Adaptation des contraintes à l'évolution de leur méta-modèle

Soraya Mesli

Stage de master recherche

Rapport de stage

Adaptation des contraintes à l'évolution de leur méta-modèle

Auteur:
Soraya MESLI

Encadrant:
Salah SADOU
ArchWare

UBS

Abstract

Les méta-modèles évoluent au fil du temps pour différentes raisons. Dans le monde MOF, des contraintes OCL sont ajoutées aux méta-modèles. L'évolution des méta-modèles peut avoir un impact sur les différents artefacts (modèles, modèles de transformation et contraintes OCL) qui lui sont liés. Une coévolution de ces artefacts est nécessaire pour garder leur conformité au méta-modèle et leur cohérence. Dans ce document, nous proposons une approche qui permet d'adapter les contraintes OCL liées à un méta-modèle lors d'une évolution progressive de ce dernier. Pour cela, une bibliothèque d'opérations d'évolution a été définie. Les opérations d"évolution et les adaptations OCL proposées sont décrites en QVT-R.

.

Mots clés : évolution méta-modèle, coévolution, adaptation OCL, QVT-R

Contents

1 Introduction

L'ingénierie dirigée par les modèles (IDM), est une approche de développement qui se concentre sur des préoccupations plus abstraites que la programmation classique pour maîtriser la complexité des systèmes qui ne cesse de croître. Il s'agit d'une forme d'ingénierie générative dans laquelle tout ou partie d'une application est engendrée à partir de modèles. Dans L'IDM les modèles sont au cœur du processus de conception.
Dans l'IDM tous les artefacts manipulés sont des modèles mais on les distingue selon 3 grand rôle : le modèle, le méta-modèle et le modèle de transformation. Tout modèle quel qu'il soit doit toujours être conforme à son méta-modèle.
Par ailleurs, les langages de méta-modélisation souvent graphiques (comme MOF) ne permettent pas d'exprimer certaines parties de la syntaxe du langage, le langage (méta-modèle) voulu. En conséquence, des contraintes sont associées aux méta-modèles afin d'affiner la syntaxe. Dans le monde MOF (*Meta Object Facility*), ces contraintes sont exprimées grâce au langage de contraintes OCL (*Object Constraint Language*).
Un méta-modèle, comme tout système logiciel, est amené à évoluer au fil du temps pour différentes raisons : satisfaire de nouveaux besoins, correction de problèmes, etc. Mais la simple modification d'un méta-modèle peut avoir des conséquences directes sur les artefacts (modèles, modèles de transformation, contraintes OCL) qui lui sont liés. La figure 1 synthétise les différents problèmes pouvant surgir lors de l'évolution d'un méta-modèle.

1. Conformité des modèles : lors de l'évolution d'un méta-modèle MMa vers un méta-modèle MMa', le modèle Ma qui est conforme au méta-modèle MMa, doit lui aussi évoluer pour devenir conforme au méta-modèle MMa'.

2. Consistance des contraintes attachées aux méta-modèles : les contraintes OCL attachées au méta-modèle MMa peuvent devenir incohérentes lors de l'évolution de ce dernier. La plupart du temps, ces contraintes sont laissées telles quelles, supprimées ou réécrites à la main. Cela fait perdre du temps aux concepteurs et peut générer des erreurs. Il est nécessaire de faire coévoluer les contraintes attachées au méta-modèle MMa pour les rendre cohérentes au méta-modèle MMa'.

3. Consistance des contraintes attachées aux modèles : l'évolution du méta-modèle MMa vers MMa', implique une évolution des modèles conformes à MMa pour les rendre conforme à MMa'. Mais dans le cas où les modèles eux même contiennent des contraintes. Ces dernières doivent elles aussi évoluer.

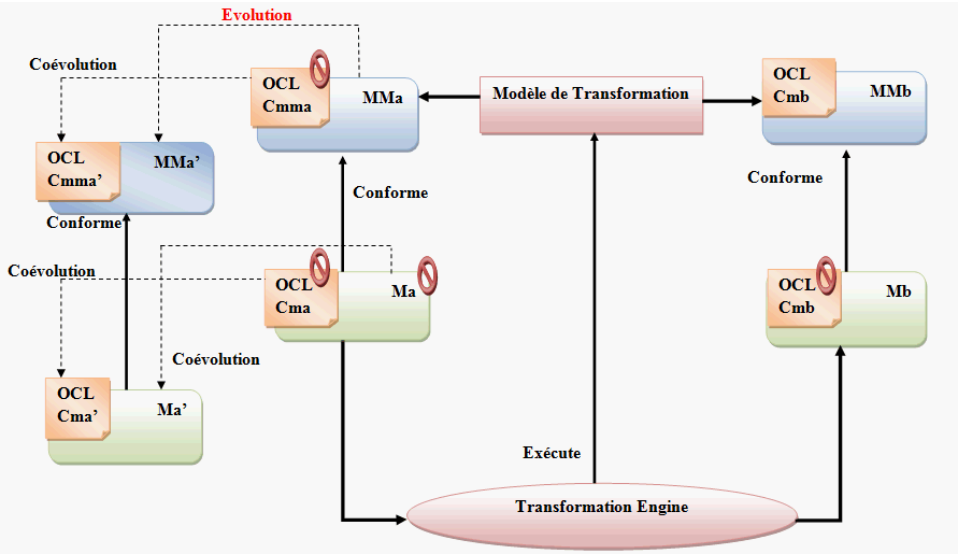

Figure 1: Différents cas de coévolutions possibles dans un processus IDM [7]

4. Consistance des contraintes lors d'une transformation de modèles : comme le montre la figure 1, lors d'une transformation d'un modèle Ma vers un modèle Mb, seuls les éléments du langage sont concernés. Ainsi, le modèle Mb respectera, certes, les contraintes édictées dans le modèle Ma, juste après la transformation. Mais, si le modèle Mb doit subir des évolutions, nous n'avons aucun moyen pour vérifier qu'il respecte les contraintes initiales (Cma). Donc les contraintes (Cma) doivent être aussi transformées et rattachées au modèle Mb.

La coévolution consiste à faire évoluer un artefact (modèle, transformation de modèle, contrainte OCL) automatiquement suite à l'évolution du méta-modèle auquel il est lié. Il est important de noter, que dans le cas général, une coévolution entièrement automatique est impossible, en raison de l'existence de changements non solubles automatiquement [12]. Les changements nécessitent donc parfois l'intervention du développeur du méta-modèle.

Mon stage vient en continuité des travaux de thèse de Hassam [7] sur la coévolution contraintes OCL / méta-modèle. Elle a proposé une bibliothèque d'opérations d'évolution (opérations qui permettent de faire évoluer progressivement un méta-modèle), ainsi que les adaptations OCL associées à chacune de ces opérations. Mon stage comportent trois parties essentielles :

- Correction et extension des opérations : cette partie consistait à vérifier et corriger la formalisation QVT-R de toutes les opérations d'évolution

proposées par [7], ainsi que la formalisation QVT-R des adaptations OCL. Puis je devais vérifier la complétude des opérations et proposer dans la cas échéant des extensions.

- Amélioration de l'outillage : toutes les opérations d'évolution et les adaptations OCL proposées en QVT-R sous forme graphique devaient être réécrites sous forme textuelle en utilisant l'outil *Borland Together architect*.

- Validation de l'approche : l'approche devait être validée sur le méta-modèle d'UML et ses évolutions de la version 2.2 à la version 2.3 et de la version 2.3 à la version 2.4.

Ce rapport de stage est organisé comme suit : la section 2 présente l'état de l'art des travaux sur le problème de la coévolution modèle/méta-modèle, la coévolution modèles de transformation/méta-modèle et la coévolution contrainte OCL/méta-modèle. La section 3 présente l'approche proposée pour résoudre le problème de la coévolution contraintes OCL. La section 4 présente les outils développés pour supporter l'approche supposée. La section 5 présente les expérimentations conduites pour valider notre approche. Enfin, le rapport se termine par une conclusion mettant en évidence les problèmes restant à résoudre.

2 Etat de l'art

Plusieurs travaux tentent de résoudre les différents problèmes liés à la coévolution. Même si le sujet de mon stage concerne seulement l'étude de la coévolution des contraintes OCL avec leur méta-modèle, il est intéressant de regarder le cas de coévolution des autres artefacts. En effet, certaines solutions peuvent être communes à tous les artefacts et ainsi réutilisées.

La sous section qui suit présente un aperçu des différentes approches utilisées (approche manuelle, utilisation d'une bibliothèque d'opérateurs, et approche basée sur le matching des méta-modèles) pour résoudre le problème de la coévolution modèle/méta-modèle. La sous section 2 donne une vue sur les travaux liés aux problèmes de la coévolution modèle de transformation/méta-modèle. La sous section 3, qui est liée directement à mon sujet de stage, concerne les travaux sur la coévolution contrainte OCL/méta-modèles.

2.1 Coévolution modèle / méta-modèle

La coévolution modèle / méta-modèle peut être implémentée par un langage de programmation (comme java), ou bien, avec un langage de transformation de modèles (ATL, QVT) [13].Nous allons présenter dans les sous sections

qui suivent ces différentes approches. Les auteurs de [14] classifient la coévolution modèle/ méta-modèle en trois approches : i) coévolution manuelle ; ii) coévolution basée sur une bibliothèque d'operateur ; iii) coévolution basée sur le matching des modèles.

2.1.1 Approche manuelle

Dans cette approche, les stratégies de coévolution sont décrites à la main par les développeurs. Ce qui nécessite généralement plus d'efforts. Afin de réduire les efforts, les développeurs utilisent des langages de transformation exogènes pour générer automatiquement les éléments du modèle qui n'ont pas été impactés par l'évolution de leur méta-modèle. Une transformation de modèle est dites exogène lorsque le méta-modèle de sortie est différents du méta-modèle d'entrée. Les autres éléments impactés par l'évolution sont coévolués manuellement.

Rose et al [12] se sont inspirés des langages de transformation de modèles (ATL et QVT) pour proposer un langage hybride nommé Epsilon Flock. Contrairement à ATL, ce langage permet de copier automatiquement les éléments du modèle qui sont toujours conformes au méta-modèle évolué. Ces éléments sont copiés à l'aide d'un algorithme nommé « Conservative copy ». Pour chaque élément du modèle initial, l'algorithme vérifie sa conformité au méta-modèle évolué. Si l'élément est toujours conforme (l'élément n'est pas impacté par l'évolution), alors l'algorithme procède à la copie automatique de cet élément dans le modèle coévolué. Des règles de transformation Flock sont écrites manuellement pour faire coévoluer les éléments impactés par l'évolution du méta-modèle.

Dans [11] les auteurs proposent un langage visuel nommé MCL (Model Change Language) pour la description de l'évolution d'un méta-modèle. Les éléments non impactés par l'évolution sont générés automatiquement. Les auteurs proposent un langage graphique proche d'UML qui permet de décrire manuellement les éléments impactés. Le Diagramme MCL a été inspiré du langage de transformation QVT Relation pour modéliser les opérations d'évolutions. Il possède un pattern gauche et un pattern droit qui sont liés par un lien dirigé avec le nom de l'opération d'évolution à appliquer.
Comme toutes les approches manuelles, les transformations décrites ne sont pas génériques, elles sont donc difficilement réutilisables. Comme il n'existe pas un support qui permet de détecter que tous les éléments du modèle ont été coévolués, des coévolutions peuvent être omises.

Refactoring	Construction	Destruction
Rename element	Introduce class	Eliminate class
Move property	Introduce property	Eliminate property
Extract class	Generalize property	Restrict property
Inline class	Pull up property	Push property
Association to class	Extract superclass	Flatten hierarchy
Class to association		

Table 1: Operations d'évolution de base selon [15].

2.1.2 Utilisation d'une bibliothèque d'opérateurs

Cette approche est basée sur la définition d'une bibliothèque d'opérateurs dits "de base". Ces opérateurs permettent à la fois de spécifier l'évolution des méta-modèles et la coévolution des modèles qui leur sont conformes.

Dans [15], l'auteur propose une approche transformationnelle pour faire coévoluer les modèles, lorsque l'évolution est réalisée par adaptation progressive. Une adaptation progressive consiste à appliquer une suite d'opérations "de base" pour atteindre l'évolution souhaitée. Les opérations d'évolution et de coévolution sont décrites en utilisant le langage de transformation *QVT Relation*. L'auteur a aussi défini les relations qui peuvent exister entre deux méta-modèles. Ces relations lui ont permis de définir des propriétés concernant la préservation de la sémantique de chaque opération. Ainsi, l'auteur a établi un classement de ces opérations selon leur impact sur un méta-modèle (voir tabeau 1). Ces impacts sont de trois types :

- Opérations qui préservent la sémantique*(refactoring)*.
- Opérations qui augmentent la taille du méta-modèle*(construction)*.
- Opérations qui réduisent la taille du méta-modèle*(destruction)*.

