Systematic Co-Evolution of OCL Expressions

Angelika Kusel[1] **Juergen Etzlstorfer**[2] **Elisabeth Kapsammer**[1] **Werner Retschitzegger**[1]
Johannes Schoenboeck[3] **Wieland Schwinger**[1] **Manuel Wimmer**[2]

[1] Johannes Kepler University Linz, Austria
[firstname].[lastname]@jku.at

[2] Vienna University of Technology, Austria
[lastname]@big.tuwien.ac.at

[3] University of Applied Sciences Upper Austria, Campus Hagenberg, Austria
[firstname.lastname]@fh-hagenberg.at

Abstract

Metamodels are the central artifacts in Model-Driven Engineering and like any other software artifact, subject to constant change. This fact necessitates the co-evolution of dependent artifacts such as models and transformations to resolve induced inconsistencies. While the co-evolution of models has been extensively studied, the co-evolution of transformations and especially OCL expressions being a substantial part thereof have been less examined up to now. To fill this gap, this paper proposes resolution actions for all atomic metamodel changes violating the syntactical correctness of OCL expressions, thus, being able to resolve induced inconsistencies. Thereby, the resolution actions establish a virtual view on the evolved metamodel such that syntactical correctness is re-established. To verify the semantical correctness of the resolution actions, we use our PaMoMo language, allowing to specify semantical correctness requirements for model transformations. Finally, to demonstrate the applicability of our approach, a proof-of-concept prototype on basis of ATL is provided.

1 Introduction

Model-Driven Engineering (MDE) proposes the use of models to conduct software development on a higher level of abstraction [2]. Thereby, model transformations play a vital role for systematic transformations of models conforming to different metamodels (MMs), which describe the syntactical constraints for models, i.e., the abstract syntax thereof. Like any other software artifact, MMs are subject to constant change, i.e., they evolve, caused by, e.g, changing requirements. During evolution, the conformance between the MM and the dependent artifacts may be violated, which demands for a co-evolution of the dependent artifacts to resolve induced inconsistencies.

While the automated co-evolution of models has been subject to extensive research in the past (cf. [12] for a survey), the automated co-evolution of transformations has been less examined so far. Although first works exists (cf., e. g., [6, 8, 9, 16]), especially the co-evolution of Object Constraint Language (OCL) [25] expressions has not been a major focus up to now, although OCL expressions make up substantial parts of model transformations [29]. The indispensable role of OCL stems from the fact that OCL expressions allow to perform complex queries on the input models, which are essential, since the results thereof are used in two important parts, namely in assignments, e.g., to produce the target model, and in steering the control flow, e.g., in case of conditions. Consequently, they represent a significant ingredient in rule-based model transformation languages, such as ATL [14] or QVT [24].

To enable the co-evolution of OCL expressions, this paper proposes *resolution actions* for all atomic MM changes violating the syntactical correctness of OCL expressions. Thus, this paper continues the work described in Kusel et al. [17], where we proposed a *complete and minimal set of changes*, which has been systematically derived from Ecore[1], thereby enabling the definition of arbitrary evolutions of Ecore-based MMs. This set of changes has been further *analyzed* with regard to its *impacts* on OCL expressions, thereby dividing the set of changes into those breaking the syntactical correctness of OCL expressions and those that do not. For breaking changes resolution actions are presented in this paper, establishing a virtual view on the evolved MM simulating the old version of the MM to the transformation definition, such that syntactical correctness is re-established. Whenever an automatic resolution is not possible, the user may specify appropriate resolution actions being supported by respective templates. The proposed resolution actions ensure both, *syntactical correctness*, which can be statically checked by a compiler, as well as *semantical correctness* by preserving the old version of target model as far as possible, which we verify by dedicated properties expressed in our "Pattern-based Modeling Language for Model Transformations" (PaMoMo) [10]. Finally, to provide a proof-of-concept, the proposed resolution actions are implemented by means of Atlas Transformation Language (ATL) [14] helpers, demonstrated on basis of a running example.

The paper is structured as follows: Section 2 briefly introduces model transformations and the role of OCL, while in Section 3 MM evolution and its impacts on OCL are discussed. Resolution actions for breaking changes are proposed in Section 4, while the formalism to check their semantical correctness is presented in Section 5. Lessons learned are discussed in Section 6, before related work is surveyed in Section 7. Section 8 concludes the paper.

2 Role of OCL in Model Transformations

This section shortly introduces model transformations in general as well as the role and importance of OCL expressions therein in particular. Although OCL might also be used in other contexts, e.g., to specify MM constraints [4] restricting the instantiability of the MM, we focus on the co-evolution of OCL in model transformations. Nevertheless, this work might also be applied to other application contexts.

