Assistance System for OCL Constraints Adaptation during Metamodel Evolution.

Kahina Hassam, Salah Sadou, Vincent Le Gloahec, Régis Fleurquin

Assistance System for OCL Constraints Adaptation During Metamodel Evolution

Kahina Hassam
Valoria laboratory,
Université de Bretagne-Sud, France
Email: kahina.hassam@univ-ubs.fr

Salah Sadou
Valoria laboratory,
Université de Bretagne-Sud, France
Email: salah.sadou@univ-ubs.fr

Vincent Le Gloahec
Alkante SAS, France
Email: v.legloahec@alkante.com

Régis Fleurquin
Valoria laboratory,
Université de Bretagne-Sud, France
Email: regis.fleurquin@univ-ubs.fr

***Abstract*—Metamodels evolve over time, as well as other artifacts. In most cases, this evolution is performed manually by stepwise adaptation. In MDE, metamodels are described using the MOF language. Often OCL constraints are added to metamodels in order to ensure consistency of their instances (models). However, during metamodel evolution these constraints are omitted or manually rewritten, which is time consuming and error prone.**

We propose a tool to help the designer to make a decision on the constraints attached to a metamodel during its evolution. Thus, the tool highlights the constraints that should disappear after evolution and makes suggestions for those which need adaptation to remain consistent. For the latter case, we formally describe how the OCL constraints have to be transformed to preserve their syntactical correctness. Our adaptation rules are defined using QVT which is the OMG standard language for specifying model-to-model transformations.

I. Introduction

Model Driven Engineering (MDE) [1] raises the abstraction's level of software development from code to models, with a great emphasis on focusing the developer concerns on the problem domain rather than on the underlying technologies. Metamodels are the definition of abstract syntax of languages. However, structural constructions of metamodeling languages do not allow to express completely the syntax of a language (e.g. context-sensitive properties). In consequence, to get coherent models, it is necessary to add constraints expressed using the Object Constraint Language *(OCL)* [2].

Moreover, metamodels evolve over time like other software artifacts [3], [4], due to several reasons: to be consistent with the evolution of the application domain scope, to improve or correct the abstract syntax of the language, etc. As a consequence, these changes may break consistency of related terminal models. The risks stem from the fact that constraints attached to metamodels are either omitted or manually rewritten, which is time consuming and error prone [4], [5].

To address this problem, we propose an approach in order to assist designers in constraint adaptation during a stepwise evolution [6] of a metamodel. This approach consists in identifying the impact of a change, made at the metamodel, on its associated constraints. After each modification of the metamodel, the assistance system shows the constraints that become obsolete or proposes a transformation of the latter, whenever it is possible. For constraint transformation we use grammar adaptation. Thus, operations on metamodel are formalized using the QVT Relation part of the OMG's standard Query/Views/Transformations (QVT). For each QVT Relation, we propose an adaptation to the constraints (also described using QVT Relation) that involve artifacts that are modified by the concerned operation.

In the next section we present an example that highlights the problem raised by metamodel evolution. we describe our approach in Section III. In Section IV, we show how to adapt constraints to a change in their metamodel. We present an implementation of our solution in Section V. And before concluding in Section VII, we give some related work in Section VI.

II. Motivating Example

The work presented in this paper was mainly initiated by several requirements of our industrial partner, Alkante, a company specialized in the design of Web applications and Geographical Information Systems. To build their applications, the company has defined its own Architecture Description Language (ADL), referred to as AlCoWeb [7]. The abstract syntax of this ADL is inspired by the UML2 Component model. Along with this ADL, the company has elaborated a complete Model-Driven platform, AlCoWeb-Builder, which allows the graphical definition of AlCoWeb models and the automatic code generation of corresponding Web applications.

To ensure the quality of produced models, the AlCoWeb metamodel comes with a large set of OCL constraints that are checked during the design stage. Due to constant improvements of its underlying Web development framework, the company makes evolve its ADL accordingly, but also has to deal with the co-evolution of OCL constraints.

The Figure 1 presents two versions of the AlCoWeb ADL abstract syntax: on the left side, the initial version of the ADL (AlCoWeb-v1), and on the right side, a revised version obtained after several stepwise adaptations (AlCoWeb-v2).

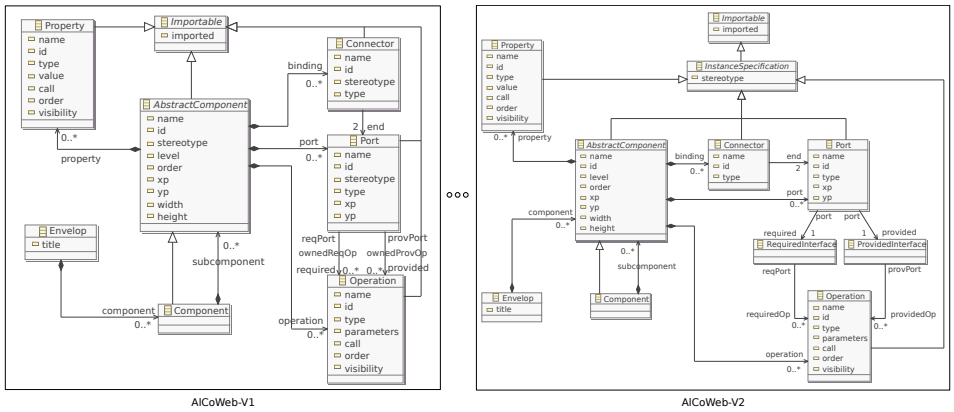

Fig. 1. Evolution of AlCoWeb Metamodel Over Time

OCL constraints defined on the initial version of the AlCoWeb metamodel allow to ensure the respect of common component model rules and those specific to AlCoWeb models. Here are some examples of OCL constraints attached to the initial AlCoWeb-v1 metamodel:

```
--(1) Provided operations must have
-- a public visibility
context: Port
inv: self.provided->forAll(o| o.visibility='public')

--(2) An envelope should not contain components
-- with the same name
context: Envelop
inv: self.component->forAll(c1,c2|c1 <> c2
     implies c1.name <> c2.name)

--(3) A port has either provided operations,
-- or required operations, but not both
context: Port
inv: self.required->notEmpty() implies
     self.provided->isEmpty()
     and
     self.provided->notEmpty()
     implies self.required->isEmpty()
```

Here, below, an example of changes that were made to get the latest version of the AlCoWeb metamodel:

(i) Transform the association between metaclasses *Envelop* and *Component* in an association between metaclasses *Envelop* and *AbstractComponent*.

