse est aussi vraie pour certaines transformations de modèles, il faut ici comparer les modèles obtenus en sortie. La comparaison de modèles est un problème à part entière auquel nous nous sommes intéressés et pour lequel nous proposons des heuristiques qui permettent d'en simplifier la réalisation, c'est la deuxième contribution de ce stage.

Puisque l'oracle par comparaison des modèles de sortie n'est pas généralisable à l'ensemble des transformations de modèles, nous proposons d'exploiter la spécification des transformations de modèles sous forme de pré et post conditions. Ces assertions nous permettent de vérifier l'intégrité des modèles obtenus et de s'assurer de la concordance entre les modèles d'entrée et de sortie et le traitement qui a été réalisé. Cela forme la troisième contribution de ce stage.

Ces trois contributions forment la base de l'article « Génération automatique de tests pour les transformations de modèles » que nous avons écrite et qui a été accepté à la conférence IDM05[1].

Ce rapport est organisé de la façon suivante : nous présentons le contexte du sujet dans la section 2 puis nous étudions les travaux en rapport avec l'étude dans la section 3. La section 4 présente la procédure suivie pour la création de l'oracle. Les sections suivantes correspondent aux principales contributions avec en section 5 l'analyse de mutation, en section 6 l'oracle et en section 7 la comparaison de modèles. La section 8 présente la mise en application des techniques proposées et leurs évaluations. Finalement la section 9 conclue cette étude. Les annexes (section 10) et les références bibliographiques clôturent ce rapport (section 11).

2 Contexte du sujet

Le but de cette partie est d'introduire le contexte du sujet de stage, d'abord du point de vue de l'ingénierie des modèles, avec les notions de modélisation et de transformation de modèles, puis en présentant l'approche générale pour le test de logiciels. Il sera ainsi possible de situer sur quels points nous avons apporté nos contributions.

2.1 Ingénierie des modèles

Dans cette première partie nous nous intéressons aux nouvelles méthodes de conception logicielle dirigées par les modèles. Les systèmes logiciels actuels sont de plus en plus complexes, les différentes plates-formes qui les accueillent de plus en plus nombreuses et hétérogènes. Les coûts de développement induits suivent malheureusement cette évolution. Le nombre de lignes de code des systèmes augmente exponentiellement, mais une telle augmentation en terme de coût n'est pas envisageable. C'est pourquoi la réalisation des systèmes complexes pose problème et nécessite l'utilisation de nouvelles méthodes. Une solution émerge actuellement : le développement dirigé par les modèles. L'**OMG** (Object Management Group) a développé pour cela le langage de modélisation UML permettant de représenter un système de manière abstraite sous des aspects statiques aussi bien que dynamiques.

UML est un langage de modélisation destiné à décrire des modèles. Ce langage aussi peut être modélisé, grâce à son métamodèle. Ainsi l'ensemble de tous les modèles UML possibles est décrit par le métamodèle d'UML. Cependant, même s'il est le plus répandu, il n'est pas universel, dans le sens où il ne permet pas de tout modéliser. Dans sa dernière version il utilise bien la notion de profil qui permet de le moduler en fonction des besoins spécifiques d'un domaine, mais cela n'est pas réalisable d'avoir un langage unique [2].

Dans le but d'avoir d'autres métamodèles et pour éviter de fortes incompatibilités, l'OMG met en avant un environnement de travail basé sur le **MOF** (Meta-Object Facility) [3] qui est en fait un méta-métamodèle permettant de décrire les métamodèles. L'ensemble forme une architecture représentable en quatre couches comme illustré figure 1 par un exemple.

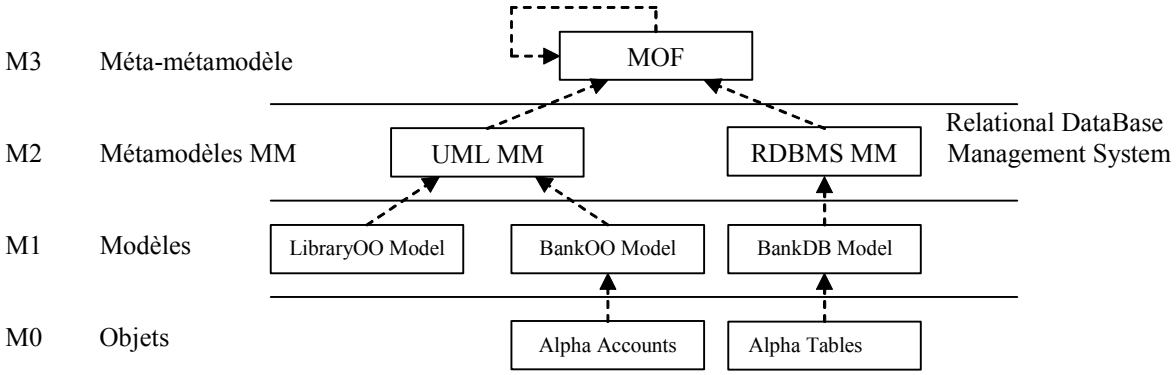


Figure 1 - Architecture 4 couches

Il s'agit d'une vision de la modélisation, métamodélisation donnée par l'OMG qui peut-être vue comme une formalisation d'approches déjà existantes [4]. On peut par exemple mettre au niveau M3 EBNF (Extended Backus-Naur Form) qui est utilisé pour écrire la grammaire d'un langage comme C (niveau M2) : au niveau M1 on aurait alors des programmes écrits en C, et au niveau M0 les exécutions spécifiques de ces programmes [5]. Cela n'est pas restreint au domaine de l'informatique : une carte par exemple est le modèle d'une région géographique particulière, sa légende est le métamodèle de la carte. [4]

En novembre 2000, l'OMG a introduit une nouvelle approche nommée **MDA** (Model-Driven Architecture) dont UML et le MOF sont les éléments centraux. Il s'agit de recentrer le développement des logiciels sur des modèles spécialisés en les séparant de l'implantation pour découpler les parties métiers et les parties technologiques dans les systèmes informatiques.

Le MDA introduit 2 niveaux de modélisation. Dans un **PIM** (Platform Independent Model), il s'agit de s'affranchir au maximum de l'implantation sur une plate-forme donnée et des contraintes et restrictions que cela implique pour rester au maximum au niveau fonctionnel du système. Cela permet aussi de pouvoir réutiliser les PIM et de capitaliser le savoir-faire métier indépendamment des technologies. Alors qu'un **PSM** (Platform Specific Model) est spécifique à une implantation et représente ses contraintes technologiques. Par des méthodes automatiques de transformation de modèles, le modèle du système est raffiné étape par étape vers l'implantation finale. On va donc chercher à élever le niveau d'abstraction grâce à la modélisation et à faciliter le travail des développeurs en leur fournissant des outils et méthodes automatiques de transformation de modèles.

L'ingénierie dirigée par les modèles (**IDM** ou **MDE** Model Driven Engineering) [4, 6] généralise le MDA. UML n'est plus considéré comme le standard central pour modéliser. On n'est pas non plus limité à des transformations destinées à se rapprocher de l'implantation, il peut aussi bien s'agir de remonter d'un langage concret vers un modèle plus abstrait (rétro ingénierie), ou de rester à un certain niveau sur le même système (réusinage). Cette nouvelle manière d'aborder le génie logiciel est censée apporter : une meilleure continuité entre spécification et réalisation, une meilleure communication entre les acteurs d'un projet, une meilleure résistance aux changements (de la spécification…)

L'OMG est en train de normaliser les langages de transformations de modèles par une technologie nommée QVT (Query View Transformation) qui fait l'objet d'une RFP (Request For Proposal) [7]. Plusieurs équipes de l'INRIA [8] s'intéressent à l'IDM et ont proposé des langages de transformation de modèles tel que **MTL** (développé au sein de l'équipe Triskell) ou ATL (par l'équipe Atlas), des industriels comme France Télécom (qui propose le langage TRL conforme à QVT2) travaillent aussi dans ce domaine.

Les perspectives sont intéressantes : la modélisation permet d'abstraire la vision du système conçu, les modèles et les transformations sont facilement réutilisables, certaines phases de développement peuvent être simplifiées. On peut par exemple simplifier la réécriture d'un programme pour le porter sur un autre système si l'on dispose d'une transformation qui va produire une implémentation à partir d'un modèle métier du programme.

L'approche orientée objet avait constitué un profond changement, par rapport au procédural, dans la façon de concevoir des systèmes. Beaucoup [2, 4, 9] croient que l'IDM peut constituer un saut technologique comparable. Nous avons placé notre étude dans ce contexte d'ingénierie des modèles.

2.1.1 Rappels sur UML et OCL

Standardisé par l'OMG en 1997, le premier but d'UML était d'unir le travail de différents corps de métier qui utilisaient leurs standards spécifiques. Cela entraînait des problèmes de communication entre eux. UML (Unified Modeling Language) est donc le résultat d'une fusion entre plusieurs langages d'analyse orientées objets (comme OOA-OOD, OMT, Use-Case…). UML s'intègre donc dans le développement de projets de divers domaines et particulièrement informatique.

UML reste un langage de modélisation parmi d'autres. Nous l'abordons ici car il est tout de même un standard dans l'ingénierie des modèles. Il sert de plus pour les expérimentations et nous permet d'illustrer les notions de métamodélisation.

2.1.1.1 Exemple du diagramme de classes UML

Le diagramme de classes est le diagramme principal dans le langage UML. Il permet de décrire statiquement les classes dans un système et leurs relations entre-elles. La Figure 2 est un extrait du métamodèle d'UML, il ne concerne qu'une partie du diagramme de classes : cela permet de créer des diagrammes comportant des classes (`Class`) ayant des associations entre elles (`AssociationEnd`, `Association`), des attributs (`Attribute`) et des tags (`Stereotype`), avec les notions d'héritages (`GeneralizableElement`, `Generalization`)

A partir de cet extrait du métamodèle, il est possible d'écrire des diagrammes d'instances de modèles UML comme par exemple celui de la Figure 3. Il s'agit d'une classe Livre avec un attribut `titre` de type `String`, liée par l'association `étatCourant` à la classe `EtatLivre` dont hérite la classe `StateReliure`.

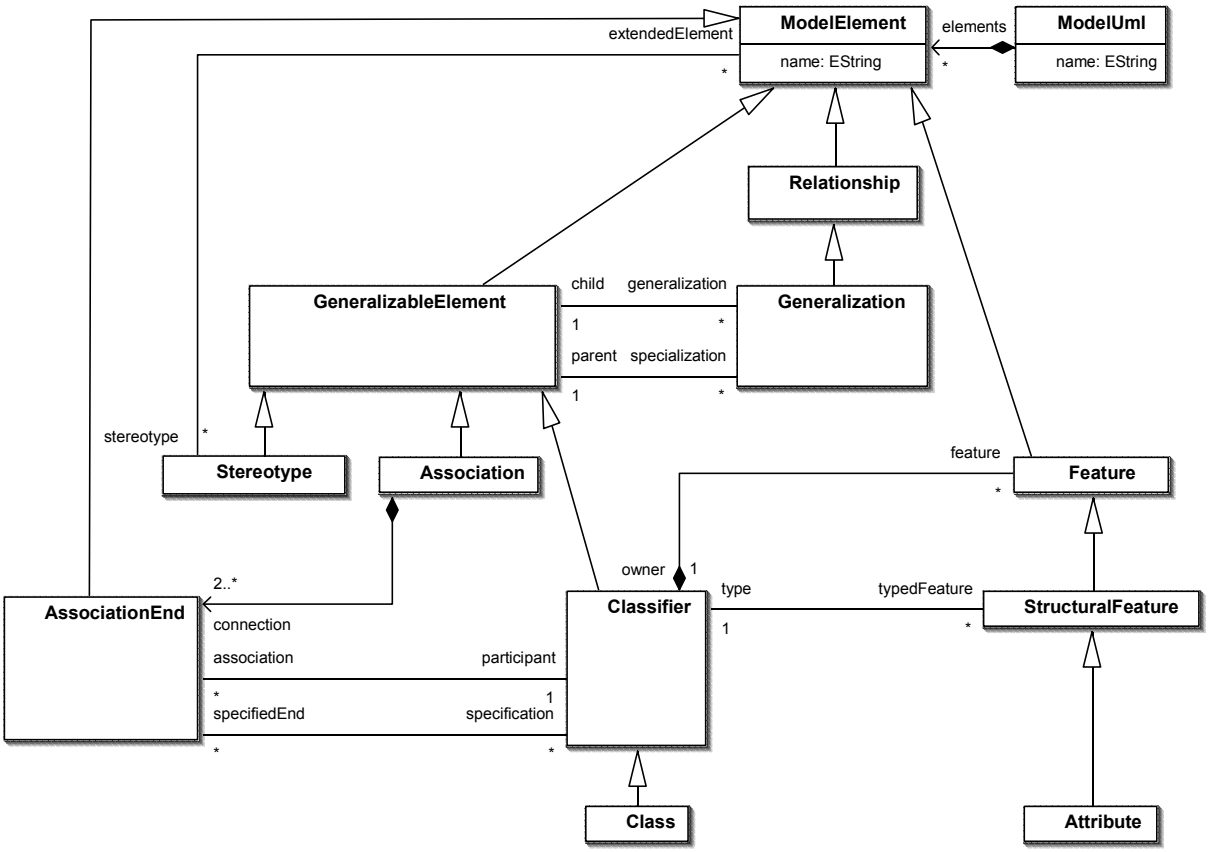


Figure 2 - Extrait du métamodèle d'UML

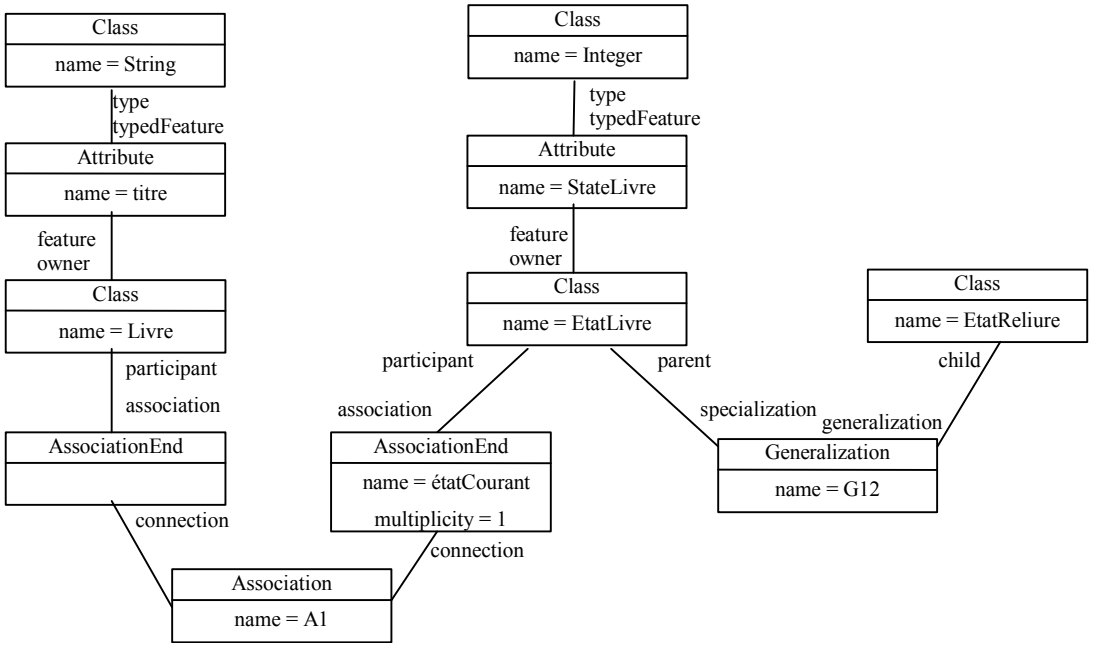


Figure 3 - Exemple de diagramme d'instances UML

La Figure 4 montre la notation UML qui correspond à ce diagramme d'instances et qui est nettement plus lisible que ce dernier.

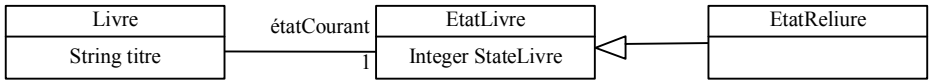


Figure 4 - Exemple de diagramme de classes UML

*2.1.1.2 **OCL***

Le langage **OCL** [10] (Object Constraint Language) est un langage formel pour exprimer des contraintes sur les diagrammes UML. Il permet par exemple d'écrire des invariants de classe, des pré et post conditions pour des méthodes, des gardes. Cela permet de transposer les principes de la conception par contrat défini par Meyer [11] dans un cadre de modélisation.

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 un type, soit une opération.

Prenons un exemple basique:

context CompteBancaire ::créditer(montant : **Integer**)
pre : montant > 0
post : solde = solde@pre + montant

Ici le contexte est l'opération `créditer` de la classe `CompteBancaire`. La pré condition impose que le montant crédité soit positif et la post condition que le solde du compte après opération soit égal au solde avant l'opération plus le montant.

Le formalisme introduit par OCL est intéressant et préférable au langage naturel (il est toutefois utile de conserver des commentaires en indication) car il supprime les ambiguïtés et permet d'être exploité de façon automatique dans les programmes.

Ainsi le couple UML/OCL permet relativement facilement, de façon claire et non ambiguë de modéliser un système à partir de sa spécification. Des études préliminaires d'évaluations de la spécification sont alors possibles avant d'avoir commencé l'implantation. L'intérêt est donc important dans le développement des systèmes actuels qui sont toujours de plus en plus compliqués et dont les spécifications varient souvent. A ce niveau (modèle) les changements sont plus simples et seront plus faciles à reporter dans l'implantation.

