Contrats de transformation pour la validation de raffinement de modèles

Eric Cariou — Nicolas Belloir — Franck Barbier

Université de Pau et des Pays de l'Adour
LIUPPA
B.P. 1155, 64013 Pau Cedex France
{Eric.Cariou, Nicolas.Belloir, Franck.Barbier}@univ-pau.fr

RÉSUMÉ. *Un processus logiciel basé sur l'ingénierie des modèles est construit à partir d'un ensemble de transformations qui sont exécutées en séquence pour partir d'un niveau de modélisation abstrait et arriver au code final ou à une spécification d'implémentation détaillée. Ces transformations correspondent pour beaucoup à des raffinements, c'est-à-dire à de l'ajout de détails ou d'information sur un modèle. Ces raffinements peuvent être entièrement automatisés ou nécessiter l'intervention manuelle du concepteur. Nous proposons ici une méthode pour valider que le résultat d'une transformation, y compris lorsque le concepteur intervient manuellement sur les modèles, respecte la spécification d'un raffinement. Nous nous basons pour cela sur des contrats de transformations de modèles écrits en OCL afin de rendre notre méthode indépendante des outils de modélisation et de transformations de modèles.*

ABSTRACT. *A model-driven engineering process relies on a set of transformations which are sequentially executed, starting from an abstract level to produce code or a detailed implementation specification. These transformations are mostly refinements, that is to say detail or data added to models. These refinements may be entirely automated or may require manual intervention by designers. In this paper, we propose a method to demonstrate that a transformation result is correct with respect to the specification of the refinement. This method both includes automated transformations and manual interventions. For that, we focus on transformation contracts written in OCL. This leads to make the proposed method independent of modeling and transformation tools.*

MOTS-CLÉS : *IDM, contrats de transformation, raffinement, OCL*

KEYWORDS: *MDE, transformation contracts, refinement, OCL*

1. Introduction

Un processus logiciel basé sur l'ingénierie des modèles est construit à partir d'un ensemble de transformations qui sont exécutées en séquence pour partir d'un niveau de modélisation abstrait et arriver au code final ou à une spécification d'implémentation détaillée. Ces transformations correspondent pour beaucoup à des raffinements, c'est-à-dire à de l'ajout de détails ou d'information sur un modèle. Ces raffinements peuvent être entièrement automatisés ou nécessiter l'intervention manuelle du concepteur. En effet, même si le but principal de l'ingénierie des modèles est d'automatiser un processus logiciel complet, le concepteur doit souvent intervenir sur les modèles et faire des choix sur certaines actions à réaliser.

Il est important de pouvoir garantir que les transformations effectuées sont valides, c'est-à-dire qu'elles respectent la spécification du raffinement. Cela est d'autant plus important quand le concepteur intervient manuellement sur le modèle. Dans ce cas là, le modèle peut être grandement modifié sans contraintes particulières et il faut s'assurer que la modification reste dans le cadre de ce qui est attendu.

Nous proposons ici une méthode pour valider qu'une transformation, y compris quand le concepteur intervient manuellement, respecte la spécification d'un raffinement. Nous nous basons pour cela sur des contrats de transformations de modèles écrits en OCL (*Object Constraint Language* (OMG, 2006)) afin de rendre notre méthode indépendante des outils de modélisation et de transformation de modèles. La validation du contrat doit en effet pouvoir se faire à partir de deux modèles – l'un représentant le modèle source et l'autre le modèle cible de la transformation – générés ou obtenus à partir de n'importe quel outil. Notre méthode peut s'appliquer sur n'importe quelle transformation endogène et donc sur un raffinement car c'est un type particulier de transformation endogène. Dans cet article, nous nous intéressons plus particulièrement aux raffinements pour illustrer ces transformations endogènes.

La section suivante résume le concept de contrat de transformations de modèles. Ensuite, un exemple de raffinement de modèle et de son contrat est donné. La section 4 présente la méthode de définition et de validation d'un contrat, appliqué à notre exemple de raffinement. Enfin, avant la conclusion, nous parlons des travaux connexes.

2. Contrats de transformations de modèles

La programmation et conception par contrat (Meyer, 1992, Beugnard *et al.,* 1999) consiste à spécifier ce que fait un élément logiciel, programme ou modèle, afin de savoir comment l'utiliser et valider que l'on a bien obtenu le résultat attendu. En termes d'opérations, qu'il s'agisse d'une méthode d'un programme ou d'une classe d'un diagramme de classes, l'approche par contrat consiste à spécifier une pré et une post-condition. La pré-condition définit l'état du système à respecter pour que l'opération puisse être appelée et la post-condition, l'état que le système s'engage à respecter après l'appel. On peut également préciser un ensemble d'invariants que l'élément logiciel doit respecter en permanence.

Dans le contexte des transformations de modèles, nous avons proposé dans (Cariou *et al.,* 2004b, Cariou *et al.,* 2004a) d'appliquer ces principes à une opération de transformation de modèles et donc de définir des contrats de transformations. Ces contrats spécifient alors des transformations de modèles en établissant des contraintes sur l'état du modèle avant la transformation (modèle source) et l'état du modèle après la transformation (modèle cible). Ils servent à garantir qu'un modèle cible est bien le résultat valide d'une transformation par rapport à un modèle source, ou bien encore qu'un modèle source peut être transformé.