Les auteurs de [6]ont proposé un outil nommé COPE basé sur le Framework de modélisation *Eclipse Modeling Framework*(EMF). Cet outil permet d'adapter l'évolution des modèles à l'évolution progressive de leur méta-modèle. Il offre une bibliothèque d'opérations (similaires a celles proposées dans [15]) de coévolution dites *Reusable Coupled Transaction*. Chaque opération spécifie en même temps l'évolution du méta-modèle et la stratégie de coévolution qui lui correspond au nouveau méta-modèle. Ces opérations sont écrites en langage *Groovy*. Quand une opération d'évolution est appliquée sur le méta-modèle, une instance de cette opération est sauvegardée dans un historique. Cet historique est affiché à l'utilisateur sous forme d'une liste d'opérations, via l'interface graphique de l'outil COPE. Ainsi, l'utilisateur peut effectuer des modifications sur ces opérations. COPE offre aussi aux

utilisateurs la possibilité de définir de nouvelles opérations d'évolution si elles ne sont pas fournies par la bibliothèque.

2.1.3 Coévolution basée sur le matching des méta-modèles

Cette coévolution est basée sur la spécification des discordances (delta) lors de l'évolution d'un méta-modèle. Le delta construit permettra par la suite d'identifier les éléments du méta-modèle impactés par le changement. Deux méthodes existent pour le calcule des deltas :i) comparaison directe entre le méta-modèle initial et le méta-modèle final ; ii) garder la suite d'opérations qui fait passer un méta-modèle d'une version initiale à une version finale. Une fois les deltas calculés, des transformations de type HOT (*Higher Order Transformation*) sont alors appliquées sur ces deltas, pour générer des modèles de transformations de coévolution. Cicchetti et all [1] ont élaboré une classification des changements qui peuvent intervenir sur un méta-modèle selon leur impact sur les modèles. Ainsi, ces changements peuvent être :

- *sans effet de bord* : après une modification de ce type sur un méta-modèle, tous les modèles demeurent conformes à leur méta-modèle.
- *avec effet de bord et soluble*: ce type de modification entraîne la non-conformité de certains modèles par rapport à leur méta-modèle, mais cela peut être réglée à travers des approches automatisées.
- *avec effet de bord et non soluble* : dans cette catégorie, les changements opérés entraînent la non-conformité de certains modèles. L'intervention des concepteurs est nécessaire pour les résoudre.

Les auteurs de [1] ont utilisé le *matching* de méta-modèles pour faire coévoluer les modèles avec leur méta-modèle. Pour cela ils calculent un modèle de différences à partir du méta-modèle initial et du méta-modèle final. Ce modèle de différences (Δ) est conforme à un méta-modèle de différences qu'ils proposent. Une fois le modèle de différences calculé, ils utilisent deux modèles de transformation TR et T$\neg R$ pour dériver (ΔR) et ($\Delta \neg R$) qui représentent respectivement, les changements *avec effet de bord et solubles* et les changements *avec effet de bord et non solubles*. Sur ces (ΔR) et ($\Delta \neg R$), ils appliquent des transformations HOT pour générer des modèles de transformation de coévolution (voir figure 2).

Dans [3], les auteurs proposent une approche basée sur le *matching* qui comprend trois étapes (voir figure 3). La première étape consiste à l'utilisation d'une bibliothèque pour le calcul des différences entre deux méta-modèles. Ces différences sont représentées par un modèle de *matching* (modèle de différences). La deuxième étape transforme ce modèle de *matching* en une transformation d'adaptation en utilisant des transformations de modèle de type HOT. Enfin, la troisième étape exécute les transformations obtenues

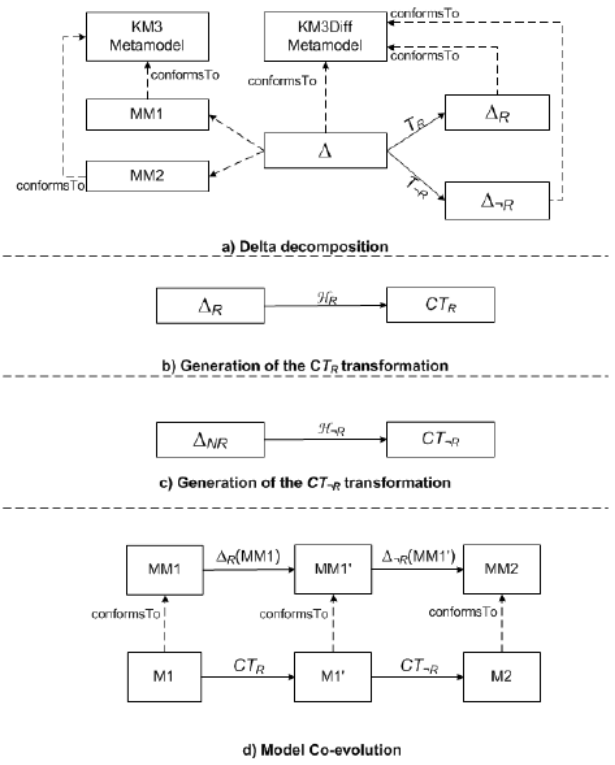

Figure 2: Approche du matching definie par [1].

pour générer le modèle coévolué.

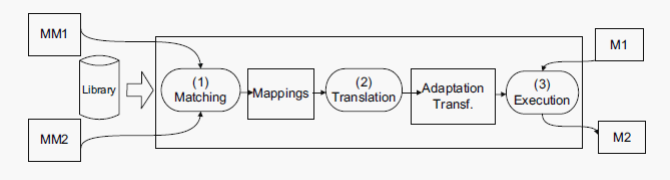

Figure 3: Approche du *matching* définie par [3].

2.2 Coévolution modèle de transformation / méta-modèle

Une transformation de modèles prend en entrée un ou plusieurs modèles conformes à un méta-modèle source et génère un modèle conforme à un méta-modèle cible. Si l'un des méta-modèles évolue, voire les deux, la transformation peut devenir incohérente. Il devient nécessaire alors de faire évoluer le

modèle de transformation pour qu'il puisse s'exécuter correctement [10].

Dans [2], les auteurs proposent une approche comportant deux étapes:

- Étape de détection : elle consiste à détecter des changements entre la version initiale et la version évoluée du méta-modèle. Les changements peuvent être "simples" ou "complexes". Les changements simples sont calculés à l'aide d'un modèle de différences. Des formules de la logique des prédicats sont appliquées sur les changements "simples" pour déterminer ceux qui sont "complexes" , puis représenter ces changements "complexes" sous forme d'un modèle de différences.

- Étape de coévolution : cette étape prend en entrée la version initiale et la version évoluée du méta-modèle, ainsi que les changements calculés lors de l'étape précédente. Les changements sont classées dans trois catégories (*sans effet de bord, avec effet de bord et soluble, avec effet de bord et non soluble*) selon leur impact sur la transformation. Ils ont utilisé des transformations de type HOT pour faire évoluer les modèles de transformation.

Les auteurs de [8] proposent une approche qui permet de faire évoluer les modèles de transformations. La première étape consiste à calculer un modèle de différences entre le méta-modèle source et sa version évoluée, puis un modèle de différences entre le méta-modèle cible est sa version évoluée. Ces modèles de différences sont conforment au méta-modèle MCL (voir sous-section 2.1.1). Enfin, les auteurs utilisent des algorithmes pour faire coévoluer le modèle de transformation.

2.3 Coévolution contrainte /(méta-) modèles

En IDM, les méta-modèles sont souvent décrits en utilisant le langage MOF. Des contraintes OCL sont le plus souvent ajoutées pour affiner la syntaxe visée par un méta-modèle. Cependant durant l'évolution d'un méta-modèles ces contraintes OCL sont, soit omises, soit réécrites manuellement. Ceci est couteux en temps et source d'erreurs [4].

Dans [4], les auteurs décrivent une approche pour aider les concepteurs à adapter les contraintes OCL attachées à un méta-modèle, lors de son évolution progressive. L'évolution d'un méta-modèle est vue comme une séquence d'opérations élémentaires. Après chaque opération élémentaire, le système vérifie l'état de toutes les contraintes pour identifier celles qui sont impactées. Pour chaque contrainte impactée, le système propose une adaptation, quand celle-ci est possible, sinon souligne l'incohérence de la contrainte.

L'approche proposée assure une cohérence syntaxique des contraintes adaptées automatiquement. Le concepteur doit ensuite s'assurer qu'elles correspondent à son intention pour une validation sémantique. Pour cela, une approche collaborative a été mise en place. Le système propose des adaptations automatiques mais c'est au concepteur de les valider, de les modifier ou de les annuler.
Les auteurs s'appuient travaux de [9] et [15]et font une synthèse des opérations de bases à prendre en compte (voir section 3).

Markovic et al [9] proposent une approche pour adapter les contraintes OCL lors d'un *refactoring* de diagramme de classe UML. Les auteurs définissent le *refactoring* d'un modèle UML comme une suite d'opérations élémentaires de refactoring appliquées par adaptation progressive sur le modèle. Ces opérations de refactoring, ainsi que les transformations des contraintes correspondantes sont décrites à l'aide du langage *QVT Realtion*. Les auteurs proposent pour les opérations élémentaires de *refactoring* (un sous ensemble des opérations présentées dans le tableau 1) une formalisation à travers des règles QVT. L'approche est supportée par un plugin *Eclipse* nommé *Roclet*.

Les auteurs de [5] se sont basé sur les travaux de Markovic et al [9] pour proposer une nouvelle approche pour l'adaptation des contraintes OCL lors d'un *refactoring* de modèles UML. Dans ce travail, un *refactoring* est considéré comme une transformation de modèles endogène. L'étape la plus importante de leur approche est la représentation du modèle de transformation. Ce dernier est représenté par un ensemble ordonné d'opérations élémentaires de *refactoring*. Les opérations de *refactoring* prises en compte sont celles décrites dans [9].
Pour obtenir la séquence d'opérations élémentaire de *refactoring*, ils ont défini trois étapes principales :

- Annotation des modèles : cette étape annote le modèle source en ajoutant des tags à chacun de ses éléments afin de les tracer. Ensuite ils exécutent la transformation du modèle source annoté pour obtenir un modèle cible annoté.

- Construction de la table de mapping : à partir des modèles source et ciblé annotés de la première étape, ils construisent une table de mapping qui permet pour chaque élément du modèle source d'identifier le(s) élément(s) correspondant(s), cette table doit être soumise au concepteur pour gérer les cas de renommage et de la suppression.

- Extraction des opérations de *refactoring* : après l'approbation de la table de mapping par le concepteur, l'ensemble des opérations de *refactoring* est calculé. Ces opérations vont permettre de transformer les invariants de contraintes OCL attachées aux diagrammes de classe

UML.cet ensemble d'Operations est généré dans un ordre indéterminé, pour cela des règles d'ordonnancement ont été défini pour ordonner cet ensemble.

Dans cette section, j'ai traité en même temps le cas des contraintes associées à un modèle et des contraintes associées à un méta-modèle. En effet, un méta-modèle n'est rien qu'un modèle du MOF et donc les problèmes sont similaires. La seule différence réside dans les opérations de base concernant leur évolution.

3 Approche

3.1 Le travail de thèse de Hassam[7]

Mon stage vient en continuité des travaux de thèse de Hassam [7]. son approche consiste à faire coévoluer les contraintes OCL attachées au méta-modèle lors d'une évolution progressive de ce dernier. Elle a proposé une bibliothèque d'opérations d'évolution, ainsi que les adaptations OCL associées à chacune de ces opérations. Les opérations et les adaptations sont décrites en QVT-R. Afin de faire évoluer le méta-modèle, le concepteur applique une succession d'opérations de base. L'application d'une opération de base permet à la fois de faire évoluer le modèle et aussi de proposer des adaptations automatiques des contraintes OCL qui lui sont associées. Le système assure une adaptation syntaxique des contraintes, mais au niveau sémantique, une collaboration avec le concepteur est nécessaire pour s'assurer que les adaptations correspondent à son intention.

Une contrainte est dite "Valide", si elle est correcte syntaxiquement et qu'elle correspond à l'intention du concepteur. Les contraintes sont gérées à l'aide d'une table d'états où l'état des contraintes est sauvegardé. Avant l'évolution, toutes les contraintes sont présumées "Valides". Après chaque adaptation les contraintes initiales et les contraintes adaptées sont sauvegardées dans le tableau avec un état "Non-valide", et c'est au concepteur de les valider, les modifier ou les supprimer (voir figure 4).

La sous section qui suit présente la première partie de mon stage et qui consiste à corriger les formalisation QVT-R des différentes opérations d'évolution et aussi les adaptations proposées par [7]. Puis, vérifier et compléter les opérations et les adaptations OCL proposées.