[1] http://www.eclipse.org/modeling/emf

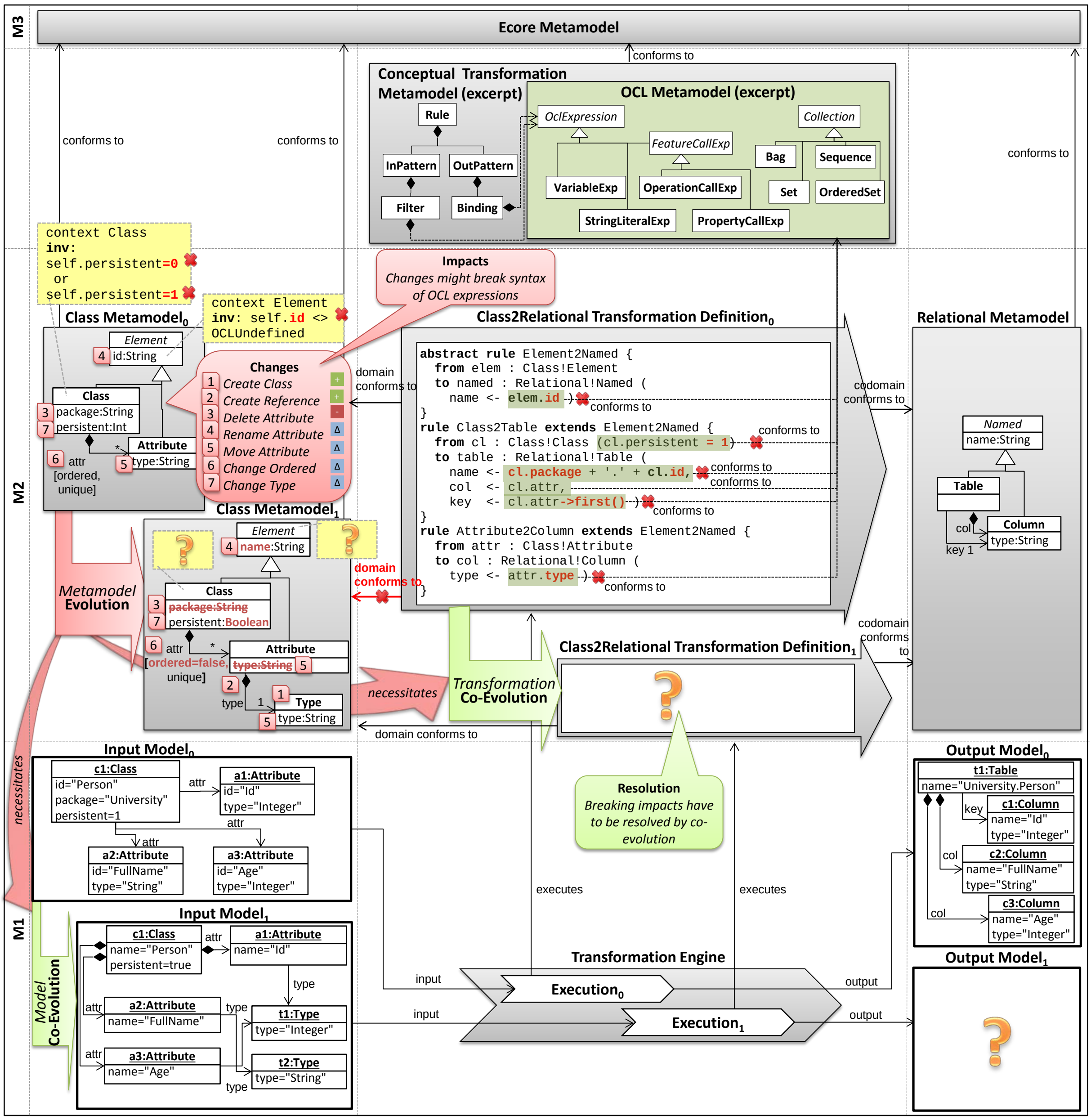

Figure 1: Metamodel Evolution and its Impact on OCL Expressions

2.1 Model Transformations in a Nutshell

Model transformations aim at transforming source models conforming to a source MM to target models conforming to a target MM, whereby both MMs conform themselves to a meta-metamodel, e.g., Ecore, being the Eclipse realization of MOF [23]. Consequently, transformation definitions realizing model transformations must conform to these two MMs in addition to the transformation MM including the OCL MM, which specifies the syntax of transformation definitions. Syntactical correctness of a model transformation may be checked by an according compiler, while the semantical correctness may be verified by given requirements the transformation has to fulfill. To define such requirements, our PaMoMo language may be utilized (cf. Sect. 5). To give an example, Figure 1 provides a small excerpt of the well-known Class2Relational transformation[2]. Please note that despite its simplicity, it is nevertheless able to serve as a running example throughout this paper.