(ii) From the association between metaclasses *Port* and *operation*, introduce a new metaclass (*RequiredInterface*),

(iii) From the association between metaclasses *Port* and *process*, introduce a new metaclass (*ProvidedInterface*),

(iv) Extraction of a super metaclass named *InstanceSpecification* from the metaclasses (*AbstractComponent*, *Connector* and *Port*),

(v) Pull-up attribute *stereotype*, from *AbstractComponent*, to the previously created metaclass *InstanceSpecification*,

(vi) Eliminate a *stereotype* property from the metaclasses *Port* and *Connector*,

The set of consistency rules defined on the AlCoWeb metamodel is about 30 OCL constraints. For the company, identifying and transforming manually each constraint affected by the evolutions of the metamodel is a tedious and time consuming task. Although the number of constraints is not particularly huge, the complexity of some constraints makes them difficult to maintain. The manual maintenance of constraints is an error-prone task, which could even lead maintainers to miss some modifications. For example, consider the constraint (3) attached to the initial metamodel. The designer in charge of the evolution of the metamodel, and its constraints, unfortunately made a mistake. In this particular example, the adaptation of the expressions *self.provided->notEmpty()* and *self.required->notEmpty()* had been forgotten. The constraint is still syntactically valid on the new version of the AlCoWeb metamodel. Indeed, the *provided* and *required* properties still exist. But unfortunately, the meaning of the constraint has changed and its application requirements are not satisfied anymore. The problem was detected during the construction of a model and after several attempts to circumvent it. At the end, it became necessary to return to the metamodel and to check the consistency of its constraints.

Such mistakes often occur, and are very time consuming for the maintainers. That is why the evolution of constraints have to be automated as much as possible.

III. Approach

Our approach is to provide i) a set of basic operations to modify a metamodel, ii) assistance in order to adapt constraints according to changes operated on their associated metamodel. These basic operations are a synthesis made from those proposed in [8], [9], [6], [10], [11].

In our approach, a metamodel evolution consists of a sequence of basic operations. Thus, after each basic operation we check the status of all constraints in order to identify those which are impacted. For each impacted constraint, we can either just point out that it becomes inconsistent, or propose an adaptation in accordance with the performed operation. In all cases, an inconsistent constraint is not removed until the evolution is not completed and that the designer did not validate its withdrawal. In other words, the set of constraints may be inconsistent in the middle of an evolution, but it must be consistent at the end.

A constraint is said to be consistent if its syntax is correct (accepted by an OCL compiler) and its semantic is in agreement with the intent of the designer. If the designer does

Constraint ID	Constraint Context	Involved Elements	Valid	Adaptation
(1)	Port	(Port,provided,Ref), **(provided,visibility,Prop)**	No	No
(1')	Port	(Port,provided,Ref), (provided,providedOp,Ref), (providedOp,visibility,Prop)	No	Yes
(2)	Envelop	(Envelop,Component,Ref), (Component,name,Prop)	Yes	No
	...	...	...	...

TABLE I
INTERMEDIATE CONSTRAINT STATE.

not change her/his intent (as in a case of refactoring), the adapted constraints preserve the semantics according to the criteria proposed by Markovic [9]. But, since we can not know whether the designer has changed or not her/his intent, adapted constraints must be validated by the latter.

Our approach is based on the one hand, on the ability to detect that a constraint becomes inconsistent and on the other hand on the ability to propose an adaptation. For this, we manage the state of the constraints associated with a metamodel through the following data:

- Constraint ID: identifier of the constraint in the meta-model.
- Constraint Context: name of the meta-class representing the context of the constraint.
- Involved Elements: involved elements are shown in pairs in order to represent the link's type that attaches them.
- Valid: defines the state of a constraint. The constraint is said to be valid when it is syntactically correct and that the designer has confirmed it.
- Adaptation: to express that the constraint is an adaptation of another, which has become inconsistent, and which requires to be validated by the designer.

Before any change constraints are all considered valid. After each change on the metamodel the table is updated, but it will be subject to validation by the designer, only at the end of evolution.

Table I shows the constraint state after a change in the metamodel. This change was to create a new meta-class *Interface* between meta-classes *Port* and *Operation*.

The value of attribute 'Valid' for constraint (1) is 'No' to indicate that it was affected by the change. The cause of the invalidity is the loss of a relationship that existed between two elements involved in the constraint (relationship in bold in the table). The identifier of the constraint (1'), by itself, indicates that its origin is the constraint (1). The 'Yes' value for its 'Adaptation' attribute indicates that it is a coherent adaptation. The 'No' value for its 'Valid' attribute means that it has not yet been validated by the designer. At the end of evolution, adaptation will be proposed to the designer. If it is validated, as it is or with some corrections, it will permanently replace the constraint (1) (i.e. it takes its identifier and 'Yes' for 'Valid' and 'No' for 'Adaptation'). If not, the constraint (1') will be removed and the designer will have to take a decision on constraint (1): either remove it or modify it and validate it. The designer is also required to take a decision on constraints that become invalid and for which no adaptation was proposed. Thus, after an evolution all constraints of the metamodel become consistent again.

In the next section, we show how to build an adaptation to a constraint affected by a change in the metamodel.

IV. CONSTRAINTS ADAPTATION

We have already seen that in some cases the assistance system proposes an adaptation to constraints impacted by a change in the metamodel. For this aim we have identified operations for which it is possible to propose the way to adapt the impacted constraints. Thus, we found very interesting the use of QVT Relation in order to define not only the different possible operations on the metamodel, but also transformations they induce on the involved constraints.

A. Concerned Operations

We did not define rules for all possible operations on a metamodel. Indeed, some of them have already been done by other works. Thus, we have reused results from mainly two works: the first one concerns the metamodel and models coevolution [6] and the second one concerns UML models and OCL constraints corefactoring [9]. From the first work, we reused the QVT Relation rules defined for metamodel adaptation and from the second one, we reused and adapted QVT Relation rules defined for constraint adaptation. Indeed, for the latter some rules require adaptation because they were written for the model level and could not be applied as such onto the metamodel level. Moreover, rules that were defined in [9] do not cover all possible operations on a metamodel, because the context was refactoring. Table II summarizes the different operations of refactoring, creation and suppression that may occur during the evolution of a metamodel.

