Génération automatique de tests pour les transformations de modèles

Mottu Jean-Marie[1], Baudry Benoit[1], Brottier Erwan[2], Le Traon Yves[2]

1 IRISA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France

{jean-marie.mottu, bbaudry}@irisa.fr

2 France Télécom R&D, 2 av. Pierre Marzin 22307 Lannion Cedex, France

{erwan.brottier, yves.letraon}@francetelecom.com

Résumé

Dans l'ingénierie des modèles, les transformations de modèles représentent un élément essentiel pour la réutilisation. Il est donc important de proposer des solutions efficaces pour valider les programmes de transformation. Ce travail propose des solutions originales pour le processus de tests de transformations de modèles. Nous proposons des critères pour la génération de données de test. Nous discutons d'une adaptation de l'analyse de mutation permettant d'évaluer l'efficacité des données générées. Enfin plusieurs pistes possibles pour l'oracle sont abordées.

1. Introduction

Dans l'ingénierie des modèles (IDM), 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 transformations automatiques de modèles deviennent donc des entités primordiales dans ce nouveau cadre de développement. Ces transformations sont développées pour être réutilisées dans divers projets et nécessitent des techniques de test efficaces.

Dans ce papier, nous abordons le problème du test de transformation de modèles dans sa globalité : *génération de données* (critères et algorithme), évaluation des données générées et production de l'*oracle*. La difficulté pour le premier point tient à la complexité des données de test manipulées : des modèles. Pour la génération de données, nous profitons que les modèles en entrée sont décrits par un métamodèle. Les critères de test proposés identifient des fragments de modèles pertinents qui doivent apparaître dans une donnée de test. Ces fragments sont appelés des objectifs de donnée de test (ODT) et définissent des parties du métamodèle qui doivent être instanciées avec certaines valeurs intéressantes pour le test. Un algorithme est proposé pour générer automatiquement un ensemble de modèles pour le test qui exhibe tous les ODT.

Le deuxième point important de ce travail est la validation des données de test générées avec les critères de test. Pour cela, nous proposons une adaptation de *l'analyse de mutation* qui est une technique proposée dans [1] pour valider une suite de tests. Cette analyse consiste à injecter des

fautes dans le programme à tester : si la suite de test peut détecter ces erreurs alors elle est considérée efficace. L'efficacité de cette approche tient en grande partie au type d'erreurs injectées. L'adaptation proposée ici consiste à proposer des modèles de fautes (opérateurs de mutation) qui sont significatifs pour les transformations de modèles.

Enfin, le problème de l'oracle est abordé : l'oracle consiste à rendre un verdict sur les modèles produits par la transformation. Ici, nous proposons deux solutions pour produire l'oracle. La première consiste à comparer le modèle attendu au modèle obtenu. Cette approche nécessite des techniques pour la comparaison de modèle, ce qui est un problème en tant que tel. De plus, cette solution requiert le modèle résultat attendu, qui n'est pas toujours disponible pour chaque donnée de test. La seconde solution consiste à utiliser les propriétés attendues sur le modèle produit, si ces propriétés sont disponibles, par exemple sous la forme de post-conditions pour la transformation.

La section 2 introduit l'exemple illustrant l'approche tout au long de l'article. Le processus et les critères pour la génération de données de test sont présentés section 3. L'adaptation de l'analyse de mutation pour les transformations de modèles est étudiée section 4. Enfin, la section 5 aborde le problème de l'oracle. La section 6 présente quelques travaux connexes et la section 7 conclut.

2. Etude 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 classes 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 1) et écrit le métamodèle SimpleRDBMS (illustré Figure 2). Pour cette étude nous avons choisi d'utiliser JAVA pour implanter la trnasformation. Ce n'est pas un langage dédié aux transformations de modèles mais une fois associé à EMF (Eclipse Modeling Framework) il est possible de créer des modèles conformes aux métamodèles eux-mêmes conformes à Ecore, qui 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).