[2]For a complete example see http://www.eclipse.org/atl/atlTransformations/

The given transformation definition states that persistent classes should be transformed into tables and attributes into columns by two dedicated transformation rules. In order to avoid code duplication, common transformation parts have been extracted to a separate base-rule Element2Named, containing the assignment of ids of elements. The lower part of Figure 1 shows a concrete model describing a persistent class named Person comprising three attributes, i.e., Id and Age of type Integer as well as Fullname of type String. This model serves as input for the transformation engine, which outputs a model, comprising a Table with name University.Person and three columns resulting from the attributes.

2.2 Role and Importance of OCL

OCL expressions as part of our exemplary transformation definition are highlighted in Figure 1, exhibiting the role and importance of OCL. From this, one may recognize that OCL expressions are used in two indispensable roles [16]. First, OCL is used in *bindings* to query source model elements, which are used to produce the target model (cf., e.g., “cl.package+’.’+cl.id” calculating the values for the target attribute Table.name). Second, OCL is utilized in *conditions* to steer the control flow (cf., e.g., “cl.persistent=1” to transform persistent classes, only). Thus, OCL expressions constitute large parts of transformations [29] and by this play an essential role.

As already stated above, OCL may also be used to constrain the instantiability of MMs. The exemplary constraints shown in Figure 1 state that for the attribute Element.id a value must be set as well as that the value of the attribute Class.persistent must be either 0 or 1.

Although transformations must conform to three MMs, OCL expressions depend on the source MM by means of a so-called “domain conforms to”-relationship [21] and the OCL MM by means of a “conforms to”-relationship [14], only. This is, since OCL expressions are used to query source models and do not refer to the target MM. Thus, this paper focuses on the evolution of the source MM and the co-evolution of OCL expressions.

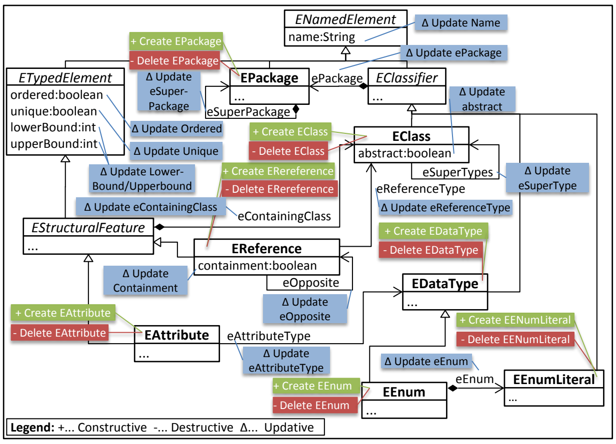

Figure 2: Set of Atomic Changes derived from Ecore

3 Metamodel Evolution and its Impacts on OCL Expressions

This section discusses the challenge of MM evolution along with a summary of arising impacts on OCL expressions. For details, the interested reader is referred to Kusel et al. [17], where we provide an in-depth analysis of this topic.

3.1 Metamodel Evolution

Like any other software artifact, MMs are subject to constant evolution, e.g., due to needs for (i) *adaptation* caused by changing software environments, (ii) *perfection* induced by user requirements, or (iii) *correction* because of errors [19]. A particular MM evolution may be described by dedicated changes that lead to a new version of the MM.

To exemplify this, an evolution of the running example is shown in the left part of Figure 1. One may see that a new version of the source MM has been created by seven dedicated changes. Thereby, a new class Type has been created (cf. ① in Fig. 1) as well as a new reference Attribute.type (cf. ② in Fig. 1) connecting the new class. Furthermore, the attribute Attribute.type has been moved to this new class (cf. ⑤ in Fig. 1)[3]. In addition to that, the attribute Class.package has been deleted (cf. ③ in Fig. 1) and the attribute Element.id has been renamed to Element.name (cf. ④ in Fig. 1). Finally, the reference Class.attr has been set to be unordered (cf. ⑥ in Fig. 1) and the type of the attribute Class.persistent has been changed from Int to Boolean (cf. ⑦ in Fig. 1).

When analyzing the *impacts of these changes* on the occurring OCL expressions, one might recognize that all changes except the two constructive changes (cf. ① and ② in Fig. 1) have a breaking impact on the OCL syntax and thereby prevent a successful execution of the existing model transformation. Consequently, a co-evolution of the OCL expressions is needed by dedicated resolution actions. But before discussing potential resolution actions, the questions which changes may arise at all and which of them might have a breaking impact on OCL expressions are discussed subsequently.