Un contrat de transformation est défini par trois ensembles de contraintes :

Contraintes sur le modèle source : contraintes à respecter par un modèle pour pouvoir être transformé

Contraintes sur le modèle cible : contraintes générales (indépendamment du modèle source) à respecter par un modèle pour qu'il soit le résultat valide de la transformation

Contraintes d'évolution d'éléments : contraintes à respecter sur l'évolution de certains éléments entre le modèle source et le modèle cible, pour que le modèle cible soit le résultat valide de la transformation par rapport au modèle source

Dans (Cariou *et al.,* 2004b, Cariou *et al.,* 2004a), nous avons établi les bases conceptuelles autour des contrats de transformations écrits en OCL. Dans cet article, nous validons en pratique cette approche via une méthode et une implémentation complète dans le contexte des transformations endogènes (c'est-à-dire à méta-modèle constant). Nous détaillons également des points que nous avions seulement abordés comme le problème essentiel de la correspondance entre éléments des modèles source et cible (voir la section 4.3.2).

Le choix d'OCL comme langage d'expression des contrats se justifie pour plusieurs raisons. La première est qu'OCL est par nature un langage d'expression de contraintes et qu'un contrat est un ensemble de contraintes. Ensuite, notre souci était de pouvoir définir un contrat le plus indépendamment possible des plates-formes et des outils. OCL est pour cela un bon candidat puisque c'est un standard utilisable sur des modèles de types variés, comme UML, le MOF ou bien encore pour la plate-forme Eclipse/EMF qui est une des principales plates-formes de développement d'outils d'ingénierie des modèles. On peut donc vérifier des contraintes OCL sur des modèles générés par une grande majorité des moteurs de transformation ou de manipulation de modèles. Enfin, OCL est un langage qui est relativement bien connu. Il est en effet difficile de définir un méta-modèle ou un modèle sans avoir besoin d'y ajouter des contraintes supplémentaires, écrites par exemple et bien souvent en OCL. OCL est également utilisé comme langage de requêtes ou est étendu dans un certain nombre de langages de transformations de modèles (par exemple ATL[1] ou le standard QVT (OMG, 2008)). (Pons *et al.,* 2006) argumente également dans le sens de l'utilisation d'OCL en expliquant qu'OCL offre l'avantage d'être plus accepté en pratique

1. Atlas Transformation Language : `http://www.eclipse.org/m2m/atl/`

par les concepteurs que d'autres langages formels, à la syntaxe et à l'utilisation plus complexes ou moins connues.

3. Raffinement de modèles

Le raffinement de modèles consiste à modifier le contenu d'un modèle sans en changer le but ou la sémantique. Il s'agit de rajouter des détails à un modèle ou bien encore de le restructurer pour en améliorer la conception. Un exemple simple de raffinement est l'ajout de méthodes d'accès à un attribut. À un certain niveau d'abstraction, dans un diagramme de classes, on peut se contenter de spécifier la liste des attributs d'une classe. Quand on se rapproche de l'implémentation, on rajoutera pour chaque attribut une méthode de lecture et une d'écriture (un *getter* et un *setter*). Le sens du modèle reste le même, des détails ont juste été ajoutés.

La réalisation d'un raffinement consiste donc à prendre un modèle et à le modifier, ce qui correspond en fait à une transformation de modèle. Par principe, le raffinement implique que les deux modèles, celui d'avant le raffinement (le modèle source de la transformation) et celui d'après (le modèle cible), sont du même type, c'est-à-dire conformes au même méta-modèle. La transformation est donc endogène. De manière plus générale, le raffinement est un cas particulier de transformation endogène. Un raffinement étant une transformation, il peut donc être spécifié par un contrat de transformation.

À titre d'exemple, nous allons étudier dans cet article un raffinement d'ajout d'interfaces sur un diagramme de classes de type UML. Afin de simplifier nos explications et l'écriture du contrat associé, nous n'avons pas utilisé directement un diagramme de classes UML et donc le méta-modèle UML standard car ce dernier est relativement complexe en terme de nombre d'éléments et de navigation entre ces éléments. Nous avons défini un méta-modèle de diagramme de classes simplifié, comme détaillé dans la section suivante.

3.1. *Méta-modèle de diagramme de classes*

La figure 1 représente le méta-modèle de notre diagramme de classes. On y retrouve les mêmes concepts de base que dans un diagramme de classes UML : classe, interface, type, méthode, attribut, association ainsi que quatre types primitifs (Void, Boolean, Integer et String). Les concepts qui ne sont pas présents, par souci de simplification, sont par exemple les visibilités ou bien encore la spécialisation.

L'élément `ModelBase` est particulier et représente, comme son nom l'indique, la base du modèle. Il ne définit pas de concept du domaine mais est une aide à la manipulation du modèle dans certains outils en référençant l'ensemble des principaux éléments d'un modèle. Nous avons implémenté ce méta-modèle via la plate-forme Eclipe/EMF et son méta-méta-modèle Ecore. Dans ce contexte, il est recommandé, par exemple pour la génération automatique d'éditeurs de modèles, d'avoir un tel élé-