2.1.2 Transformation de modèles

Les transformations de modèles forment une base de l'approche MDE, puisqu'il s'agit d'automatiser la transformation d'un modèle source en un modèle cible. La Figure 5 schématise ce processus.

2.1.2.1 Principe

Disposant d'un modèle d'entrée décrit par un métamodèle, la transformation produit un modèle de sortie décrit par son métamodèle. Les métamodèles d'entrée et de sortie peuvent être identiques : il est possible de faire des transformations d'un modèle UML en un autre modèle UML par exemple (surtout qu'UML peut décrire aussi bien un PIM qu'un PSM). Une spécification de la transformation est disponible et nous faisons l'hypothèse réaliste suivante: la spécification est définie de manière formelle ou pseudo formelle. Une spécification formalisée peut être définie par des pré et des post conditions en OCL (nous verrons au 6.2 que c'est un élément important pour l'oracle). Le premier travail du testeur est de formaliser la spécification.

L'intérêt des transformations de modèles est important dans le processus de développement. Il s'agit d'établir un premier modèle à partir de la spécification qui sera indépendant de la plate-forme (PIM). Ce modèle peut par exemple être affiné (à l'aide de transformations ou pas). Ensuite intervient la transformation vers un autre modèle qui sera dépendant d'une architecture cible (PSM). Ce modèle pourra être amélioré en déterminant par exemple les lacunes dans la spécification. Par une suite de transformations, on peut finalement obtenir le code source implantable. On sera ainsi passé de la spécification à l'implantation de façon automatique, qui est l'objectif principal des transformations de modèles tel qu'il est défini dans le MDA.

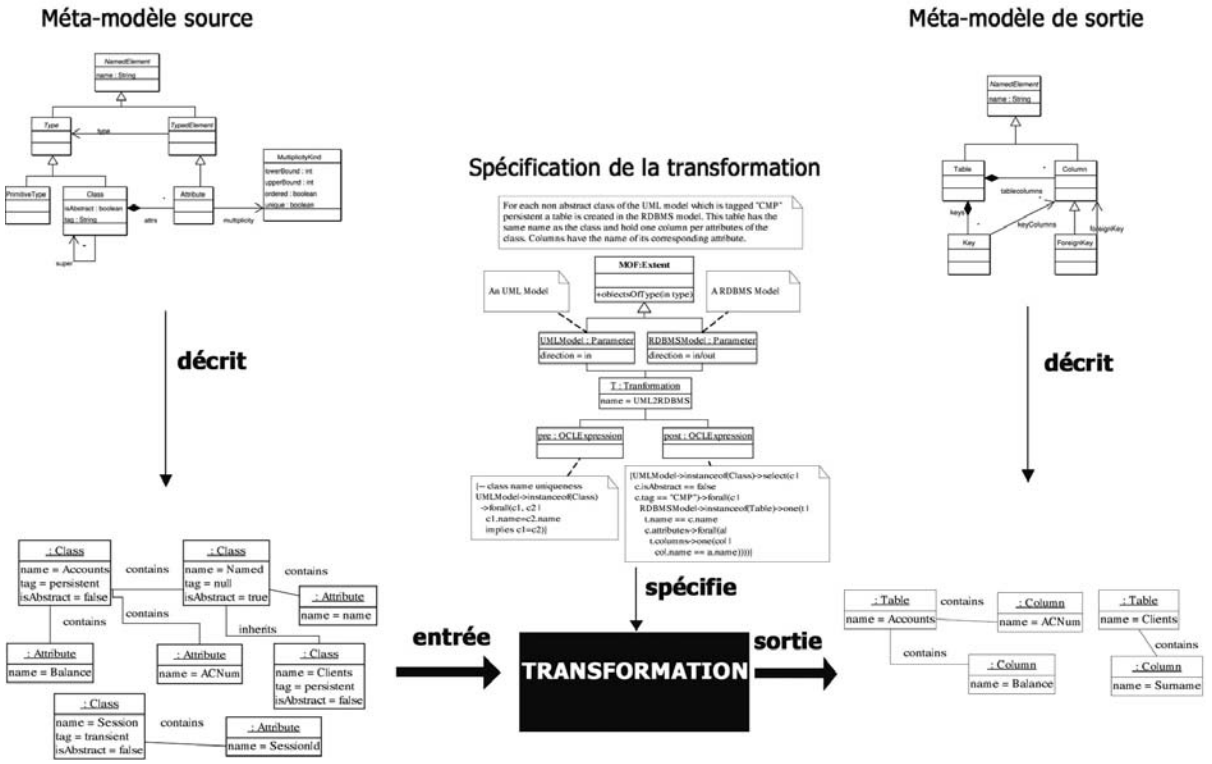


Figure 5 - Transformation de modèles

Cependant les applications des transformations de modèles ne sont pas uniquement du type PIM vers PSM mais peuvent par exemple être de PSM vers PIM. Il s'agit alors de « reverse engineering » (ou retro ingénierie). Nous pouvons enfin avoir des transformations de PIM vers PIM ou de PSM vers PSM (la génération de code appartient à cette catégorie).

Les transformations de modèles offrent d'autres intérêts:

- Pour appliquer un design pattern (patron de conception) par exemple qui est connu sous un métamodèle particulier mais pas avec celui de notre modèle.
- Pour simplifier un changement de plate-forme : on prend le programme existant, par le reverse engineering on obtient un modèle PIM et avec la spécification de la nouvelle plate-forme on peut créer une transformation pour obtenir le programme final voulu. L'opération peut être répétée automatiquement pour tous les programmes à adapter.
- Pour effectuer du réusinage (refactoring) : tout en restant à un même niveau (PIM…) on peut appliquer des transformations à un modèle, par exemple pour le rendre plus facilement testable ou réutilisable.

2.1.2.2 Mise en œuvre des transformations

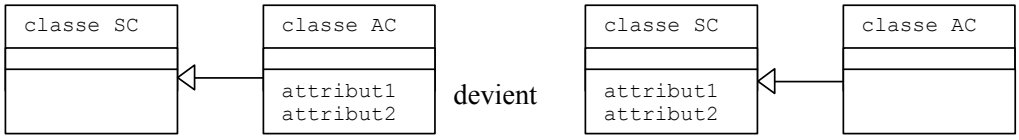
Dans [6], l'approche réflexive est proposée, qui consiste à appliquer les concepts du MDA aux transformations de modèles. Il s'agit donc de commencer par définir une transformation abstraite qui soit indépendante de toute plateforme : PIT (Platform Independant Transformation). Il est ensuite possible de dériver de façon plus ou moins automatique une PST (Platform Specific Transformation) qui soit spécifique à l'environnement de modélisation utilisé. La transformation elle-même est un modèle, décrite par son métamodèle. 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, surtout associé à OCL. Cependant, leur restriction en matière d'exécutabilité est un inconvénient : c'est pourquoi il est proposé de leur rajouter (en particulier à OCL) des fonctionnalités exécutables [2, 12].

Dans tous les cas, de manière générale, une transformation sera créée à partir des métamodèles des modèles d'entrée et de sortie ainsi que d'une spécification de la transformation à effectuer. En paramètre nous aurons le modèle source et en résultat le modèle cible.

Un point de discordance n'a pas encore été résolu : le choix entre une orientation impérative ou déclarative. Les propositions qui ont fait suite à la RFP OMG/QVT l'ont montré. L'approche déclarative permet de définir un ensemble de règles en utilisant les pré-conditions sur le modèle source et les post-conditions qui contraignent le modèle de sortie. Cette technique peut permettre de définir des transformation mais qui ne pourront pas être très complexes [6].

Le choix de l'impératif semble avoir été retenu pour la très prochaine norme QVT2 et l'équipe Triskell avait fait ce choix pour son langage MTL [8]. **MTL** est un langage résolument orienté objet qui se base sur OCL pour pouvoir naviguer dans les modèles tout en en y ajoutant les effets de bord permettant de les modifier.

Prenons un exemple simple qui consiste à remonter les attributs (features) d'une classe dans sa super-classe (la classe dont elle hérite) :



Il faut pour cela associer chaque attribut de la classe fille à la classe mère, puis les dissocier de la classe fille, ce qui s'écrit par la fonction moveFeature en MTL :

```
moveFeature( AC : UML::Core::Class , SC : UML::Core::Class)
{
     foreach(f : UML::Core::Feature) in (AC.feature)
     {
        associate(owner = SC:UML::Core::Class , feature= AC:UML::Core::Class);
        dissociate(owner= SC:UML::Core::Class , feature= AC:UML::Core::Class);
     }
}
```

2.2 Le test de logiciel

Un programme, quel qu'il soit, a besoin d'être validé tout au long de son cycle de vie, et pas uniquement dans le cas de logiciels critiques. Un des principaux buts de l'approche MDA est la réutilisabilité, aussi est-il important qu'un composant qui va se retrouver dans de nombreux logiciels ait été vérifié et validé. Dans notre étude, la validation par le test nous intéresse et nous l'introduisons dans cette partie.

2.2.1 Le processus de test :

Le processus de test peut être schématisé, comme illustré Figure 6:

Le processus de test commence par la génération de **données de test**. Elles sont exécutées sur le programme à valider (appelé **programme sous test**), ce qui donne un résultat. La fonction d'**oracle** valide ou non ce résultat. Si le résultat ne correspond pas à ce qui est attendu, il faut corriger le programme. Sinon le jeu de test est réussi. Le critère d'arrêt peut demander que d'autres tests soient effectués ; il dépend de la qualité attendue du logiciel. Il pourrait au pire exiger une étude exhaustive.

Cette partie forme le **processus de test** qui permet de détecter les *défaillances* du programme, il faut la distinguer de la partie mise au point qui permet de localiser les *fautes* et de corriger les *erreurs*.

Dans la suite nous détaillons chaque activité du processus.

2.2.2 Génération de données de test

Le test exhaustif n'est pas envisageable, soit parce qu'il n'est pas réalisable (espace d'entrée infini : les entiers par exemple), soit parce qu'il est trop coûteux (combinatoire…). Il est donc nécessaire de mettre au point des techniques pour la génération de cas de test. Ces techniques peuvent être classées selon de 2 approches [13]:

- le test **structurel** (ou boite blanche) qui tient compte de la structure interne du programme sous test

- le test **fonctionnel** (ou boite noire) qui ne tient pas compte de la structure du programme mais de sa spécification, cette technique est retenue pour l'instant dans les travaux de l'équipe Triskell pour la générations de données de test pour les transformations de modèles (abordés au 4.2). Les travaux réalisés au cours de ce stage se concentrent sur l'oracle mais présentent aussi des techniques de génération de donnée de test que nous avons aussi étudiés.

Une fois que les données de test sont générées, il est nécessaire de vérifier leur pertinence, l'analyse de mutation, que nous étudions en détail partie 5, a cette fonction.

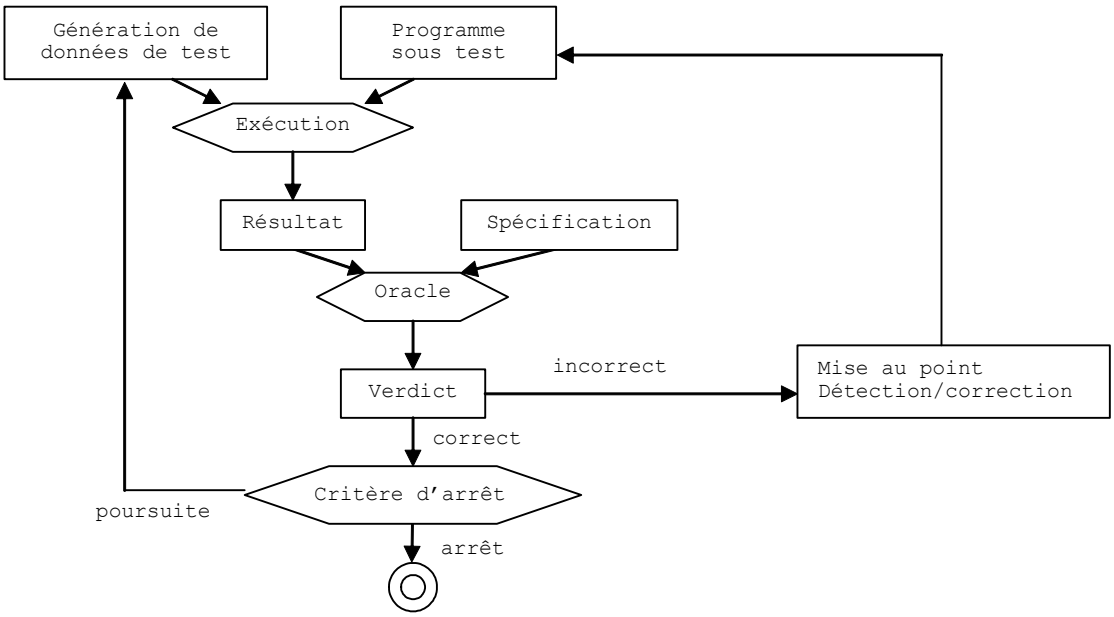


Figure 6 - Processus de test

2.2.3 L'oracle

Une fois que les données de test ont été obtenues et appliquées au programme sous test, nous obtenons un résultat. L'***oracle*** est la fonction qui détermine si le résultat obtenu est correct ou non. Une première approche consiste à calculer le résultat attendu à partir des spécifications et à le comparer au résultat obtenu avec les données de test. Il est aussi parfois utile d'utiliser un même logiciel disponible mais sur une plate-forme différente et alors l'oracle au niveau du test système sera une simple comparaison des résultats (encore faut-il faire confiance au premier logiciel). A part dans les cas de changement d'implantation (un logiciel est adapté à une autre plate-forme), il n'est généralement pas possible de connaître à l'avance le résultat souhaité avec exactitude.

L'oracle peut être manuel ou automatique. Chaque cas de test va donner un résultat qu'il faudra vérifier. Dès qu'il est nécessaire d'avoir de nombreux cas de test, il faut disposer d'un oracle efficace et automatique pour soulager le testeur.

L'oracle automatique est un problème à part entière. Malheureusement dans de nombreuses études du test, l'hypothèse est faite soit de disposer de l'oracle, soit de disposer du résultat attendu à l'avance.

L'oracle peut être défini ainsi: si l'on teste un programme P créé à partir d'une spécification F. Si In(P) et Out(P) sont les domaines d'entrée et de sortie. Si $X \in In(P)$ alors l'oracle est la fonction ok_F tel que :

$$ok_F : In(P) \times Out(P) \rightarrow Boolean$$

$$ok_F(X , P(X)) \text{ ssi } P(X) = F(X)$$

Le cas le plus simple est le cas des fonctions inversibles. Si nous appliquons la transformation inverse à notre résultat et que nous retrouvons notre donnée de test, alors ce cas de test est vérifié :

$F^{-1}(F(X)) = X$?

Il est possible par exemple de vérifier qu'une racine carrée est correcte en la remettant au carré ou de contrôler l'inversion d'une matrice A si $A.A^{-1}$ = matrice identité. Mais l'inversibilité n'est pas systématique, loin de là, et rarement dans le cas des transformations de modèles.

2.2.4 Critère d'arrêt

Le critère d'arrêt fixe une limite au processus de test. Il y a généralement une infinité de cas de test (dès qu'une variable d'entrée à son domaine infini) ou tout au moins un nombre trop important pour que le test exhaustif soit envisageable. Le critère de test sert donc d'objectif pour le test, quand il sera rempli, nous aurons atteint un degré de confiance suffisant dans notre programme. Le critère d'arrêt est lié à la génération des données de test, il est lui aussi soit fonctionnel, soit structurel. Les critères fonctionnels ne tiennent pas compte de l'implantation mais seulement de la spécification. Les critères structurels se servent du code, ils peuvent imposer la couverture du code par exemple. L'analyse de mutation est une technique pouvant servir de critère d'arrêt, elle est étudiée au 5.

La couverture de code, des prédicats, des chaînes définition/utilisation … sont des exemples de critères structurels; la couverture des scénarios au niveau système, des cas d'utilisation … sont des exemples de critères fonctionnels.

2.3 Le test de transformation de modèles

Les transformations de modèles sont effectuées par des programmes qui doivent suivre un processus de développement classique, comme illustré Figure 7 :

- Nous commençons par le développement d'un environnement de travail spécifique.
- Un développeur va ensuite créer une transformation particulière.
- Enfin l'utilisateur va pouvoir appliquer cette transformation sur ses modèles.