3.2 Complete and Minimal Set of Changes

To be able to describe arbitrary evolutions and analyze their impacts on OCL expressions, a systematic set of atomic changes has been derived from Ecore [17] which is shortly summarized in the following. This set of changes fulfills two criteria – *completeness* to allow for any possible change and *minimality* to avoid the analysis of overlapping changes as it may be the case for composite changes. For deriving this set of changes, we referred to the Ecore meta-MM. Thereby, all potential *constructive* and *destructive* changes have been derived by resorting to all concrete meta-classes, e.g., EClass. In addition to that all potential *updative* changes have been derived by referring to all meta-features, e.g., EClass.abstract. Figure 2 shows the resulting set of atomic changes.

This set of changes has been analyzed according to its effects with respect to *structural complexity*, i.e., the number of instantiable types, and *information capacity*, i.e., the potential number of valid model instances, since these two criteria are significant for the impacts on OCL expressions as well as subsequently for resolution. Thereby, changes affecting structural complexity indicate impacts in *accessing MM elements* in OCL expressions and have been evaluated by counting the number of all instantiable types according to [27]. In contrast, changes concerning information capacity indicate impacts on the *result set* of OCL expressions and have been evaluated by counting the potential number of all valid instances of a MM following [22]. These two criteria are able to partition the set of updative changes into six groups according to their behavior, which are used to analyze impacts and resolution actions in the following. The resulting partitions comprising ① *renaming updates*, ② *moving updates*, ③ *relaxing updates*, ④ *restricting updates*, ⑤ *constructive updates*, and finally, ⑥ *destructive updates* may be found in Table 2.

3.3 Impacts on OCL

In order to know, which atomic changes of the systematic set of changes require resolution, the impacts of changes on OCL expressions are summarized in this section. Thereby, we distinguish between *non-breaking* changes, i.e., those not affecting the syntactical correctness of OCL expressions, and *breaking* changes. Please note that the evaluation assumes that changed MM elements have been used by at least one OCL expression and the worst case scenario is considered, i.e., changes are evaluated as breaking, if there exists at least one case that breaks the syntactical correctness of the OCL expression. The results of the evaluation are presented in Table 2.

[3] Although composite changes are not part of this work, this sequence of changes represents the composite change “Extract Class” [7].

Ecore Meta-Feature		OCL Type				
		Scalar Type	Collection			
			Bag	Sequence	Set	OrderedSet
lowerBound		no impact on OCL type				
upperBound = 1	unique/ordered not applicable	✓				
upperBound > 1	unique = false and ordered = false		✓			
	unique = false and ordered = true			✓		
	unique = true and ordered = false				✓	
	unique = true and ordered = true					✓

Table 1: Bounds of ETypedElements with their Effect on OCL Types

3.3.1 Constructive/Destructive Changes

Constructive changes do not impact OCL expressions, since newly created elements can not have been referred to. In contrast, destructive changes always have breaking impact on OCL expressions, since having a destructive effect on the structure.

3.3.2 Updative Changes

In the following, updative changes are evaluated on basis of the introduced groups.

Group ① Renaming Updates: Renames are always breaking, since renamed elements are no longer accessible under their original name.

Group ② Moving Updates: Moving Updates are always breaking, since moves change the structure of instances by changing the position of elements.

Group ③ Relaxing Updates: Although relaxing updates leave the structural complexity unaffected, and relax the instantiability of the MM, i.e., increase the information capacity, only, they may nevertheless break the syntax of OCL expressions, since they are able to change the underlying OCL datatype and by this, change the set of valid operations (cf. Table 1). Please note that any change affecting the underlying OCL collection type may break the OCL syntax, since the OCL collection types are in not connected by inheritance relationships [3].

Group ④ Restricting Updates: Restricting updates, including the opposite cases of group ④ may break the syntactical correctness again due to changes in the underlying OCL types. Additionally, a restriction of ETypedElement.upperBound may break OCL expressions accessing elements by index, i.e., by the operation at(index).

Group ⑤ Constructive Updates: Constructive updates increase structural complexity and information capacity alike and are, thus, non-breaking with respect to syntax comparable to constructive changes.

Group ⑥ Destructive Updates: Since destructive updates decrease structural complexity and information capacity, these changes are comparable to destructive changes and thus, always breaking.

4 Resolution Actions for Breaking Changes

In this section, the purpose of resolution actions is discussed, before the conceptual approach for our resolution actions is presented. As a proof-of-concept, an exemplary realization of the approach is demonstrated by means of ATL helpers, which are comparable to methods in object-oriented programming languages. An overview of the resolution actions may be found in Table 2.

4.1 Purpose of Resolution Actions

The purpose of resolution actions is threefold. First, they must re-establish *syntactical correctness* in order to re-enable the execution of the model transformation to process the evolved instances. Second, the semantics of the original transformation should be preserved as far as possible, i.e., *semantical correctness* should be ensured by the resolution actions, which means in our case the preservation of the specified requirements the transformation has to fulfill. While the achievement of syntactical correctness may be verified automatically by a compiler, the verification of the achievement of semantical correctness is more challenging and, thus, will be discussed in detail in Section 5. Third, resolution actions must be consistent with the co-evolution of other dependent artifacts, like model co-evolution as well as semantical requirements.