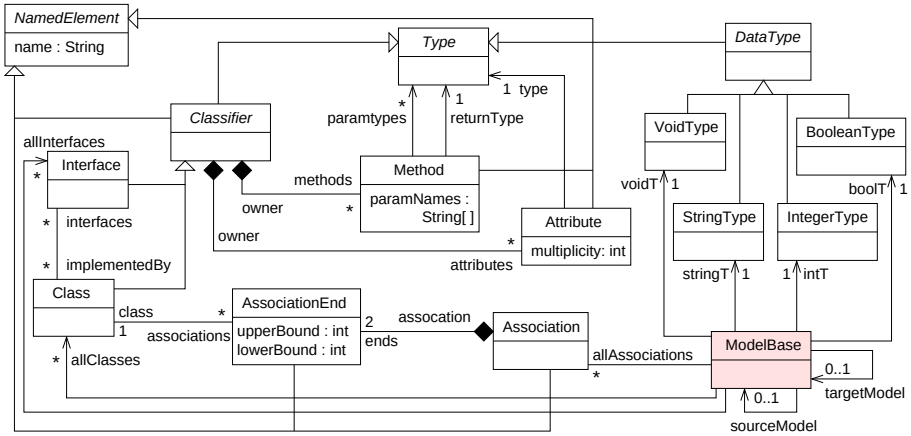

Figure 1. *Méta-modèle du diagramme de classes simplifié*

ment de base. De même, dans d'autres outils, tel que Kermeta[2], où la navigation sur le modèle se fait de manière explicite dans le code de la manipulation du modèle, une telle base de modèle est utile. Cette base référence donc l'ensemble des classes, interfaces et associations du modèle ainsi qu'une instance de chacun des quatre types primitifs. Les références `targetModel` et `sourceModel` serviront à la concaténation de deux modèles (voir la section 4.2).

Afin de compléter la définition du méta-modèle, au diagramme de classe de la figure 1 s'ajoute un ensemble de régles de bonne formation, exprimées sous la forme d'invariants OCL (ces règles ne sont pas présentées ici faute de place).

3.2. *Exemple de raffinement : ajout d'interfaces*

Le principe du raffinement que nous allons utiliser en exemple est de rajouter une ou plusieurs interfaces à chaque classe et de déplacer une partie ou toutes les méthodes implémentéees par cette classe dans ses interfaces. Ce raffinement se fait en deux étapes :

1) Ajout automatique d'une interface par défaut : pour chaque classe du diagramme, on crée (sauf si elle existait déjà) et lui associe une interface nommée `I`*`nomDeLaClasse`*`.` Toutes les méthodes implémentées directement par la classe sont ensuite déplacées dans cette interface. Cette opération est entièrement automatisable et est réalisée par un outil de transformation de modèles.

2. `http://www.kermeta.org/`

2) Réarrangement éventuel des méthodes et interfaces : l'ajout d'une interface par défaut ne convient pas forcément au concepteur. Il peut alors modifier manuellement pour chaque classe la liste des interfaces (renommage, suppression, ajout) ainsi que la localisation des méthodes (déplacement d'une méthode vers une autre interface ou la classe).

Au final, après ces deux étapes, on obtient un modèle cible qui est la transformation du modèle source et correspond à un raffinement d'ajout d'interfaces. La principale contrainte à vérifier pour que le raffinement soit correct est que chaque classe implémente toujours la même liste de méthodes, directement ou via ses interfaces. De manière détaillée, le contrat de transformation associé à ce raffinement est le suivant :

– Contraintes sur le modèle source : aucune, on peut raffiner un diagramme de classes quelconque

– Contraintes sur le modèle cible : chaque classe implémente au moins une interface (même si elle ne contient pas de méthodes)

– Contraintes d'évolution des éléments du modèle source vers le modèle cible : toutes les classes sont conservées[3] et chacune implémente, directement ou indirectement via ses interfaces, la même liste de méthodes qu'avant la transformation

La figure 2 montre un exemple d'un tel raffinement, avec ses deux étapes. On peut y noter que le concepteur a renommé les deux interfaces de la classe `Compte` et a replacé la méthode `getNom` directement dans la classe `Client`. On constate que le raffinement est correct puisque toutes les classes sont conservées, que chaque classe implémente toujours, directement ou indirectement, la même liste de méthodes et qu'elle implémente au moins une interface.

4. Méthode de validation de transformation par contrat

Dans cette section, nous allons détailler notre méthode de définition et de validation de contrats. Cette méthode est la plus générale possible et doit donc s'appliquer sur deux modèles, l'un supposé source et l'autre cible de la transformation ou du raffinement, définis ou obtenus à partir d'une grande variété d'outils.

4.1. *Discussion sur la forme du contrat*

Dans (Cariou *et al.,* 2004b), nous avons défini deux techniques pour écrire le contrat de transformation, et plus précisément les contraintes sur l'évolution des éléments entre le modèle source et le modèle cible. Ces contraintes sont particulières car elles nécessitent de référencer à la fois le modèle source et le modèle cible. Les

3. D'autres éléments du modèle doivent être également conservés sans modification, comme les associations entre les classes. Par souci de simplification, nous laisserons ces éléments de côté dans nos explications.