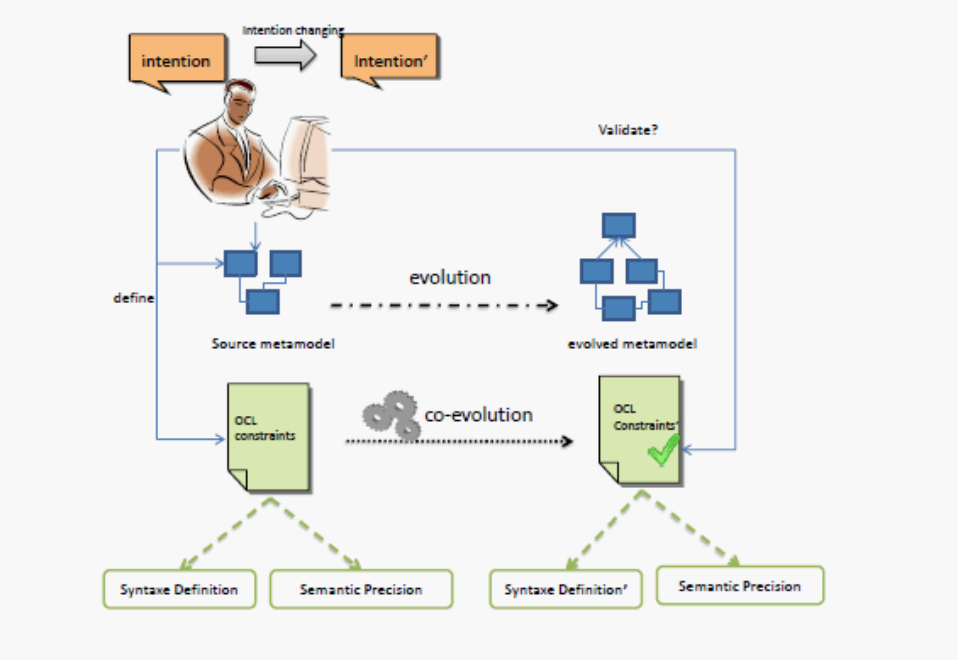

Figure 4: Approche proposée par [7]

3.2 Adaptation des contraintes OCL

Les opérations d'évolution de base et les adaptations des contraintes OCL qui leur sont associées ont été décrites en utilisant le langage de transformation QVT Relation. Ces opérations de bases ont été classifiées selon leur impact sur les contraintes OCL [7] en trois catégories :i) opérations sans influence sur les contraintes OCL; ii) opérations avec influence sur les contraintes OCL; iii) opérations nécessitant la suppression des contraintes OCL.

Les différentes opérations sont décrites en détail dans les sous sections suivantes. Chaque opération est présentée de la manière suivante : un exemple d'application est donné pour expliquer l'opération en question, suivi d'une partie discussion, où toutes les formalisations en QVT de l'opération au niveau MOF et les formalisations QVT des adaptations OCL (dans le cas des opérations qui altèrent et qui nécessitent des adaptations des contraintes OCL) proposées par [7] sont discutées en mettant en évidence les erreurs et les manques recensés sur ces formalisations et adaptations. Les solutions que je propose pour corriger, compléter et étendre l'opération sont décrites dans la partie proposition.

3.2.1 Opérations sans influence sur les contraintes OCL

Ce sont toutes les opérations qui ne nécessitent pas une adaptation des contraintes OCL.

RenameElement

- **Définition et exemple :** cette opération consiste à renommer une méta-classe, un méta-attribut, une méta-opération, ou bien une extrémité d'association. L'opération *RenameElement* est sans influence sur les contraintes OCL. Car renommer un élément nécessite le changement du nom de cet élément dans la contrainte, sans changer la structure de la contrainte. Chaque changement de nom effectué dans le méta-modèle, est automatiquement propagé à toutes les expressions OCL qui utilisent l'élément. La figure 5 présente un exemple d'application de l'opération *RenameMetaAttribute*.

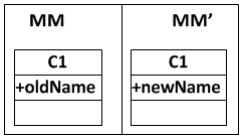

Figure 5: Opération *RenameElement*

La figure 6 décrit la règle QVT qui permet de renommer une propriété (méta-attribut, extrémité d'association). Le pattern gauche (LHS) est doté d'une clause *when* qui permet de décrire les conditions à vérifier avant l'application du pattern droit (RHS). Dans ce cas, la condition doit s'assurer d'une part que le nouveau nom de la propriété est différent de l'ancien nom, et d'autre part que la méta-classe C1 ainsi que ses sous-méta-classes et ses super-méta-classes ne contiennent pas un méta-attribut, une méta-opération ou bien une extrémité d'association qui porte le nom *newName*. Pour vérifier la deuxième partie de la condition, l'opération *allConflictingName()* a été utilisée. Une fois la condition vérifiée, le pattern droit sera exécute et ainsi la propriété sera renommée.

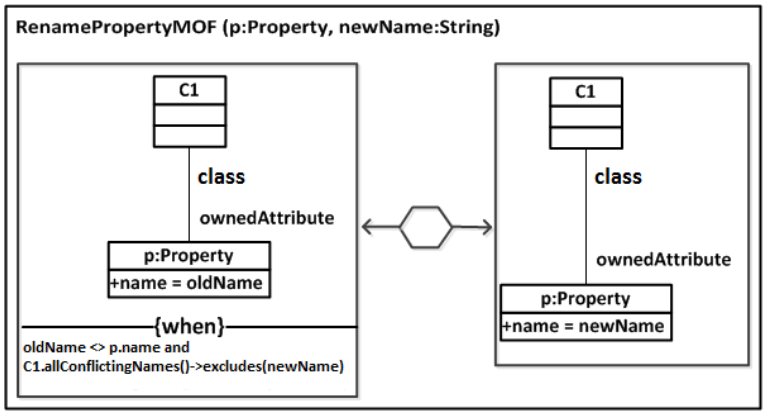

Figure 6: Description en QVT-R de l'opération *RenameElement* au niveau MOF

- **Discussion :** l'opération *allConflictingNames()* était à l'origine proposée par [9] pour le méta-modèle UML 1.5. Elle est liée a une méta-classe et elle retourne tous les noms des attributs, des extrémités d'association de la méta-classe, des sous-méta-classes et les super-méta-classes. Plusieurs opérations déjà définis en UML1.5 (telles que *allChildren(),attribute()* ,... etc.) n'existent plus dans les versions actuelles de UML. Ainsi, les opérations *allConflictingNames()* et *allUsedNames()* ne sont plus valides.

- **Proposition :**

  - je propose d'actualiser ces deux opérations au méta-modèle actuel du MOF comme suit :

    context *Classifier* ***def:***
    allUsedNames()*:Bag(String)= self.allParents()->including(self)*
    ->iterate(c; acc:Bag(String)=Bag| acc->union(c.name)
    ->union(c.attributes.name) ->union(c.ownedElement.name))

    context *Classifier* ***def:***
    allConflictingNames()*:Bag(String)= self.allUsedNames()*
    -> union(self.inheritedMember()->including(self)
    ->iterate(c; acc:Bag(String)=Bag|acc->union(c.opposite().name)
    -> union(c.attributes.name)->union(c.ownedElement.name)))

  - La condition dans la clause *when* de la règle QVT proposée par [7] est de la forme : *newName <> oldName*

and C1.ownedAttribute.allConflinctingNames()->excludes(newName).
Le nom *oldName* est le nom que porte la propriété *p* avant l'application de l'opération *RenameElement*. Je suggère de le remplacer par *p.name* car le paramètre *oldName* est indéfini puisque cette règle QVT prend en paramètres une propriété *p* et un String *newName*.
La condition *C1.ownedAttribute.allConflictingNames()->excludes(newName)* doit être transformer en *C1.allConflictingNames()->excludes(newName).*
Car l'opération *allConflictingNames()* est définie dans le contexte de Classifier et s'applique sur des méta-classes, elle ne peut pas être utilisée sur une Property (ownedAttribute référence des Property).

PullUpMetaAttribute/MetaOperation

- **Définition et exemple :** cette opération consiste à faire monter un méta-attribut, ou bien une méta-opération appartenant à plusieurs sous-méta-classes, vers la super-méta-classe commune à ces dernières (voir figure 7). Elle permet d'éliminer la duplication des propriétés (méta-attribut, méta-opération). Les sous-méta-classes peuvent toujours utiliser ces propriétés à travers la relation d'héritage.

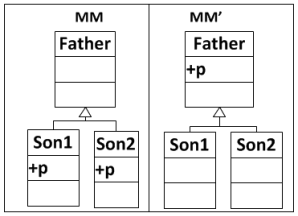

Figure 7: Opération *PullUpMetaAttribute/MetaOperation*

La figure 8 schématise la règle QVT qui permet d'appliquer l'opération *PullUpMetaAttribute*. La condition impose que la super-méta-classe (*Father*) ne contient pas déjà une propriété qui porte le nom de la propriété déplacée.

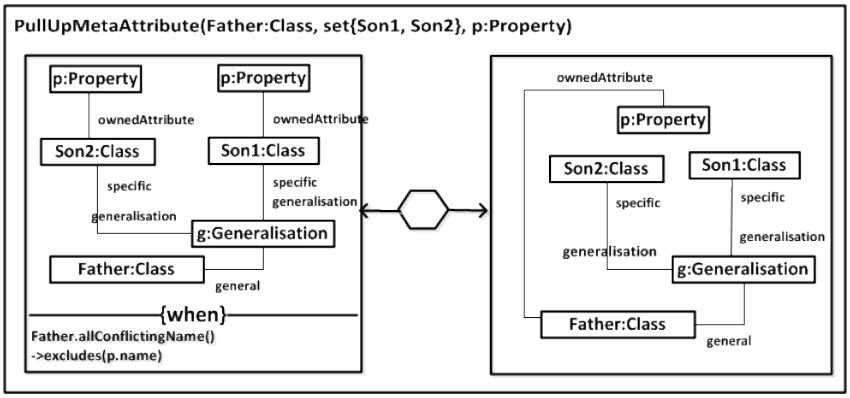

Figure 8: Description en QVT-R de l'opération *PullUpMetaAttributeM/ eta-Operation* au niveau MOF

- **Discussion :** Les contraintes OCL qui sont écrites dans le contexte des sous-méta-classes et qui portent sur la propriété déplacée peuvent être impactées par cette opération d'évolution. Deux cas se présentent lors de l'application de cette opération.
  - Cas 1 : le but du concepteur est de déplacer les propriétés sans déplacer les contraintes. Les contraintes figurent dans le contexte des sous-méta-classes, et elles restent toujours valides. Cette évolution n'a pas d'influence sur les contraintes.
  - Cas 2 : le but du concepteur est de déplacer la propriété et les contraintes OCL vers la super-méta-classe afin que les contraintes OCL puissent être appliquées sur toutes les sous-méta-classes. Dans ce cas, les contraintes doivent être dans le contexte de la super-méta-classe.
- **Proposition :** actuellement cette opération proposée par [7] traite juste le 1er cas. Je propose d'ajouter une nouvelle opération d'évolution nommée *FullPullUpProperty*. Cette nouvelle opération traite le 2ème cas. Elle influence les contraintes. Elle sera traitée dans la catégorie des opérations influençant les contraintes OCL.

ExtractMetaClass

- **Définition et exemple :**L'opération *ExtractMetaClass* consiste à créer une nouvelle méta-classe et de la relier à une méta-classe déjà existante par une association. Les deux extrémités de l'association doivent être de cardinalité (1-1). Dans le cas du *refactoring*, cette opération est utilisée en conjonction avec l'opération *MoveProperty*(détaillée dans les sections suivantes) pour diminuer le nombre des propriétés d'une

méta-classe (figure 9).

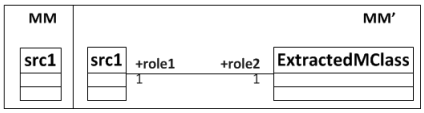

Figure 9: Opération *ExtractMetaClass*

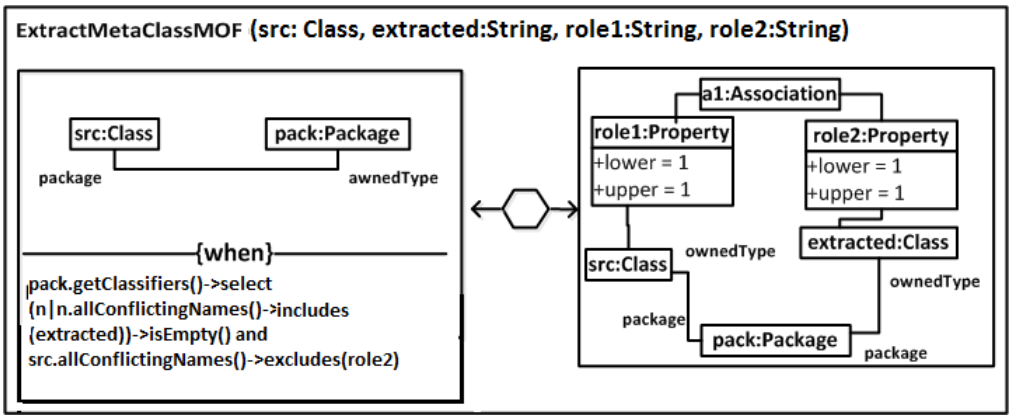

Figure 10: Description en QVT-R de l'opération *ExtractMetaClass* au niveau MOF

- **Discussion :** sur la formalisation QVT proposée par [7], la condition dans la clause *when* est : *pack.ownedType.allConflictingNames()->excludes(extracted) and src.allConflictingNames()->excludes(role1)*. Or que, la condition doit vérifier que le *package (pack)* ne contient pas déjà une méta-classe qui porte le nom *extracted*. Après l'application de cette opération, une nouvelle propriété (*role2* de type extrémité d'association) est ajoutée à la méta-classe *src*. Cette opération n'a pas d'influence sur les contraintes OCL. Ajouter une nouvelle méta-classe au méta-modèle n'altère en aucun cas les contraintes déjà existantes.
- **Proposition :** réécrire la condition de la manière suivante: *pack.getClassifiers()->select(n|n.allConflictiongNames()->includes(extracted)) ->isEmpty() and src.allConflictingNames()->excludes(role2)*(voir figure 10). La transformation QVT corrigée et qui décrit cette opération au niveau MOF est illustrée dans la figure 10.