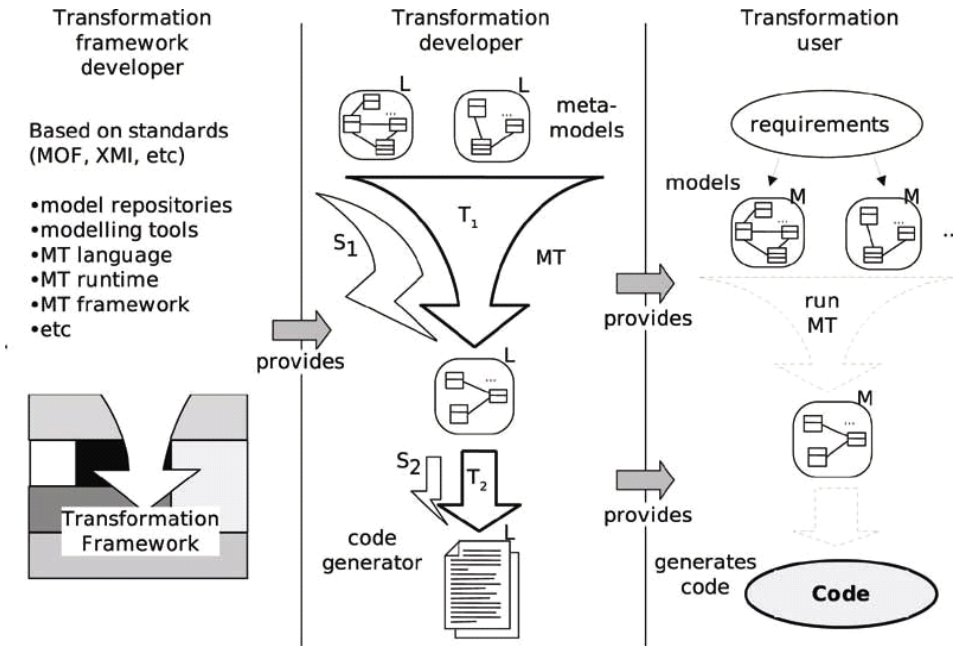


Figure 7 - Séparation des rôles en ingénierie des modèles

Si l'utilisateur final constate des fautes dans le code généré, 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 validation pendant son développement comme tout système logiciel. Cette phase de validation 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, arriver à ce point elle sera très difficile à localiser.
- 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 on risque de générer les mêmes erreurs un certain nombre de fois.

Cela justifie la réalisation de phases de tests [6, 9] et des études sont actuellement en cours (en particulier dans l'équipe Triskell) car on ne dispose encore d'aucun outil pour le faire. 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 métamodèles et les données d'entrée sont des modèles. Nous sommes donc confrontés à de nouveaux problèmes pour la génération des données d'entrée et le contrôle des données de sorties (ce qui nous intéresse particulièrement).

Notons aussi qu'il y a une autre différence majeure entre le test classique et le test de transformations de modèles : en test classique les données en entrée et en sortie n'ont plus rien à voir (leur seul lien est la méthode qui a permis de passer de l'une à l'autre), alors qu'en transformation de modèles on a 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, du moins partiellement. La conservation de la sémantique des modèles au cours d'une transformation est un sujet d'étude encore ouvert.

3 Etat de l'art

Dans la suite nous étudions les quelques travaux qui concernent le test, d'abord dans un contexte de développement dirigé par les modèles, puis pour les transformations de modèles et enfin spécifiquement pour l'oracle.

3.1 Les différentes phases de test dans un cycle dirigé par les modèles

Dans [14] le problème du test dans un cadre MDA est traité, dans le but de proposer une méthode de test dans un contexte de modélisation. L'attention est ainsi portée sur la séparation entre PIM et PSM pour le test. Les générations des données de test et de l'oracle sont faites à partir des modèles et sont considérées comme indépendantes de la plate-forme. Par contre l'exécution des tests est spécifique à la plate-forme.

La fonction d'oracle nous intéresse particulièrement, elle consiste ici en un modèle exécutable. C'est pourquoi il est proposé de rajouter au langage UML des fonctionnalités pour qu'il puisse remplir ce rôle : il s'agit d'intégrer dans une même vue les aspects statiques, dynamiques et l'effet d'opérations agissant sur les données.

3.1.1 Test de modèles UML

Certains travaux utilisent les diagrammes UML décrivant un programme (comme d'une spécification) pour générer des données de tests pour ce programme [15]. Dans [16] , il est proposé de tester le modèle UML, il est toutefois nécessaire de le mettre dans une forme exécutable [2]. Il s'agit de déterminer des critères de test basés sur les diagrammes de classes et d'interaction/collaboration UML pour couvrir le modèle. Plusieurs critères sont proposés et étudiés ; pour le diagramme de classes par exemple il y a :

- Critère sur les multiplicités des associations (Association-End) : il s'agit d'intervalles, par exemple pour [0..n] on prend les limites 0 et n et une valeur entre. Un produit cartésien est effectué pour obtenir un ensemble de couples à partir des 2 extrémités de l'association.

- Critère sur les attributs de classe : chacun peut prendre un ensemble de valeur défini par leur type et parfois des contraintes OCL, il faut alors prendre des valeurs significatives et effectuer un produit cartésien.
- Critère sur l'héritage : quand on veut utiliser une classe, on doit pouvoir lui substituer un de ses fils, ces différents cas doivent être étudiés.

Cette technique de partitionnement permet d'obtenir un vaste ensemble de cas de test couvrant les cas d'utilisation les plus sensibles. Nous pouvons ainsi obtenir à partir de diagrammes UML des données de tests intéressantes malgré le niveau d'abstraction élevé de la représentation. C'est-à-dire que dès les premières phases d'un projet (puisque c'est surtout à ce niveau d'avancement qu'est réalisée la modélisation) il est déjà possible de commencer à valider l'application. Cela évite de se rendre compte trop tard (au moment de l'implantation) que des erreurs ont été commises dans les fondements du système.

3.1.2 Test d'intégration de composants

Même si un composant passe avec succès la phase de test unitaire, rien n'indique que son intégration dans un système, impliquant des rapports avec d'autres composants, sera réussie. Dans un cadre client-serveur, le serveur agissait peut-être différemment dans l'environnement où il a été créé, ou le client peut s'attendre à des résultats de mêmes syntaxes mais dont le sens n'est pas le même. Dans [17], Atkinson et al. montrent l'intérêt d'intégrer dans le composant des contrats qui vont permettre d'en tester l'intégration. Le but étant de permettre au composant de vérifier son intégration et l'environnement dans lequel il est intégré. Cela renforce l'intérêt d'utiliser et de réutiliser des composants puisque cela assure un maximum de fiabilité, un composant est capable de signaler s'il est mal utilisé, s'il ne peut pas assurer les taches qu'on lui confie. Cela diminue encore plus le coût du développement d'un système malgré le coût que cela implique en amont. Les composants étant plus robustes, ils seront davantage utilisés, ce qui économisera des phases de développement. Cette technique présente aussi l'avantage de diminuer la phase de test système.

3.1.3 Test de programmes issus de transformations

Modest [18] est un outil destiné principalement à générer du code mais aussi des données de tests, cette fonction étant le propos de cet article. Le but de cet outil est de tirer parti de la modélisation et du niveau d'abstraction qu'elle apporte en générant automatiquement le code de l'application. Sans être conforme au MDA, cela s'inscrit dans un contexte MDE.

Modest permet de créer à partir de spécifications modélisées, un système qui n'est pas final mais qui va être enrichi et spécialisé en fonction des besoins de l'utilisateur. Les tests sont aussi générés et fournis pour faciliter et contraindre l'évolution du système.

L'évaluation montre que le coût de la génération de données de tests est faible comparé à celui du code de l'application. Cela permet d'en générer un maximum et d'en tirer différents avantages : validation de la partie créée par Modest, formation d'une base de tests utile pour faire évoluer et maintenir l'application.

3.2 Le test de transformations de modèles

Si les transformations de modèles sont un sujet dont l'étude s'est bien développée, il n'en est pas encore de même de la validation de ce type de programme avec seulement quelques travaux.

Dans [9] il est proposé de transposer l'approche de [16] pour le test de transformation de modèles dans le cas d'une approche de test fonctionnel. En effet une transformation de modèles est créée à partir du métamodèle du modèle à transformer. Puisque les métamodèles 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 [9] peuvent être utilisés pour couvrir les métamodèles. Ceci nous donne des critères pour la génération de données de test pour la transformation. Une fois que la transformation a été exécutée sur les modèles d'entrée, il va falloir valider les modèles résultats grâce à un oracle. C'est cette partie validation qui manque à cette étude et qui le but de mon stage.

Dans [19], les auteurs présentent les problèmes qu'ils ont rencontrés lors du développement de leur outil de transformation de modèles et les solutions qu'ils ont pu apporter. Ils notent la similarité entre cette activité et

celle de tester les transformations elles-mêmes, et présentent un certain nombre de techniques qui se rapportent à l'utilisation des modèles en tant que données de test. L'utilisation de critères de couverture pour la génération de données de test est proposée comme une possibilité et appliquée manuellement dans leur étude.

Dans [20], Lin et al., identifient ce qui doit être résolu dans le test de transformation de modèles, et proposent un cadre de travail pour cela. Le problème de la génération de données de test n'est pas étudié, les auteurs se focalisent particulièrement sur le problème de la comparaison de modèles propre à l'oracle. Ils proposent un premier algorithme s'appuyant sur la comparaison de graphes. Le problème de la comparaison de modèles est aussi abordé dans [21] dans un contexte d'outil de contrôle de version de modèles, plusieurs algorithmes indépendants des métamodèles sont proposés pour calculer les différences entre modèles.

Dans [22], Küster considère les transformations à base de règles et le problème de la validation de ces règles qui définissent la transformation de modèles, c'est-à-dire la correction et la complétude de l'ensemble des règles.

3.3 Oracle

Certains travaux s'appuient sur la spécification pour générer l'oracle. C'est le cas dans [23] où est décrit comment générer un oracle à partir d'une spécification formelle. Le travail du développeur ou du testeur sera d'écrire la spécification du programme de façon formelle. Un formalisme précis et compréhensif est proposé pour cela. L'essentiel de la méthode consiste ensuite à convertir la spécification dans une forme exécutable.

La fonction d'oracle peut aussi s'appuyer sur les assertions pour. La conception par contrat a été introduite par Meyer, elle permet d'assurer la qualité au niveau unitaire (les post-conditions contraignent une partie de code) et de se rendre compte de problème d'intégration (les pré-conditions révèlent des problèmes dans ce qui les précèdent). L'étude faite dans [24], qui se fonde principalement sur une analyse de mutation (le principe est expliqué dans la partie 5), montre que dans une certaine mesure les contrats peuvent servir de fonction d'oracle. En effet l'amélioration des contrats montre une augmentation du nombre de mutants tués, ce qui signifie que plus d'erreurs sont signalées grâce aux contrats. Ce sont aussi les contrats qui sont utilisés dans [16].

Dans [25], 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.

3.4 Conclusion

Cette étude tend bien à soutenir l'importance du test dans chaque étape du développement logiciel. Les transformations de modèles qui vont certainement prendre une place de plus en plus importante dans le procédé doivent être soumises à une phase de validation.

L'importance de la spécification et la possibilité de la formaliser sont d'intéressantes pistes pour l'élaboration de l'oracle. Nous constatons aussi que le caractère exécutable qui manque actuellement au MDA sera une nécessité aussi bien pour les transformations [12] elles-mêmes que pour leur test.

4 Procédure de création d'un oracle automatique

L'oracle pour le test de transformation de modèle consiste, comme dans le cadre de programmes classiques, à confirmer ou réfuter le résultat d'un test. Cette donnée obtenue en sortie est le point essentiel que l'oracle va analyser. D'autres points doivent être pris en compte comme la donnée d'entrée et la transformation elle-même.

Sur quoi va-t-on baser l'oracle dans le test de transformations de modèles ? On cherche à savoir si le modèle produit par la transformation de modèles correspond à celui attendu. On a vu dans la partie introduisant l'oracle (2.2.3) qu'on ne dispose pas systématiquement d'une prédiction de la sortie qu'il suffirait de comparer au résultat. En effet, cela revient à exiger qu'en l'absence de modèle attendu, le testeur soit contraint de l'écrire à la main pour le comparer au modèle obtenu. Le gain apporté par l'automatisation de la génération des données

(modèles) de test serait perdu avec un oracle manuel. Il faut donc pouvoir contrôler la validité des modèles obtenus sans dépendre de l'existence d'un modèle attendu, et si possible de façon automatique.

4.1 Besoin d'erreurs

Le but de l'oracle est la détection des erreurs, aussi notre matière est-elle donc un ensemble de programme dans lequel il y a des erreurs censées être détectées. Il faut que l'oracle révèle la présence d'erreur quand il y en a et apporte une confirmation quand il n'y en a pas (complétude et correction). Pour disposer d'un cadre d'expérimentation scientifique de l'efficacité des oracles, le premier problème qui se pose est de disposer de programmes incorrects sur lesquels nous ayons un contrôle des erreurs qui s'y trouvent. Dans un premier temps, il faut créer nos propres programmes de transformation de modèles erronés, contenant des erreurs dont nous avons la maîtrise.

La technique la plus appropriée pour l'insertion d'erreur est l'analyse de mutation qui est présentée dans la partie 5. Cependant l'analyse de mutation n'a pas encore été étudiée dans le cadre de programmes de transformation de modèle, nous avons donc pris en compte ce nouveau besoin qui est une des principales contributions de notre étude. La première partie du stage a donc été consacrée à l'étude et la réalisation d'une adaptation de l'analyse de mutation dans le cadre des transformations de modèles.

4.2 Travaux connexes utilisés : Génération de données de test

Le premier élément nécessaire dans un processus de test est un ensemble de données de test qui servent en entrée des programmes sous test. Cette section aborde donc le problème de la génération de données de test pour les transformations de modèles qui est un sujet étudié simultanément dans l'équipe Triskell. L'approche proposée définit des critères de test fonctionnels qui s'appuient sur la description du domaine d'entrée par un métamodèle. Le test fonctionnel a été choisi pour proposer une technique indépendante du langage de transformation utilisé. Les critères identifient, sur ce métamodèle, des structures d'objets intéressantes pour le test, appelées *objectifs de données de test* (ODT), et qui doivent être présentes dans les modèles d'entrée. Un partitionnement sur les types simples et les cardinalités du métamodèle permet de définir des valeurs pour les attributs de ces ODT.

La Figure 8 décrit les différentes étapes que nous avons identifiées dans [9] :

1 *du métamodèle effectif aux partitions*. Cette étape permet de construire et d'identifier les classes d'équivalence pour toutes les propriétés de type simple du métamodèle. Ces classes d'équivalence sont calculées sur la partie du métamodèle d'entrée qui est effectivement utilisée dans la transformation (appelé métamodèle effectif).

2 *des partitions aux objectifs de données de test (ODT)*. A partir des classes d'équivalence identifiées au cours de l'étape 1, il est possible de définir des ODT. Les critères de test proposés pour les transformations de modèles sont tous fondés sur la notion d'ODT.

3 *des objectifs aux modèles*. Des modèles contenant tous les ODT identifiés à l'étape 2 sont générés.

Ces étapes sont détaillées dans la suite de cette section.

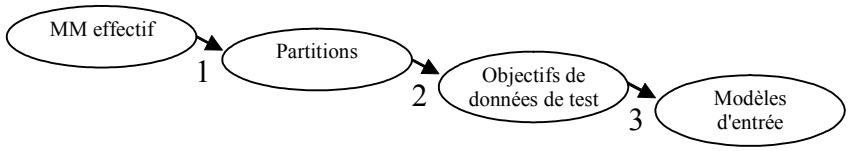


Figure 8 - Processus pour la génération de modèles d'entrée

4.2.1 Métamodèle effectif

Dans de nombreux cas, le domaine d'entrée défini par le métamodèle d'entrée d'une transformation est plus large que nécessaire; autrement dit, la transformation ne s'applique souvent que sur un sous-ensemble du métamodèle qui est appelé *métamodèle effectif*.

Ce manque de précision au niveau de la spécification n'est pas un problème dans le cadre de l'exécution d'une transformation. En revanche, dans une perspective de test, cette réalité est beaucoup plus gênante. En effet, le but étant de couvrir le domaine d'entrée, il est clair que restreindre le métamodèle d'entrée permet de limiter grandement le nombre de données de test à générer et, du même coup, le temps de calcul. Le métamodèle effectif peut être vu comme le type des données d'entrée d'une transformation. Dans [9] les auteurs donnent plusieurs pistes pour identifier automatiquement le métamodèle effectif. Ce point n'est pas abordé plus en détail ici.

4.2.2 Partitions

Pour couvrir l'ensemble du domaine d'entrée d'une transformation, une adaptation du test par partition présenté dans [26] est effectuée. Cette technique consiste à définir un ensemble de classes d'équivalence pour le domaine d'entrée du logiciel à tester, puis à choisir une donnée de test tirée dans chacune des classes. Dans le cas du test de transformation de modèles, une partition est définie pour chaque propriété (au sens du MOF) de chaque classe du métamodèle d'entrée.

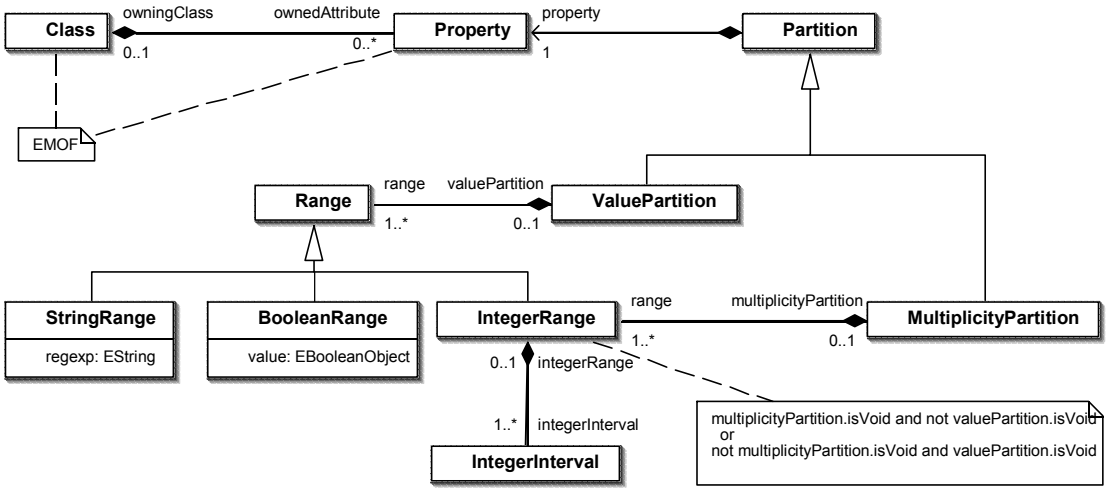


Figure 9 - Le métamodèle Partition

Définition – Partition. *Une partition Part d'une propriété Pr d'une classe C est un ensemble de classes d'équivalence CE sur l'ensemble des valeurs de Pr. Une partition est notée : Part(C::Pr) = { CE_i | $i \in [1..\#classesEq]$} où #classesEq est le nombre de classes d'équivalence pour Pr.*

Deux types de partition sont définis. Le premier partitionne les types primitifs (ValuePartition). Le deuxième partitionne les multiplicités, c'est-à-dire le nombre d'éléments de listes composées d'instances de type primitif (attribut liste) ou d'instances de classe (référence). Dans ce cas, le type partitionné est un sous-ensemble des entiers naturels. La Figure 9 décrit le métamodèle pour les partitions sur un métamodèle.