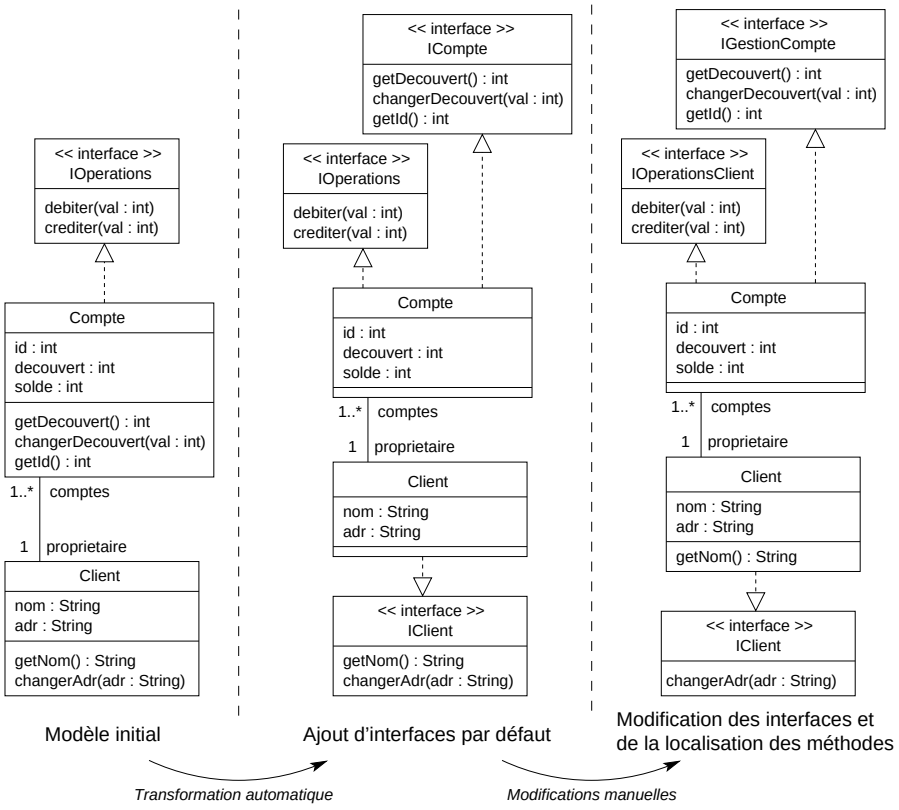

Figure 2. *Exemple de raffinement : ajout et modification d'interfaces*

contraintes OCL s'exprimant dans un contexte – et donc par rapport à un seul méta-modèle – il faut donc pouvoir manipuler les deux modèles à partir de ce seul contexte[4].

Pour réaliser cela, la première technique consiste à attacher l'opération de transformation à un élément du méta-modèle et à la spécifier en OCL : le modèle avant la transformation est le modèle source et une fois la transformation exécutée, le modèle modifié par l'opération est le modèle cible. Dans la post-condition, on peut alors référencer à la fois les éléments du modèle cible et ceux du modèle source via la construction `@pre` d'OCL. Au final, la pré-condition permet d'écrire la partie du contrat relative au modèle source et la post-condition contient les deux autres parties du contrat

4. Cela implique par défaut que le méta-modèle soit le même pour le modèle source et le modèle cible, d'où la restriction à des transformations endogènes. En pratique, les méta-modèles source et cible peuvent être relativement différents si l'on arrive à en déterminer un méta-modèle commun, comme expliqué dans (Cariou *et al.,* 2004a)

(sur le modèle cible et l'évolution des éléments). Pour notre exemple d'ajout d'interfaces, on pourrait avoir une écriture du contrat de la forme suivante :

context ModelBase : : addInterfaces()
pre : - - *contraintes sur le modèle source*
post :
- - *contraintes sur le modèle cible et*
- - *contraintes d'évolution entre le modèle source et le modèle cible*
allClasses -> size() = allClasses@pre -> size() **and** ...

Ici, l'opération de transformation s'appelle `addInterfaces` et est attachée au méta-élément `ModelBase`. La post-condition de l'opération vérifie, entre autres, que le nombre de classes après la transformation est le même qu'avant. On voit que l'on manipule bien les éléments avant la transformation (modèle source) et les éléments après la transformation (modèle cible).

Si cette méthode est conceptuellement élégante car on associe un contrat à l'opération de transformation comme on associe un contrat en OCL à n'importe quelle opération d'une classe (via une pré et une post-condition) ou bien encore un contrat à une méthode d'un programme objet, elle n'est pas adaptée à notre problème, et cela pour deux raisons. La première est que ce contrat n'est vérifiable que lorsqu'on exécute la transformation. Cela implique qu'on utilise un outil qui permet à la fois d'exécuter une transformation et d'en vérifier les contraintes associées. Cela limite fortement les outils utilisables. On peut tout de même citer Kermeta qui permet d'associer des opérations aux méta-éléments et de leur associer une pré et une post-condition[5]. La seconde raison est la conséquence de la première : il faut exécuter une transformation pour valider son contrat, ce qui oblige à une exécution entièrement automatique. Si le concepteur modifie à la main le modèle généré, il n'est donc pas possible de valider le contrat.

4.2. *Concaténation des modèles source et cible*

Lors de la définition du contrat, pour éviter les problèmes que nous venons de décrire, nous allons utiliser la deuxième technique que nous avons définie. Le principe général est de concaténer ou de fusionner le modèle source et le modèle cible dans un modèle plus global. Ensuite, on définit en OCL un ensemble d'invariants s'appliquant sur le modèle global, et donc à la fois sur les éléments du modèle source et du modèle cible. Cette méthode est à adapter en fonction du type de modèle. Par exemple, dans le cadre d'un diagramme de classes UML, on définira un modèle simple contenant deux packages : le premier correspondra au diagramme de classes source et le second au diagramme de classes cible (on peut utiliser cette méthode du double package pour n'importe quel type de diagramme UML).

5. Kermeta utilise un langage dédié pour l'écriture des contraintes. On n'écrirait donc pas le contrat directement en OCL mais dans un langage à la sémantique et l'expressivité similaires.