ExtractSuperMetaClass

- **Définition et exemple :** l'opération *ExtractSuperMetaClass* consiste à créer une nouvelle méta-classe à partir des méta-classes déjà existantes et de les lier avec cette dernière par une relation d'héritage. Elle est souvent utilisée dans le cadre du *refactoring* pour gérer le problème de la duplication des propriétés communes à plusieurs méta-classes. Un exemple d'application de cette opération est présenté à la figure 11.

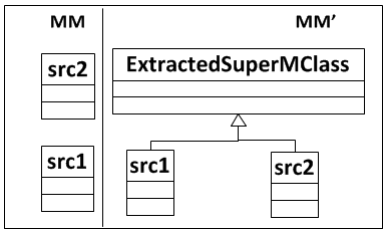

Figure 11: Opération *ExtractSuperMetaClass*

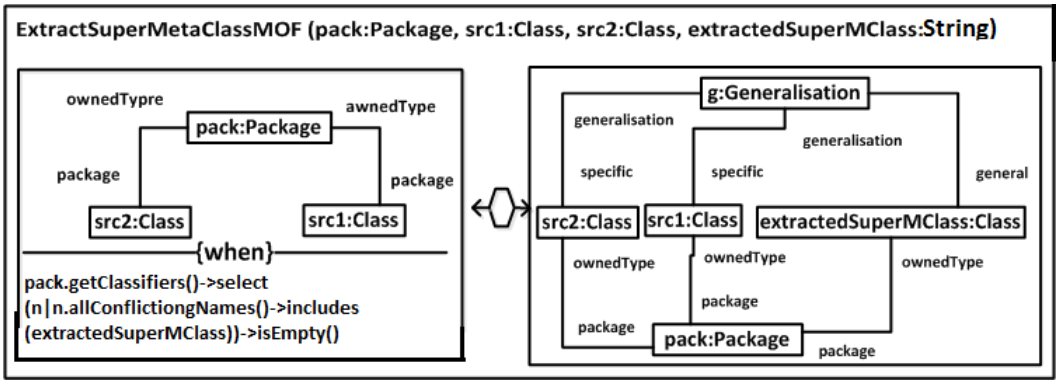

Figure 12: Description en QVT-R de l'opération *ExtractSuperMetaClass* au niveau MOF

- **Discussion :** cette opération n'a pas d'influence sur les contraintes OCL.
- **Proposition :** réécrire la condition dans la clause *when* proposée par [7] de la manière suivante:
pack.getClassifiers()->select(n|n.allConflictingNames()->includes(extracted))->isEmpty()(voir figure 12). La transformation QVT que j'ai corrigé et qui décrit cette opération au niveau MOF est à la figure 12.

3.2.2 Opérations influençant les contraintes OCL

L'application des opérations qui sont décrites dans cette section impactera les contraintes OCL attachées au méta-modèle. Des adaptations seront proposées pour conserver la conformité des contraintes OCL.

FullPullUpProperty

- **Définition et exemple :** cette opération est un cas particulier de l'opération décrite à la figure 7. Le concepteur a pour intention de faire monter la propriété p ainsi que les contraintes OCL qui portent sur cette dernière vers la super-méta-classe. Le but de cette opération est d'appliquer les contraintes sur toutes les sous-méta-classes de la super-méta-classe *Father*.

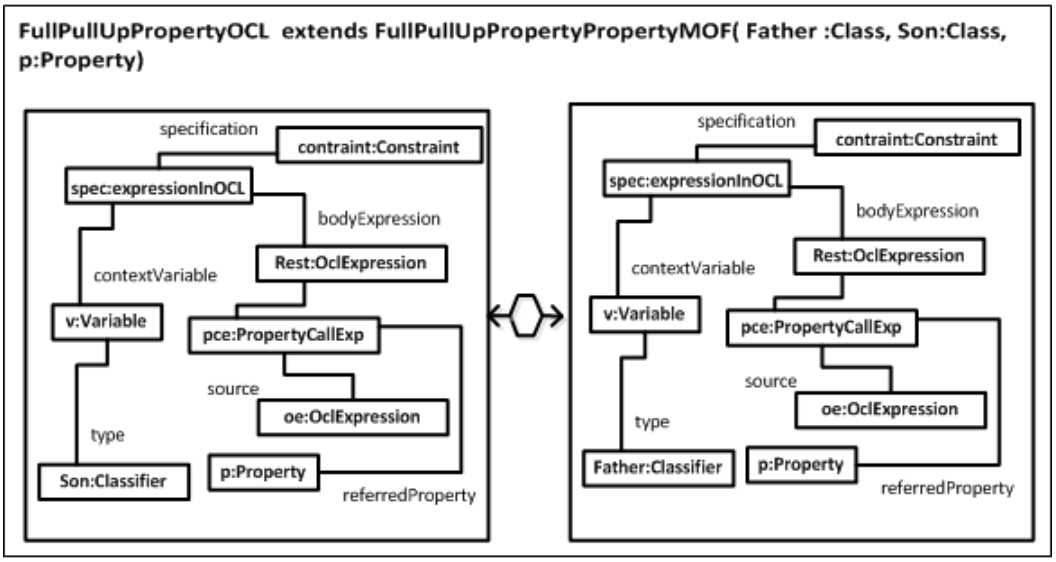

Figure 13: Opération *FullPullUpProperty* au niveau OCL

- **Proposition :** l'adaptation des contraintes consiste ici, à écrire les contraintes dans le contexte de la super-méta-classe. Donc les contraintes qui étaient de la forme: ***context*** *Son* ***inv*** : *oclExpression.p[Rest]*, sachant que *oclExpression* et *[Rest]* sont des expressions OCL et p est la propriété déplacée. Ces contraintes deviennent
context *Father* ***inv*** : *oclExpression.p[Rest]*.

La formalisation QVT de cette adaptations des contraintes OCL est décrites à la figure 13.

Opération d'évolution *PullUpProperty* extrémité d'association

- **Définition et exemple :** cette opération permet de faire monter une propriété de type extrémité d'association d'une méta-classe *Son1* vers la super-méta-classe *Father* comme le montre la figure 14.

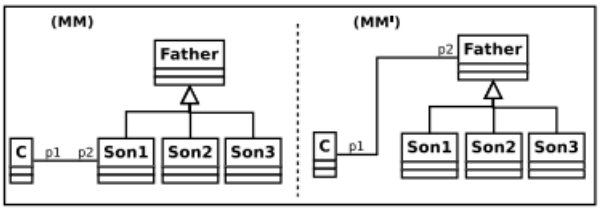

Figure 14: Opération *PullUpProperty* extrémité d'association

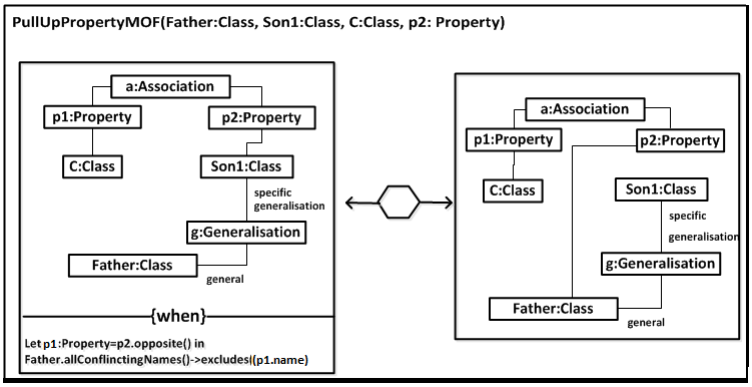

Figure 15: Description en QVT-R de l'opération *PullUpProperty* au niveau MOF

La figure 15 présente la règle QVT corrigée et qui décrit cette opération au niveau MOF. La condition dans la clause *when* impose que la méta-classe *Father* ne contient pas une propriété nommée *p1*.

- **Discussion :** les expressions OCL impactées par l'application de cette opération, sont des contraintes de contexte C qui naviguent vers des instances de la méta-classes *Son1*. Elles sont de la forme:

context *C* ***inv*** *:*
oclExpression.p2[Rest].

Si ces contraintes restent les même, elles vont être appliquées sur toutes les instances des sous-méta-classes de la méta-classe *Father*. Pour garder l'idée initiale du concepteur qui est d'appliquer les contraintes sur seulement les instances de la méta-classes *Son1*. Hassam [7] a proposé de les transformer ainsi:

context *C* ***inv*** :
oclExpression.p2-> select(v | v.oclIsTypeOf(Son1)) -> forAll(s | s[Rest])
.

- **Proposition :**cette adaptation a pour but de faire des sélections sur le type de *p2*. Si *p2* était de type *Son1* alors ils appliquent le reste de la contrainte. Or, si *p2* apparait plusieurs fois dans la contrainte, sa taille augmente en proportion du nombre d'apparitions de la propriété *p2*. Je propose d'introduire une condition *if* dans la contrainte qui permet de vérifier le type de *p2* une seule fois de la manière suivante :

context *C* ***inv*** *:*
if *oclExpression.p2.oclIsTypeOf(Son1)*
then *oclExpression.p2[Rest]* ***else*** *true* ***endif***
.

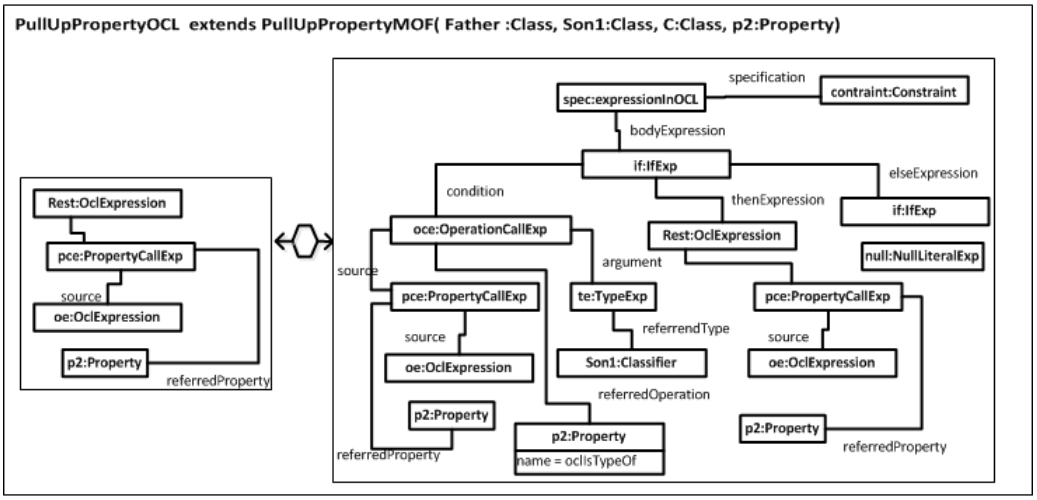

Figure 16: Opération *PullUpProperty* au niveau OCL

L'adaptation des contraintes OCL que je propose pour cette opération

est décrite en QVT-R à la figure 16.

PushDownProperty

- **Définition et exemple :** l'operation *PushDownProperty* est l'inverse de l'operation *PullUpProperty* . Elle consiste à faire descendre une propriété d'une super-méta-classe *Father* vers une de ses sous-méta-classes. Si une propriété de la super-méta-classe *Father* est utilisée juste par une seule sous-méta-classe *Son*, il serait intéressant de faire descendre cette propriété vers la méta-classe *Son*. (voir figure 17).

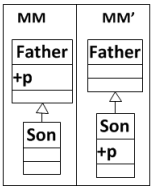

Figure 17: Opération *PushDownProperty*

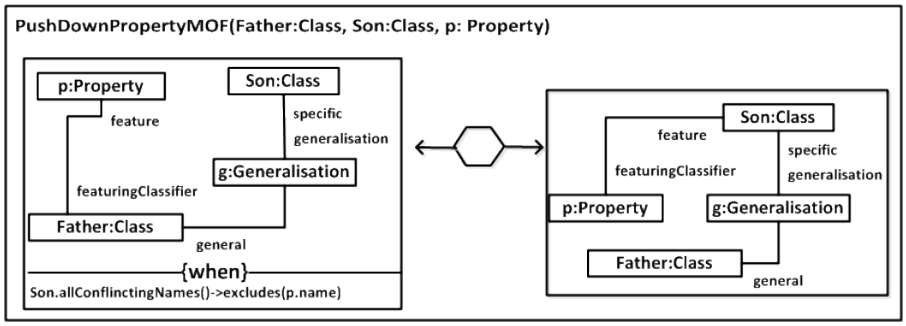

Figure 18: Description en QVT-R de l'opération *PushDownProperty* au niveau MOF

La figure 18 décrit la formalisation de l'opération *PushDownProperty* en niveau du méta-modèle MOF.

- **Discussion :** toutes les contraintes OCL qui portent sur la propriété p sont impactées. L'adaptation proposée par Hassam [7] est la suivante :

context *C* ***inv*** :
oclExpression->select(v | v.oclIsTypeOf(Son))->forAll(v1 | v1.p[Rest]),
pour toutes les expressions qui sont sous la forme :

context *C* ***inv*** *:oclExpression.p[Rest]*, où *oclExpression* est de type *Father*.