The meaning of the different columns of the table is as follows:

- *Operations:* lists the different types of evolution operations;
- *Influence:* indicates if the operation may affect the syntax of OCL constraints associated to the metamodel;
- *QVT for MOF:* indicates whether an operation has been formalized in the literature, using QVT Relation at the MOF level;
- *QVT for UML/OCL:* indicates whether an adaptation of OCL constraints was formalized using QVT Relation, in order to fit a refactoring of their associated UML class diagram;
- *QVT for MOF/OCL:* indicates if, for a given operation, some work deals with the coevolution of metamodels and their constraints.

Operations presented in the first column come from [6] and were inspired by work on refactoring [8], [11], work on grammar adaptation [12], and also on [10] which identifies concrete operations for metamodel evolution. The second column highlights operations that have an influence on the syntax of constraint. The "Yes" with a white background are those which have already been identified by [9]. Those whose

Operations	Influence	QVT for MOF	QVT for UML/OCL	QVT for MOF/OCL
RenameElement	Yes	Yes	Yes	Yes
MoveProperty	Yes	Yes	Yes	Yes
ExtractClass	No	Yes	Yes	Yes
InlineClass	No	No	No	No
ExtractSuperClass	No	Yes	No	No
FlattenHierarchy	No	No	No	No
Introduce property	No	Yes	No	No
eliminate property	Yes	Yes	No	No
PullUpProperty	Yes	Yes	No	No
PushDownProperty	Yes	Yes	No	No
AssociationToClass	Yes	Yes	No	No
ClassToAssociation	Yes	No	No	No
IntroduceClass (in a package)	No	Yes	No	No
EliminateClass (in a package)	No	No	No	No
IntroduceProperty	No	Yes	No	No
GeneralizeProperty	Yes	Yes	No	No
RestrictProperty	No	No	No	No
Eliminate property	No	No	No	No
InheritanceToCompostion	Yes	No	No	No

TABLE II
SYNTHESIS OF MOF/OCL ADAPTATION RULES

background is gray are operations that we believe have an influence on the constraint and we will deal with in the following. An operation with a "Yes" in the third column means that its formalization in the QVT already exists for the meta-metamodel MOF. These formalizations are from [6]. Thus, we propose to formalize, using QVT, operations with a "No" value in the table. In the fourth column we have identified operations for which rules for OCL constraint adaptation were formalized in [9] to fit with UML class diagram refactoring. Finally, the "Yes" set in the last column of the table corresponds to operations resulting from [9] that can be easily extended to the MOF meta-metamodel.
The rows whose value is "No" at the last column and "Yes" at the second column (the gray lines in the table) correspond to operations for which we propose a formalization with QVT in order to adapt OCL constraints to the corresponding metamodel evolution.

B. PullUpProperty *Operation*

In the MOF meta-metamodel, a *property* refers to either an *attribute* or an *association end*. Pulling a property into a super-class is a well known object-oriented refactoring [11], [8]. Figure 2 shows an example of a *PullUpProperty* operation. *MM* represents a part of a metamodel concerned with the *PullUpProperty* operation. On the other side, *MM'* represents the evolution of *MM* after application of the operation.

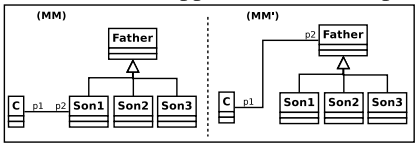

Fig. 2. *PullUpProperty* Operation Example

In [13], authors proposed a formalization of this operation in the context of UML class diagrams refactoring. As both the UML metamodel and the MOF meta-metamodel are conceptually close, QVT rules defined on the UML metamodel can be easily adapted to the MOF meta-metamodel as shown in Figure 3.

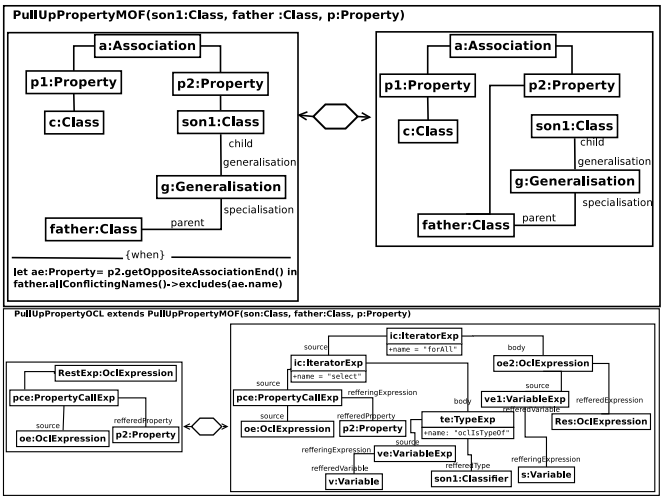

Fig. 3. QVT Rules for *PullUpProperty* Operation

Figure 3 is composed of an upper part, which represents the transformation rule corresponding to the operation applied on the metamodel and a bottom part, which represents the transformation rule to apply on constraints impacted by the operation. The upper part is itself divided into two parts: the left hand side (LHS) represents the pattern on which the operation can be applied. And if an instance of this pattern is found in the current metamodel, it will be transformed so that it matches to the pattern given in the right hand side (RHS). If the operation is completed, the constraint transformation rule (bottom part) will be applied. Thus, the constraints that contain parts that match with the LHS pattern will be considered as impacted by the operation. Therefore the proposed adaptation will be built thanks to the RHS pattern.

Sometimes, some conditions are associated with transformation rules. For example, the "when" clause of the rule from Figure 3, states that it should not already exist an association of the same name between *C* and *Father* metaclasses.

In [13], authors did not propose a rule to deal with the OCL constraints attached to UML class diagrams in case of *PullUpProperty*. This is true if constraints that use this property have to be applied on all the other son of the metaclass *Father*. But sometimes, the need is to pull up only the property and leave constraints applying only on the metaclass *Son1*. Thus, in our metamodel evolution assistance system, we propose two distinct operations: *FullPullUpProperty* for the first case and *PartialPullUpProperty* for the second case. In case of *FullPullUpProperty* operation constraints remain consistent, while in case of *PartialPullUpProperty* operation constraints are impacted. In the latter case we propose the following.