Dans [9] les auteurs proposent deux politiques permettant d'identifier une partition pour une propriété Pr. La première, le *partitionnement par défaut*, consiste à définir à priori une partition basée sur la structure ou le type de la propriété Pr. Le Tableau 1 donne un exemple succinct de partitionnement par défaut. La deuxième, le *partitionnement contextuel*, consiste à extraire des valeurs représentatives du type de la propriété partitionnée, à partir de la spécification de la transformation et/ou du code (white box testing). Cette politique raffine le partitionnement par défaut.

Chaîne	{ {null}, {""}, {s tel que |s| > 0} }
Entier	{]-∞, -2], {-1}, {0}, {1}, [2, +∞[}
Booléen	{ {vrai}, {faux} }

Tableau 1 -Un exemple de partitionnement par défaut

4.2.3 Objectifs de données de test et critères de test

Un concept important pour la définition des ODT est le concept de fragment d'objet défini ci-dessous. Les objectifs de données de test sont définis comme un ensemble de fragments d'objet. Le métamodèle pour les ODT (notés coverage item) est illustré Figure 10.

> ***Définition – fragment d'objet (OF)***. *Un fragment d'objet spécifie partiellement une instance d'une classe C du métamodèle effectif. Il lie une ou plusieurs propriétés de la classe avec une classe d'équivalence dans laquelle la propriété devra prendre une valeur. Une propriété ne peut être présente qu'une fois dans un fragment d'objet. Un fragment est un ensemble de contraintes sur les propriétés comme illustré Figure 10.*

Exemple :

- un fragment d'objet pour la classe ModelElement (du métamodèle de la Figure 17), qui défini une classe d'équivalence pour la propriété name est un ensemble constitué d'un élément qui lie la propriété avec une classe d'équivalence. On peut la noter ainsi : {(ModelElement::name, {""})}.
- un ODT peut regrouper tous les fragments d'objet sur la propriété name de ModelElement :

  {{(ModelElement::name, {null})},

  {(ModelElement::name, {""})},

  {(ModelElement::name, { s tel que |s| > 0})}}

Cet ODT contient trois fragments d'objet ne contenant qu'une contrainte sur une propriété. Il spécifie qu'un modèle pour le test devra contenir trois objets de type ModelElement. Un de ces objets devra avoir sa propriété name à null, un autre name à vide et le dernier avoir un name contenant au moins un caractère.

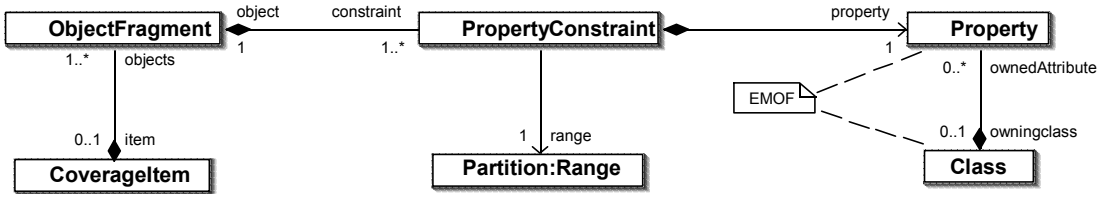


Figure 10 - Le métamodèle ODT

Un *critère de test* pour une transformation de modèles spécifie un ensemble d'ODT à partir des partitions spécifiées sur les propriétés des classes du métamodèle effectif (étape 2 du processus illustré Figure 8). Un critère de test impose de plus que toute classe concrète du métamodèle d'entrée E soit instanciée au moins une fois (au niveau des données de test générées) et que toute classe d'équivalence de toute partition de E soit utilisée au moins une fois.

Dans [27] les auteurs proposent 6 critères qui diffèrent sur la manière de regrouper les fragments d'objet pour définir les ODT devant apparaître dans les modèles pour le test. La génération de données de test pour une transformation consiste alors à générer un ensemble de modèles qui satisfait un de ces critères. La définition ci-dessous donne une explication intuitive de la satisfaction d'un critère par un ensemble de modèles (une définition formelle est disponible dans [27]).

> **Définition – satisfaction à un critère de test**. *Un ensemble de modèles d'entrée satisfait un critère de test si et seulement si l'ensemble de ses modèles couvre l'ensemble des objectifs de données de test spécifiés par le critère i.e. tous les objectifs sont exhibés par au moins un modèle d'entrée.*
> *Un ODT est exhibé par un modèle si tous ses fragments d'objets sont exhibés par le modèle. Un modèle exhibe un fragment d'objet défini sur une classe C si ce modèle contient une instance de C prenant ses valeurs dans les classes d'équivalence spécifiées par les fragments d'objet.*

4.2.4 Génération de modèles

La dernière étape pour la génération de données de test pour une transformation (Figure 8) consiste à générer un ensemble de modèles d'entrée qui satisfait un critère de test. Un algorithme a été défini pour générer cet ensemble à partir d'un ensemble d'ODT spécifié par un critère de test.

L'algorithme 1 présente la partie fixe de l'algorithme. Dans [27] il est montré qu'il existe plusieurs stratégies pour certaines opérations. Par exemple, il faut choisir le nombre d'ODT couverts par un modèle, c'est-à-dire trouver un compromis entre un grand modèle qui exhibe tous les ODT et de nombreux petits modèles qui exhibent chacun un ODT. Un modèle unique présente l'avantage d'une seule donnée de test, mais si une erreur est détectée dans la transformation il est difficile de savoir quelle partie du modèle l'a déclenchée. En revanche, des petits modèles facilitent la localisation d'erreurs mais augmentent le nombre de données de test.

```
Entrée : ensemble d'ODT définissant le critère C, métamodèle d'entrée MM, taille limite du modèle généré
Sortie : une liste L de modèles vérifiant le critère C
Tant qu'il reste des ODT non couverts Faire
    Créer un modèle M
    Tant que la limite de taille n'est pas atteinte et que M peut encore être augmenté Faire
        Choisir un objectif ODT non couvert
        Pour tous les fragments d'objets OF de ODT Faire
            Trouver un objet O instance de la classe partiellement spécifiée par OF.
            Pour chaque contrainte CT de OF portant sur une propriété P Faire
                Si CT contraint une valeur (cas d'une ValuePartition) alors
                    Choisir une valeur et l'affecter à P
                Sinon (cas d'une MultiplicityPartition)
                    Choisir une cardinalité N
                    Si le type de P est Classe alors
                        Trouver N objets du type de P et les associer à O pour la propriété P
                    Sinon   Trouver N valeurs parmi la partition de P et les associer à O pour la propriété P
                    fSi
                fSi
            fPour
            Ajouter O à M
        fPour
    fTant
    Marquer ODT comme couvert
    Compléter le modèle M pour qu'il soit conforme au MM (Complétion)
    Ajouter M à L
fTant
```

Algorithme 1 - Génération des données de test

Tout ce qui a été présenté dans cette section est automatisé par un prototype. Il implante le partitionnement d'un métamodèle d'entrée, la spécification d'un ensemble d'ODT à partir de ces partitions et la génération de modèles exhibant cet ensemble d'ODT et est actuellement en cours d'évaluation chez France Télécom R&D.

Un ensemble de critères de test couvrant le domaine d'entrée d'une transformation à partir de son partitionnement a été défini ainsi qu'un algorithme permettant de générer un ensemble de données de test satisfaisant ces critères et étant conformes au métamodèle. Il nous faut désormais tester la pertinence de ces données. Pour ce faire, nous utilisons l'analyse de mutation, présentée dans la section suivante, en l'adaptant au test de transformations de modèles.

5 Analyse de mutation

Pour l'évaluation de l'efficacité des fonctions d'oracle proposées dans ce rapport, nous avons besoin d'un cadre d'expérimentations reproductibles. L'analyse de mutation est une technique qui, par l'introduction de fautes connues, permet d'évaluer et l'efficacité des données de test, et celle de l'oracle. Appliquer la mutation aux transformations de modèles est un problème nouveau que nous étudions dans ce chapitre. Les résultats que nous obtenons dépassent la question de l'évaluation des oracles pour les transformations de modèles, puisqu'ils sont directement exploitables pour la qualification des générateurs de données de test (étudiés section 4.2). Cette

application à la qualification des données de test générées pour tester les transformations de modèles fait partie des travaux de l'équipe Triskell ainsi que le critère d'arrêt, auxquels j'ai contribué durant mon stage.

5.1 Présentation

L'***analyse de mutation*** est une technique introduite dans [28], dont le but premier est l'évaluation de la qualité d'un ensemble de données de test. Un ***mutant*** est une altération du programme à tester dans lequel on a injecté une et une seule erreur simple. Le processus représenté Figure 11 exécute les données de test une à une avec tous les mutants du programme P. Une fonction d'oracle est ensuite utilisée pour vérifier si l'erreur a été détectée. Cette fonction compare le résultat de chaque mutant avec le résultat du programme sous test qui est supposé correct. S'il y a inégalité alors l'erreur injectée a été détectée : la donnée de test a ***tué*** le mutant. Deux cas sont possibles quand un mutant n'est pas tué:

- c'est un mutant équivalent : quelque soit la donnée de test, le mutant produit le même résultat que le programme P
- aucune donnée de test n'est adaptée pour mettre en évidence cette erreur, dans ce cas le mutant est ***vivant***

Les mutants équivalents sont supprimés de l'ensemble des mutants et on obtient finalement une liste de mutants tués qui permet de calculer un ***score de mutation*** pour l'ensemble des données de test qui est la proportion de mutants tués par rapport au nombre total de mutants principaux. Ce score est l'indicateur de la qualité des données de test.

$$\text{score de mutation} = \frac{\text{nombre de mutants tués}}{\text{nombre total de mutants principaux}}$$

Si le score n'est pas suffisant il faut améliorer l'ensemble des données de test ce qui peut être réalisé en ajoutant de nouvelles données de test.

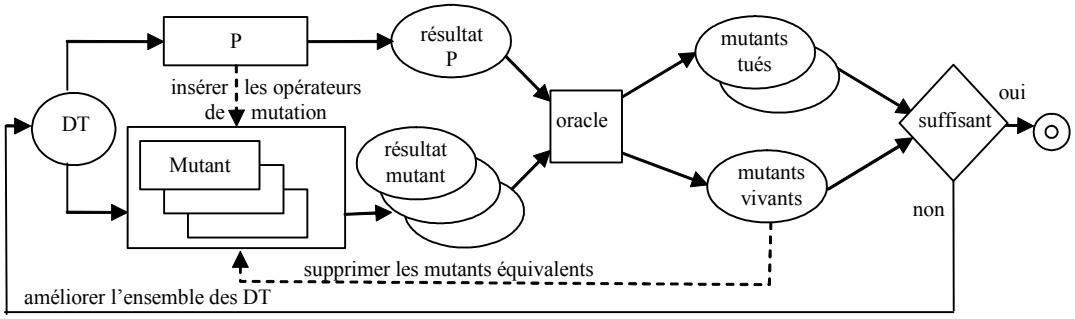


Figure 11 - processus de mutation

La valeur de l'analyse de mutation est fondée sur la pertinence des mutants, donc sur la pertinence des erreurs introduites. Des travaux antérieurs ont permis de spécifier un ensemble (minimal) de fautes qui sont modélisées par des ***opérateurs de mutation***. Ces opérateurs insèrent uniquement des erreurs simples, c'est-à-dire qui ne touchent qu'une seule instruction. Des opérateurs classiques ont été proposés pour les langages procéduraux : sur les opérations arithmétiques (par exemple remplacer un + par un -), logiques (par exemple remplacer un true par un false), de comparaison (par exemple remplacer un > par un <), ou encore sur les instructions, comme étudiés dans [29]. Pour exécuter une analyse de mutation avec ces opérateurs, les erreurs sont injectées systématiquement partout où cela est possible. La génération de mutants est donc purement syntaxique : elle ne s'appuie que sur la syntaxe du langage pour placer les erreurs.

D'autres travaux ont proposé des opérateurs spécifiques aux langages orientés objet pour générer des mutants qui contiennent des erreurs liées aux notions de classes, d'héritage, de polymorphisme [30]. Ces opérateurs prennent en compte des spécificités liées à la sémantique des langages, mais restent des erreurs simples qui peuvent être introduites par une analyse syntaxique du programme.

5.2 Création des mutants spécifiques aux transformations de modèles

Pour évaluer efficacement les données de test générées pour les transformations de modèles, il est nécessaire de proposer des opérateurs de mutation adaptés aux transformations. Ces opérateurs doivent refléter le type des erreurs qui apparaissent communément lors de l'implantation d'une transformation.

La première particularité de ces opérateurs est qu'ils ne peuvent pas s'appuyer sur la syntaxe d'un langage. En effet, la nouveauté des langages de transformations de modèles, leurs spécificités et leur hétérogénéité (impératif orienté objet, déclaratif, mixte) imposent de s'affranchir de l'aspect mise en œuvre dans un premier temps. Les opérateurs classiques de mutation (qu'ils soient orientés objet ou non) restent utiles mais ils seront employés en fonction des langages qui servent à l'implantation. Les opérateurs proposés doivent donc être définis par rapport à une vue abstraite d'un programme de transformation.

Cette vue abstraite, nous la proposons en tentant de répondre à la question suivante : quel type d'erreur peut-on faire en implantant une transformation de modèles ? Par exemple, une transformation va naviguer le modèle d'entrée pour reconnaître certains éléments à transformer, une erreur peut se produire en navigant les mauvais éléments ou sur la manière de trier ces éléments. Lors d'une transformation, il faut aussi créer des éléments du modèle de sortie, une erreur peut se produire sur le type d'élément créé ou sur son initialisation. L'analyse des erreurs possibles pour une transformation de modèles nous a conduit à distinguer quatre opérations abstraites correspondant aux traitements principaux effectués lors d'une transformation.

(1) navigation : le modèle est parcouru grâce à une relation définie sur son métamodèle, et un ensemble d'éléments est obtenu.

(2) filtrage : un ensemble d'éléments est étudié et seuls ceux qui répondent à un critère sont sélectionnés.

(3) création du modèle de sortie : les éléments du modèle de sortie sont créés à partir d'élément(s) extrait(s).

(4) modification du modèle d'entrée devenant le modèle de sortie : s'il s'agit de modifier le modèle d'entrée alors des éléments peuvent être créés ou supprimés.

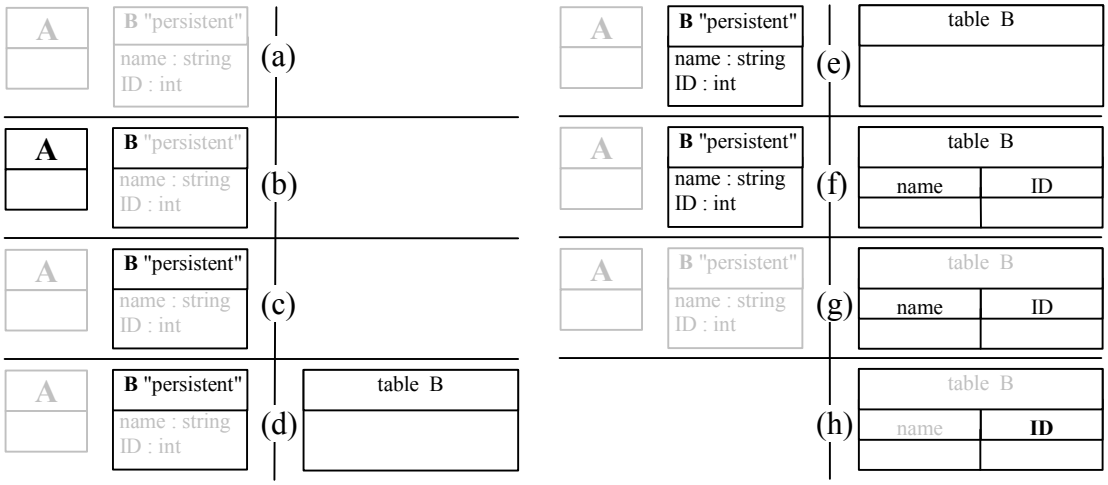


Figure 12 – Décomposition d'une transformation

La transformation UML vers RDBMS donne un exemple de décomposition grâce à ces opérations, illustrée par la Figure 12. Dans le diagramme de classes donné en entrée, il faut sélectionner les classes persistantes et créer des tables correspondantes avec pour colonnes les attributs des classes correspondantes. Le modèle d'entrée (a) est d'abord navigué pour trouver les classes (b) qui sont filtrées pour ne garder que les persistantes (c). Une table est créée pour chacune (d). Puis la transformation navigue les classes persistantes pour récupérer leurs attributs (e) (y compris ceux hérités) et les colonnes correspondantes sont créées (f). Finalement les colonnes sont filtrées (g) pour trouver une clef appropriée qui est créée (h).