4.2 Conceptual Approach

For achieving syntactical correctness as well as semantical correctness, we propose a conceptual approach that establishes a “virtual view” on source MM_1 (cf. Fig. 3), which basically maps the newly structured input models conforming to the source MM_1 such that they appear to the transformation in the original structure, i.e., conforming to the source MM_0. Consequently, this approach tries to re-establish the information contained in the original source MM_0 on basis of the information still available in the new source MM_1. Thus, differently structured information might be recovered automatically by dedicated resolution actions (cf., e.g., the information of the attribute Attribute.type might be recovered by accessing Type.type instead). However, one might recognize that deleted information can not be restored automatically and, consequently, demands for user intervention in the resolution process (e.g., the information contained in the attribute Class.package is lost as exemplified by the change “Delete Attribute” in Fig. 1).

One may see that this approach is able to ensure syntactical correctness, since the view simulating MM_0 is compatible with the original transformation definition. By this means it also ensures semantical correctness preserving the specified correctness requirements (cf. Sect. 5). The original transformation is extended by helpers, only, realizing the virtual view approach that restores all the information still available in MM_1 or demands for user intervention otherwise.

4.3 Proof-of-Concept Realization

In this section, a proof-of-concept realization of the conceptual approach is presented. Although this implementation relies on ATL, the conceptual approach is not limited to a certain transformation language.

For realizing the virtual view, basically two approaches might be followed. First, the resolution actions realizing the virtual view may be ”inlined”, i.e., the original transformation gets adapted at the corresponding positions, being closely related to *program transformation* [28]. Sec-

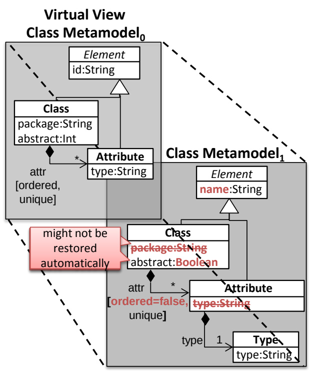

Figure 3: Exemplary Virtual View

	Change					Impact	Resolution	
	Name / Group	Meta-Feature	State Change of Meta-Feature	Structural Complexity	Information Capacity	Non-Breaking	Resolution Action	Template for ATL Helper
Constructive Changes	Create EPackage	n.a.	n.a.	+	+	✓	*not needed*	n.a.
	Create EClass					✓		
	Create EAttribute					✓		
	Create EReference					✓		
	Create EDataType					✓		
	Create EEnum					✓		
	Create EEnumLiteral					✓		
Destructive Changes	Delete EPackage	n.a.	n.a.	-	-	×	*user intervention*	Listing 2
	Delete EClass					×		
	Delete EAttribute					×		
	Delete EReference					×		
	Delete EDataType					×		
	Delete EEnum					×		
	Delete EEnumLiteral					×		
Updative Changes	① Renaming Updates	ENamedElement.name	oldName → newName	o	o	×	*rename*	Listing 4 (except for EClass and EDataType)
	② Moving Updates	EPackage.eSuperpackage	oldESuperPackage → newESuperPackage	- & +	o	×	*adapt navigation path*	Listing 6 (except for ENumLiteral)
		EClassifier.ePackage	oldEPackage → newEPackage			×		
		EStructuralFeature.eContainingClass	along reference with upperBound = 1			×		
		EENumLiteral.eEnum	oldEENum → newEEnum			×		
	③ Relaxing Updates	EClass.abstract	true → false	o	+	✓	*not needed*	n.a.
		EReference.eOpposite	remove			✓		
		ETypedElement.lowerBound	x → y, y<x			✓		
		EReference.containment	true → false			✓		
		ETypedElement.upperBound	x → y, x>1, y>x			✓		
			1 → >1			×	*cast to single valued element*	Listing 9
		ETypedElement.ordered	true → false			×	*cast to original datatype*	Listing 8
		ETypedElement.unique	true → false			×		
	④ Restricting Updates	EClass.abstract	false → true	o	-	✓	*not needed*	n.a.
		ETypedElement.lowerBound	x → y, y>x			✓		
		EReference.containment	false → true			✓		
		EReference.eOpposite	add			✓		
		ETypedElement.upperBound	x → y, y>1, y<x			×	*user intervention*	n.a. (direct adaptation)
			>1 → 1			×	*cast to collection*	Listing 10
		ETypedElement.ordered	false → true			×	*cast to original datatype*	Listing 8
		ETypedElement.unique	false → true			×		
	⑤ Constructive Updates	EClass.eSuperType	add*	+	+	✓	*not needed*	n.a.
		EStructuralFeature.eContainingClass	pull up*			✓		
		EAttribute.eAttributeType	specialize			✓		
		EReference.eReferenceType	specialize			✓		
	⑥ Destructive Updates	EClass.eSuperType	remove	-	-	×	*user intervention*	Listing 2
		EStructuralFeature.eContainingClass	push down and other cases			×		
		EAttribute.eAttributeType	generalize and other cases			×		
		EReference.eReferenceType	generalize and other cases			×		