Impacted constraints are of the form:

oclExpression.p_2*[RestExp]*
so we propose to transform the constraint as follows:

oclExpression.p_2 → ***select(v | v.oclIsTypeOf(Son1))*** → ***forAll(s | s[RestExp])***

Thus, the transformed constraint will apply only on instances of the meta-class. This transformation is formalized with QVT in the bottom part of Figure 3. In *RHS* of the QVT rule we add an instance of metaclass *PropertyCallExpression* which introduces the *select* operator and an instance of *TypeExp* to introduce the *oclIsTypeOf*.

C. PushDownProperty *Operation*

PushDownProperty is a refactoring operation [8], [11]. This operation moves an attribute from the parent to a selected subclass, as it is described in Figure 4. In this example, we consider the property value equal to an attribute.

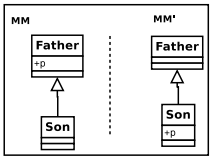

Fig. 4. *PushDownProperty* Operation Example

The formalization of *PushDownProperty* operation to MOF metamodels is given in the upper part of Figure 5. This formalization was initially given in [13], but Authors did not propose to transform the constraints which have *oclExpression.p*, such as *oclExpression* is type of *Father*. To check for non-existence of constraints attached to the property *p* whose contextual metaclass is *Father*, they added a condition into the *When-clause* of QVT rule.

To address these constraints, we have removed this condition, and we keep only the condition that ensures that the property *p* does not exist in metaclass Son before applying the operation. Moreover, contrary to rule given in [13], we consider the constraints of the form of *oclExpression.p[RestExp]* to take into account the rest of the constraint. As stated earlier, *oclExpression* is type of *Father*. So, OCL constraints with expression ***oclExpression****.p[RestExp]* will be transformed into:

oclExpression →***select(v|*** ***v.oclIsTypeOf(Son))***
→***forAll(***v_1***|*** v_1***.p[RestExp])***

Thus, the constraint will apply on all instances of the metaclass *Son*.

The bottom part of Figure 5 formalizes the adaptation to be applied on impacted constraints after a *PushDownProperty* operation on a metamodel.

D. AssociationToClass *Operation*

The *AssociationToClass* operation is a kind of generic refactoring [14]. It consists of extracting a metaclass from an existing association between two metaclasses, and to affect 1-1 cardinalities to their containers as shown in Figure 6.

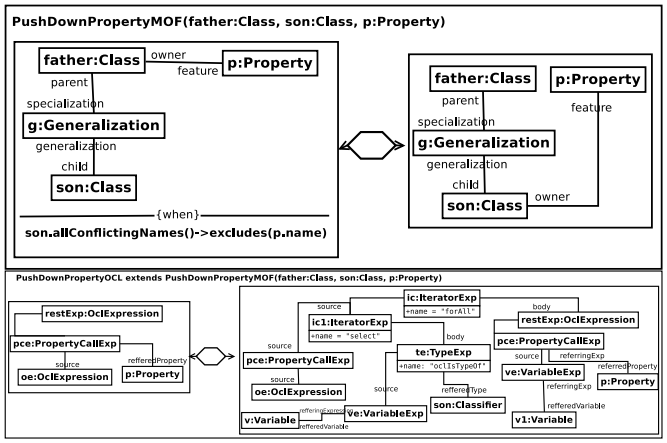

Fig. 5. QVT Rules for *PushDownProperty* Operation

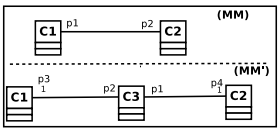

Fig. 6. *AssociationToClass* Operation Example

The upper part of Figure 7 represents the QVT formalization of the *AssociationToClass* operation for MOF meta-metamodel that was proposed in [6].

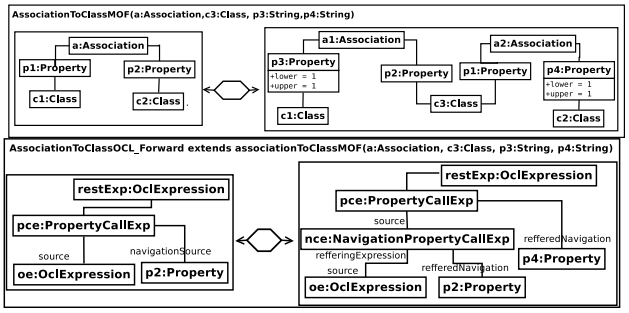

Fig. 7. QVT Rules for *AssociationToClass* Operation

Applying an *AssociationToClass* operation on a metamodel may impact the attached OCL constraints. These can be defined either in the context of C1 or either in the context of C2. Suppose a constraint defined in the context of C1, with the following form:

oclExpression.p_2*[RestExp]*

Such a constraint is syntactically correct after the metamodel evolution, but its meaning has completely changed. Indeed, the constraint will no longer be evaluated on the instances of C2 but on the instances of C3. To solve this problem we propose the following transformation whose formalization is given in the bottom part of Figure 7:

oclExpression.p_2***.***p_4***[RestExp]***

In the new form of the constraint, we added a simple navigation through the property *p4*. In the case of constraints defined in the context C2, the problem is symmetric to that raised by the constraints defined in the context C1. The solution is also symmetric. The formalization of this solution is, of course, written for the assistance system, but it provides

no new information to the reader.

E. ClassToAssociation *Operation*

The *ClassToAssociation* operation is the inverse operation of the *AssociationToClass* operation. It consists of removing a metaclass related with two other metaclasses, and whose associations have a 1-1 cardinality. Figure 6 describes this evolution from *MM'* to *MM*. We propose a formalization of this operation in QVT at the MOF meta-metamodel level, as shown in the upper part of Figure 8. The *ClassToAssociation* operation need three metaclasses and two associations as parameters. To apply this operation on a metamodel, it requires the property cardinality to be 1-1 on both sides of the removed class and there is no other association with the same properties in the two remaining classes. These two conditions are expressed through the *When*-clause of the QVT rule.