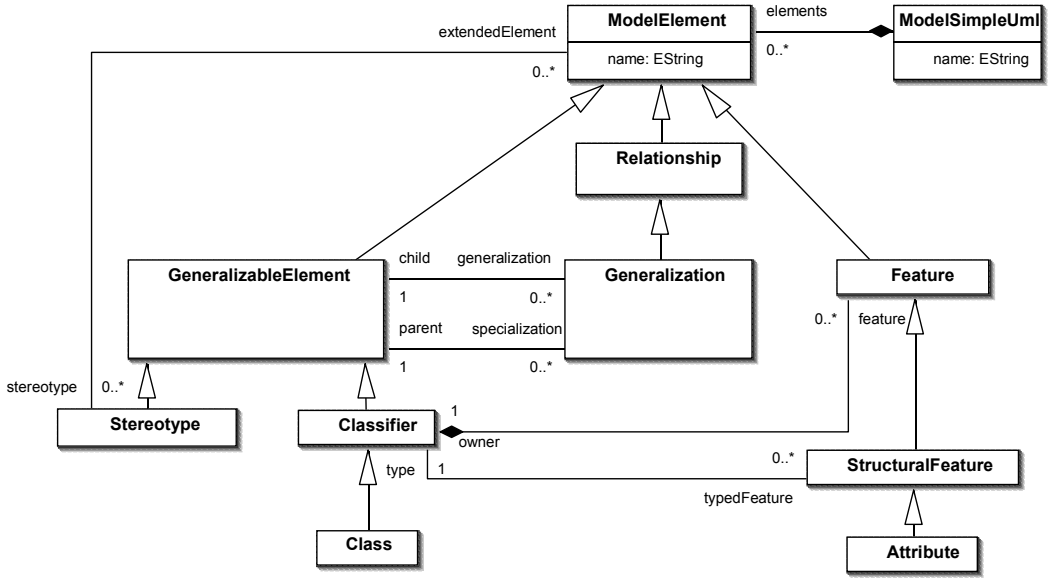

Figure 1 - Le métamodèle SimpleUML

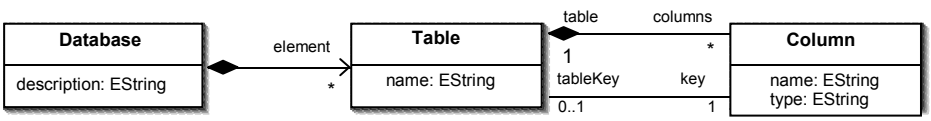

Figure 2 – Le métamodèle SimpleRDBMS

3. Génération de données de test

Cette section aborde le problème de la génération de données de test pour les transformations de modèles. L'approche proposée définit des critères de test fonctionnels (indépendants du langage de transformation utilisé) qui s'appuient sur la description du domaine d'entrée par un métamodèle. 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 3 décrit les différentes étapes que nous avons identifiées dans [2] :

1 *du métamodèle effectif aux partitions*. Cette étape permet de construire et d'identifier des 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.

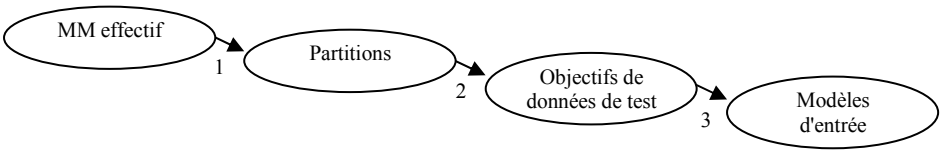

Figure 3 - Processus pour la génération de modèles d'entrée

3.1 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, notre 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 [2] nous donnons plusieurs pistes pour identifier automatiquement le métamodèle effectif. Dans notre étude de cas, le métamodèle SimpleUML (Figure 1) est le métamodèle effectif utilisé dans la transformation et extrait du métamodèle d'UML. Nous n'abordons pas ce point plus en détail dans cet article.

3.2 Partitions

Pour couvrir l'ensemble du domaine d'entrée d'une transformation, nous adaptons le test par partition présenté dans [3]. 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.