Legend: + ... *increase* o ... *neutral* - ... *decrease* *... *decreasing effect if required features are introduced* n.a. ... *not applicable*

Table 2: Atomic Changes by Groups with their Impacts on OCL and Potential Resolution Actions

ond, the resolution actions realizing the virtual view may be implemented by the concept of ATL helpers, without modifying the original transformation definition, being related to *data transformation* [18]. Since the first approach has the disadvantage that the border between the original transformation and the resolution actions gets blurred and by this hinders a subsequent manual adaptation, we decided for an approach basing on data transformation. This allows us to keep the resolution actions separated and thus, facilitates understandability as well as maintainability by the user. For realizing the helper approach, the transformation has to be adapted marginally by adding an empty parameter list to accesses of the affected elements in order to redirect the accesses to the helpers, e.g., calling cl.package() instead of cl.package. In the following, the proposed ATL helpers realizing the resolution actions for each breaking change are presented. Thereby, the general idea behind the resolution actions is discussed, before an example as well as the generic templates for building the helpers are provided.

4.3.1 Destructive Changes

All destructive changes decrease structural complexity, thereby invalidating OCL expressions that access the deleted elements. Since the information held by the deleted elements is lost, it might not be restored automatically in the virtual view. However, a generic template is provided to support the user in the resolution process. For example, the user may decide to compensate the deletion of the attribute Class.package (cf. ③ in Fig. 1) by a substitution with an empty string, resulting in Listing 1.

Listing 1: Resolution Action for Destructive Updates

```
-- resolution action
helper context ClassD!Class def : package() : String =
    ''; -- user-defined resolution

-- transformation definition
rule Class2Table extends Element2Named {
    from cl : Class!Class (cl.persistent = 1)
    to table : Relational!Table (
        name <- cl.package() + '.' + cl.id,
        col <- cl.attr,
        key <- cl.attr->first() )
}
```

The generic template for creating the concrete helper is shown in Listing 2. Thereby, the context of the helper is set to the EClass the deleted feature was originally contained in, i.e. featureV0.eContainingClass. The name of the helper defined by "def" is the name of the original feature, i.e., featureV0.name, and also its return type is set to the original type, i.e., featureV0.type. The body of the helper provides a place-holder for a user defined resolution action.

Listing 2: Template for Destructive Changes of EStructuralFeature

```
-- template for resolution action
helper context ≪featureV0.eContainingClass≫ def :
 ≪featureV0.name≫() : ≪featureV0.type≫ =
    ≪user-defined resolution action≫;
```

Please note that the deletion of an EClass or EDataType might not be compensated with this approach in ATL, since ATL does not allow for "virtual classes", which would be needed to emulate the deleted EClass or EDataType. Furthermore, all OCL operations that operate directly with type definitions, i.e., oclAsType(T), oclIsTypeOf(T), oclIsKindOf(T), and T::allInstances(), can not be resolved automatically, and thus, require user intervention by, e.g., substituting the deleted type with another one.

4.3.2 Updative Changes

In the following, resolution actions for updative changes with breaking impact are proposed.

Group ①: Renaming Updates: Each renaming update has breaking impact on the OCL expressions that access the renamed element. Thus, a resolution action is needed, which returns the renamed element. Listing 3 shows the resolved transformation definition of change "Rename Attribute" (cf. ④ in Fig. 1) of the running example.

Listing 3: Resolution for Rename

```
-- resolution action
helper context ClassD!Element def : id() : String =
    self.name;

-- transformation definition
abstract rule Element2Named {
    from elem : Class!Element
    to named : Relational!Named (
        name <- elem.id() )
}
```

The generic template for this helper is shown in Listing 4. The context of the helper is set to its original containing class, the name of the helper is set to the original name, and the return type is also set to its original datatype. In the body of the helper the value of the renamed element is returned. Please note, that renaming of EClasses may not be resolved in ATL using this approach, since ATL does not support the definition of virtual classes, as stated above. However, renaming of EClasses may be resolved by program transformation.