Ainsi la navigation et le filtrage sont applicables à la fois sur le modèle d'entrée et de sortie, ces 2 phases sont fortement liées : la navigation renvoie des éléments qui sont généralement filtrés puis qui servent à la

création (ou la modification), cela forme un cycle qui peut-être répété pour former finalement une transformation de modèles.

Les opérateurs de mutation définis dans la suite correspondent à des modèles de faute pour ces opérations. Ces opérateurs sont ainsi indépendants du langage utilisé pour l'implantation (remarquons qu'il faudra peut-être ajouter des opérateurs spécifiques au langage lors de l'analyse de mutation). De plus, les opérateurs ainsi définis s'appuient réellement sur la sémantique d'une transformation et permettent donc de valider les tests sur des erreurs liées à la nature du programme plus qu'à la syntaxe du langage.

5.3 Opérateurs de mutations pour les transformations de modèles

Nous proposons plusieurs opérateurs de mutation nouveaux, spécifiques aux transformations de modèles qui agissent sur la navigation et le filtrage effectués par la transformation sur les modèles d'entrée ou de sortie et sur la création du modèle de sortie. La Figure 13 représente un métamodèle et sert à illustrer les exemples utilisés par la suite.

Opérateurs portant sur la **navigation** :

- **remplacement d'une relation vers une même classe (RRMC)** : s'il existe plusieurs relations entre 2 classes navigables par une transformation, l'opérateur remplace la relation naviguée par les autres possibles. Par exemple la classe A possède 3 relations `b1`, `b2`, `b3` vers la classe B : si la transformation navigue `A.b1` alors l'opérateur remplace `b1` par `b2` et `b3`, cela crée 2 mutants.
- **remplacement d'une relation vers une autre classe (RRAC)** : si depuis une classe il existe plusieurs relations navigables vers différentes classes, l'opérateur remplace la relation naviguée par une autre pointant vers une classe différente. La classe B possède 2 relations : f (vers F) et d (vers D), si la transformation navigue `B.f` alors l'opérateur crée un mutant en remplaçant `f` par `d`.
- **modification d'une succession de relation avec manque (MSRM)** : lors d'une navigation, la transformation peut naviguer successivement plusieurs relations. Ainsi à partir d'une instance de A, il est possible d'obtenir un ensemble d'instances de F par la navigation composée `A.b1.f`. Cet opérateur enlève la dernière étape d'une navigation composée. Dans le mutant créé ici, la navigation devient `A.b1` et ce n'est plus une collection d'instances de F mais de B qui est obtenue.
- **modification d'une succession de relation avec ajout (MSRA)** : cet opérateur effectue le contraire de MSRM. Une relation est ici ajoutée: `A.c` devient `A.c.d` par exemple. Le nombre de mutants créés dépend du nombre de relations sortant de la classe obtenue dans la transformation (ici la transformation obtient une instance de la classe C qui n'a qu'une relation sortante donc un seul mutant est créé).

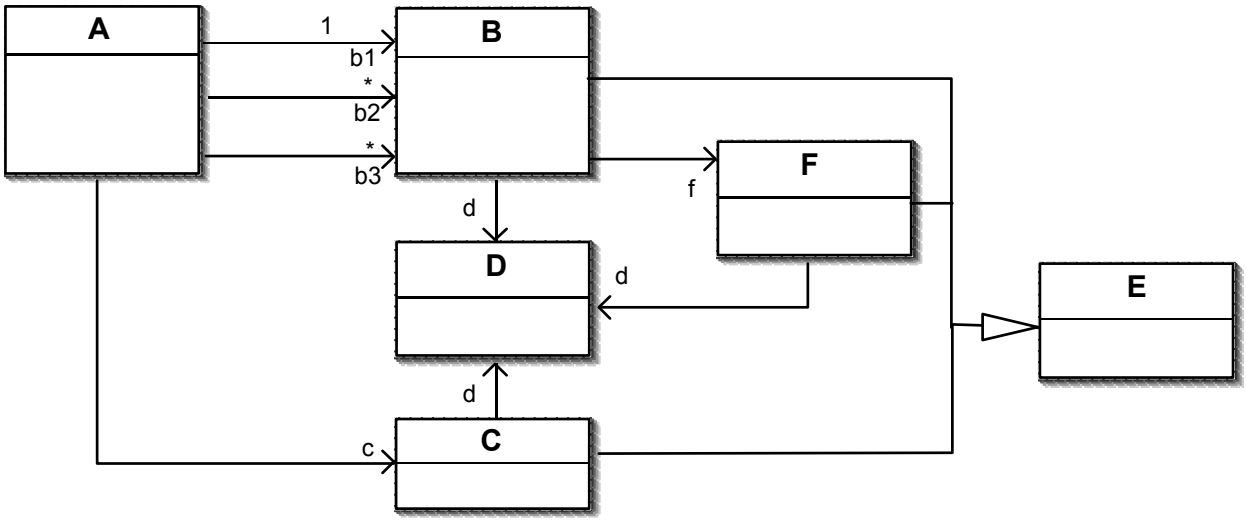


Figure 13 – Exemple de métamodèle

Opérateur portant sur le **filtrage** :

- **modification du filtrage sur une collection avec perturbation (MFCP)** : le filtrage agit sur une collection pour n'en conserver que les éléments qui intéressent la transformation. Il s'agit ici de modifier un filtrage existant en influençant les paramètres du filtrage. Dans un cadre général un filtrage peut être considéré comme une conditionnelle appliquée à une collection et qui dépend d'un critère particulier. Dans notre transformation UML vers RDBMS au lieu de filtrer les classes persistantes, le mutant filtre en fonction d'un autre stéréotype. Il y a aussi des filtrages en fonction du type de classes qui peuvent être perturbés. Ainsi certaines relations mènent à des classes qui ont plusieurs fils (comme la classe E). Les classes filles récupèrent ces relations et donc la navigation d'une telle relation renvoie un ensemble de classe qui peuvent être soit du type de la classe mère ou de ses fils. Ainsi si une relation arrive à la classe E, alors sa navigation renvoie un ensemble d'instance de classe des types A, B, C, E ou F. Il peut y avoir un filtrage sur ce type de classe (pour ne garder que les instances de la classe A par exemple) et cet opérateur va le modifier en fonction des différentes possibilités (ici cela crée donc 4 mutants).
- **modification du filtrage d'une collection avec manque (MFCM)** : Cet opérateur supprime un filtrage sur une collection, le mutant renvoie la collection qu'il était censé filtrer à l'identique. Dans l'exemple de la transformation UML vers RDBMS, seuls les classes persistantes sont utilisées, la navigation fournit l'ensemble des classes et le filtrage doit sélectionner celles qui sont persistantes pour que les tables correspondantes soient créées. L'opérateur va supprimer le filtrage et la transformation va donc créer des tables pour toutes les classes. Cela peut être un filtrage sur le type de classe : si une navigation parcoure une relation menant à la classe E et s'il existe un filtrage sur cette collection qui ne garde que les instances de F (fils de E) ce filtrage est supprimé, ainsi toutes les instances de A, B, C, E, F qui étaient dans cette collection sont conservées.
- **modification du filtrage d'une collection avec ajout (MFCA)** : cet opérateur effectue le contraire de MFCM, il prend une collection en entrée et lui applique un filtrage inutile. Comme cet opérateur pourrait donner une infinité de mutant, nous devons le restreindre. Il prend donc une collection et effectue un filtrage inutile qui retourne un seul élément choisi arbitrairement.

Opérateurs portant sur la **création** :

- **remplacement de la création d'un fils par un autre (RCFA)** : si la transformation veut créer la spécialisation d'une classe ayant plusieurs fils, alors cet opérateur crée un fils au lieu d'un autre. Par exemple, si la transformation crée une instance du fils B de E alors le mutant crée une instance de C au lieu de B.
- **modification d'une mise en relation de classes avec retrait (MMRR)** : quand la transformation dispose de 2 instances de classes qui nécessitent la création d'une relation entre elles alors l'opérateur retire cette création. Par exemple si la transformation crée la relation `b1` entre une instance de A et une instance de B alors le mutant ne crée pas cette relation.
- **modification d'une mise en relation de classes avec ajout (MRCA)** : quand la transformation dispose de 2 instances de classes et que le métamodèle autorise d'avoir une relation entre elles alors l'opérateur de mutation crée cette relation. Dans notre exemple le métamodèle de la Figure 12 autorise 3 relations différentes entre des instances de A et de B. Donc s'il existe une instance de A et une de B alors l'opérateur génère 3 mutants, chacun créant une des relations `b1`, `b2`, `b3` entre A et B, même si l'une d'elle existe déjà.

Ces erreurs sont directement liées à la manière de programmer des transformations de modèles qui se décomposent suivant les 4 phases abstraites (navigation, filtrage, création/modification).

5.4 Justification de ces opérateurs

Nous discutons ici l'impact de ces opérateurs sur une transformation de modèles basés sur le métamodèle de la Figure 13. La navigation se fait par une succession de relations parcourues, donc une erreur commise au début de la navigation peut rendre la suite irréalisable. Par exemple si l'opérateur RRAC altère une navigation de A à F en la forçant à atteindre C alors il ne sera plus possible de retrouver F. Ce cas particulier montre que les opérateurs de mutation proposés peuvent créer des mutants inexploitables, mais dans d'autres cas les erreurs sont beaucoup plus pertinentes. Quand la transformation veut naviguer de A vers D par exemple. La navigation peut être décomposée en 2 étapes : `A.b1` puis `B.d`, si l'opérateur MSRA ajoute une relation inutile vers F : `A.b1.f`, il

est toujours possible d'atteindre D par `F.d`. Dans les 2 cas, la deuxième étape de la navigation utilise une relation `d` et obtient des instances de D, l'erreur passe alors inaperçue. Ce sont les particularités des modèles et de leurs métamodèles qui ressortent ici montrant que de nombreuses situations masquent les erreurs au moment de la programmation et même de l'exécution. Dans de nombreux cas ces erreurs ne causeront pas de problème lors de la compilation et l'exécution ne provoquera pas de défaillance.

Nous étudions les particularités des opérateurs qui les rendent pertinents, en s'appuyant sur le métamodèle de la Figure 13 :

- **remplacement d'une relation vers une même classe (RRMC)** : plusieurs cas sont possibles lorsque la navigation utilise une mauvaise relation entre 2 classes, en fonction de la cardinalité. Par exemple, remplacer A.b1 par A.b2 entraîne une différence de cardinalité qui a des chances d'être repérée. Par contre différencier l'emploi de b2 et de b3 sera beaucoup plus difficile : dans les 2 cas (avec ou sans erreur) une collection d'instances de B est obtenue. Ainsi la suite de la transformation n'est pas affectée, et la faute passe inaperçue.

- **remplacement d'une relation vers une autre classe (RRAC)** : plusieurs cas sont à considérer.
  Par exemple, pour atteindre D depuis A, il est possible de naviguer indifféremment par B ou par C. Une (collection d') instance de D est obtenue mais pas la même dans un cas ou dans l'autre et cela passera inaperçu.
  Cet opérateur exploite aussi l'héritage, les classes B et C héritent d'une même classe, dans ce cas elles héritent des mêmes éléments. Les 2 classes (B et C) peuvent aussi avoir des attributs communs (non hérités). Dans les 2 cas, si le mutant navigue une mauvaise relation et renvoie des instances de C au lieu de B, si la transformation exploite ensuite leurs éléments communs, alors il n'y aura pas de différence repérable. Ainsi si 2 classes ont soit une relation vers une autre classe commune, soit une classe parente commune, soit des attributs communs, alors une erreur de navigation conduisant vers ces 2 classes est pertinente car elle passe inaperçue.

- **modification d'une succession de relation avec ajout/manque (MSRA** et **MSRM)** : ces opérateurs conduisent aux même cas que RRAC, si la transformation navigue pour obtenir une (collection d') instance de B et qu'une mauvaise succession de relation renvoie finalement une instance de F (pour MSRA ou l'inverse pour MSRM), alors la faute passe inaperçu si B et F partagent une relation commune vers une même classe, un lien d'héritage d'une même classe ou des attributs communs.

- **modification du filtrage d'une collection avec manque (MFCM) :** nous supposons que l'opérateur a supprimé le filtrage sur le type de classe, ainsi si les classes sont toutes du type de la classe parent vers qui va la relation navigue alors cela n'entraînera pas de problème détectable. De même si les seules propriétés, relations utilisées sont communes aux différents types de classes de la collection parce qu'elles sont héritées par toutes ces classes sans changement. Dans tous les cas certaines classes auraient du être retirées de l'ensemble et leur présence constitue une faute qui passe inaperçue. Même si l'ensemble ne contient pas de classe qui aurait du être retiré, il peut y avoir problème : si une des instances est du type d'une classe ayant fait une redéfinition d'une fonction, l'appel de cette fonction sera celui de la classe parente et non de la classe voulue.

- **remplacement de la création d'un fils par un autre (RCFA)** : cet opérateur crée une instance d'une mauvaise classe mais héritant d'un même parent que la classe créée dans le programme sous test. L'opérateur est intéressant quand les 2 classes (celle créée et celle qui aurai du l'être) sont accessibles par une même relation, héritée du parent commun donc. Ainsi l'inversion lors de la création ne gênera pas cette navigation (on est revenu dans le deuxième cas de l'opérateur RRAC présenté juste avant).

- **modification d'une mise en relation de classes avec retrait (MMRR)** : cet opérateur élimine des relations entre classes. Si la relation qui n'est pas créé est de cardinalité 1 dans le métamodèle alors son absence peut poser problème dans la suite de la transformation. Par contre en cas de cardinalité plus grande, la suite de la transformation sera peu altéré (la navigation de cette relation amènera à une collection juste privé d'un élément).

- **modification d'une mise en relation de classes avec ajout (MRCA)** : cet opérateur crée des relations inutiles, les collections qui seront obtenues par navigation dans la suite de la transformation seront donc fausse mais resteront conforme au métamodèle donc non décelable. Puisque A ne peut avoir qu'une seul classe par la relation b1, cela peut conduire à un écrasement de la relation existante par la relation mutante.

L'intérêt de ces opérateurs est la pertinence des erreurs qu'ils injectent dans le programme puisque ce sont des erreurs simples par rapport au métamodèle et au raisonnement suivi pour écrire une transformation.

Le deuxième point intéressant est qu'elles vont facilement passer inaperçu pour le programmeur à tout niveau (programmation, compilation, exécution).

Tout cela montre à quel point les opérateurs peuvent être « malin » et discret. Nous pouvons alors faire l'hypothèse que les données de test générées automatiquement à partir de la couverture du métamodèle vont avoir des difficultés pour tuer ces mutants. C'est une nouveauté par rapport à la mutation classique.

Pour conclure, notons que les opérateurs classiques restent utiles pour les parties arithmétiques ou logiques présentes au moins dans les phases de filtrage ou de création. L'emploi de langages OO (orientés objet) pour l'implantation nécessite aussi l'emploi d'opérateurs OO.

5.5 Complexité de l'implantation de ces opérateurs

L'étude de cas présentée section 8.1 est implantée en JAVA. Ce langage n'est pas au même niveau qu'un langage dédié aux transformations de modèles, ce qui entraîne des difficultés pour mettre en œuvre les opérateurs de mutation spécifiques aux transformations de modèles : il ne s'agit plus d'erreurs simples mais complexes dans leur mise en œuvre, c'est-à-dire ne touchant plus une seule instruction mais un ensemble d'instructions liées fonctionnellement (navigation, création…). L'implantation complexe des opérateurs n'est probablement pas spécifique à JAVA, l'utilisation de langages impératifs, c'est-à-dire manipulant les modèles à un bas niveau, proche de la structure des modèles (et de la façon de les stocker comme ici avec EMF, présenté au 8.2, ou avec le format XMI) conduira aussi à des opérateurs complexes. Prenons l'exemple d'une transformation qui manipule une base de donnée, dont les clés des tables ont besoin d'être initialisées. La spécification de la transformation demande qu'une colonne appropriée soit choisie, si possible de type "integer".

La transformation filtre les colonnes avec pour critère de filtrage le type de la colonne qui doit être "integer" :

```
1.    EList columnsatraiter= tableUse.getColumns();          //navigation
2-1.  Iterator itINT = columnsatraiter.iterator();           //début de filtrage
2-2.  List columnsdetypeINT = new ArrayList();
2-3.  while(itINT.hasNext()){
2-4.  Columns cINT = (Columns) itINT.next()
2-5.  if(cINT.getType().equals("integer"))
2-6.       columnsdetypeINT.add(cINT);
2-7.  }
2-8.  if( ! columnsdetypeINT.isEmpty())
2-9.  Columns columnKey = columnsdetypeINT.get(0);
2-10. else
3.         Columns columnKey = columnsdetypeINT.get(0);      //fin de filtrage
4.    tableUse.setKey(columnKey);                            //création
```

En fait il y a 2 étapes de filtrage : le filtrage des colonnes de type integer puis la sélection de la première d'entre-elles (correspondant à l'instruction `2-9. ou 3.`). L'opérateur MFCM peut supprimer le premier filtrage, ainsi la première colonne de la table est-elle sélectionnée sans distinction sur le type :

```
1.  EList columnsatraiter= tableUse.getColumns();           //navigation
2.  Columns columnKey = (Columns) columnsatraiter.get(0);   //filtrage
3.  tableUse.setKey(columnKey);                             //création
```

Dans la mise en œuvre cela ne se traduit pas par une erreur simple mais par la suppression de toute une partie du programme : si, dans les deux versions, 3 instructions sont communes (`1. 3. 4.` correspondant à *1. 2. 3.*), le filtrage manquant se traduit par 10 lignes de code en moins (de `2-1.` à `2-10.`). La complexité des opérateurs de mutation appliqués en JAVA est donc réelle et rend la création des mutants plus difficilement automatisable.