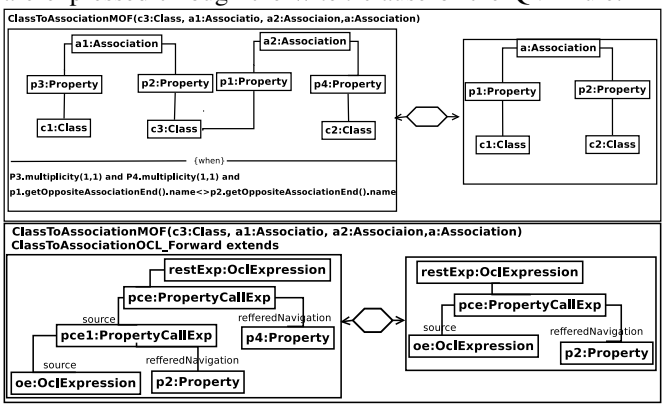

Fig. 8. QVT Rules for *ClassToAssociation* Operation

When *ClassToAssociation* operation is applied the constraints attached to the involved metaclasses may be affected. Thus, we propose the transformation of these constraints by using the rule defined in the bottom part of Figure 8. In other words, a constraint in the form ***oclExpression***$.p_2.p_4$*[RestExp]*, such as *oclExpression* is of type *C1*, will be transformed into ***oclExpression.***p_2***[RestExp]***. Thus the property p_4 has been removed.

We also defined a transformation rule for the case where the type of *oclExpression* is *C1*. As with the previous operation, this rule is symmetrical to the one shown in Figure 8. Thus, we do not describe it in this paper.

F. GeneralizeProperty *Operation*

A property can be generalized or restricted in terms of its multiplicity and its type. The case of restriction does not lead to constraint transformation. This will be discussed at the end of this sub-section. *GeneralizeProperty* operation consists in generalizing the cardinality of an *association end* so that the old cardinality is included in the new one. An example of such operation is illustrated in Figure 9. The lower bound of the cardinality remains unchanged while the upper bound is affected by the generalization.

In [6], the author proposes a QVT formalization of this rule at the MOF meta-metamodel level. This rule is described in the upper part of Figure 10.

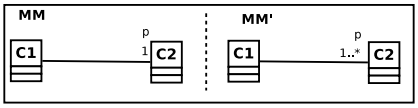

Fig. 9. *GeneralizeProperty* Operation Example

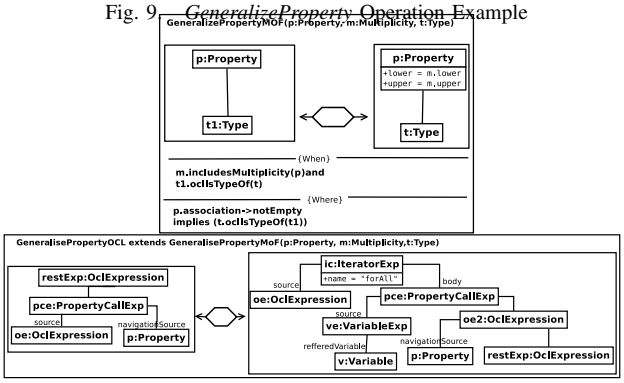

Fig. 10. QVT Rules for *GeneralizeProperty* Operation

This kind of operation implies some modifications on OCL constraints attached to the metamodel. Suppose a constraint defined with an expression whose form is:

oclExpression$.p$*[Rest]*, where *oclExpression* is of type *C1*.

After the generalization of the cardinalities, instances of the metaclass *C1* can be linked to several instances of the metaclass *C2*. To apply this constraint on all the instances of the metaclass *C2*, it is necessary to transform it by adding the OCL collection operator *forAll*. Thus, the previous expression is rewritten as follows:

oclExpression*$\rightarrow$*forAll(v | v.p[RestExp]).

The QVT formalization of the adaptation on the OCL metamodel is described in Figure 10.

The *RestrictProperty* operation is the inverse operation of *GeneralizeProperty*. This operation has no impact on the constraint, because by definition in the OCL language, a collection operator can be applied to either a set full of elements or to a set with only one element. Thus, it is not necessary to transform the constraint. This can be dealt with in the case of constraint optimization.

G. InheritanceToComposition *Operation*

The *InheritanceToComposition* Operation consists in the conversion of an inheritance relationship into a composition relationship as shown in Figure 11. This is a refactoring operation described in [11] which is usually performed in the context of metamodels evolution [10].

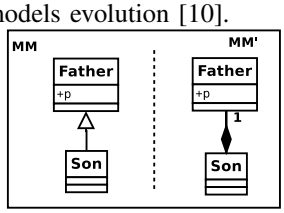

Fig. 11. *InheritanceToComposition* Operation example

The upper part of Figure 12 shows the formalization of the *InheritanceToComposition* operation at the MOF meta-metamodel level that we propose. This operation needs three

parameters: a super-class, a sub-class, and the inheritance relationship between them. The inheritance relationship is transformed as a composition relationship: two properties are created with a 1-1 cardinality. If another cardinality associated with the *Son* metaclass is needed, we can use the *GenarlizeProperty* operation.

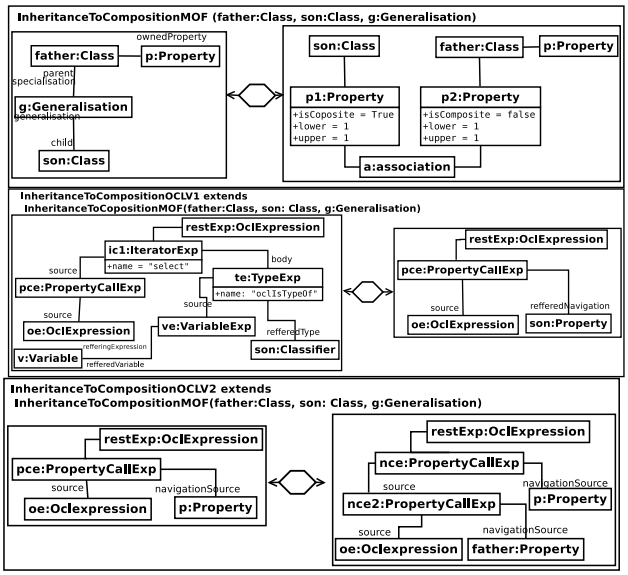

Fig. 12. QVT Rules for *inheritanceToComposition* Operation

To explain how to adapt OCL constraints impacted by this operation, we must consider separately those whose context is the metaclass *Father* and those whose context is the metaclass *Son*. In the case of the *Father* context, impacted constraints are those that conform to the following form:

oclExpression$\rightarrow$*select(v|v.oclIsTypeOf(Son))[RestExp]*

This expression allows to restrict the application of the defined constraint to instances of the *Son* metaclass. After applying the *InheritanceToComposition* operation, *Son* has not the same type than *Father* In *MM'*. Thus, we need to transform the constraint as follows:

***oclExpression.son*[RestExp]**

The formalization of this transformation is illustrated in the middle part of Figure 12.