Listing 4: Template for Renaming Updates

```
-- template for resolution action
helper context ≪featureV0.eContainingClass≫ def :
    ≪featureV0.name≫() : ≪featureV0.type≫ =
    self.≪featureV1.name≫;
```

Group ②: Moving Updates: While renames change the identifier under which the information may be accessed, moving updates change the position from where the information may be obtained. Consequently, moves always entail a breaking impact, while being again automatically resolvable by the virtual view without any loss of information. For realizing the virtual view, the original path has to be adapted to redirect to the new location of the moved element. The change "Move Attribute" (cf. ⑤ in Fig. 1) of our running example is thus, resolved by a helper that is responsible for this redirection (cf. Listing 5). Therefore, the new navigation path is defined in the body of the helper, starting from the element itself along the reference type to the original feature type.

Listing 5: Resolution for Moving Update

```
-- resolution action
helper context ClassD!Attribute def : type() : String
    = self.type.type;

-- transformation definition
rule Attribute2Column extends Element2Named {
    from attr : Class!Attribute
    to col : Relational!Column (
        type <- attr.type() )
}
```

The template for moving updates is shown in Listing 6. In this template, the context and return type are set to the original types, while the name of the helper is set to the name of the moved feature with an empty parameter list. In the body of the helper, the newly created reference is added in order to navigate over this reference to the moved feature.

Listing 6: Template for Moving Updates

```
-- template for resolution action
helper context ≪featureV0.eContainingClass≫ def :
    ≪featureV0.name≫() : ≪featureV0.type≫ =
    self.≪reference≫.≪featureV0.name≫;
```

In case of inlining a feature, i.e., moving it along a reference inside the class from which the feature navigated stems from, we suggest to optimize the navigation by removing this indirection. This optimization prevents vulnerability with respect to a potential deletion of the references in the future, which is, e.g., entailed in a composite change like "Inline Class".

Group ③: Relaxing Updates: Relaxing updates may have breaking impact on OCL expressions, since they might cause changes to the underlying OCL datatype as introduced before (cf. Table 1). The potential datatype changes may be classified into two cases: (i) collection type to collection type and (ii) single valued element to collection type. The virtual view approach allows to automatically resolve both cases by casting the changed OCL types back to their original types. While for casts between a collection with ordering information and a collection without ordering information, the original ordering information might have been lost during model co-evolution, for the cast from a collection type to a single valued element, no information loss occurs in case that still at most a single element is contained in the collection after model co-evolution. Consequently, although syntactical correctness might be achieved by the virtual view approach, semantical correctness in the strict sense of remaining the observable behavior is not guaranteed in both cases, since the inputs might have been changed during model co-evolution, which demands for a relaxed notion of semantical correctness as done in Section 5.

Case (i): Collection Type to Collection Type. For the change "Change Ordered" in the running example, this means to cast the type of the reference attr back to its original type, i.e., OrderedSet, which is demonstrated in Listing 7.

Listing 7: Resolution of Collection Type Change

```
-- resolution action
helper context ClassD!Class def : attr() : OrderedSet(
    Class!Attribute) =
    self.attr.asOrderedSet();

-- transformation definition
rule Class2Table extends Element2Named {
    from cl : Class!Class (cl.persistent = 1)
    to table : Relational!Table (
        name <- cl.package + '.' + cl.id,
        col <- cl.attr(),
        key <- cl.attr()->first() )
}
```

The template to generate helpers for casting the OCL collection types is shown in Listing 8. Thereby, the return type is set to the original type of the collection. In the body of the helper the corresponding cast to the original type of the feature is determined by checking the new type of the collection and applying the corresponding cast operation on this type.

Listing 8: Template for OCL Collection Type Casts

```
-- template for resolution action
helper context ≪featureV0.eContainingClass≫ def :
    ≪featureV0.name≫() : ≪featureV0.type≫ =
    <if (≪featureV0.name≫->oclIsTypeOf(OrderedSet))
      then self.≪featureV0.name≫.asOrderedSet()
```

```
    else if (≪featureV0.name≫->oclIsTypeOf(Set))
      then self.≪featureV0.name≫.asSet()
    else if (≪featureV0.name≫->oclIsTypeOf(Sequence))
      then self.≪featureV0.name≫.asSequence()
    else
        self.≪featureV0.name≫.asBag()
    endif endif endif >;
```

Case (ii): Single Valued Element to Collection. Relaxing updates also comprise the increase of the upperBound from 1 to > 1, which causes the underlying OCL type to change from T to Collection(T). As a consequence, the operations applied on elements of type T are now invalidated, since the OCL datatype changed to a collection. The corresponding resolution action is to extract a single value from the collection, which is then used in the transformation definition.

Listing 9 shows the template for an ATL helper to extract a single valued element from a collection. In the body of the helper, an arbitrary element from the collection is selected and returned.