Ainsi la mutation dans un contexte de transformation de modèles est à prendre sous un angle nouveau, on ne peut plus se baser uniquement sur la syntaxe du programme ou la sémantique des langages d'implantation. Les opérateurs de mutation proposés se basent donc davantage sur le raisonnement qui conduit à la création d'une transformation et sur sa façon de manipuler les modèles.

5.6 Bilan sur la mutation

Il ne faut pas voir dans les mutants présentés l'envie de notre part d'être le plus sournois possible, cela au dépend de l'hypothèse de base de la mutation qu'est la simplicité des opérateurs. Mais nous estimons que ses opérateurs sont réalistes et parfaitement adaptés à l'idée générale que nous nous faisons d'une transformation de modèles. A terme, il semble même probable que ces opérateurs soient simples à implanter avec des langages dédiés (QVT 2, MTL, Kermeta).

6 Oracle

Notre étude a permis de cerner plusieurs éléments qui permettent de former un oracle et de l'automatiser en utilisant la conformité des modèles à leur métamodèle et la spécification de la transformation par des contrats.

6.1 Utilisation du métamodèle

La première chose à vérifier pour un modèle de sortie est sa conformance au métamodèle qui impose sa structure. L'utilisation de JAVA associé au dépositaire de modèles EMF assure la conformance structurelle vis-à-vis du métamodèle. L'utilisation d'un langage de transformation qui manipule les modèles sans contrôle du métamodèle peut engendrer des erreurs de structure. Par exemple, dans le cas d'une relation de cardinalité 1, il serait possible d'avoir une cardinalité n. Les modèles étant stockés en XMI selon les normes OMG (dérivé du XML), l'utilisation d'un langage sans contrôle comme XSLT peut engendrer ce genre d'erreur.

Il est possible de faire une comparaison avec un problème de typage. En effet, dans le cas d'un programme classique qui rend en résultat des données simples, une opération ou un programme peut avoir une signature qui définit le type du résultat obtenu. Si le résultat ne correspond pas au type attendu, il n'est pas nécessaire de chercher plus loin pour pouvoir conclure que le résultat est erroné. Obtenir des booléens avec un programme censé renvoyer des entiers ne sera pas acceptable.

Dans le cadre des transformations de modèles, la vérification de type est compliquée par la nature des données manipulées. Le typage des modèles est encore en pleine étude, dans l'équipe Triskell [31] entre autres. Les résultats qui seront obtenus dans ce domaine pourront être directement exploitables dans notre étude. Les transformations pourront alors être signées et les données manipulées seront plus facilement contrôlables.

6.2 Oracle par contrats

Dans une transformation de modèles les éléments disponibles sont les modèles d'entrée, les métamodèles d'entrée et de sortie et la spécification de la transformation. Ces deux derniers éléments peuvent comporter des informations formelles réutilisables.

6.2.1 Evaluation de l'utilisation des contrats pour la robustesse

Une deuxième méthode peut être utilisée comme oracle pour valider les tests, elle consiste à utiliser les assertions. Les travaux présentés dans [32] montrent l'intérêt de l'utilisation des contrats dans les systèmes orientés objet pour en contrôler la robustesse et la diagnosabilité. Il s'agit de tirer parti des apports d'une conception par contrats (design by contract [11]). Les bénéfices de cette approche peuvent être transposés dans l'ingénierie des modèles et particulièrement dans la partie oracle du test de transformation de modèles.

Les assertions sont présentes dans différentes parties d'une transformation de modèles et représentent des contrats à respecter. Nous allons voir celles qui peuvent être utiles pour l'oracle parmi les 4 types que l'on trouve particulièrement :

- Les contrats sur le modèle de sortie
- Les contrats de la spécification de la transformation
- Les contrats de la transformation
- Les contrats sur le modèle d'entrée

Ce dernier point n'est pas utile ici, il ne correspond pas à la partie oracle qui nous intéresse. L'oracle cherche à valider le résultat d'une transformation, sous l'hypothèse que la donnée de test soit correcte. Par contre l'étude de l'apport de ces contrats sur la génération de modèles d'entrée est une perspective.

Contrats sur le modèle de sortie : Le modèle obtenu en sortie de transformation est censé correspondre au métamodèle de sortie. Le modèle de sortie sera conforme à son métamodèle sur le plan structurel si la transformation utilise un environnement qui se base sur les métamodèles pour la création des instances du modèle. C'est le cas avec EMF qui assure la conformance des modèles en fonction de leurs métamodèles. Cependant, des aspects plus sémantiques des modèles ne sont pas représentés dans leur métamodèle comme dans le cas de notre exemple: le métamodèle ne spécifie pas qu'une base de données ne peut pas avoir des tables de même nom par exemple. Le métamodèle doit donc être enrichi de contrats, le plus souvent écrits en OCL (Object Constraint Language) qui vont améliorer la précision des modèles. Cela donne pour ce cas particulier :

```
context database : self.element->forall(t1,t2|t1.name=t2.name implies t1=t2)
```

Les contrats qui portent sur la conformance du modèle de sortie vis-à-vis de son métamodèle doivent donc être contrôlés pour valider le résultat d'un test, ils doivent donc faire partie de l'oracle.

Contrats de la spécification : Une hypothèse qui nous semble réaliste est de considérer que la transformation dispose de contrats qui définissent des pré et post-conditions, des invariants pour la transformation (comme proposé dans [25]). Les pré conditions imposent des contraintes sur les modèles d'entrée, qui ne sont pas utilisées pour l'oracle mais pourraient être intéressantes pour la génération des données de test. Les invariants peuvent être exploités pour l'oracle puisqu'ils doivent être vrais tout au long de la transformation, y compris quand elle se termine et rend son résultat. Enfin les post-conditions expriment des propriétés attendues sur les modèles en sortie et des propriétés qui lient les entrées aux sorties. Ce dernier type de contrat est le plus intéressant pour l'oracle puisqu'il permet d'évaluer le résultat. S'il s'agit de contrats écrits en OCL leur utilisation pour l'oracle est simple puisqu'il suffit de les appliquer sur le modèle de sortie obtenu avec une donnée de test. C'est le cas dans notre exemple qui est spécifié par l'expression OCL :

```
UMLmodel->instanceOf(Class)
    ->select(c|c.stereotype.name='persistent')
    ->forAll(c|RDBMSmodel   ->instanceOf(Table)
                            ->one(t|t.name=c.name and c.attributes
                            ->forAll(a|t.columns->one(col|col.name=a.name))))
```

Ce qui précise bien que, pour chaque classe de stéréotype persistent (`select`), il existe une et une seule (`one`) table de même nom dont les colonnes correspondent aux attributs. Si les attributs hérités doivent être considérés alors il faudra utiliser une fonction récursive définie dans la classe `Classifier` qui collecte tous les attributs. Il faut plusieurs contrats de ce type pour spécifier complètement une transformation. Si la transformation n'est pas (entièrement) formalisée, le développeur (de la transformation de préférence ou des tests à défaut) devra écrire ces contrats.

Contrats de la transformation : Une transformation de modèle va être implantée dans un programme écrit dans un langage, elle peut être décomposée en plusieurs phases abstraites comme présentée au 5.2. Mais suivant les langages, il est possible de regrouper certaines étapes, c'est le cas dans notre exemple de la création de tables, la création des colonnes et la création de la clef. Il y a donc une factorisation possible et chaque fonction peut être enrichie de contrats, des pré et post-conditions et des invariants. Ces contrats sont utiles pour contrôler que la transformation s'exécute convenablement et que ses étapes intermédiaires respectent certaines contraintes. Ils servent aussi d'oracle au moment du test unitaire des fonctions séparément les unes des autres. Dans [32], les auteurs ont montré que ces contrats étaient aussi très utiles pour aider à localiser les erreurs.

Les contrats que l'on trouve dans les différentes parties d'une transformation de modèle permettent de réaliser un oracle automatisable même si les contrats directement disponibles sont parfois seulement partiels.

6.2.2 Limites d'OCL

En OCL, il n'est par exemple pas possible de définir des fonctions et donc d'appliquer des étapes de navigation récursivement. Si l'on reprend l'exemple d'UML (Figure 1), le parcours de plusieurs classes (Generalization, GeneralizableElement) est nécessaire pour récupérer les classes parents ou fils d'une classe. Avec OCL, on ne peut pas gérer le nombre de succession fils-parent à naviguer, ainsi on ne peut pas par exemple

à partir d'une classe récupérer la totalité de ses classes parents (les classes dont elle héritent et les classes dont celles-ci héritent…).

6.3 Validation de l'oracle

Pour comparer et évaluer les différentes solutions proposées pour l'oracle, il faut évaluer la qualité de chacune. Pour ça, nous proposons d'adapter l'analyse de mutation.

Les opérateurs de mutation proposés au 5.3 ainsi que les autres (classiques et OO), nous permettent de créer des programmes comportant des erreurs pertinentes (les mutants). L'analyse de mutation appliquée aux données de test générées permet de distinguer quelle donnée de test tue quel mutant. Nous espérerons que l'oracle soit aussi capable de signaler que les résultats renvoyés par ces données de test avec les mutants tués sont faux. Cela est illustré par la partie haute (**complétude**) de la Figure 14, l'oracle est complet si quand il y a une erreur il l'annonce. La présence d'une erreur détectable ayant été prouvée par une analyse de mutation sur les données : nous savons qu'au moins une des données de test a tué le programme mutant utilisé.

La complétude est la propriété principale recherchée pour l'oracle. Nous pouvons aussi vérifier que l'oracle est correct, c'est-à-dire qu'il n'annonce pas d'erreur quand il n'y en a pas. La **correction** est plus subtile à prouver que la complétude, car il faut disposer d'un programme vraiment parfait ayant la confiance total du testeur pour pouvoir contredire le résultat négatif (« présence d'erreur ») de l'oracle sur ce programme. Pour cela nous proposons une solution originale qui consiste à réutiliser les mutants équivalents (ceux qu'aucune donnée de test n'est capable de tuer).

L'analyse de mutation a permis de définir quels mutants restent vivants avec l'ensemble des données de tests utilisé, parmi ceux-là un certain nombre sont équivalents au programme sous test. Nous utilisons les mutants équivalents pour prouver la correction de l'oracle. Cette phase d'analyse de l'oracle est effectuée une fois que la complétion de l'oracle a été prouvée. Ainsi nous sommes assurés que le programme sous test ne comporte pas d'erreur, car la complétion de l'oracle assure que toutes les fautes sont révélées. Prouver la complétion du programme sous test est nécessaire pour assurer qu'il ne contient plus d'erreur et donc que les mutants équivalents ne comportent pas non plus d'erreurs.

Une fois que le programme sous test est correct, nous pouvons vérifier que l'oracle n'annonce pas que les résultats des mutants sont faux, et prouver ainsi la correction de l'oracle. La Figure 14 illustre le processus d'analyse de la pertinence de l'oracle dans sa globalité. L'ensemble des données de test est exécuté avec les mutants tués, l'oracle doit alors signaler que les résultats obtenus sont faux. En cas de résultat contradictoire, il faut améliorer l'oracle. Puis l'ensemble des données de test est exécuté avec les mutants équivalents, cette fois le résultat ne doit pas signaler de faute, sans quoi il faut l'améliorer.

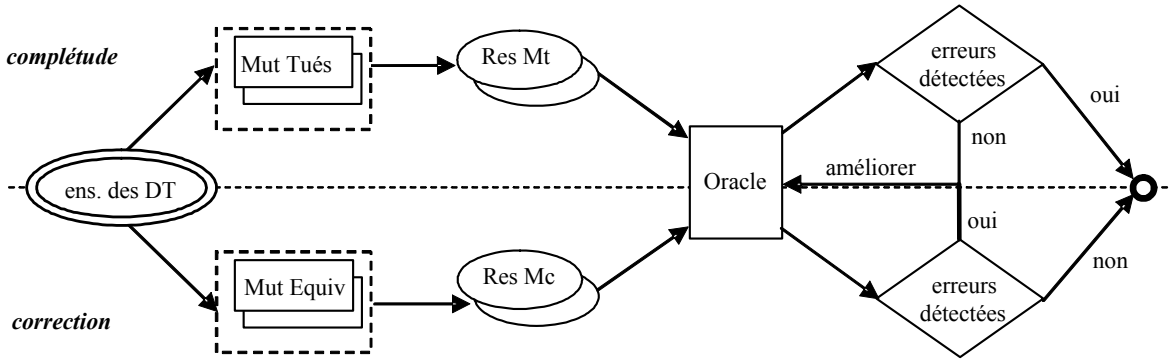


Figure 14 - Processus de l'analyse de l'oracle

7 Comparaison de modèles

Dans cette partie nous présentons les deux utilisations possibles de la comparaison qui nous intéressent, puis étudions les spécificités complexes des comparaisons de modèles et proposons une méthode pour simplifier leur mise en œuvre.

7.1 Utilisation dans l'analyse de mutation

La comparaison de modèles est utilisée comme fonction d'oracle dans l'analyse de mutation (Figure 11), dans ce cas la comparaison est faite entre les résultats d'une donnée de test exécutée avec le programme et avec chaque mutant. Dans le cadre de notre étude, nous avons eu besoin de pouvoir effectuer des comparaisons de modèles pour mettre en pratique l'analyse de mutation dans nos expérimentations (partie 8). Ne disposant pas d'outil général de comparaison de modèles, nous avons écrit un programme JAVA ad hoc écrit pour la transformation, les explications se trouvent au 8.3.2.

7.2 Utilisation comme fonction d'oracle

Pour le test, une fonction d'oracle doit rendre un verdict lors de l'exécution d'un cas de test. Classiquement, cette fonction compare le résultat attendu avec le résultat obtenu. Dans le cas du test de transformation de modèles, cela revient à comparer 2 modèles.

Pour utiliser la comparaison de modèle comme oracle il faut disposer du modèle attendu, ce qui n'est pas évident. C'est le cas pour le test de non-régression qui vérifie que les changements dans une partie d'un système n'affectent pas d'autres parties déjà validées. Dans le cadre d'un changement de plate-forme, les résultats des tests peuvent aussi être disponibles. On peut citer par exemple le jeu vidéo *Tom Clancy's Splinter Cell Chaos Theory* qui a été adapté en plusieurs centaines de versions différentes pour fonctionner sur un nombre important de téléphones mobiles, dans différents pays et donc dans différentes langues.

Toutefois, dans un contexte global du test de transformation de modèle, nous ne pouvons pas supposer disposer systématiquement du modèle espéré. Pour cette raison, nous avons cherché d'autres méthodes pour l'oracle, plus indépendantes et ne se basant que sur des éléments réellement disponibles, basée sur l'utilisation de contrats comme oracle, qui a été présenté section 6.

7.3 Spécificités de la transformation de modèles

Comparer des modèles a été une tache difficile dans notre étude, nous allons voir pourquoi puis comment nous proposons de la simplifier.

7.3.1 Propriétés d'une comparaison de modèles

Les modèles sont des données complexes, bien au-delà des données habituellement manipulées dans les programmes, et cela même par rapport aux objets des langages orientés objet. En effet les modèles peuvent être utilisés pour représenter des systèmes complexes et sont instanciés comme un ensemble d'objets en relation.

Les modèles peuvent être considérés comme des graphes, en considérant les éléments qui le composent comme des nœuds et leurs relations comme des arcs. Dans le cas général, l'isomorphisme entre 2 graphes est un problème NP-complet [33], il est donc de complexité exponentielle, ce qui est un obstacle important, aggravé par les notions de modélisation qui sont ajoutées aux graphes. En effet, la correspondance modèle/graphe n'est pas directe, ce sont les diagrammes d'instances qu'il faut considérer (et non les diagrammes dans une forme plus commune comme les diagrammes de classes UML, la différence est clair entre la Figure 3 et la Figure 4), cela donne des graphes non orientés mais dont les nœuds sont annotés par les attributs des classes et sont susceptibles de porter des noms identiques.

Dans la partie qui suit, nous proposons des heuristiques pour permettre les comparaisons de modèles.

7.3.2 Éléments de solution basé sur les métamodèles et la sémantique

Il est possible de simplifier la comparaison de modèles de plusieurs façons. Nous avons déjà dit que les modèles sont conformes à leur métamodèle, nous mettrons cela sur le plan syntaxique. D'un point de vue sémantique, les modèles ont aussi des propriétés à respecter, comme nous l'avons vu (section 6) avec le problème de l'oracle. Dans le cadre des transformations de modèles, la combinaison des propriétés sémantiques et structurelles peut conduire à d'importantes simplifications.

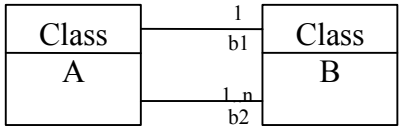


Figure 15 – Métamodèle pour la simplification de comparaison

La Figure 15 illustre un exemple d'un métamodèle qui crée clairement une structure de graphe dans les modèles qui s'y conforment. La partie gauche de la Figure 16 illustre un modèle possible conforme à ce métamodèle, dans ce cas il s'agit clairement d'un graphe. Les 2 relations différentes entre 2 mêmes classes du métamodèle compliquent la comparaison de ses modèles. Par contre il est possible de disposer d'une propriété sémantique qui impose que l'instance de la classe B renvoyée par la relation b1 soit une des instances de la relation b2, le modèle ressemble alors à la partie droite de la Figure 16. On n'est plus alors réellement dans une structure de graphe quelconque puisque la relation b1 est indissociable d'une relation b2. Cette exemple de simplification est parfaitement réaliste comme le montre l'étude de cas (8.3.2).