Dans le cas de définition d'un méta-modèle dédié à un domaine, il est intéressant d'y ajouter directement les éléments nécessaires à la manipulation simultanée des deux modèles. Dans notre méta-modèle de diagramme de classes simplifié, nous avons par exemple rajouté dans la classe `ModelBase` deux références, nommées `sourceModel` et `targetModel` de type `ModelBase`. Ces deux références permettent de charger et référencer deux modèles de type diagramme de classes, à savoir un modèle source et un modèle cible. L'intérêt de rajouter ces références de cette manière permet de rester dans le contexte du même méta-modèle, ce qui simplifie l'écriture du contrat.

À partir de deux modèles, un source et un cible, il s'agit donc d'en créer un troisième. D'un point de vue pratique, pour l'implémentation en Ecore de notre méta-modèle, il suffit de créer un nouveau modèle et de positionner les références `sourceModel` et `targetModel` de la base de ce modèle avec les deux modèles à fusionner via les mécanismes standards d'instantiation et d'édition de modèles de la plateforme EMF. Une autre méthode consiste à passer par un petit programme dédié, par exemple écrit en Kermeta, qui prend en paramètre les noms des deux modèles et sauvegarde leur fusion dans un troisième modèle.

Ce type de programme dédié permet également d'effectuer le cas échéant des traitements supplémentaires. Dans notre exemple, les quatre types primitifs (Void, Boolean, Integer et String) sont gérés comme des singletons : la base du modèle référence une instance de chacun de ces types. Chaque élément du modèle devant utiliser un type primitif va utiliser cette référence unique. Le problème ici est, qu'après concaténation des modèles source et cible, on se retrouve avec trois bases de modèle (la base du modèle global, la base du modèle source et la base du modèle cible) et donc avec les types primitifs chacun présents en trois exemplaires. Pour revenir à une unique instance de chaque type, il faut parcourir le contenu des modèles cible et source et remplacer chaque référence vers un type primitif par la référence sur le type primitif de la base globale. Ainsi, on pourra facilement comparer que deux éléments de modèles différents utilisent le même type primitif via une simple égalité de référence.

4.3. *Écriture du contrat*

Comme nous l'avons vu, le contrat comporte trois parties : les contraintes que doit respecter le modèle source, les contraintes que doit respecter le modèle cible et les contraintes sur l'évolution des éléments du modèle source vers le cible. Les deux premières parties sont simples à exprimer, il s'agit de définir des invariants associés au méta-modèle, comme cela se fait de manière classique. Pour notre exemple d'ajout d'interface, le modèle source est un diagramme de classes quelconque donc il n'y a pas de contraintes particulières à appliquer. Pour le modèle cible, il s'agit d'un diagramme de classes dans lequel chaque classe implémente au moins une interface. Cela s'exprime de la manière suivante :

---

context Class **inv** hasAnInterface :
self.interfaces -> notEmpty()

Il suffit ensuite via un évaluateur OCL de vérifier que cette contrainte est validée par le modèle cible.

La troisième partie concernant les contraintes d'évolution entre les deux modèles est plus complexe à établir, parce qu'elle s'applique sur le modèle global qui concatène le modèle source et le modèle cible au sein d'un seul modèle. Il faut donc, pour commencer, déterminer explicitement la référence sur le modèle source et sur le modèle cible. Ensuite, il faut disposer d'un ensemble de fonctions utilitaires permettant de déterminer que tel ou tel élément du modèle cible a une correspondance dans le modèle source et de récupérer cet élément correspondant.

4.3.1. *Détermination des modèles source et cible dans le modèle global*

Selon la façon dont le modèle global est construit, récupérer les références sur le modèle source et le modèle cible peut être direct ou nécessiter une requête OCL. C'est ce que nous avons besoin de faire dans notre exemple et notre implémentation. En effet, notre modèle global contient trois bases de modèles (trois instances de `ModelBase`) et il faut déterminer laquelle correspond au modèle source et laquelle correspond au modèle cible. Cela peut être réalisé par la définition de variables OCL comme suit :

```
context ModelBase
def: base : ModelBase = ModelBase.allInstances() -> any ( base |
    not(base.sourceModel.oclIsUndefined()) and
    not(base.targetModel.oclIsUndefined()) )

def: sourceModel : ModelBase = base.sourceModel
def: targetModel : ModelBase = base.targetModel
```

L'idée appliquée ici est de récupérer la base globale et via ses références `sourceModel` et `targetModel` de récupérer alors les modèles source et cible. La base globale est en fait la seule qui a ses deux références `sourceModel` et `targetModel` positionnées. On fait donc une requête sur l'ensemble des bases pour récupérer celle qui respecte cette caractéristique.

4.3.2. *Correspondance entre éléments du modèle source et du modèle cible*

Dans notre exemple de raffinement, le but principal du contrat, concernant l'évolution des éléments entre le modèle source et le modèle cible, consiste à vérifier que chaque classe du modèle cible implémente (directement ou via ses interfaces) les mêmes méthodes que sa classe équivalente du modèle source. Pour valider cela, il est donc nécessaire de pouvoir déterminer quelle est sa classe équivalente. De manière plus générale, il est nécessaire de pouvoir déterminer que tel élément dans un modèle possède ou pas un élément du même type qui lui correspond dans l'autre modèle ainsi que de récupérer cet élément correspondant. Pour l'exemple de la figure 2, la classe `Compte` du modèle cible (modèle final) correspond dans le modèle source (modèle initial) à la classe `Compte`. Cette correspondance se vérifie en comparant les noms des classes et en vérifiant, entre autres, que leurs attributs (ici `id`, `decouvert` et

`solde`) sont les mêmes. Il est donc nécessaire de vérifier également la correspondance des attributs.