In case of *Son* context, impacted constraints are those that conform to the following form:

oclExpression.*p[RestExp]*

It is obvious that the property *p* can not be accessed in this manner after the change resulting from the operation. What makes the constraint syntactically incorrect. Thus, we propose the following transformation:

oclExpression.father.p[RestExp]

This transformation is formalized in QVT in the bottom part of Figure 12.

H. Example of rule application

Consider the example presented in Section II. In this example we presented three constraints associated with the metamodel AlCoWeb. To evolve this metamodel from its version 1 to version 2, the following basic operations have been applied:

- PullUpProperty(Component, AbsractComponent, component)
- AssociationToClass(ownedReqOp, RequiredInterface, port, ReqInt)
- AssociationToClass(ownedProOp, ProvidedInterface, port, ProvInt)
- ExtractSuperClass(setAbstractComponent, Connector, Port, InstanceSpecification)
- PullUpProperty(AbstractComponent, InstanceSpecification, stereotype)
- EleminateProperty(Connector, stereotype)
- EleminateProperty(Port, stereotype)

This sequence of operations is considered by the designer as an evolution. So it was only at the end of this sequence of operations that the assistance system provides the designer with an adaptation for each of the three constraints.

The proposals of the assistance system are the following:

```
--(1) Provided operations must have
-- a public visibility
context: Port
inv: self.provided.providedOp->forAll(o| o.visibility='public')

--(2) An envelope should not contain components
-- with the same name
context: Envelop
inv: self.component->select(v|v.oclIsTypeOf(Component))
    ->forAll(c1,c2|c1 <> c2 implies c1.name <> c2.name)

--(3) A port has either provided operations,
-- or required operations, but not both
context: Port
inv: self.required.requiredOp->notEmpty() implies
     self.provided.providedOp->isEmpty()
     and
     self.provided.providedOp->notEmpty() implies
     self.required.requiredOp->isEmpty()
```

Even if this proposal appears to be correct, the designer does not necessarily validate them because some change may be accompanied by a change in semantics. Thus, the role of semantic validation of adapted constraints is left to the designer.

V. IMPLEMENTATION

To implement all adaptation rules related to the evolution of metamodels and their attached OCL constraints, we have developed our own tool, named METAEVOL. This tool has been designed to provide assistance to designers when they make evolve their metamodels and associated constraints. In this way, our tool consists in a set of plug-ins that contribute to the Eclipse platform, providing extensions to the standard Ecore Diagram metamodel editor and OCL editor. The architecture of this tool is inspired by an existing tool named Roclet [15]. Roclet proposes its own UML Class diagram editor and a collection of refactoring operations (operations

marked as 'Yes' in the column "QVT Rule for UML/OCL" in Table II), and when applying a refactoring operation on an UML model element, the OCL constraints are automatically adapted. Our tool works in a similar way to Roclet. However, it supports the adaptation of OCL constraints to metamodel evolution, whereas Roclet works at the model level only. The general architecture of the METAEVOL tool is depicted in Figure 13.

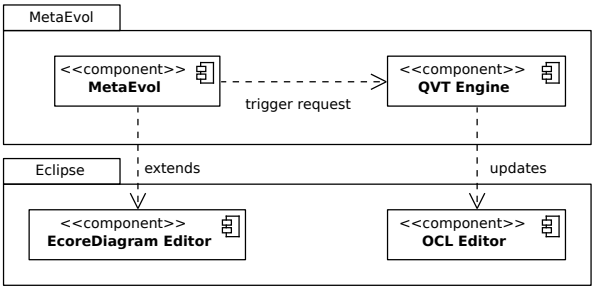

Fig. 13. Architecture of the METAEVOL tool

A. Tool Architecture

As described in Figure 13, the architecture of the tool is separated in two distinct layers. The bottom layer is composed of standard editors offered in the Eclipse platform:

- The EcoreDiagram Editor is a graphic editor used to design Ecore models. Ecore being an implementation of the EMOF part the MOF meta-metamodel, Ecore and the Eclipse EMF framework [16] allow to define metamodels. This is the tool used by designers to built and make evolve their metamodels;
- The OCL editor is also provided by the MDT-OCL project [17] of the Eclipse platform. This textual editor allows designers to describe OCL constraints attached to metamodels.

Our contribution lies in the top layer of the presented architecture. In this layer, its components work as follows:

- The METAEVOL component extends the behavior of the Ecore Diagram editor, so that it offers both features to apply the refactoring operations, and the automatic adaptation of attached OCL constraints.
- The QVT Engine is responsible of the adaptation of OCL constraints. This engine consists in the collection of the adaptation rules, implemented in QVT. Based on the set of graphical rules listed in Section IV, we have transformed each refactoring operation in a QVT textual syntax. For this, we used Together Architect 2008 [18], a tool which offers a concrete graphical syntax for QVT Relation. This tool allows to define QVT rules in a graphical way, and to generate their corresponding textual syntax. Once the additional refactoring rules have been transformed in a set of Java classes, we integrated them in our QVT Engine.

B. Adaptation of OCL Constraints

METAEVOL proposes an implementation of the approach presented in Section III. Indeed, the assisted adaptation of OCL constraints relies on a table which stores all constraints associated to a metamodel, and which also contains, for each constraint, its validity state and a field that tells if an adaptation has been proposed (see Table II in Section IV). This table allows to trace all adaptations realized by the designer during the evolution process of metamodels.