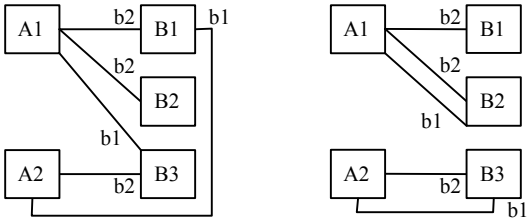


Figure 16 - Sérialisation de modèle

8 Mise en application

Les travaux de recherche que nous avons menés durant ce stage ont pu être mis en application et évalués, cela est présenté dans cette partie.

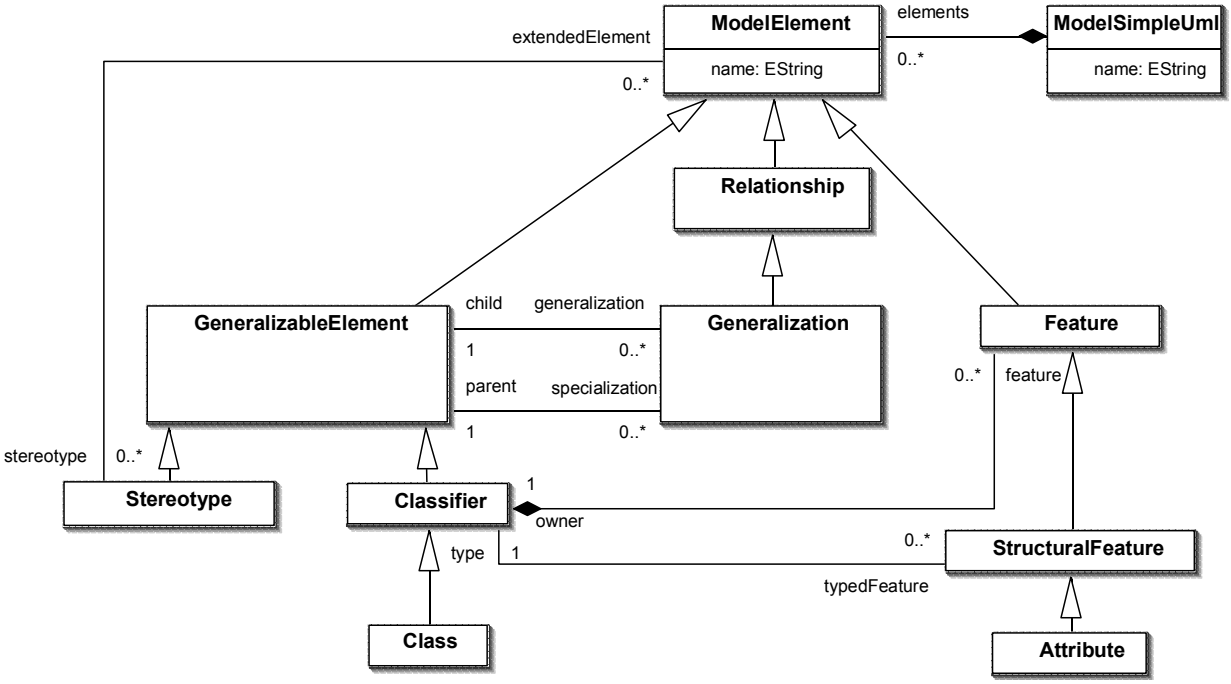


Figure 17 - Le métamodèle SimpleUML

8.1 Présentation de l'étude de cas

Toutes les techniques qui sont présentées par la suite ont été mises en application dans le cadre d'une transformation de modèles qui convertit un diagramme de classe UML en une base de données RDBMS. Pour chaque classe persistante il faut créer une table de même nom dont les colonnes correspondent aux attributs de la classe ; une colonne appropriée est finalement choisie comme clef pour chaque table. Nous avons d'abord extrait la partie du métamodèle UML qui nous intéresse (illustrée Figure 17) et écrit le métamodèle SimpleRDBMS (illustré Figure 18).

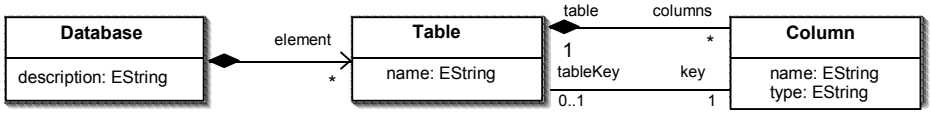


Figure 18 – Le métamodèle SimpleRDBMS

Il s'agit d'une transformation relativement simple mais toutefois représentative de ce qu'est une transformation, puisqu'elle comporte un certain nombre de navigations et de filtrages à la fois sur les modèles d'entrée et de sortie, et de créations d'instances (il s'agit en fait de l'exemple de la partie 5.2 avec la Figure 12). L'automatisation des techniques de test étant un des buts de ce stage, il a fallu dans un premier temps réaliser des opérations fastidieuses manuellement. Le problème de la comparaison de modèles n'étant pas encore complètement résolu et mis en œuvre, le métamodèle de sortie se devait d'être suffisamment simple pour réaliser un programme de comparaison ad hoc.

Plusieurs spécifications sont possibles pour la transformation comme par exemple sur le choix des attributs qui seront créés comme colonnes des tables. Il est possible de conserver uniquement les attributs des classes persistantes, ou de récupérer aussi ceux des classes parentes. Ainsi plusieurs niveaux de contrats sont envisageables, ils seront plus ou moins perspicaces et rigoureux, sur la provenance des attributs par exemple. L'intérêt est de nous permettre d'évaluer les différents niveaux de contrats possibles et ainsi fixer l'effort nécessaire à fournir pour créer des contrats suffisamment perspicaces sans être trop complexes.

L'étude de transformations plus complexes est prévue dans les semaines à venir en partenariat avec France Telecom R&D.

8.2 Utilisation de JAVA, EMF, KMF, mujava

Pour cette étude nous avons choisi d'utiliser JAVA. Sans être un langage spécialement dédié aux transformations de modèles il permet, une fois associé au dépositaire de modèles EMF (Eclipse Modeling Framework), de créer des modèles conformes aux métamodèles eux-mêmes conformes à Ecore. Ecore est la correspondance JAVA de ce qu'est le MOF (Meta Object Facility) dans le MDA (Model Driven Architecture) de l'OMG (Object Management Group).

8.3 Outils créés

Nous avons apporté notre contribution dans trois techniques nouvelles que sont la mutation de programmes de transformations de modèles, la comparaison de modèles et l'oracle du test de transformations de modèles. Puisqu'il n'y avait pas encore de théorie sur ces sujets de recherche, il n'y avait pas non plus d'outillage pour leur mise en œuvre. Nous avons donc créé nos propres outils, présentés ici.

8.3.1 Mutation

D'après le procédé présenté Figure 11, plusieurs étapes composent l'analyse par mutation. Il faut commencer par insérer les erreurs définies par les opérateurs de mutation dans le programme original pour créer les mutants. Cette étape nécessite un outil automatique car le nombre de mutants augmente fortement avec la taille du programme sous test. Nous avons utilisé Mujava pour expérimenter l'insertion automatique des erreurs. C'est un outil de recherche créé spécialement pour les opérateurs de mutation orientés objet [34]. Cet outil, peut dans le cadre de programme simple, effectuer tout le processus de mutation avec un oracle par différence de trace. Dans le cadre de notre transformation de modèle, cet outil est peu adapté. Il ne considère que 5 modèles de

faute non orientés objet et 23 modèles de fautes orientés objet alors que notre programme contient peu de ses spécificités (pas d'héritage, pas de polymorphisme). Nous avons pu générer 29 mutants automatiquement ce qui est trop peu. Finalement nous avons créé manuellement une centaine de mutant grâce à l'utilisation de plus de modèles de fautes.

Il n'y avait aucun outil capable de prendre en compte les nouveaux modèles de fautes que nous proposons, nous avons donc manuellement créé une cinquantaine de mutant avec nos opérateurs. L'automatisation (indispensable) de cette étape reste à étudier. Il ne s'agit plus ici d'une étude du code de la transformation comme dans le cas des autres opérateurs de mutation. Pour savoir quelle relation peut être altérée, il faut disposer des métamodèles d'entrée et de sortie. La complexité de l'implantation en Java de ces opérateurs rend l'opération plus difficilement automatisable. Cela semble tout de même possible et sera certainement étudié par la suite.

Puis on considère les données de test une à une, elles sont exécutées avec le programme original pour donner un résultat resP et avec les n programmes mutants pour obtenir autant de résultat $resMut_i$. Puis chaque résultat d'un mutant est comparé au résultat du programme sous test. Le procédé d'exécution des données de test a été automatisé. Nous disposons de 146 mutants et les ensembles de données de test utilisés atteignent facilement plusieurs dizaines d'éléments, ce qui fait plusieurs milliers voire même plusieurs dizaine de milliers de comparaison à effectuer. Le processus de mutation est un processus long, comme nous le constatons avec notre étude de cas. Son automatisation est indispensable, sans quoi il devient immédiatement coûteux en temps nécessitant la mobilisation de personnes pour réaliser cette phase fastidieuse. Si le coût en temps « humain » est considérablement diminué par une automatisation du processus, il reste tout de même important en termes de délai et d'utilisation de temps machine.

8.3.2 Comparaison ad hoc

Nous ne disposions pas d'outil pour réaliser des comparaisons de modèles, il a donc été nécessaire de créer un programme ad hoc, pour chacune de nos transformations, qui permette la comparaison des modèles de sortie, vis-à-vis du résultat du programme de référence.

Dans la Transformation uml2rdbms, les modèles de sortie sont des bases de données. Pour comparer deux modèles résultats, nous n'avons pas uniquement tiré parti du métamodèle simpleRDBMS mais aussi de la sémantique d'une base de données, comme présenté au 7.3.2. La difficulté de la transformation vient des 2 relations entre les classes Table et Column (Figure 18) qui font de ce diagramme un graphe. En supposant qu'une base de donnée n'a pas 2 tables de même nom et que la clef d'une table est forcément une de ses colonnes, le type de simplification expliqué au 7.3.2 a pu être appliqué : la relation de clef ne peut être dissocier d'une relation de colonne, comme illustré Figure 19 (la partie gauche est impossible alors que la partie droite est conforme sémantiquement et beaucoup plus simple à comparer). Cette simplification ramène le problème à une comparaison d'arbre. Un parcours du modèle permet de le sérialiser en une liste de listes et la comparaison devient possible.

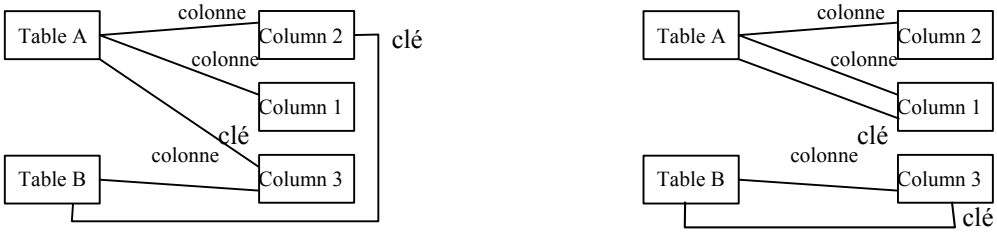


Figure 19 - Simplification de comparaison de modèles

Le contrôle des propriétés sémantiques des modèles peut être formalisé au moyen d'assertions. Un modèle UML par exemple doit répondre à plusieurs dizaines de contraintes pour être conforme. L'utilisation d'OCL est une nouvelle fois une bonne solution.

8.4 Expérimentations

Le but des expérimentations est l'évaluation des nouvelles techniques proposées pour l'analyse de mutation et l'évaluation de l'oracle par contrat pour les programmes de transformation de modèles.

Le processus a débuté par la création des mutants, comme expliqué au 8.3.1, nous avons ensuite récupéré des ensembles de données de test générés automatiquement suivant différents critères, puis nous avons procédé au calcul des scores de mutation.

8.4.1 Etude des résultats

La Figure 21 illustre l'efficacité de chaque donnée de test avec des mutants classiques. Les données de test sont représentées par les colonnes de points, chaque point correspondant au mutant tué par la donnée. On voit donc clairement que la 7^{ième} donnée de test tue beaucoup de mutants alors que la 5^{ième} en tue assez peu. Ce constat est intéressant pour l'évaluation de l'efficacité des données de test. Cependant, dans un premier temps, nous nous restreignons à l'étude de la pertinence des mutants. Il faut alors raisonner ligne par ligne. Ainsi nous constatons que certaines lignes sont complètement vides, par exemple de la ligne 88 à la ligne 97, ce qui signifie que ces mutants n'ont été tués par aucune donnée de test et sont donc particulièrement difficiles à détecter.

La Figure 22 illustre l'efficacité de chaque donnée de test avec les nouveaux mutants proposés. Il s'agit des opérateurs sémantiques que nous avons définis dans ce rapport pour être représentatif des fautes spécifiques survenant dans une transformation de modèles. Nous constatons que certains de ces mutants spécifiques ne sont pas tués, que d'autres sont tués par toutes les données de test (20, 28), et que certains le sont par une seule donnée de test (35, 37, 41). Il est plus intéressant de faire une étude par opérateur de mutation : les mutants de 1 à 5 sont issus de l'opérateur RRMC (la répartition se trouve dans le Tableau 3). Ainsi certains opérateurs génèrent des mutants résistants qui ne sont tués par aucune donnée de test (opérateur RRMC, MMRR). Au contraire, d'autres opérateurs tels que l'opérateur RFTC, sont plus faibles puisque tous les mutants qu'il produit sont tués par l'ensemble des données de test. Cependant, aucun des mutants de cet opérateur n'est tué par toutes les données de test. Ce type d'opérateurs est donc intéressant puisqu'il oblige à avoir des données de test qui diffèrent en fonction des mutants produits.

Le tableau 2 synthétise les résultats que nous avons obtenus avec la transformation UML2RDBMS et un ensemble pertinent de données de test. Les trois lignes correspondent aux différents types de mutants obtenus avec les différents opérateurs de mutation, en commençant par les opérateurs classiques (orientés objet ou non) puis en présentant les opérateurs dédiés aux transformations de modèles que nous proposons. La dernière ligne est calculée sans distinction sur les opérateurs (classiques et spécifiques). La deuxième colonne correspond aux scores de mutation obtenus avec un oracle basé sur la comparaison de modèles et en utilisant le programme initial comme référence. Les scores élevés montrent que l'ensemble des données de test est pertinent et qu'il signale les erreurs injectées. La troisième colonne concerne uniquement les contrats qui assurent que le modèle de sortie est conforme (structurellement et sémantiquement) à son métamodèle. Les quatre colonnes suivantes donnent les résultats des contrats qui lient les modèles d'entrée et de sorties, en ordre croissant de robustesse. On les appelle « lien » parce qu'ils lient la donnée résultat à la donnée d'entrée.

lien 0 : cette post condition vérifie que chaque classe persistante a une table correspondante

lien 1 : cette post condition vérifie lien 0 et que les attributs de chaque classe persistante (non hérités) ont leurs colonnes correspondantes dans la bonne table

lien 2 : cette post condition vérifie en plus que le type des colonnes correspond au type des attributs

lien 3 : cette post condition vérifie en plus qu'il n'y a pas de table ni de colonne inutile et considère aussi les attributs hérités des parents directs (et pas des parents des parents…)

A chaque nouvelle post condition, nous avons augmenté la précision du lien assuré entre le modèle d'entrée et son modèle de sortie correspondant.

Finalement la post condition de la dernière colonne utilise à la fois les contrats de conformité et de lien 3.

	Nombre de mutants	Score de mutation (%)	Contrat de conformité (%)	Contrat de lien 0 (%)	Contrat de lien 1 (%)	Contrat de lien 2 (%)	Contrat de lien 3 (%)	Contrat global (%)
classique	96	91,7	64,6	74,0	83,3	88,5	91,7	91,7
modèle	46	89,1	67,4	58,7	67,4	71,7	78,3	87,0
total	142	90,8	65,5	69,0	78,2	83,1	87,3	90,1

Tableau 2- Scores de mutation et efficacité des contrats

Les données de test tuent 90% des mutants, avec un oracle « parfait » de comparaison des modèles résultats avec les modèles attendus (les premiers obtenus avec les mutants, les seconds avec le programme sous test). Les contrats ne peuvent pas dépasser ces scores, quelque soit leur efficacité. Relativement à ces scores, on constate que les contrats basés uniquement sur les propriétés des modèles de sortie détectent plus de 60% (65.5) des mutants soit plus des deux tiers du score de mutation idéal. Cependant, la seule utilisation de ce type de contrat est insuffisante car la transformation pourrait toujours rendre la même base de données valide sans que ces contrats puissent le détecter. Il faut donc combiner les 2 types de contrats pour obtenir un score de 90% qui est très proche (0.7% entre 90.1 et 90.8) du score obtenu par comparaison de modèles. L'augmentation de la précision des contrats induit une augmentation du score. Nous pouvons donc conclure que l'intérêt des contrats comme fonction d'oracle est important. Ce score de 90% est souvent considéré satisfaisant et l'effort produit pour l'obtenir (pour la création de la post condition lien 3) n'est pas excessif. Il faut être prudent lors de l'écriture de tels contrats qui deviennent très rapidement complexes et risquent eux aussi d'être erronés. Ainsi pour illustrer cette complexité de l'écriture des contrats, il suffit de considérer la post condition lien 3 en OCL :

```
context MetaUmlRdbms inv:self.modelUml.elements
  ->select(e|e.oclIsTypeOf(Class) and e.stereotype->exists(s|s.name='persistent'))
  ->collect(ecp|ecp.oclAsType(Class))
  ->forAll(cp|self.database.elements
  ->one(t|t.name=cp.name and
  cp.feature.union(cp.generalization.parent->collect(p|p.oclAsType(Class)).feature)
  ->select(f|f.oclIsTypeOf(Attribute))->collect(fa|fa.oclAsType(Attribute))
  ->forAll(a|t.columns->one(tc|tc.name=a.name and tc.type=a.type.name))
  and t.columns.size()=cp.feature.union(cp.generalization.parent
  ->collect(p|p.oclAsType(Class)).feature)
  ->select(f|f.oclIsTypeOf(Attribute)).size()))
  and self.database.elements.size()=self.modelUml.elements
  ->select(e|e.oclIsTypeOf(Class) and e.stereotype
  ->exists(s|s.name='persistent')).size()
```

Nous pouvons tirer un autre résultat de ce tableau en constatant que les mutants issus des opérateurs spécifiques sont moins bien détectés que les opérateurs classiques. Cela montrent que les opérateurs spécifiques que nous proposons sont plus pertinents.