La correspondance d'un élément d'un modèle à un élément du même type d'un autre modèle peut être de trois natures :

Correspondance totale : un élément correspond à un élément identique de l'autre modèle, dans le sens où tous ses attributs et toutes ses références ont également une correspondance[6] dans l'autre modèle.

Par exemple, dans notre méta-modèle, un méta-élement attribut possède un nom, une référence sur un type et une référence sur son propriétaire. Deux instances d'attribut seront en correspondance totale si elles ont le même nom, si leurs types sont en correspondance et si leurs propriétaires sont également en correspondance.

Correspondance partielle : un élément possède un élément correspondant dans l'autre modèle dans le sens où une sous-partie de ses atttributs et de ses références ont également une correspondance[6] dans l'autre modèle.

Dans notre exemple, une classe est en correspondance partielle avec une classe de l'autre modèle. En effet, si le nom doit être le même et si la liste d'attributs doit être en correspondance pour les deux classes, il ne faut en revanche pas vérifier que les listes des méthodes ou des interfaces implémentées sont en correspondance. Ces dernières concernent les parties modifiées entre le modèle source et le modèle cible et leur validité est contrôlée à part.

Pas de correspondance : il n'est pas nécessaire de vérifier qu'un élément possède une correspondance dans l'autre modèle.

Pour notre exemple, il ne faut pas vérifier qu'une interface est en correspondance avec une interface de l'autre modèle. En effet, pendant le raffinement, la liste des interfaces change (renommage, création, suppression d'interface, modification de la liste des méthodes d'une interface) en fonction des choix du concepteur.

Ces correspondances entre éléments sont implémentées en OCL par une série de fonctions utilitaires (via le constructeur `def`). Selon la complexité du méta-modèle et le nombre de correspondances requises entre les éléments, le nombre de ces fonctions peut être important et au final, fastidieux à écrire à la main. Ce problème est facilement contournable via un outillage approprié en générant automatiquement ces fonctions pour un méta-modèle donné. En effet, ces fonctions sont toujours basées sur le même schéma, en vérifiant la correspondance des attributs et des références d'un élément de manière transitive. Pour une liste de références ou d'attributs, on vérifie la

6. Les correspondances sont donc vérifiées de manière transitive (la correspondance des classes implique la correspondance des attributs de ces classes qui implique à son tour la correspondance des types de chacun des attributs) mais il n'est pas nécessaire de rester dans le même mode de correspondance (totale partout ou partielle partout). Par exemple, la correspondance des classes pourra être totale mais la correspondance de ses attributs sera partielle.

correspondance des éléments de la liste un par un. L'outil pourrait analyser un méta-modèle quelconque et proposer au concepteur de choisir, pour chaque type d'élément de ce méta-modèle, le type de correspondance souhaitée (pour une correspondance partielle, on choisira les attributs et références requis). À partir de là, l'outil générera toutes les fonctions OCL de correspondance qu'il suffira de compléter ou de modifier légèrement au besoin.

Lors de la définition de ces fonctions, il faut tout de même faire attention à un détail important : éviter les cycles de correspondance. Par exemple, un attribut possède un champ `owner` de type `Classifier` référençant le propriétaire de l'attribut. Quand on vérifie la correspondance de deux classes, on vérifie transitivement la correspondance des attributs ; mais si pour un attribut on vérifie la correspondance des champs `owner`, on reboucle dans la vérification des correspondances des classes initiales. Il ne faut donc pas vérifier la correspondance de ce champ. Si l'on veut avoir une correspondance en profondeur, c'est-à-dire s'appliquant transitivement sur un grand nombre de types d'éléments, ce problème devient d'autant plus important et complexe à gérer. On pourrait là encore imaginer que notre outil vérifierait, en fonction des choix de correspondances, qu'un tel cycle n'est pas présent.

4.3.3. *Contraintes sur l'évolution des éléments pour l'exemple d'ajout d'interfaces*

La troisième partie du contrat, concernant l'évolution des éléments entre le modèle source et le modèle cible est exprimée sur la figure 3 par l'invariant des lignes 10 et 11 qui s'applique sur les modèles source et cible.

L'invariant consiste à appeler la fonction `sameClasses` (1) qui, pour deux bases de modèle, commence par vérifier que le nombre de classes est le même (2). Ensuite, pour chacune des classes du premier modèle (3), on récupère sa liste de méthodes en concaténant les méthodes de la classe et de toutes ses interfaces (4). On vérifie via la fonction `hasMappingClass` que la classe courante à une correspondance dans l'autre modèle (5) et si ça n'est pas le cas, le contrat n'est pas validé (9). On récupère via la fonction `getMappedClass` la classe associée (6) ainsi que sa liste complète de méthodes (7). Puis on vérifie que les deux classes ont la même liste de méthodes via l'appel de `sameMethodSet` (8).

Faute de place, nous ne détaillerons pas l'ensemble des fonctions de correspondance utilisées par ce contrat. Celles qui concernent les classes sont néanmoins définies en figure 4. La fonction principale en est `classMapping` qui applique une correspondance partielle entre deux classes : comparaison du nom et de la liste des attributs (via l'appel de `sameAttributes`). Les fonctions de correspondance d'attribut auront une forme similaire.