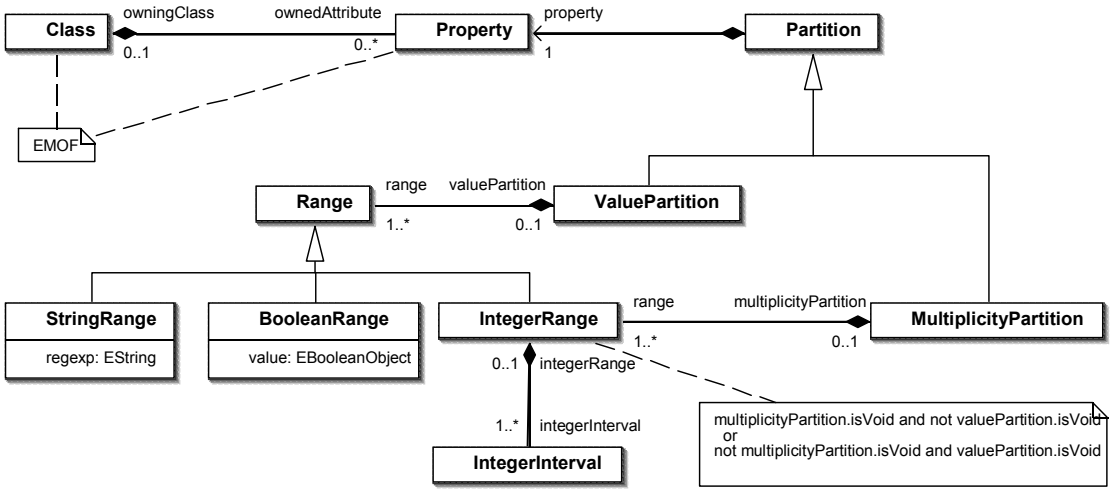

Figure 4 - Le métamodèle Partition

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.*

Nous définissons deux types de partition. 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 4 décrit le métamodèle pour les partitions sur un métamodèle.

Dans [2] nous proposons 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

3.3 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 5.

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 5.*

Exemple :

- un fragment d'objet pour la classe ModelElement (du métamodèle de la Figure 1), qui défini une classe d'équivalence (Partition:Range) 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. Un tel ODT est noté :

  {{(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.

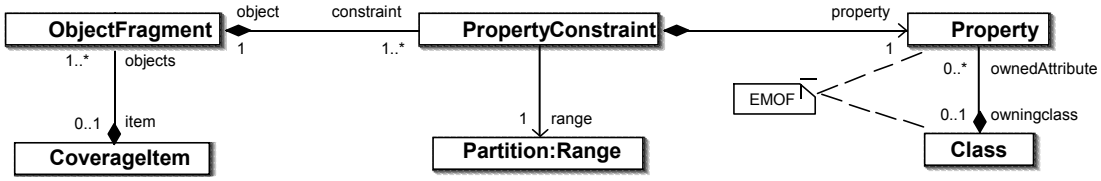

Figure 5 - Le métamodèle ODT

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 3). Un critère de test impose de plus que toute classe concrète du métamodèle d'entrée 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 [4] nous proposons 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 [4]).

Définition – satisfaction à un critère de test*. Un ensemble de modèles d'entrée satisfait un critère de test si et seulement si l'ensemble des 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 objectif 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.*

3.4 Génération de modèles

La dernière étape pour la génération de données de test pour une transformation (Figure 3) consiste à générer un ensemble de modèles d'entrée qui satisfait un critère de test. Nous avons défini un algorithme 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 [4] nous montrons 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.