9 Conclusion

Cette étude a porté sur des briques essentielles pour fiabiliser les transformations de modèles : l'analyse de mutation, la comparaison de modèles et l'oracle.

Nous avons proposé une nouvelle manière d'aborder la mutation dans ce nouveau contexte de l'ingénierie des modèles. La décomposition des transformations en étapes fonctionnelles nécessaires (navigation, filtrage, création), nous a permis de proposer des opérateurs de mutation originaux. Ces opérateurs permettent d'implanter des erreurs plus sémantiques que syntaxiques ce qui est une manière originale de définir la mutation et qui permet d'être indépendant du langage utilisé.

L'étude de la comparaison de modèles nous a permis d'en cerner les difficultés. Nous avons ainsi pu proposer des heuristiques se basant à la fois sur la structure et la sémantique des modèles pour simplifier les comparaisons qui sont particulièrement complexes au premier abord.

Les comparaisons de modèles peuvent servir d'oracle. Cependant pour être applicable dans un contexte général, nous avons proposé d'automatiser l'oracle en utilisant la spécification des transformations sous forme de

contrats exécutables. Les contrats permettent ainsi d'assurer l'intégrité des modèles de sorties vis-à-vis de leur métamodèle et de contrôler que les modèles de sorties correspondent (en tenant compte des modifications apportés) aux modèles d'entrée. De plus nous exposons une technique qui permet d'assurer en plus de la complétude de l'oracle (classiquement étudiée), la correction de cet oracle. Pour cela, nous tirons parti des résultats obtenus avec l'analyse de mutation, en exploitant les mutants équivalents.

Des perspectives intéressantes s'offrent à nous. L'analyse de mutation et l'oracle automatique peuvent être exploités pour la création de composants fiables embarquant leur données de test et un oracle qui assure leur robustesse [35]. Il s'agit de considérer un composant comme la composition de trois facettes : une spécification, une implantation et des tests embarqués (Figure 20). L'oracle est fondé sur la spécification et les tests embarqués sont composés d'un ensemble de données de test évalué par une analyse de mutation. Nous pouvons donc envisagé d'adapter nos travaux pour la construction de composants dans le contexte de l'ingénierie des modèles.

D'autres expérimentations sont en cours et devraient permettre d'aller plus loin dans cette étude en étudiant d'autres transformations de modèles.

10 ANNEXES

Opérateur de mutation	Mutants correspondants
RRMC	1 à 5
RRMA	6 à 18
MSRM	19 et 20
MSRA	21 à 27
MFCM	28 et 29
MFCP	30 à 35
MFCA	36 à 42
MMRR	43 à 45
MRCA	46
RFCA	∅

Tableau 3 - Répartition des opérateurs

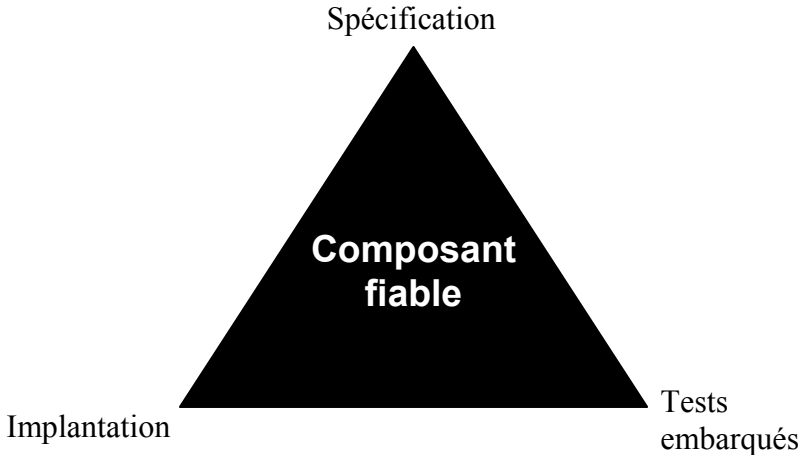


Figure 20 - Composant autotestable

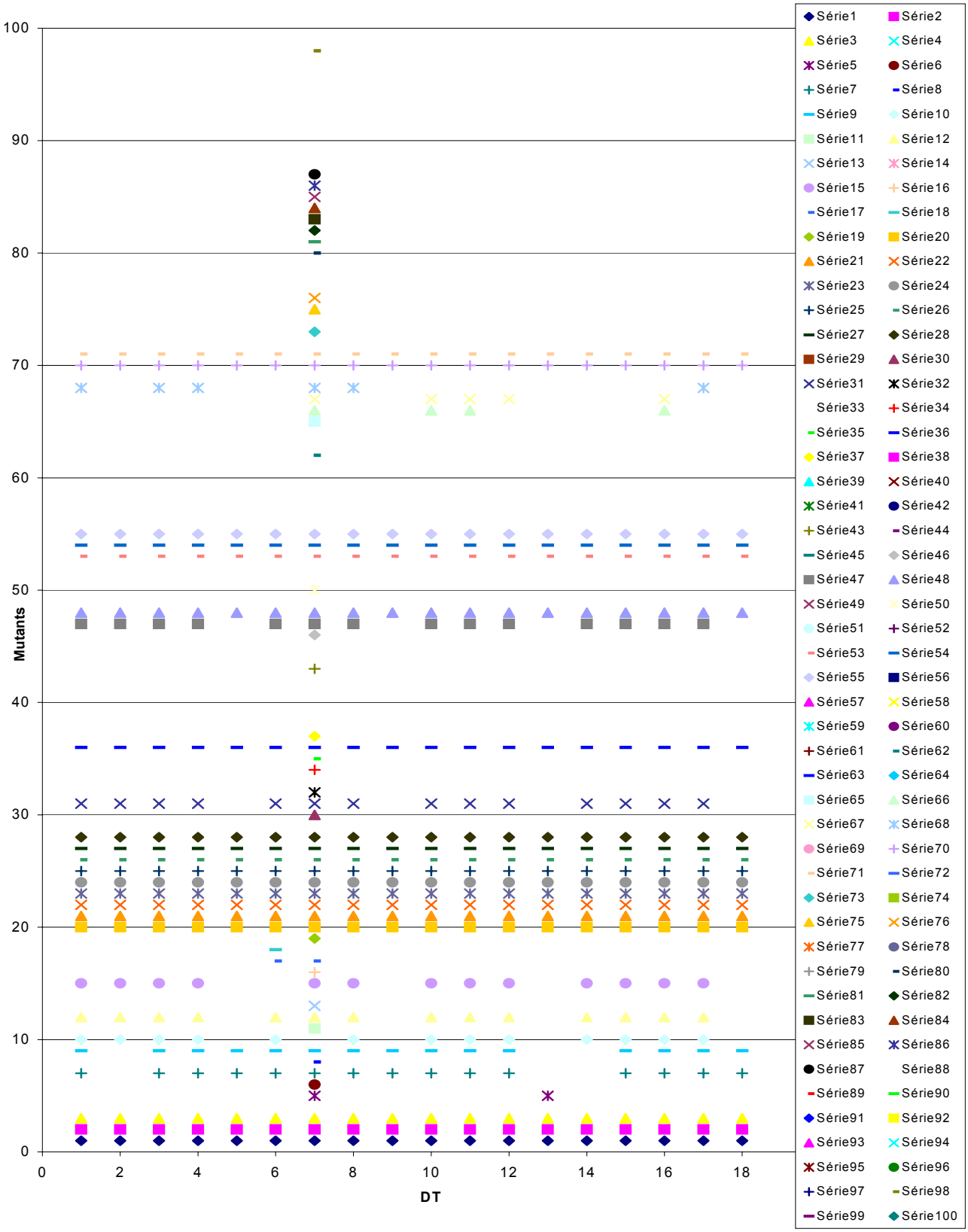


Figure 21 - nuage de mutants classiques

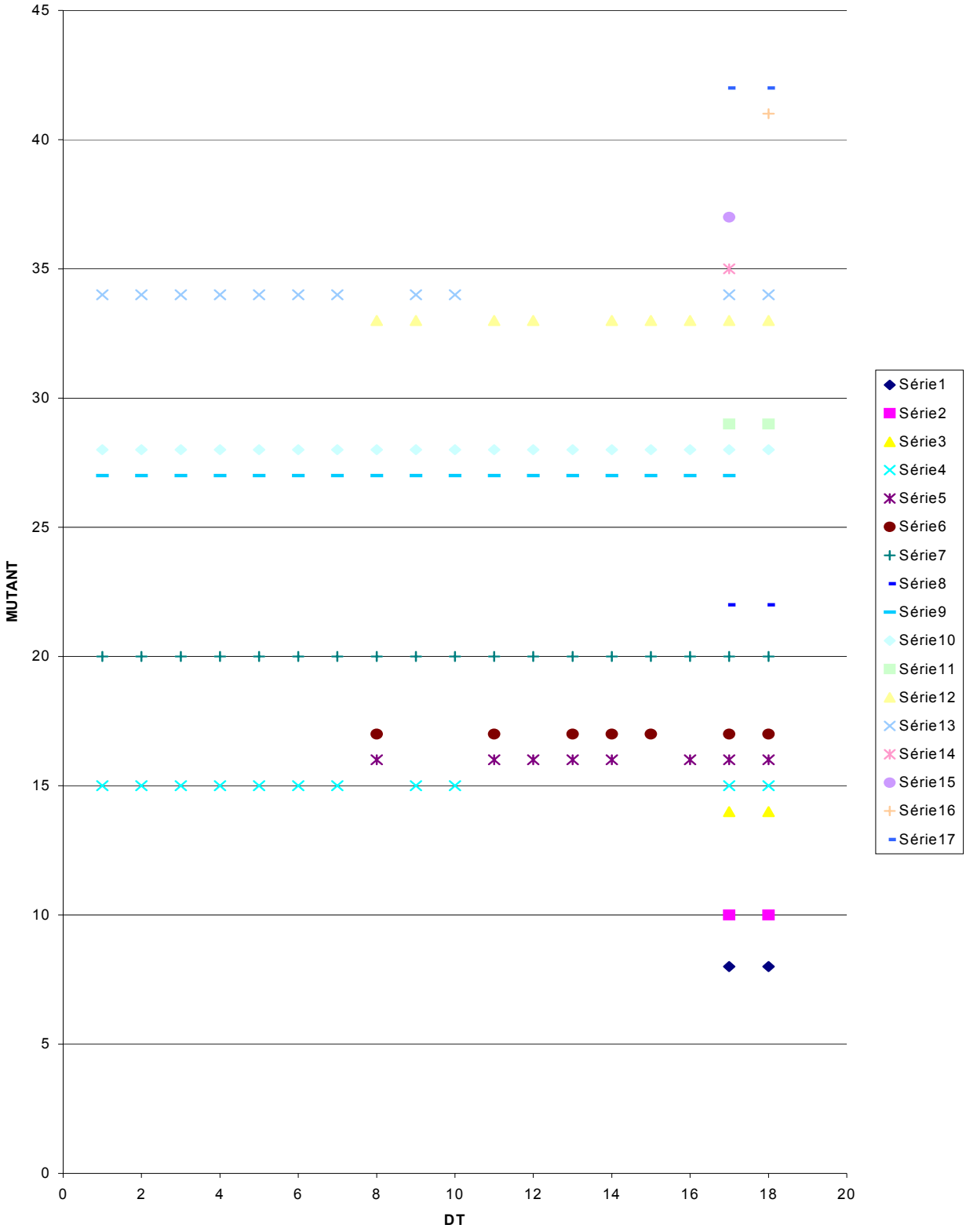


Figure 22 - nuage de mutants modèles

11 Références

[1] 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.

[2] Clark, T., A. Evans, P. Sammut, and J. Willans, *Applied Metamodelling: A Foundation for Language Driven Development*. 2004, Xactium.

[3] OMG, *Meta Object Facility (MOF) Specification*, in *OMG Document*. 1997.

[4] Bézivin, J., M. Blay, M. Bouzhegoud, J. Estublier, J.-M. Favre, S. Gérard, and J.-M. Jézéquel, *Rapport de Synthèse de l'AS CNRS sur le MDA (Model Driven Architecture)*. 2004, CNRS.

[5] Bézivin, J. *From Object Composition to Model Transformation with the MDA*. in *39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*. 2001. Santa Barbara, California. p. 350-354.

[6] Bézivin, J., N. Farcet, J.-M. Jézéquel, B. Langlois, and D. Pollet. *Reflective model driven engineering*. in *UML'03*. 2003. San Francisco, CA, USA. p. 175 -189.

[7] OMG/RFP/QVT, *Query/Views/Transformations RFP*. 2002.

[8] Inria, *Model Transformation at Inria*. http://modelware.inria.fr/.

[9] Fleurey, F., J. Steel, and B. Baudry. *Validation in Model-Driven Engineering: Testing Model Transformations*. in *MoDeVa*. 2004. Rennes, France.

[10] Briand, L.C., Y. Labiche, H.-D. Yan, and D.P. M. *A Controlled Experiment on the Impact of the Object Constraint Language in UML-Based Development*. in *20th IEEE International Conference on Software Maintenance (ICSM'04)*. 2004. Chicago, Illinois. p. 380-389.

[11] Meyer, B., *Applying Design by Contract.* IEEE Computer, 1992. **25**(10): p. 40 - 51.

[12] Pollet, D., D. Vojtisek, and J.-M. Jézéquel. *OCL as a core UML transformation language*. in *WITUML 2002*. 2002. Malaga, Spain.

[13] Xanthakis, S., P. Régnier, and K. Karapoulios, *Le test des logiciels*. 2000, Hermès.

[14] Heckel, R. and M. Lohmann, *Towards Model-Driven Testing.* Electronic Notes in Theoretical Computer Science, 2003. **82**(6).

[15] Binder, R.V., *Testing Object-Oriented Systems: Models, Patterns and Tools*. 1999, Addison-Wesley.

[16] 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.

[17] Atkinson, C. and G. Hans-Gerhard. *Built-In Contract Testing in Model-driven, Component-Based Development*. in *1st International Working Conference on Component Deployment*. 2002. Austin, TX, USA.

[18] Rutherford, M.J. and A.L. Wolf. *A case for test-code generation in model-driven systems*. in *The second international conference on Generative programming and component engineering*. 2003. Erfurt, Germany. p. 377 - 396.

[19] 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.

[20] 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.

[21] Alanen, M. and I. Porres. *Difference and Union of Models*. in *UML'03*. 2003. San Francisco, CA, USA.

[22] Küster, J.M. *Systematic Validation of Model Transformations*. in *WiSME'04(associated to UML'04)*. 2004. Lisbon, Portugal.

[23] Peters, D. and D.L. Parnas. *Generating a test oracle from program documentation: work in progress*. in *ISSTA'94 (Int. symposium on Software Testing and Analysis)*. 1994. Seattle, Washington, United States. p. 58 - 65.

[24] Baudry, B., *Assemblage testable et validation de composants*. 2003, Université de Rennes 1: Rennes. p. 180.

[25] 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.

[26] 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.

[27] Le Traon, Y., B. Baudry, F. Fleurey, and J. Steel, *Model Transformation Testing*. http://www.irisa.fr/triskell/results/ModelTransfoTesting/.

[28] 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.

[29] Offutt, A.J., J. Pan, K. Tewary, and T. Zhang, *An experimental evaluation of data flow and mutation testing.* Software Practice and Experience, 1996. **26**(2).

[30] 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. p. 352 - 363.

[31] Steel, J. and J.-M. Jézéquel. *Model Typing for Improving Reuse in Model-Driven Engineering*. in *MoDELS 2005*. 2005. Jamaica.

[32] Baudry, B., J.-M. Jézéquel, and Y. Le Traon. *Robustness and Diagnosability of Designed by Contracts OO Systems*. in *Metrics'01 (Software Metrics Symposium)*. 2001. London, UK: IEEE Computer Society Press, Los Alamitos, CA, USA. p. 272 - 283.

[33] Garey, M.R. and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, ed. F. co. 1979, New York.

[34] Ma, Y.-S., J. Offutt, and Y.R. Kwon, *MuJava : An Automated Class Mutation System.* Software Testing, Verification and Reliability, 2005. **15(2)**: p. 97-133.

[35] 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*. 2000. San Jose, CA: Kluwer Academic Publishers, Dordrecht, NL. p. 69 - 76.