La fonction `sameMethodSet` (figure 5) compare deux ensembles de méthodes. Cela se fait en vérifiant d'abord que les ensembles ont le même nombre d'éléments. Ensuite, on vérifie, via la fonction de correspondance de méthode `methodMapping`, que chaque élément du premier ensemble se trouve dans le second.

```
1   context ModelBase def: sameClasses(mb : ModelBase) : Boolean =
2       self.allClasses -> size() = mb.allClasses -> size() and
3       self.allClasses -> forAll( c |
4           let myMethods : Set(Method) = c.interfaces -> collect(i | i.methods)
                                              -> union(c.methods) -> flatten() in
5           if c.hasMappingClass(mb)
            then
6               let eqClass : Class = c.getMappedClass(mb) in
7               let eqClassMethods : Set(Method) = eqClass.interfaces -> collect(i |
                        i.methods) -> union(eqClass.methods) -> flatten() in
8               c.sameMethodSet(myMethods, eqClassMethods)
            else
9               false
            endif)

10  context ModelBase inv checkInterfaceContract:
11  targetModel.sameClasses(sourceModel)
```

Figure 3. *Contraintes d'évolution entre les modèles source et cible*

```
context Class def: classMapping(cl : Class) : Boolean =
    self.name = cl.name and
    self.sameAttributes(cl)

context Class def: hasMappingClass(mb : ModelBase) : Boolean =
    mb.allClasses -> exists( cl | self.classMapping(cl))

context Class def: getMappedClass(mb : ModelBase) : Class =
    mb.allClasses -> any ( cl | self.classMapping(cl))
```

Figure 4. *Fonctions de correspondance pour une classe*

```
context Classifier def: sameMethodSet(
                            mets1 : Set(Method), mets2 : Set(Method)) : Boolean =
    mets1 -> size() = mets2 -> size() and
    mets1 -> forAll ( m1 |
        mets2 -> exists ( m2 | m1.methodMapping(m2)) )
```

Figure 5. *Fonction de correspondance d'ensembles de méthodes*

Toutes ces fonctions et invariants formant le contrat sont écrits en OCL standard. Il est donc vérifiable par n'importe quel évaluateur OCL implémentant le standard et pouvant lire des modèles Ecore. D'un point de vue pratique, nous avons opté pour l'outil ATL pour valider un modèle global par rapport au contrat. Bien qu'étant à la base un outil de transformation de modèles, ATL s'est avéré pertinent pour plusieurs raisons. D'abord, il implémente entièrement le standard OCL et permet de manipuler des modèles Ecore ou UML. Ensuite, il est relativement simple de valider des contraintes OCL sur un modèle via une transformation, comme expliqué dans (Bézivin *et al.,* 2005). On commence par définir un méta-modèle cible qui définit simplement un élément d'information détaillant un résultat, avec deux spécialisations selon la validité du résultat : *Correct* et *Error*. On définit ensuite une règle de transformation qui, pour une instance de la base globale, génère dans le modèle cible soit une instance de *Correct* si la base respecte l'invariant du contrat (celui de la figure 3) soit une instance d'*Error* dans le cas contraire. On obtient ainsi, via ce modèle cible généré, le résultat global pour la validation du contrat. De plus, on peut être plus précis dans le détail du résultat en établissant des règles de validation de la même façon pour chacune des classes. Ainsi, on peut connaître facilement la liste des classes qui ne respectent pas leur part du contrat.

4.4. *Résumé de la méthode*

Notre méthode se résume par les étapes suivantes :

1) Définir un mécanisme permettant de concaténer deux modèles au sein d'un troisième modèle (les trois étants conformes au même méta-modèle).

2) Pour une transformation donnée, définir un contrat composé de trois parties, chacune étant un ensemble d'invariants OCL :

- Contraintes que doit respecter le modèle source
- Contraintes que doit respecter le modèle cible
- Contraintes d'évolution entre le modèle source et le modèle cible, définies à partir d'un ensemble de fonctions de correspondance entre types d'éléments

3) Pour un couple de modèles, valider le contrat en vérifiant que les contraintes sur le modèle source (respectivement cible) du contrat sont respectées par le modèle source (respectivement cible) et que la troisième partie du contrat est respectée par le modèle concaténant les modèles source et cible.

L'approche a été expérimentée en mettant notamment en œuvre le raffinement d'ajout d'interfaces et en utilisant un ensemble varié d'outils sans que cela ne pose de problème d'interopérabilité. Nous avons utilisé la plate-forme Eclipse/EMF pour définir le méta-modèle et créer ou modifier des modèles. Les transformations automatiques ont été réalisées en Kermeta et en ATL. ATL a également été utilisé pour vérifier le contrat sur un couple de modèles.

5. Travaux connexes

Notre approche a pour cœur l'utilisation d'OCL car, comme nous l'avons expliqué, c'est un langage relativement bien connu et suffisamment ouvert pour être le meilleur candidat pour la définition de contrats de transformations. Nous allons donc comparer ici notre méthode à d'autres approches se basant également sur OCL soit pour spécifier des raffinements de modèles, soit des contrats de transformations similaires aux notres. Par exemple, (Pons *et al.,* 2006) définit formellement en OCL des raffinements de modèles et (Song *et al.,* 2007) se base sur OCL pour effectuer des raffinements. (Van Grop, 2008) définit des contrats de transformations et du *refactoring* de modèles en OCL. (Baudry *et al.,* 2006, Mottu *et al.,* 2006) propose également d'utiliser des contrats de transformations de modèles en OCL pour spécifier l'oracle d'un test de transformation.