- **Proposition :**
  - La taille de la contrainte augmente en fonction du nombre d'apparition de la propriété *p*. Je propose de la transformer ainsi :

    context *C* ***inv*** *:*
    if *oclExpression.oclIsTypeOf(Son)*
    then *oclExpression.p[Rest]* ***else*** *true* ***endif***

  - Pour les contraintes écrites dans le contexte de la super-méta-classe *Father*. Cette adaptation est valide dans le cas où le concepteur a pour intention de faire déplacer la propriété sans déplacer les contraintes. Mais, dans le cas où le concepteur veut faire descendre les contraintes et la propriété vers la méta-classe *Son*, je propose une autre opération *FullPushDown* qui permet de changer le contexte des contraintes de *Father* vers *Son*, comme dans le cas 2 de l'opération *PullUpProperty*.

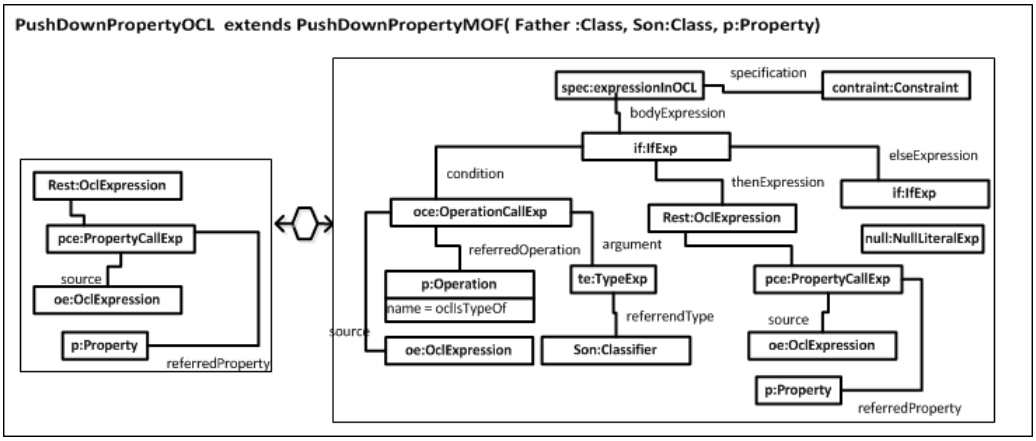

Figure 19: Opération *PushDownProperty* au niveau OCL

La figure 19 présente ma proposition de l'adaptation OCL pour cette opération .

AssociationToClass

- **Définition et exemple :** l'opération *AssociationToClass* est une opération de *refactoring*, qui consiste à extraire une méta-classe à partir d'une association entre deux méta-classes déjà existantes (voir figure 20). Les deux propriétés déjà existantes *p1* et *p2* qui faisaient référence respectivement à C2 et C1 vont faire référence à C3. Les nouvelles propriétés *p3, p4* qui sont contenues dans C3 feront référence respectivement à C1 et C2. Elles doivent être de cardinalité (1-1).

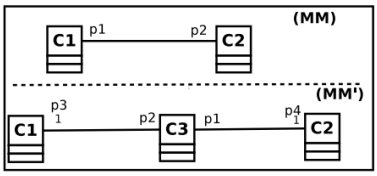

Figure 20: Exemple d'application de l'opération *AssociationToClass*.

- **Discussion :** Les contraintes impactées par cette opération d'évolution sont de la forme : ***oclExpression.p2[Rest]*** telle que *oclExpression* est de type C1. Ces contraintes ont pour but de contraindre les instances de la méta-classe C2. Si les contraintes restent telles quelles, les expressions ***oclExpression.p2[Rest]*** vont porter sur les instances de C3 car la propriété *p2* référence actuellement C3. Hassam [7] a proposé de les transformer comme suit :
oclExpression.p2.p4[Rest].

  Même cas pour les expressions qui sont de la forme : ***oclExpression.p1[Rest]*** telle que *oclExpression* est de type C2 et qui porte sur les instances de C1. Ces expressions vont être adaptées comme suit: ***oclExpression.p1.p3[Rest]***.

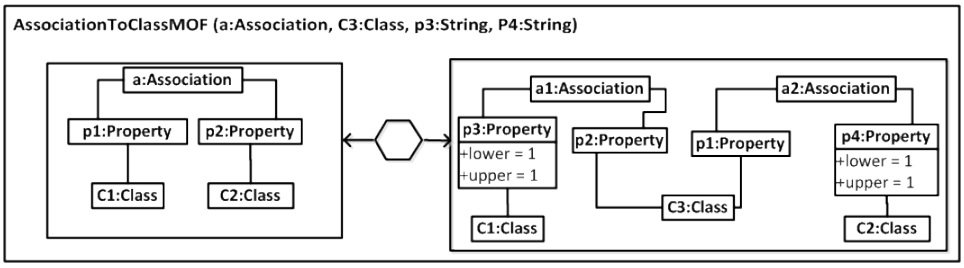

Figure 21: Description en QVT-R de l'opération *AssociationToClass* au niveau MOF

La figure 21 représente les transformation en QVT-R de l'opération *AssociationToClass*.

- **Proposition :**
  - Les expressions OCL ***oclExpression.p1.p3[Rest]*** et ***oclExpression.p2.p4[Rest]*** sont valides dans le cas ou les propriétés *p1* respectivement *p2* sont de cardinatilté (0-1) ou (1-1). Je propose d'étendre ces adaptations pour les cas ou la différence entre la cardinalité maximale et la cardinalité minimale >1. Donc, les contraintes de la forme :
    oclExpression.p1->Iterator(n/n[Rest]), où *Iterator* peut être (*collect, select, forAll, ...*) je propose des les transformer comme suit:
    oclExpression.p1->Iterator (n/n.p3[Rest]).

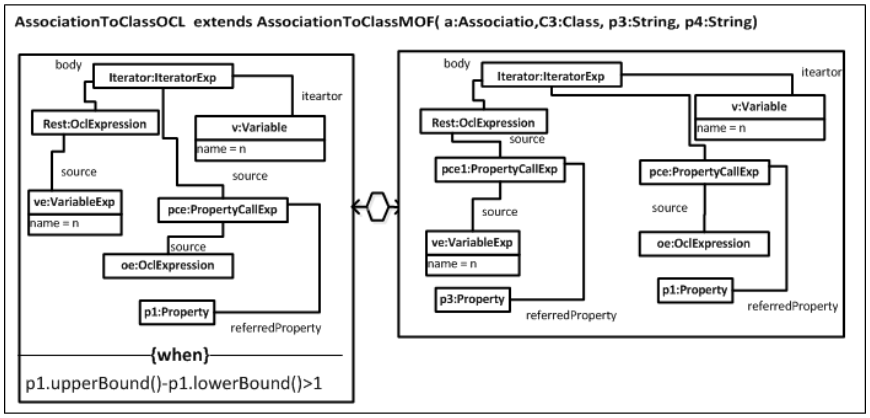

Figure 22: Opération *AssociationToClass* niveau OCL

Je propose aussi l'adaptation suivante :
oclExpression.p2->Iterator (n/n.p4[Rest]) pour les expressions de la forme ***oclExpression.p2->Iterator(n/n[Rest]***)

- Je propose d'ajouter une condition dans la clause *when* de la règle QVT proposée par [7]. Cette condition impose que la transformation est réalisée dans le cas où la différence entre la cardinalité maximale et la cardinalité minimale de *p1* ou bien *p2* est <=1. Pour les différences de cardinalités qui sont >1 , la proposition que je fait est décrite en figure 22 avec une condition sur les cardinalité qui doivent être >1.

ClassToAssociation

- **Définition et exemple :** c'est l'opération inverse de l'opération *AssociationToClass*(Figure 20), elle supprime une méta-classe qui comporte une référence *p3* de cardinalité (1-1) vers une méta-classe C1, et une autre référence *p4* de cardinalité (1-1) vers une autre méta-classe C2. Cette meta-classe est remplacée par une association entre les méta-classes C1 et C2.

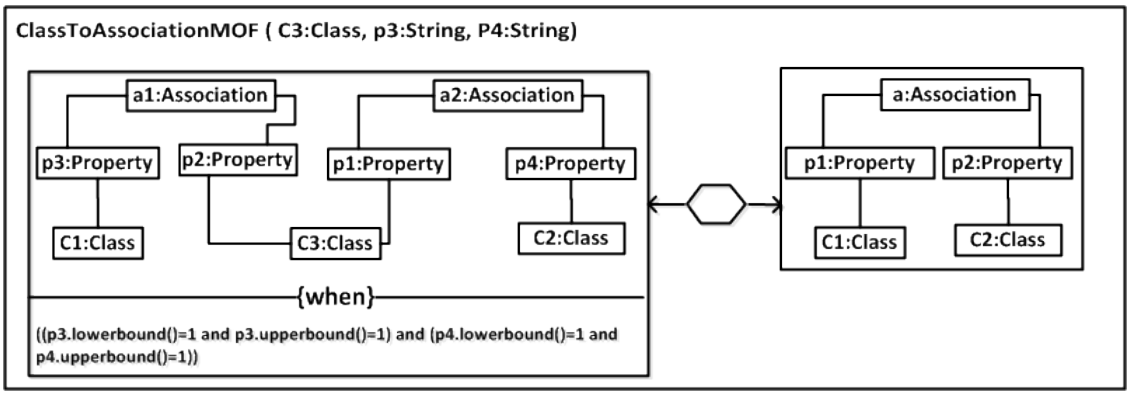

Figure 23: Opération *ClassToAssociation* au niveau MOF

La figure 23 présente les transformation en QVT-R au niveau MOF de l'opération *ClassToAssociation*.

- **Discussion :** Les contraintes impactées par cette opération d'évolution sont toutes les contraintes qui sont de la forme:
oclExpression.p2.p4[Rest] telle que *oclExpression* est de type C1. Ces contraintes ont pour but de contraindre les instance de la méta-classe C2. A l'application de l'opération *ClassToAssociation*, les références *p4* et *p3* seront supprimées. Les contraintes dont la forme est :***oclExpression.p2.p4[Rest]*** telle que *oclExpression* est de type C1 ne seront plus valides. hassam [7] a proposé l'adaptation suivante :***oclExpression.p2[Rest]***.

  Même cas pour les contraintes qui sont de la forme de: ***oclExpression.p1.p3[Rest]*** telle que *oclExpression* est de type C2. Hassam [7] a proposé l'adaptation suivante :***oclExpression.p1[Rest]***. Toutes les contraintes qui portent sur la méta-classe C3 (méta-classe supprimée) seront supprimées.

- **Proposition :** je propose de transformer les expressions OCL qui sont de la forme : ***oclExpression.p1->Iterator(n/n.p3[Rest]***), telle que *oclExpression* est de type C2 et *Iterator*={*collect, select, forAll, ...*} en : ***oclExpression.p1->Iterator (n/n[Rest]***). Ainsi que les expressions qui sont de la forme :***oclExpression.p2->Iterator (n/n.p4[Rest]***) vont être transformer en : ***oclExpression.p2->Iterator (n/n[Rest]***). Cette proposition que je fait est schématisée en QVT-R à la figure 24

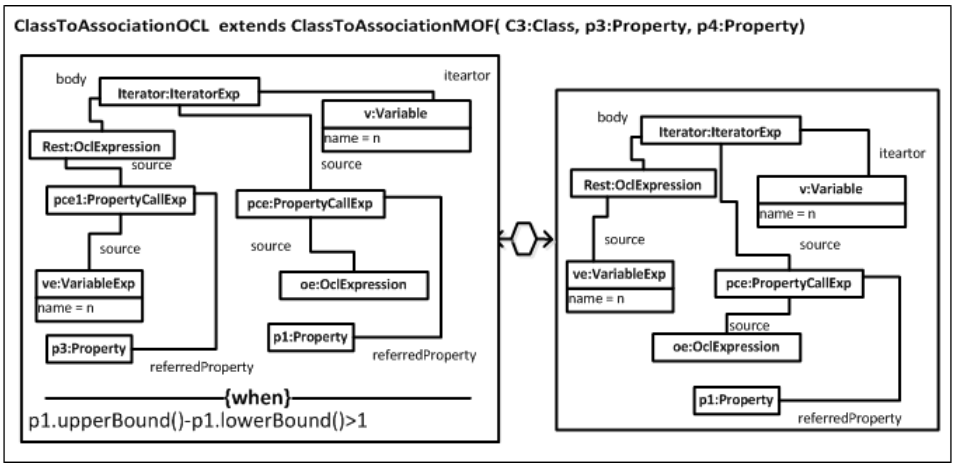

Figure 24: Opération *ClassToAssociation* niveau OCL

InheritanceToComposition

- **Définition et exemple :** c'est une opération du *refactoring* qui consiste à convertir une relation d'héritage entre deux méta-classes en relation de composition. La sous-méta classe devient la classe conteneur de la super-méta-classe. La cardinalité de l'extrémité d'association à côté de la classe contenue (ancienne super-méta-classe) est de (1-1)(voir figure 25).

- **Discussion :** lors de l'application de l'opération *InheritanceToComposition*, deux types de contraintes peuvent être touchées par cette évolution.