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.

```
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

Nous avons défini un ensemble de critères de test couvrant le domaine d'entrée d'une transformation à partir de son partitionnement 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.

4. L'analyse de mutation pour le test de transformations de modèles

L'*analyse de mutation* est une technique introduite dans [1], dont le but 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 6 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 (au contraire d'un mutant principal)
- 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.

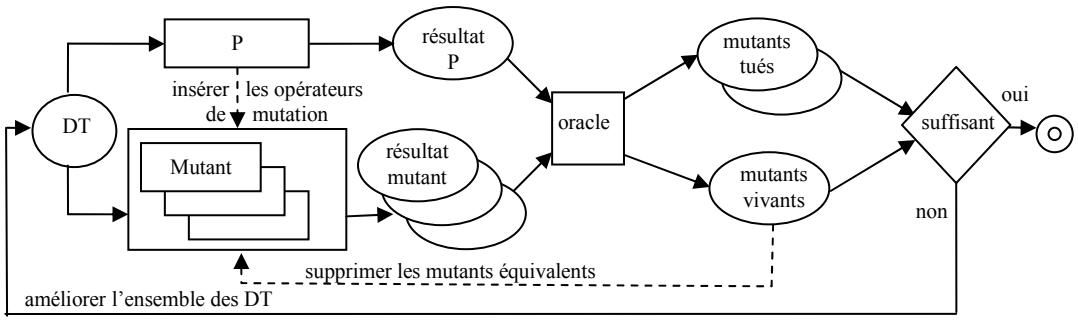

Figure 6 - processus de mutation

4.1 Les opérateurs 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 [5]. Pour exécuter une analyse de mutation avec ces opérateurs, les erreurs sont injectées systématiquement partout où c'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 [6]. 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.

4.1.1 Approche spécifique aux transformations de modèles

Pour évaluer efficacement les données de test générées pour les transformations de modèle, nous estimons nécessaire de proposer des opérateurs adaptés aux transformations. Ces opérateurs doivent refléter le type d'erreur qui apparaît 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 dans le modèle d'entrée pour reconnaître certains éléments à transformer, une erreur peut se produire en navigant de 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.

La transformation UML vers RDBMS donne un exemple de décomposition grâce à ces opérations, illustrée par la Figure 7. 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. La transformation navigue dans le modèle d'entrée (a) 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 dans 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).

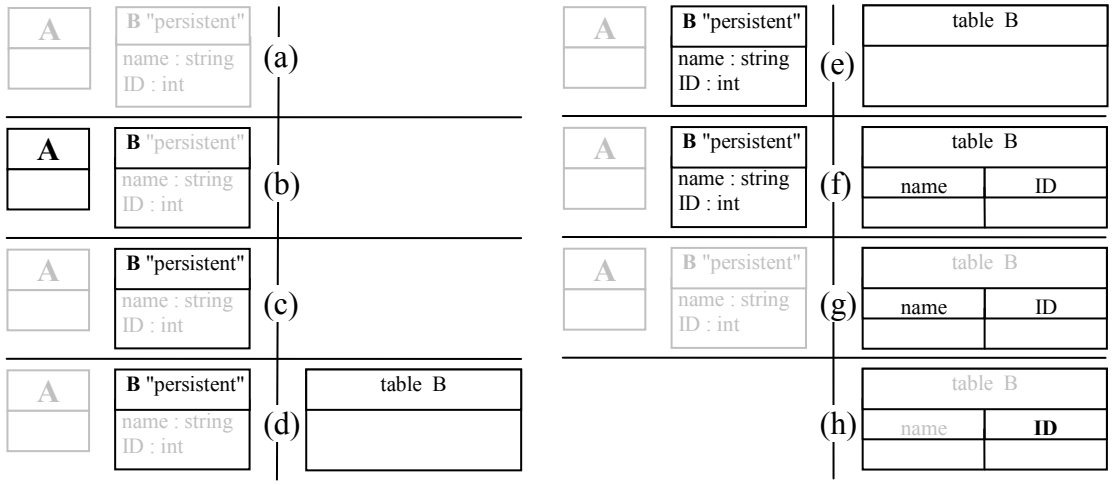

Figure 7 – Déroulement de la transformation de l'étude de cas

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.

4.1.2 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 8 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).

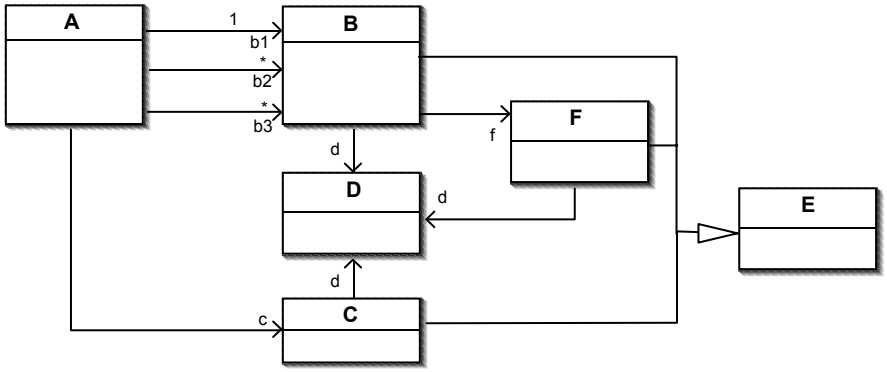

Figure 8 – Exemple de métamodèle

Opérateur portant sur le **filtrage** :

- **modification du filtrage d'une collection avec manque (MFCM)** : le filtrage agit sur une collection pour n'en conserver que les éléments qui intéressent la transformation. 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.
- **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.
- **modification du filtrage sur une collection avec perturbation (MFCP)** : ici il s'agit 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.

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 7 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).

Nous discutons ici l'impact de ces opérateurs sur une transformation de modèles basés sur le métamodèle de la Figure 8. 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.

L'intérêt de ces opérateurs est donc 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. Enfin, 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.

4.2 Conséquence d'une implantation JAVA pour l'analyse de mutation

L'étude de cas présentée section 2 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. Ce n'est probablement pas spécifique à JAVA, l'utilisation de langages trop impératifs, c'est-à-dire manipulant les modèles à un niveau trop bas, trop proche de la structure des modèles (et de la façon de les stocker comme ici avec EMF ou avec le format XMI)

conduira aussi à des opérateurs complexes. En voici un exemple : dans la transformation UML vers RDBMS il faut choisir une clé appropriée parmi les colonnes de chaque table.

La transformation filtre les colonnes pour choisir une clef de type integer ce qui est approprié :