Dans quasiment toutes ces approches, il y a nécessité de définir des correspondances entre éléments du modèle source et éléments du modèle cible, comme pour la notre. Mais ces correspondances sont toujours définies de manière *ad-hoc* pour le contexte considéré, et parfois seulement de manière implicite. Nous avons proposé au contraire une méthode générale pour définir explicitement et en détail les correspondances entre éléments, dans le contexte des transformations endogènes.

L'autre différence est que la plupart de ces approches sont moins générales que la notre du point de vue de l'obtention et de la manipulation des modèles ou sont dédiées à des environnements logiciels particuliers. Par exemple, la spécification de raffinement dans (Pons *et al.,* 2006) se fait en utilisant la relation de dépendance UML stéréotypée par `«refine»` qui oblige à définir explicitement un diagramme UML dans lequel des éléments sont liés par cette dépendance. Mais il n'est pas proposé de méthode pour, à partir de deux modèles obtenus de manière quelconque, définir automatiquement un modèle conforme à cette représentation. (Song *et al.,* 2007) propose de guider le concepteur en exécutant automatiquement des raffinements pendant l'édition d'un modèle quand ce dernier est dans un certain état (cet état est spécifié en OCL). Le raffinement est donc effectué de manière instantanée par l'outil et il n'est pas possible de vérifier a posteriori que deux modèles respectent la spécification du raffinement comme nous le proposons.

6. Conclusion et perspectives

Nous avons présenté une méthode facilement outillable pour spécifier et valider un contrat de transformation de modèles dans le cadre de n'importe quel type de transformation endogène. Nous nous sommes ici plus particulièrement intéressés au cas des raffinements de modèle. L'idée générale est de considérer un couple de modèles, l'un étant le source et l'autre le cible d'une transformation ou d'un raffinement, et de vérifier que ce couple respecte bien le contrat de la transformation.

Cette méthode a été conçue pour être la plus générale possible : on peut prendre en compte des modèles obtenus à partir d'outils et de méta-modèles variés (UML, défini

à partir du MOF ou de Ecore). La transformation de modèle peut ainsi être réalisée automatiquement et/ou via une intervention manuelle d'un concepteur. Le contrat est écrit en OCL pour être là encore le plus indépendant possible des outils.

Nous avons mis en avant la nécessité de détailler les correspondances entre éléments des deux modèles ainsi que de pouvoir concaténer deux modèles au sein d'un modèle plus global afin d'élargir la possibilité en termes d'outils et de techniques pour obtenir les modèles source et cible. Au niveau des correspondances, le prochain travail à réaliser est de développer un outillage permettant de générer automatiquement les fonctions OCL de correspondance afin de grandement simplifier l'écriture du contrat.

Enfin, la suite naturelle de notre approche est de la généraliser à des transformations exogènes, c'est-à-dire pour des méta-modèles source et cible différents. La principale difficulté est de passer outre la limitation d'OCL qui est que les contraintes ne sont exprimables que par rapport à un seul méta-modèle. Pour cela, on peut par exemple imaginer de fusionner les deux méta-modèles en un troisième plus global pour se retrouver avec un contexte unique, comme proposé dans (Baudry *et al.,* 2006).

7. Bibliographie

Baudry B., Dinh-Trong T., Mottu J.-M., Simmonds D., France R., Ghosh S., Fleurey F., Le Traon Y., « Model Transformation Testing Challenges », *ECMDA workshop on Integration of Model Driven Development and Model Driven Testing.*, July, 2006.

Beugnard A., Jézéquel J.-M., Plouzeau N., Watkins D., « Making Components Contract Aware », *IEEE Computer*, vol. 32, n° 7, p. 38-45, 1999.

Bézivin J., Jouault F., « Using ATL for Checking Models », *Intl. Workshop on Graph and Model Transformation (GraMoT 2005)*, vol. 152 of *ENTCS*, p. 69 - 81, 2005.

Cariou E., Marvie R., Seinturier L., Duchien L., Model Tranformation Contracts and their Definition in UML and OCL, Technical Report n° 2004-08, LIFL, April, 2004a.

Cariou E., Marvie R., Seinturier L., Duchien L., « OCL for the Specification of Model Transformation Contracts », Workshop OCL and Model Driven Engineering of UML'04, 2004b.

Meyer B., « Applying "Design by Contract" », *IEEE Computer (Special Issue on Inheritance & Classification)*, vol. 25, n° 10, p. 40-52, 1992.

Mottu J.-M., Baudry B., Le Traon Y., « Reusable MDA Components : A Testing-for-Trust Approach », *MoDELS 2006*, vol. 4199 of *LNCS*, Springer Verlag, 2006.

OMG, « Object Constraint Language (OCL) Specification, version 2.0 », 2006. `http ://www.omg.org/spec/OCL/2.0/`.

OMG, « Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, version 1.0 », 2008. `http ://www.omg.org/spec/QVT/1.0/`.

Pons C., Garcia D., « An OCL-Based Technique for Specifying and Verifying Refinement-Oriented Transformations in MDE », *MoDELS 2006*, LNCS, Springer Verlag, 2006.

Song H., Sun Y., Zhou L., Huang G., « Towards Instant Automatic Model Refinement Based on OCL », *APSEC '07*, IEEE Computer Society, p. 167-174, 2007.

Van Grop P., Model-Driven Development of Model Transformations, PhD thesis, University of Antwerp, Dept. of Mathematics and Computer Science, 2008.