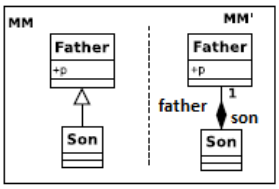

Figure 25: Opération *InheritanceToComposition*

- Type 1: ce sont toutes les contrainte dont les expressions sont de la forme : ***oclExpression->select(v|v.oclIsTypeOf(Son))[Rest]***, telle que *oclExpression* est de type *Father*. Le but de ces contraintes est d'utiliser la relation d'héritage afin de restreindre les instances de *Son*. L'application de *InheritanceToComposition* transforme la relation d'héritage en relation de composition. Ces contraintes deviennent non valides. Afin d'adapter les contraintes à la nouvelle version du méta-modèle, Hassam [7] propose l'adaptation suivante : ***oclExpression.son[Rest]***.
- Type 2 : les expressions des contraintes sont de la forme : ***oclExpression.p[Rest]***, telle que *oclExpression* est de type *Son*. Ces contraintes portent sur des propriétés héritées de la super-méta-classe (*Father*). Hassam [7] a proposé également une adaptation à ce type de contrainte. En effet, ces expressions vont être adaptées de la maniéré suivante :
  oclExpression.father.p[Rest]

  La figure 26 décrit la règle QVT-R que j'ai corrigé de l'opération *InheritanceToComposition* au niveau MOF.

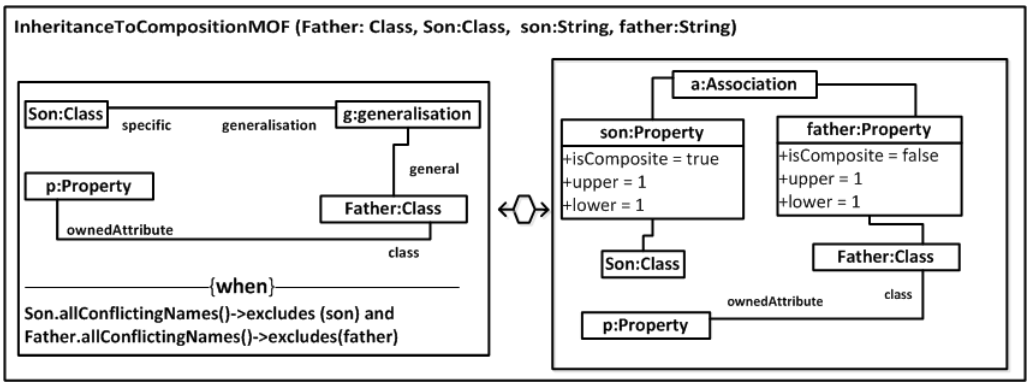

Figure 26: Description en QVT-R de l'opération *InheritanceToComposition* au niveau MOF

GeneralizeProperty/RestrictProperty

- **Définition et exemple :** cette opération consiste à généraliser ou restreindre la cardinalité d'une propriété. La généralisation de la cardinalité exige que l'ancienne cardinalié soit incluse dans la nouvelle. La figure 27 montre un exemple d'une généralisation d'une extrémité d'association. On voit bien que la propriété p avait une cardinalité de (1-1), sa généralisation a augmenté sa cardinalité à (1-N).

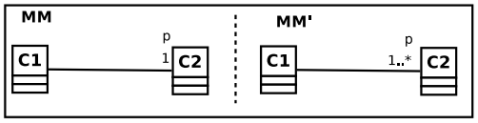

Figure 27: Opération *GeneralizeProperty*

- **Discussion :**
  - Cas d'une généralisation : La formalisation en QVT de l'opération *GeneralizeProperty* au niveau MOF proposée par [7] est dotée d'une clause *when* qui impose la condition que l'ancienne cadinalité soit incluse dans la nouvelle cardinalité, et que l'ancien type de p et le nouveau type *t1* (passé en paramètre) sont exactement les mêmes. Cette opération contient aussi une clause *where*, que je trouve inutile car elle porte sur les extrémités d'association et impose que leur type soit le même que le type *t1* passé en paramètre. une condition déjà vérifiée dans la clause *when*.

    Lors de l'application de l'opération *GeneralizeProperty*, les contraintes OCL dont les expressions sont de la forme : ***oclExpression.p[Rest]***, telle que p est une extrémité d'association généralisée. Ces contraintes sont invalides sur la nouvelle version du méta-modèle car la cardinalité de p a augmentée. Hassam [7] propose l'adaptation suivante:
    oclExpression.p->forAll(v | v[RestExp]).

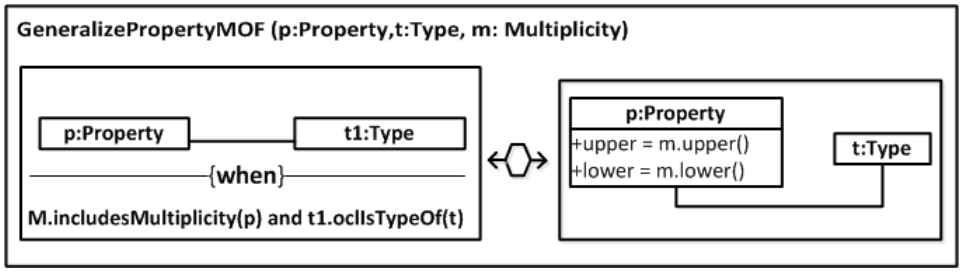

Figure 28: Description en QVT-R de l'opération *GeneralizeProperty* au niveau MOF

- Cas d'une restriction : cette opération est l'inverse de l'opération *GeneralizeProperty*. Elle permet de restreindre la cardinalité d'une extrémité d'association p , les conditions à l'application de cette propriété imposent que la nouvelle cardinalité soit incluse dans l'ancienne, et que le type de p et le type *t1* passé en paramètre sont les mêmes. Les contraintes OCL dont les expressions sont de la forme: ***oclExpression.p->forAll(v | v[Rest])*** sont toujours valides sur la version évoluée du méta-modèle. Donc y a pas nécessité d'une adaptation de contrainte.

- **Proposition :**
  - suppression de la clause *where* sur la formalisation de l'opération *GeneralizeProperty* [7], car elle porte sur une condition qui est déjà vérifiée dans la clause *when*. La formalisation en QVT que je propose pour cette opération au niveau MOF est en figure 28.
  - Ajouter une condition dans la clause *when* pour vérifier que la différence entre la cardinalité maximale et la cardinalité minimale de m est supérieure à 1 ($m.upper()$-$m.lower() > 1$). Si on avait une propriété p avec des cardinalité (1-1), et on lui applique une *GeneralizeProperty* tel que la nouvelle cardinalité est de (0-1) et de type *t1* qui est le même avec *t*. Comme la cardinalité(0-1) inclut (1-1) et ($t1 = t$), alors la condition dans la clause *when* est vérifiée et l'adaptation proposée sera appliquée. Or que, si p est de cardinalité (0-1). les expression OCL qui sont sous la forme ***oclExpression.p[Rest]*** doivent rester telles quelles.

MoveProperty/Metaoperation

- **Définition et exemple :** *MoveProperty* est une opération de *refactoring*, elle permet de déplacer une propriété (méta-attribut, méta-opération, extrémité d'association) d'une méta-classe A vers une autre

méta-classe C. Les deux méta-classes A et C sont liées par une association dont les deux extrémités sont de cardinalités (1-1). La figure 29 montre un exemple d'application de l'opération *MoveProperty*, où la propriété à déplacer est de type extrémité d'association. Les deux extrémités *initial* et *destination* doivent être de cardinalité (1-1). Le but de cette opération est de déplacer la propriété *roleB* (qui est de type extrémité d'association) de la méta-classe A vers la méta-classe B.

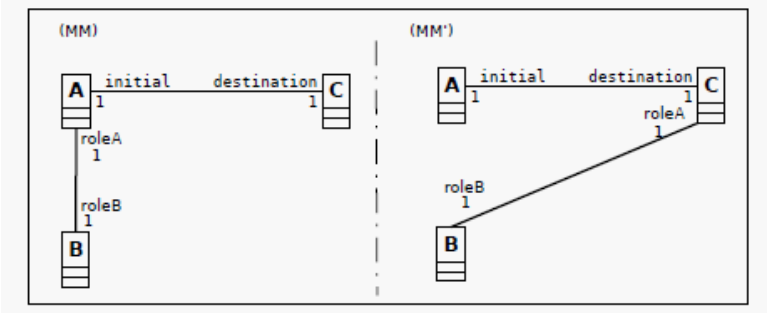

Figure 29: Opération *MoveProperty/Metaoperation*

La figure 30 décrit la règle QVT de l'opération *MoveProperty* telle que la propriété à déplacer est de type extrémité d'association.

- **Discussion :** l'application de cette opération altère toutes les contraintes OCL dont les expressions sont de type A et qui ont pour but de contraindre les instances de la méta-classe B. Ces expressions de la forme: ***oclExpression.roleB*** ne sont plus valides après l'application de l'opération *MoveProperty*, car la méta-classe A ne contient plus la propriété *roleB*. Hassam [7] propose de les adapter comme suit: ***oclExpression.destination.roleB***.

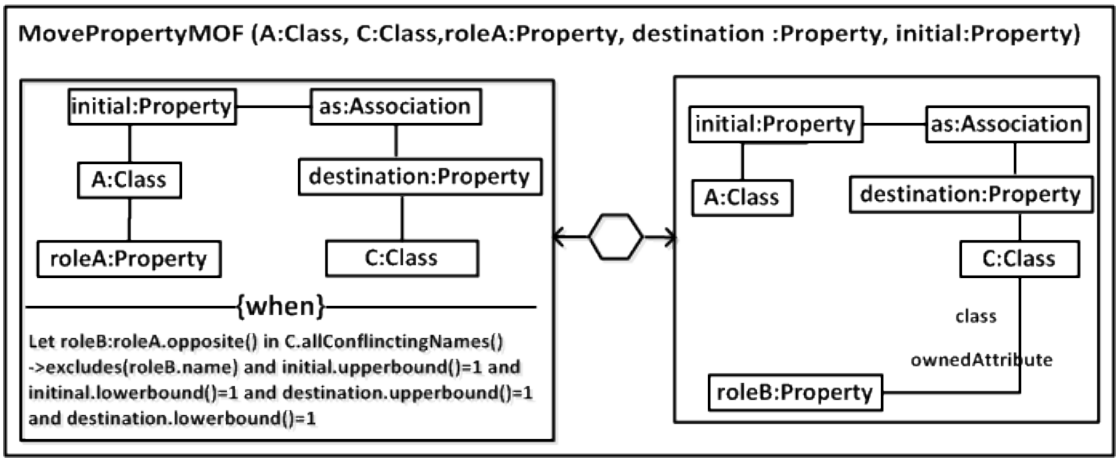

Figure 30: Description en QVT-R de l'opération *MoveProperty* au niveau MOF

Les contraintes dont les expressions sont de la forme ***oclExpression.roleA***, telle que *oclExpression* est de type de la méta-classe B sont impactées. En effet, Ces expressions portaient avant l'application de l'opération *MoveProperty* sur les instances de la méta-classe A, si ces expressions restent telles-quelles, elles vont porter sur les instances de la méta-classe C. l'adaptation proposée par [7] est la suivante : ***oclExpression.roleA.initial***.

3.2.3 Opérations nécessitant la suppression des contraintes OCL

Cette section est consacrée à la description des opérations d'évolution dont l'application impose la suppression des contraintes OCL.

Opération d'évolution *InlineMetaclass*

- **Définition et exemple :** cette opération est l'inverse de l'opération *ExtractMetaclass*, elle consiste à supprimer une méta-classe liée à une autre par une association dont les deux extremité sont de cardinalité (1-1).

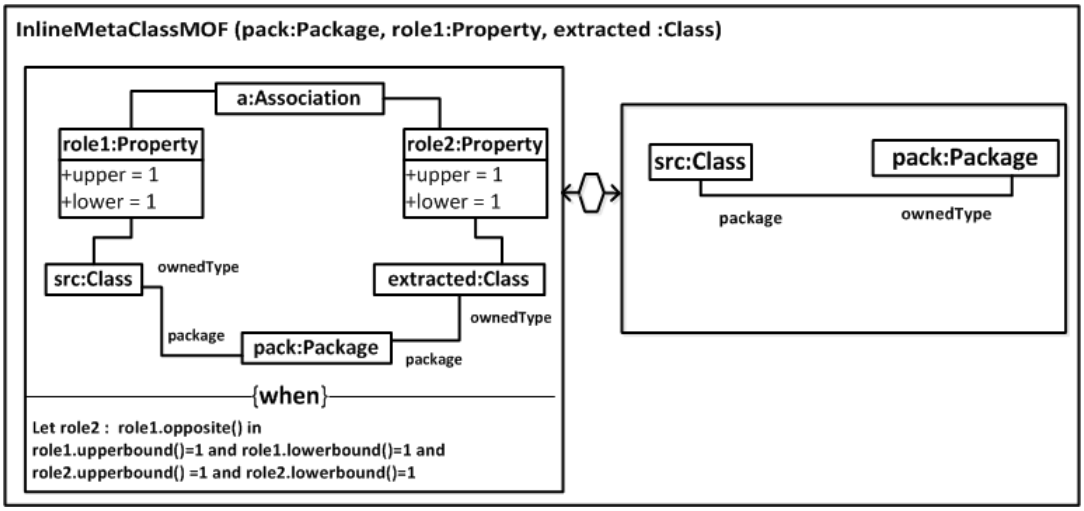

Figure 31: Opération *InlineMetaClass* au niveau MOF

La figure 31 décrit la formalisation de cette opération en QVT que j'ai corrigé en ajoutant une condition dans la clause *when*au pattern gauche(LHS), cette condition impose l'existence d'une association entre les deux méta-classes (*src* et *ExtractedMClass*), ainsi que les cardinalités des deux extrémités de cette association sont de (1-1). Lorsque ces deux conditions sont vérifiées, le pattern droit est appliqué et la méta-classe *ExtractMClass* est supprimée.