```
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. }
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 `3.`). L'opérateur MFCM peut supprimer le premier filtrage ainsi la première colonne de la table est 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 2 versions 3 instructions sont communes (`1.` `3.` `4.` correspondant à *1.* *2.* *3.*), le filtrage manquant se traduit par 7 lignes de code en moins (de `2-1.` à `2-7.`). Donc la complexité des opérateurs de mutation appliqués en JAVA est réelle, elle 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. L'oracle pour le test de transformation de modèles

Une fois l'ensemble des données de test généré et validé par les 2 techniques qui viennent d'être présentées, il reste à effectuer les tests puis à rendre un verdict. L'oracle a cette fonction, c'est lui qui détermine si les résultats obtenus avec les données de test sont corrects ou pas, ce qui révèlerait la présence d'erreurs dans le programme.

Cette section propose 2 solutions pour l'oracle. La première fonction consiste à comparer le résultat obtenu avec le résultat attendu, il s'agit donc d'une comparaison entre 2 modèles. Dans un deuxième temps, nous proposons une alternative qui consiste à fonder l'oracle sur les assertions du programme de transformation de modèles.

5.1 Différence de modèles

Pour l'oracle la différence de modèles est réalisée entre le modèle obtenu par l'exécution d'une donnée de test et le résultat attendu (qui est censé être disponible). La comparaison de 2 modèles est une opération complexe comme expliquée dans la suite.

5.1.1 Propriétés d'une comparaison de modèles

Les modèles sont des données complexes, bien au delà des données habituellement manipulées dans les programmes même par rapport aux objets des langages orientées objet. En effet les modèles peuvent être utilisés pour représenter des systèmes entiers ou tout du moins des programmes.

Dans un souci d'harmonisation et de clarification, des normes ont été proposées pour l'ingénierie des modèles par l'OMG par exemple qui propose le MDA (Model Driven Architecture) et le MOF (Meta-Object Facility). Ainsi les métamodèles qui définissent comment écrire des modèles sont eux-mêmes conformes aux MOF (ou à Ecore pour EMF). Cela peut-être exploité pour proposer des outils généraux de comparaison de modèles. Les métamodèles fournissent des heuristiques sur la structure des modèles à comparer, cela donne des indications sur leur complexité, leur particularité. Un exemple est donné dans la partie suivante (5.1.2).

On peut aussi considérer nos modèles comme des graphes, en considérant les éléments qui le composent comme des nœuds et leurs relations comme des arcs. L'isomorphisme entre 2 graphes est un problème NP-complet [7] dans le cas général, de complexité exponentielle donc, ce qui est un obstacle important, et les notions de modélisation qui sont ajoutées aux graphes ne simplifient pas la tâche. En effet la correspondance modèle/graphe n'est pas directe, ce sont les diagrammes d'instance qu'il faut considérer (et non les diagrammes dans une forme plus commune comme les diagrammes de classes UML cela évite par exemple d'avoir plusieurs types de relations différents), cela donne des graphes non orientés mais dont les nœuds sont complexifiés par les attributs des classes et sont susceptibles de porter les mêmes noms.

La comparaison de modèles est aussi utilisée dans l'analyse de mutation, dans ce cas la comparaison est faite entre les résultats d'une donnée de test exécutée avec le programme et avec un 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 pour notre étude de cas. Ne disposant pas d'outil général de comparaison de modèles, nous avons écrit un programme JAVA ad hoc qui se base sur le métamodèle SimpleRDBMS (Figure 2) écrit pour la transformation.