When a designer starts making evolve a metamodel with METAEVOL, the tool's first task is to initialize the table of constraints. Of course, in this step, each constraint is marked as syntactically valid, and no adaptation is yet proposed. METAEVOL also makes a copy of the initial OCL constraints. Actually, all OCL constraints transformed by the application of adaptation rules are stored in this copy. This copy and the table of OCL constraints are always kept in sync, so that they reflect the current state of modifications performed on metamodels and their constraints. For each modification performed on a metamodel, METAEVOL launches an update process. Those modifications correspond to the application of an operation of evolution (those presented in Table II), which are provided in a contextual menu of the editor. At each modification of the metamodel, the update process performs the following operations:

- A parsing of all initial OCL constraints, to tell which of them are still syntactically valid on the current state of the metamodel. This test updates the "validity" field of the table of constraints;
- If an operation of evolution (i.e., operations presented in this paper) is requested, METAEVOL will forward the request to the QVT Engine, that will then adapt OCL constraints impacted by the requested adaptation. Those adaptations are stored in the copy of the initial OCL constraints. The table of constraints is also updated to indicate if, for each constraint, an adaptation has been proposed or not by the METAEVOL tool.

Finally, when the designer has finished to make evolve a metamodel, he launches the finalization step. This action will assist the designer by providing the up-to-date table of OCL constraints. Depending on the validity state and if an adaptation have been proposed, the designer is able to decide, for each constraint, if he wants to either remove the constraint, keep the automatically adapted constraints transformed by METAEVOL, or manually modify a constraint if the proposed adaptation is not satisfactory.

Figure 14 shows the METAEVOL tool. This tool allows to manage the co-evolution of metamodels (left-side editor) and their OCL constraints (right-side editor). Metamodels adaptation rules defined in Section IV are present in a contextual menu.

VI. RELATED WORK

Our work concerns mainly two aspects of MDE, which are evolution of the metamodels and transformation of OCL constraints. Thus, this related work focuses only on these two aspects.

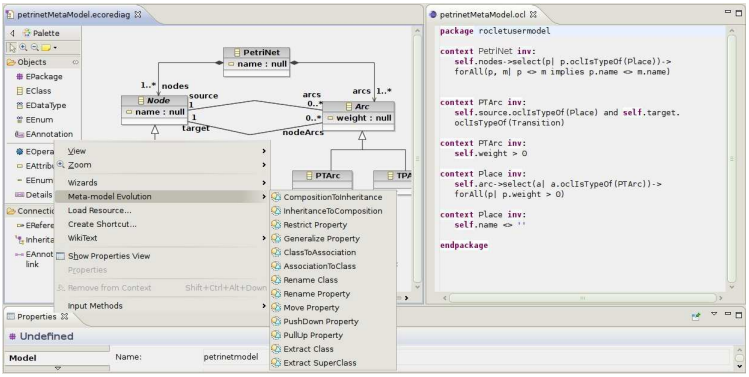

Fig. 14. METAEVOL: assisting metamodels and OCL constraints evolution

A. Constraint Transformation

In [9], the authors have presented an approach to transform OCL constraints associated with an UML class diagram in the context of refactoring. Each refactoring operation is formalized using QVT Relation. As in this work we use a stepwise adaptation approach. But in our case the operations go beyond the context of refactoring, where by definition the semantics is preserved. So we had to define the transformation rules for operations that are not allowed under refactoring context and redefine the rule for certain operations, because the assumption of semantic preservation is no longer valid.

Similarly in [19], author proposed a method, called interpretation function, for transforming constraints which allows for the extension of a mapping defined on a few model elements. This function is used first to redesign UML class diagram models. The same function is re-used to automatically translate the OCL constraints. The applicability of the approach is demonstrated in the case of refactoring UML class diagrams models. This approach can be applied at the metamodel level, since the MOF meta-metamodel was built on the UML infrastructure. However, their approach is theoretical and its implementation is not obvious.

In [20], authors proposed to improve the badly structured OCL constraints by identifying the OCL bad smells (such as *duplicate code* in the constraint). This is done for only very limited extent relationships between OCL constraints and the underlying class diagrams. In [21], authors proposed an approach to assist the designer during the definition of the constraints by means of automatically transforming the initially defined constraints into equivalent alternatives by changing the context of the constraints. The proposed approach is formalized as a path problem over a graph representing the diagram. Each path between two vertices corresponds to a different context to represent the set of constraints defined over the first vertex. In [22], authors proposed an approach to simplify the OCL constraints generated in the context of design pattern instantiation templates, i.e constraints expressing requirements associated with patterns. They identify various kinds of rules needed for OCL simplification and pointed out differences to usual term rewriting systems. All these works concern the simplification of constraints. Indeed, the simplification of constraints is complementary to our work. As we do not insure that the transformed constraint is optimal, a step for simplification may be very useful.

B. Metamodel Evolution

In [6], authors proposed a transformational approach to assist metamodel evolution by stepwise adaptation. They adopt ideas from grammar engineering and model refactoring. They presented a library of QVT Relations for the stepwise adaptation of MOF compliant metamodels. What we proposed is a complementary work by providing a library of QVT rules for automatically transform the OCL constraints attached to the evolved metamodel.

In [10] authors considered coupled model evolution based on small evolution steps. They focused on the Eclipse Modeling Framework (EMF) [16], in which ECore meta-metamodel implements a subset of MOF. In their approach, authors introduced a new language (called COPE) to support the adaptation of models with respect to metamodel updates. Our solution overlaps this solution in the sense of considering both simple and complex changes.

In [23] authors, presented an architecture to automate coupled evolution on an arbitrary software domain. They compute equivalences and differences between any pair of metamodels (e.g., representing schemas, UML models, ontologies, grammars) to derive adaptation transformations from them, and they apply these adaptations as stepwise automatic transactions on the initial metamodel to obtain the final metamodel. In our approach, adaptations are defined using QVT Relation, and not computed from traces, but the adaptation rules are done in the same way, by stepwise adaptation. In [24], authors used the Epsilon Transformation Language (ETL) to migrate models from a metamodel revision to another one. The mechanism for metamodel evolution representation is not explicitly specified, even though the problem of change detection is discussed. In fact, the authors suggest the use of changes trace as opposed to direct comparison, in order to be able to detect more complex manipulations like element movements. This approach offers an alternative way to treat the evolution of metamodel based on

traces. As our approach, authors also consider that the various adaptations performed on a metamodel are already known.