- **Discussion :** l'application de cette opération sur un méta-modèle impacte toutes les contraintes qui sont dans le contexte de la méta-classe supprimée (*ExtractedMClass*). Ces contraintes doivent être supprimées car elles ne servent à rien. Ainsi, que toutes autres contraintes qui portent sur les extrémités d'association (*role1* ou *role2*).

Opération d'évolution *FlattenHierarchy*

- **Définition et exemple :** cette opération est l'inverse de l'opération *ExtractSuperMetaClass*. Elle consiste à supprimer une super-méta-classe et la relation d'héritage qui la lie avec ses sous-classes.

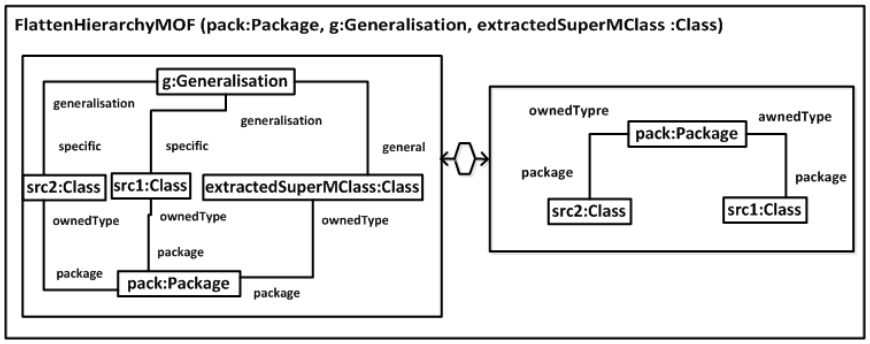

Figure 32: Opération *FlattenHierarchy* au niveau MOF

La figure 32, décrit la règle QVT qui permet d'appliquer cette opération au niveau MOF.

- **Discussion :** La suppression de cette super-méta-classe dans un méta-modèle rend les contraintes qui utilisent ou qui sont définies sur cette super-méta-classe invalides. Par conséquent, elles doivent êtres supprimées.

RemoveMetaclass/Propery/Operation

- **Définition et exemple :** Les opérations *RemoveMetaclass, Remove-Propery et RemoveMetaOperation* sont des opérations qui peuvent intervenir dans un cadre d'évolution de méta-modèle. Leur application peut rendre invalides les contraintes OCL définies sur les méta-classes supprimées, ainsi que les contraintes qui font référence à des propriétés ou méta-opérations qui viennent d'être supprimées. C'est pour cela que [7] propose de supprimer toutes les contraintes associées à ces éléments supprimés.

- **Discussion :** toutes les contraintes qui portent sur des propriétés supprimées sont invalides et doivent être elles aussi supprimées.

4 Outillage

l'approche présentée dans la section 3 a été implémentée sous forme d'un prototype sous la forme d'un *Plugin Eclipse* pendant la thèse de Hassam [7]. Le *Plugin Eclipse* presenté à la figure 33 a été repris pour correction, amélioration et extension dans le cadre de ce stage.

Pour la reprise de ce *Plugin*, j'ai travaillé en collaboration avec Fabien CHIFFRE un étudiant de 2ème année IUT. J'avais comme tâche l'implantation

en QVT des opérations au niveau ECORE et les adaptations OCL. De son côté, il a repris la partie interface graphique du plugin *Eclipse*.

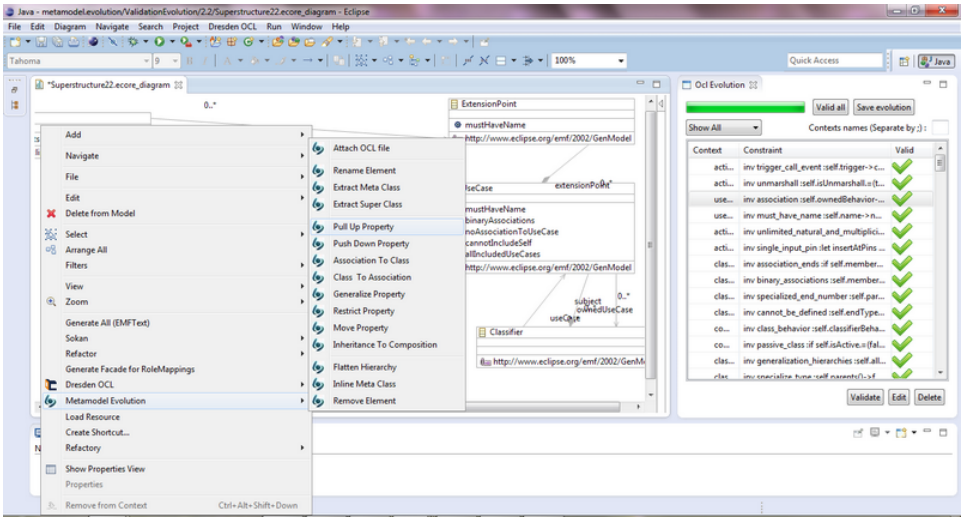

Figure 33: *Plugin*

Lors de l'application d'une opération d'évolution, le système applique une première transformation QVT qui consiste à faire évoluer le méta-modèle (Formalisation au niveau MOF), puis une autre transformation QVT qui permet d'adapter les contraintes à cette évolution (Formalisation au niveau OCL). J'ai implanté toutes les règles QVT décrites au niveau MOF de toutes les opérations et aussi les règles QVT pour l'adaptation des contraintes des opérations qui ont une influence sur les contraintes OCL.

Mon travail au niveau *Plugin* a comporté deux parties :

- **1ère Partie** : consiste à l'implantation de toutes les opérations au niveau ECORE. Toutes les formalisations au niveau MOF en QVT-R vues précédemment devaient être réécrites avec une syntaxe textuelle. Car il n'existe pas un outil qui permet de passer d'une syntaxe graphique de QVT-R en une syntaxe textuelle. L'outil *Borland Together Architect* a été utilisé pour réécrire toutes les formalisations QVT graphiques sous formes textuelles.
  Comme toute transformation de modèle, la transformation QVT prend en entrée un méta-modèle et un modèle qui lui est conforme et fournit un modèle conforme au méta-modèle de sortie. Dans notre cas le méta-modèle d'entrée est le méta-modèle ECORE défini par EMF qui est aussi le méta-modèle de sortie. Le modèle d'entrée est le méta-modèle fourni par le concepteur et sur lequel les opérations seront appliquées

(ce méta-modèle est un modèle conforme au méta-modèle ECORE de EMF). Le modèle de sortie n'est rien d'autre que le méta-modèle du concepteur après évolution.

- **2ème Partie** : toutes les adaptations des contraintes OCL proposées dans la section précédente ont été écrites dans la syntaxe textuelle de *Borland Together Architect*. Cette partie nécessitait une connaissance parfaite du méta-modèle OCL 2.0. Les contraintes OCL sont généralement écrites sous forme textuelle dans un fichier (Contraintes.ocl). Ce fichier doit être parsé et transformé en un autre fichier (Contraintes.xmi) qui n'est rien d'autre qu'un modèle conforme au méta-modèle OCL. J'ai utilisé l'outil OCLMETRICS (Triskel 2012) pour transformer les contraintes d'une forme textuelle e forme modèle. Ce modèle est utilisé comme un modèle d'entrée pour la transformation QVT-R au niveau OCL d'une opération d'évolution. La transformation QVT parcours les contraintes une à une et applique l'adaptation OCL décrite dans la règle QVT. A la fin de la transformation, le modèle fournit en sortie est un modèle conforme au méta-modèle OCL et qui contient toutes les contraintes adaptées. Les contraintes adaptées écrites sous forme modèle (Contrainte.xmi) sont transformées en une forme textuelle (Contraintes.ocl) et affichées au concepteur. Ce dernier peut ensuite de valider, éditer ou bien supprimer les adaptations proposées.

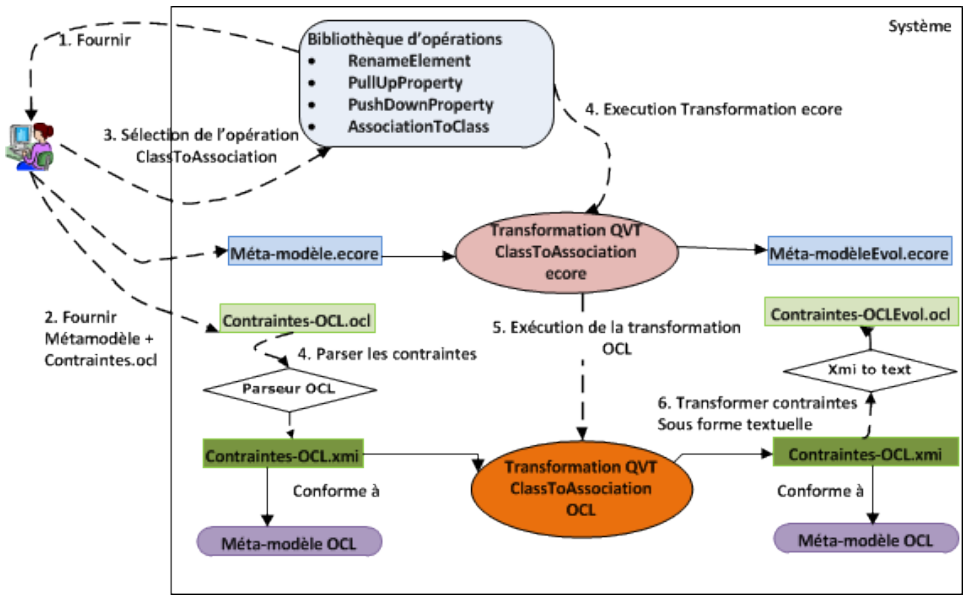

Figure 34: Exemple de l'application de l'opération d'évolution *ClassToAssociation*

La figure 34 présente un exemple d'application de l'opération d'évolution *ClassToAssociation*. Le *Plugin* fournit au concepteur une liste d'opérations d'évolution (bibliothèque d'opérations de base). Le concepteur fournit au système le méta-modèle ECORE (le méta-modèle à évoluer) ainsi que le fichier qui contient toutes les contraintes OCL définis sur ce méta-modèle (Contraintes.ocl). Ces contraintes sont sous forme textuelle. Le système doit les transformer en un modèle OCL exploitable. Quant le concepteur choisit d'appliquer l'opération *ClassToAssociation*, la transformation QVT-R au niveau *ecore* de l'opération *ClassToAssociation* est lancée avec comme entrée le méta-modèle du concepteur. A la fin de la transformation, le méta-modèle est évolué. Le système lance alors la transformation QVT-R pour l'adaptation des contraintes OCL impactées par cette évolution. Cette transformation prend en entrée le modèle OCL et génère un modèle OCL conforme au méta-modèle OCL. Ce modèle OCL est ensuite transformé sous forme textuelle où toutes les adaptations sont proposées au concepteur à travers l'interface graphique (figure 33). Le concepteur collabore avec le système pour valider, éditer, ou supprimer les adaptations proposées. Les formes textuelles de la transformation QVT-R au niveau ECORE de l'opération *ClassToAssociation* est fournies en annexe.

5 Validation de l'approche

Mon stage avait parmi ses objectifs de valider l'approche sur des évolutions en "vrai grandeur" des méta-modèles. Nous avons prévu une validation de notre approche sur le méta-modèle UML lors de ses évolutions de la version 2.2 à 2.3 et de la version 2.3 à 2.4. Les méta-modèles d'UML des différentes versions sous forme Ecore ont été récupérés auprès de l'équipe Triskel de IRISA (Rennes). Chaque version est composée de deux fichiers de spécification OMG : Superstructure.xmi et Infrastructure.xmi. Le fichier Superstructure.xmi décrit les spécifications de chaque diagramme d'UML. Le fichier Infrastructure.xmi représente le cœur d'UML. Il porte les spécifications sur les éléments de base composant les différents diagrammes d'UML. Ces deux fichiers de spécifications sont des modèles MOF qui décrivent le méta-modèle UML. Ils portent des contraintes OCL. Ces contraintes OCL ont été extraites par un script écrit par l'équipe de Triskel.

Le but de cette validation consiste à : i) comparer deux versions du méta-modèles UML (par exemple le méta-modèle UML2.2 et le méta-modèle UML2.3); ii) recenser les différentes opérations qui permettent de faire évoluer le méta-modèle UML2.2 au UML2.3; iii) appliquer la suite d'opérations sur le méta-modèle UML2.2 afin d'atteindre le méta-modèle UML2.3. Comparer les fichiers OCL générés par notre outil et les fichiers OCL existants. Élaborer une étude comparative et statistique.