5.1.2 Élément de solution basé sur les métamodèles

Dans notre étude de cas, nous avons commencé par simplifier le problème de la comparaison de 2 modèles SimpleRDBMS en rejetant les modèles qui ne correspondent pas structurellement au métamodèle ou au sens d'une base de données. L'utilisation de JAVA et EMF assure la conformance structurelle vis-à-vis du métamodèle. Certains contrats sur les modèles sont exprimées sous une autre forme, en OCL par exemple, ainsi une base de données ne peut pas avoir 2 tables de même nom, la clef d'une table fait parti de ses colonnes... Il est utile de se servir de ces contrats (ce qui est aussi abordé au 5.2) pour rejeter une partie des modèles qui dans tous les cas ne sont pas intéressants (ce qui permet de facilement relever des fautes et tuer des mutants). Ainsi l'utilisation du métamodèle pour baser la comparaison de modèles est moins complexe, cela reste généralement une comparaison de graphes mais dans certain cas comme notre exemple, il peut s'agir d'une comparaison d'arbres beaucoup plus simple à mettre en oeuvre. De plus, un modèle a un point d'entrée ce qui diminue encore la complexité de la comparaison.

5.1.3 Critique de la comparaison de modèle pour l'oracle

Pour utiliser la comparaison de modèle comme oracle il faut disposer du modèle attendu ce qui n'est pas évident. Pour le test de non-régression ou dans le cadre d'un changement de plate-forme les résultats des tests sont disponibles mais ce n'est pas toujours le cas. Cela nous conduit à proposer une alternative dans la partie suivante.

5.2 Contrats de la transformation

Une deuxième méthode peut être utilisée comme oracle pour valider les tests qui consiste à utiliser les assertions. Les travaux présentés dans [8] montrent l'intérêt de l'utilisation des contrats de 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 [9]). Les bénéfices de cette approche peuvent être transcrits dans l'ingénierie des modèles et particulièrement dans la partie oracle du test

de transformation de modèles, ce que nous nous efforçons à faire.

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 si la transformation utilise un environnement qui se base sur les métamodèles pour la création des instances du modèle, en tout cas sur le plan structurel. 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, c'est le cas dans 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 parti 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 [10]). Les pre-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 puisqu'ils doivent être vrais tout au long de la transformation, y compris quand elle se termine et rend son résultat, ce qui intéresse l'oracle. 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 c'est au développeur (de la transformation de préférence ou des tests à défaut) d'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 4.1.1. 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 [8] nous avons 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. Etat de l'art

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 [11], 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 [12], 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 [13] 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 [14], 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.

Dans [10], 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 contrats liant les modèles d'entrée et de sortie. Cela montre que les hypothèses que nous avons faites sur les possibilités de formalisation de la spécification sont justifiées et nous permettent d'envisager l'utilisation des contrats pour l'oracle.

7. Conclusion

Les travaux présentés dans cet article abordent le test de transformations de modèles dans sa globalité en proposant des solutions pour trois activités importantes : la génération de données de test, l'évaluation de la pertinence de ces données et l'oracle. A chaque étape les solutions étudiées prennent en compte la complexité du test dans le cadre des transformations de modèles, et s'appuient aussi sur leurs caractéristiques pour proposer des solutions spécifiques et adaptées. Tout d'abord nous tirons parti de la description du domaine d'entrée d'une transformation sous forme d'un métamodèle pour proposer des critères de test et un algorithme de génération automatique de données de test. Ensuite nous proposons une analyse générique de la conception et l'implantation d'une transformation pour adapter les modèles de fautes utilisés pour l'analyse de mutation. Enfin,

nous abordons deux solutions possibles pour l'oracle : en comparant les modèles obtenus et attendus, ou en utilisant les assertions et propriétés définies pour la transformation.

Une première implantation de l'algorithme de génération de données de test en fonction des critères définis est actuellement disponible et des expériences sont en cours pour évaluer l'algorithme. Nous outillons maintenant l'analyse de mutation et l'oracle pour valider les modèles de faute ainsi que la pertinence des heuristiques proposées pour l'oracle.

8. Références

[1] 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.

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

[3] 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.

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

[5] 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).

[6] 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.

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

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

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

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