In [3] authors have presented a work to deal with the coupled transformation of metamodels and models by using high-order model transformations which take a difference model for the metamodel level as input and produce a model transformation able to co-evolve the involved models as output. In [25] authors proposed refinement of this approach, present a prototype and demonstrate its applicability in two case studies. Like our approach a transformational approach is used, unlike us, the adaptation rules are computed using metamodel differences.

VII. Conclusion and Future Work

MDE technologies promotes simpler models with a greater focus on problem space. Combined with executable semantics, this elevates the level of possible automation in the software lifecycle. But to reach this level of abstraction, developers must create and manipulate new types of artifacts (Domain Specific Languages) designed for a given company or a given project. Metamodels, as a convenient way to express the syntax of such languages, become prevalent and the heart of MDE. Thus, operations related to them such as the management of the impact of their evolution must be as automated as possible.

In this paper we have dealt with the problem of constraints adaptation in order to be conform to the evolution of their associated metamodel. Because maintaining OCL constraints can be a tedious task, we propose to assist the developer to rewrite them after each evolution of their associated metamodels. The first aim of this assistance is to identify the impacted constraints and then to propose for each of them, if possible, a new form preserving their original semantics. We made this work in the case of stepwise adaptation of metamodels. Indeed, this way of evolution offers a coherent and well-known framework allowing to reason about the semantics of the realized evolutions. Based on previous works on metamodel evolution, we have listed the different operations, with a well known semantic, that allow the evolution of a metamodel. We described, using QVT language, rules that transform OCL constraints after each of these operations. Thus, we developed an assistance tool based on these rules in order to propose the adaptation of an impacted OCL constraint after an evolution of the metamodel. The designer can accept or reject the proposed adaptation if the intention of its evolution does not preserve the original semantics of the constraint.

As future work, we will deal with the situation where we know only the initial and final state of the metamodel. The approach we will apply is to use tools like EMFCompare [26] to identify differences between the two models, and thus deduce a possible list of stepwise operations which allow us to change the initial metamodel to reach the final metamodel. Based on these list of ordered operations, we can identify situations where an adaptation of the associated OCL constraints can be proposed. Such a result can be extended in the case of model transformation with attached constraints, provided that the models are instances of MOF or UML class diagram.

References

[1] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, p. 25, 2006.

[2] OMG, *Object Constraint Language Specification, version 2.0*, Object Modeling Group, June 2005. [Online]. Available: http://www.omg.org/spec/OCL/2.0/

[3] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "Meta-model differences for supporting model co-evolution," in *Proc. of the 2nd Int. Workshop on Model-Driven Software Evolution, MoDSE 2008*, Atene (Greece), 2008.

[4] J. Favre, "Meta-model and model co-evolution within the 3d software space," in *ELISA'03: Proceedings of the International Workshop on Evolution of Large-scale Industrial Software Applications*, Amsterdam, The Netherlands, 2003, pp. 98–109.

[5] G. Deng, G. Lenz, and D. Schmidt, "Addressing domain evolution challenges in software product lines," in *Bruel, J.-M. (ed.) MoDELS 2005*, Heidelberg, 2006, pp. 247–261.

[6] G. Wachsmuth, "Metamodel adaptation and model co-adaptation," in *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07)*, ser. Lecture Notes in Computer Science, E. Ernst, Ed., vol. 4609. Springer-Verlag, jul 2007, pp. 600–624.

[7] R. Kadri, C. Tibermacine, and V. Le Gloahec, "Building the presentation-tier of rich web applications with hierarchical components," in *The 8th International Conference on Web Information Systems Engineering*, ser. Lecture Notes in Computer Science, vol. 4831S. Nancy, France: Springer-Verlag, pp. 123–134.

[8] M. Fowler, *Refactoring: Improving the Design of Existing Programs*. Boston, MA, USA: Addison-Wesley, 1999.

[9] S. Markovic and T. Baar, "Refactoring OCL annotated UML class diagrams," *Software and System Modeling*, vol. 7, no. 1, pp. 25–47, 2008.

[10] M. Herrmannsdoerfer, S. Benz, and E. Juergens, "Automatability of coupled evolution of metamodels and models in practice," in *MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 645–659.

[11] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, 1992.

[12] R. Lämmel, "Grammar adaptation," in *FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*. London, UK: Springer-Verlag, 2001, pp. 550–570.

[13] S. Markovic, "Model refactoring using transformations," Ph.D. dissertation, 2008.

[14] R. Lämmel, "Towards generic refactoring," in *Proc. of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02*, Pittsburgh, USA, 2002.

[15] RoclET Team, "Roclet project," http://www.roclet.org/, 2007.

[16] Eclipse, "Eclipse Modeling Framework (EMF)," http://www.eclipse.org/modeling/emf/, 2009.

[17] ——, "MDT OCL Project," http://www.eclipse.org/modeling/mdt/, 2007.

[18] Borland, "Together technologies," www.borland.com/together/, 2008.

[19] P. Kosiuczenko, "Redesign of UML class diagrams: a formal approach," *Software and System Modeling*, vol. 8, no. 2, pp. 165–183, 2009.

[20] A. L. Correa and C. M. L. Werner, "Applying Refactoring Techniques to UML/OCL Models," in *UML*. Springer-Verlag, 2004, pp. 173–187.

[21] J. Cabot and E. Teniente, "Transforming OCL constraints: a context change approach," in *SAC*, 2006, pp. 1196–1201.

[22] M. Giese and D. Larsson, "Simplifying Transformations of OCL Constraints," in *MoDELS*, 2005, pp. 309–323.

[23] S. Vermolen and E. Visser, "Heterogeneous coupled evolution of software languages," in *MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 630–644.

[24] B. Gruschko, D. S. Kolovos, and R. F. Paige, "Towards synchronizing models with evolving metamodels," in *in Proc. Workshop on Model-Driven Software Evolution (MODSE), 11th European Conference on Software Maintenance and Reengineering*, Amsterdam, the Netherland, 2007.

[25] K. Garcés, F. Jouault, P. Cointe, and J. Bézivin, "Managing model adaptation by precise detection of metamodel changes," in *ECMDA-FA '09: Proceedings of the 5th European Conference on Model Driven*

Architecture - Foundations and Applications. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 34–49.

[26] A. Toulmé, "Presentation of EMF Compare Utility," in *In 10th Eclipse Modeling Symposium*, 2006.