- **Évolution UML 2.2 à 2.3 et 2.3 à 2.4**: les deux méta-modèles *UMLSuperstructure2.2.ecore* et *UMLSuperstructure2.3.ecore* ont été comparés avec *EMFCompare* dans le but de déterminer la suite d'opérations (DIF2.2-2.3Superstructure, table 4 colonne2) de base à appliquer sur *UMLSuperstructure2.2.ecore* pour avoir le méta-modèle *UMLSuperstructure2.3.ecore*.
  Les contraintes OCL ont été extraites sur les deux fichiers et sauvegardées dans les fichiers
  UMLInfrastructure2.3.ocl, UMLSuperstructure2.3.ocl.

  Le but est d'appliquer la suite (DIF2.2-2.3Superstructure) sur le méta-modèle *UML2.2Superstructure.ecore* et le fichier *UML2.2Superstructure.ocl* et comparer le fichier OCL obtenu avec *UML2.3Superstructure.ocl*

  La même démarche a été appliquée pour l'évolution d'UML de la version 2.3 à 2.4. Le (DIF2.3-2.4Superstructure, table 4 colonne3) présente les opérations d'évolution recensées entre les deux méta-modèle UML 2.3 et 2.4.

- **Résultats** : La partie validation n'est pas encore achevée, il reste à comparer les deux méta-modèle UML (*Infrastructure.ecore*) pour toutes les versions citées précédemment, recenser toutes les opérations d'évolutions, puis appliquer ces opérations sur les méta-modèle UML (*Infrastructure.ecore*).

  La table 2 présente le nombre de contraintes OCL parsant et non-parsant de chaque version au niveau *Superstructure*. La table 3 illustre l'étude statistique qu'on a faite afin de détecter le nombre de contraintes ajoutées, supprimées, ou bien modifiées d'une version à une autre.

UML Superstructure	**ContraintesParsées**	**Contraintes Parsent pas**	**Total**
Version UML 2.2	134	69	203
Version UML 2.3	150	66	216
Version UML 2.4	150	66	216

Table 2: Nombre de contraintes parsants et non-parsants en UML Superstructure Version 2.2, 2.3, 2.4

Différence UML Superstructure	Contraintes OCL			
	ajoutées	modifiées	supprimées	sans changements
UML2.2 ->2.3	15	11	1	190
UML 2.3 ->2.4	1	6	1	209

Table 3: Differences des contraintes entre les Version 2.2, 2.3, 2.4 de UML Superstructure

Opération	Évolution UML Superstructure	
	2.2->2.3	**2.3->2.4**
IntroduceProperty	33	32
EliminateProperty	9	5
IntroduceClass	3	5
EliminateClass	1	8
RenameElement	2	7
GeneralizeProperty	2	4
RestrictProperty	2	0

Table 4: Opérations d'évolutions déterminées entres les Version 2.2-> 2.3 et 2.3->2.4 de UML Superstructure

6 Conclusion

Dans ce rapport, je me suis intéressé aux travaux qui traitent des problèmes qui surgissent après l'évolution d'un méta-modèle, dont la synthèse est donnée par la figure 1. L'étude bibliographique effectuée dans le cadre de ce master a permis d'identifier les aspects communs entre ces différents travaux. Pour résoudre ces problèmes, la plupart des travaux reposent sur l'approche de coévolution. Le principe étant de faire des modifications sur les éléments (modèles, modèles de transformation ou contraintes OCL) du méta-modèle après chaque évolution de ce dernier. Les différents travaux définissent la granularité des modifications à travers des opérations dites de base. Les travaux se distinguent, le plus souvent, par l'ensemble d'opérations de base pris en compte et leur mode d'application :

- progressif : où les éléments impactés sont modifiés après chaque opération de base ;
- a posteriori : où les opérations de base sont déduites après une comparaison entre deux versions (méta-) modèles.

Les travaux de Hassam et al [4], Markovic et al [9], Wachsmuth [15] ont constitué pour moi une base pour élaborer une approche pour ce qui concerne

la coévolution des contraintes OCL lors d'une évolution de méta-modèle.

Durant ce rapport, j'ai proposé des corrections et des extensions pour les opérations d'évolution et les adaptations OCL proposée par Hassam [7] lors de son travail de thèse. J'ai participer à l'amélioration de l'outil proposé par Hassam [7], en implantant toutes les transformations QVT-R, soit au niveau MOF ou bien au niveau OCL. L'approche est en cours de validation le cas d'évolution du méta-modèle UML de la version 2.2 à 2.3 et de la version 2.3 à 2.4. Valider notre approche sur autres méta-modèles comme CORBA par exemple figure parmi nos perspectives.

Dans ce stage, nous n'avons considéré que les problèmes liés à l'évolution d'un méta-modèle et son impacte sur les contraintes OCL qui lui sont associées. Dans ce cas, nous sommes dans un carde de méta-modèle constant (MOF). Selon la figure 1, il reste le problème des contraintes attachées à un modèle après sa transformation à travers un modèle de transformation. Dans ce cas, on parle d'une évolution exogènes : le méta-modèle du modèle source est diffèrent du méta-modèle du modèle cible. Il est nécessaire de transformer les contraintes OCL du modèle source pour avoir leurs équivalents pour le modèle cible. Ce travail mérite d'être traité afin de permettre au modèle cible d'évoluer tout en restant conforme aux contraintes initiales.

References

[1] Cicchetti, A., Ruscio, D. D., Eramo, R., and Pierantonio, A. Automating co-evolution in model-driven engineering. In *12th International IEEE Enterprise Distributed Object Computing Conference, ECOC 2008, 15-19 September 2008, Munich, Germany* (2008), IEEE Computer Society, pp. 222–231.

[2] Garcea, J., Diaz, O., and Azanza, M. Model transformation co-evolution: A semi-automatic approach. In *Software Language Engineering*, K. Czarnecki and S. Hedin, Eds., vol. 7745 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 144–163.

[3] Garces, K., Jouault, F., Cointe, P., and Bezivin, J. Managing model adaptation by precise detection of metamodel changes. In *Model Driven Architecture - Foundations and Applications*, R. Paige, A. Hartman, and A. Rensink, Eds., vol. 5562 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009, pp. 34–49.

[4] Hassam, Sadou, S., Gloahec, V. L., and Fleurquin, R. Assistance system for ocl constraints adaptation during metamodel evolution. In *15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany* (2011),

T. Mens, Y. Kanellopoulos, and A. Winter, Eds., IEEE Computer Society, pp. 151–160.

[5] Hassam, K., Sadou, S., and Fleurquin, R. Adapting OCL Constraints After a Refactoring of their Model Using an MDE Process. In *BElgian-NEtherlands software eVOLution seminar (BENEVOL 2010)* (Lille, France, 2010), pp. 16–27.

[6] Herrmannsdoerfer, M., Benz, S., and Jurgens, E. Cope - automating coupled evolution of metamodels and models. In *ECOOP* (2009), S. Drossopoulou, Ed., vol. 5653 of *Lecture Notes in Computer Science*, Springer, pp. 52–76.

[7] Kahina, H. *Adaptation des contraintes OCL lors de l'évolution des méta-modèles et les modèles.* PhD thesis, Université de Bretagne Sud, 2011.

[8] Levendovszky, T., Balasubramanian, D., Narayanan, A., Shi, F., Buskirk, C., and Karsai, G. A semi-formal description of migrating domain-specific models with evolving domains. *Software and Systems Modeling* (2013), 1–17.

[9] Markovic, S., and Baar, T. Refactoring ocl annotated uml class diagrams. In *Model Driven Engineering Languages and Systems*, L. Briand and C. Williams, Eds., vol. 3713 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2005, pp. 280–294.

[10] Mendez, D., Etien, A., Muller, A., and Casallas, R. Towards Transformation Migration After Metamodel Evolution. In *Model and Evolution Wokshop* (Olso, Norway, Oct. 2010).

[11] Narayanan, A., Levendovszky, T., Balasubramanian, D., and Karsai, G. Automatic domain model migration to manage metamodel evolution. In *Model Driven Engineering Languages and Systems* (2009), vol. 5795 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg.

[12] Rose, L., Kolovos, D., Paige, R., Polack, F., and Poulding, S. Epsilon flock: a model migration language. *Software and Systems Modeling* (2012), 1–21.

[13] Rose, L. M., Herrmannsdoerfer, M., Williams, J. R., Kolovos, D. S., Garces, K., Paige, R. F., and Polack, F. A. C. A comparison of model migration tools. In *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I* (2010), D. C. Petriu, N. Rouquette, and ystein Haugen, Eds., vol. 6394 of *Lecture Notes in Computer Science*, Springer, pp. 61–75.

[14] Rose, L. M., Paige, R. F., Kolovos, D., and Polack, F. A. C. An analysis of approaches to model migration. In *Proc. Models and Evolution (MoDSE-MCCM) Workshop, 12th ACM/IEEE International Conference on Model Driven Engineering, Languages and Systems* (2009).

[15] Wachsmuth, G. Metamodel adaptation and model co-adaptation. In *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings* (2007), E. Ernst, Ed., vol. 4609 of *Lecture Notes in Computer Science*, Springer, pp. 600–624.

```
transformation ClassToAssociation;

metamodel  'http://www.eclipse.org/emf/2002/Ecore';


mapping main(in Metamodel:ecore ::EPackage,in souspackage:ecore ::EPackage): ecore::EPackage {
  init {




    var C1:=souspackage.getEClassifier('attrib');
    var C2:=souspackage.getEClassifier('Classifier');
    var C3:=souspackage.getEClassifier('c3');
    var P3:='p3';
    var P4:='p4';
    var P1:='p1';
    var P2:='p2';
    var d:=
classtoassociation(Metamodel,souspackage,C1.oclAsType(ecore::EClass),C2.oclAsType(ecore::EClas
s),C3.oclAsType(ecore::EClass),P3,P4);

    result:=Metamodel;
    }


}
query classtoassociation(inout Metamodel:ecore ::EPackage,inout souspackage:ecore
::EPackage,inout C1:ecore::EClass,inout C2:ecore::EClass,inout C3:ecore::EClass,inout
P3:String,in P4:String):ecore::EPackage{

    if findLHSMOF(Metamodel,souspackage,C1,C2,C3,P3,P4) then
        applyRHSMOF(Metamodel,souspackage,C1,C2,C3,P3,P4 )=1

    else true
    endif;
    undefined
}


query findLHSMOF(inout Metamodel:ecore ::EPackage,inout souspackage:ecore ::EPackage,inout
C1:ecore::EClass,inout C2:ecore::EClass,inout C3:ecore::EClass,inout P3:String,in
P4:String):Boolean{




     whentest(Metamodel,souspackage,C1,C2,C3,P3,P4 )


}


query whentest(inout Metamodel:ecore ::EPackage,inout souspackage:ecore ::EPackage,inout
C1:ecore::EClass,inout C2:ecore::EClass,inout C3:ecore::EClass,inout P3:String,in P4:String):
Boolean{
     C1.eStructuralFeatures->select(n|n.oclIsKindOf(ecore::EReference )and
n.oclAsType(ecore::EReference).eOpposite.eContainingClass=C3 )->notEmpty() and
C2.eStructuralFeatures->select(n|n.oclIsKindOf(ecore::EReference )and
n.oclAsType(ecore::EReference).eOpposite.eContainingClass=C3 )->notEmpty() and
```

```
     C3.eStructuralFeatures->select(p|p.name=P3 and p.upperBound=1 and
p.lowerBound=1)->notEmpty()and
      C3.eStructuralFeatures->select(p|p.name=P4 and p.upperBound=1 and
p.lowerBound=1)->notEmpty()

}


mapping applyRHSMOF(inout Metamodel:ecore ::EPackage,inout souspackage:ecore ::EPackage,inout
C1:ecore::EClass,inout C2:ecore::EClass,inout C3:ecore::EClass,inout P3:String,in
P4:String):ecore::EPackage{

        init{


            var reference1:ecore::EReference
:=C1.eStructuralFeatures->select(n|n.oclIsKindOf(ecore::EReference )and
n.oclAsType(ecore::EReference).eOpposite.eContainingClass=C3
)->first().oclAsType(ecore::EReference) ;
            var reference2:ecore::EReference
:=C2.eStructuralFeatures->select(n|n.oclIsKindOf(ecore::EReference )and
n.oclAsType(ecore::EReference).eOpposite.eContainingClass=C3
)->first().oclAsType(ecore::EReference) ;
           var
p3:ecore::EReference:=C3.eStructuralFeatures->select(n|n.oclIsKindOf(ecore::EReference )and
n.oclAsType(ecore::EReference).eOpposite.eContainingClass=C1
)->first().oclAsType(ecore::EReference) ;
           var
p4:ecore::EReference:=C3.eStructuralFeatures->select(n|n.oclIsKindOf(ecore::EReference )and
n.oclAsType(ecore::EReference).eOpposite.eContainingClass=C2
)->first().oclAsType(ecore::EReference) ;
            reference1.eOpposite:=reference2;
            reference2.eOpposite:=reference1;


            reference1.eType:=C2;
            reference2.eType:=C1;


       souspackage.
eClassifiers:=souspackage.eClassifiers->excluding(C3.oclAsType(ecore::EClassifier))->asOrdered
Set();


            }

    }
```