Listing 9: Template for Collection to Single Valued Element

```
-- template for resolution action
helper context ≪featureV0.eContainingClass≫ def :
    ≪featureV0.name≫() : ≪featureV0.type≫ =
    self.≪featureV0.name≫->any();
```

Group ④: Restricting Updates: Analogous to relaxing updates, restricting updates may also have breaking impact on OCL expressions, since the underlying OCL datatypes may have changed. Again two cases might be distinguished according to the arising type change: (i) collection type to collection type and (ii) collection type to single valued element. The virtual view approach compensates both cases with corresponding casts. However, for restricting updates no additional information loss occurs, since no ordering information may get lost.

Case (i): Collection Type to Collection Type. The template for resolution has already been presented in Listing 8 and may be applied to this case as well.

Case (ii): Collection Type to Single Valued Element. In case of decreasing the upperBound from > 1 to 1 the underlying OCL type changes from Collection(T) to T causing the invalidation of collection operations such as first(). Thus, the corresponding resolution action is to wrap the now single valued element into its former collection type.

Listing 10 shows the template for a helper, which returns a collection containing a single element, i.e., the original value of the feature.

Listing 10: Template for Single Valued Element to Collection

```
-- template for resolution action
helper context ≪featureV0.eContainingClass≫ def :
    ≪featureV0.name≫() : ≪featureV0.type≫ =
    ≪featureV0.type≫{self.≪featureV0.name≫};
```

Finally, a collection might also be downsized by decreasing the upperBound from a value > 2 to a value > 1. This change has breaking impact on OCL expressions which access elements directly by their index, i.e., the operation at(index). If the index is greater than the upperBound, the transformation will break, since the access is out of bounds. However, since an automatic resolution action can only guarantee syntactical correctness by setting the index between 1 and lowerBound, thereby ensuring a return value for this operation, in this case user intervention is required to redefine the index.

Group ⑥: Destructive Updates: Analogous to destructive changes, destructive updates have breaking impact on OCL expressions. Furthermore, type changes may introduce a destructive effect, e.g., changing the type from Integer to Boolean as done by the change "Change Type" (cf. ⑦ in Fig. 1). Since there is no dedicated cast operation for Integer values to Boolean values in OCL, user intervention is required to specify a suitable cast operation. A potential user-defined resolution action is shown in Listing 11.

Listing 11: Resolution of Type Change

```
-- resolution action
helper context ClassD!Class def : persistent() :
     Integer =
    if (self.persistent = true) then 1 else 0 endif;

-- transformation definition
rule Class2Table extends Element2Named {
    from cl : Class!Class (cl.persistent() = 1)
    to table : Relational!Table (
        name <- cl.package + '.' + cl.id,
        col <- cl.attr,
        key <- cl.attr->first() )
}
```

Although no automatic resolution action may be generated, a template is provided (cf. Listing 2) which has to be completed by the user according to the migrations rules for model co-evolution and the semantic requirements.

4.4 Composition of Resolution Actions

After having dealt with the resolution of a single change, this section deals with the composition of resolution actions of more than one change affecting the same MM element, to provide a single helper as resolution action. Thereby, basically four combinations of meaningful changes on the same element are possible. First, a rename and a move might be arbitrarily combined without any additional type change. Second, a rename and a move might be arbitrarily combined with a change of a collection type. Third, a rename and a move might be arbitrarily combined with a type switch from a collection type to a single value typed element. Finally and forth, a rename and a move might be arbitrarily combined with a switch from a single value typed element to a collection type. For being able to produce templates covering those potential combinations, Listing 12 shows an EBNF, which is able to produce corresponding combined templates, covering the composition of resolution actions.

Listing 12: Composition of Resolution Actions

```
VirtualView = HelperSignature "="
 NoTypeSwitch |  CollectionTypeSwitch |
 CollectionToSingleValueSwitch |
 SingleValueToCollectionSwitch;
HelperSignature =
 "helper context ≪featureV0.eContainingClass≫
 def : ≪featureV0.name≫() : ≪featureV0.type≫";
Renaming =  ".≪featureV0.name≫" |
            ".≪featureV1.name≫";
Moving = "" | ".≪reference≫";
NoTypeSwitch = "self" [Moving] Renaming;
CollectionTypeSwitch =  NoTypeSwitch
 (".asSet()" | ".asBag()" |
  ".asSequence()" | ".asOrderedSet()");
CollectionToSingleValueSwitch =
 NoTypeSwitch "->any()";
SingleValueToCollectionSwitch =
 "≪featureV0.type≫{" NoTypeSwitch "}";
```

5 Ensuring Semantical Correctness of Co-Evolved Model Transformation Definition

As discussed previously, resolution actions must ensure syntactical as well as semantical correctness. While the former may be easily verified by a compiler, the verification of the latter is more challenging. Thus, this section discusses notions of semantics and introduces a formalism for the automatic verification of semantical correctness.

5.1 Notions of Semantics

When surveying dedicated literature, ideas for the verification of semantical correctness may be found for a restricted set of change operations, namely for refactorings, only. Thereby, a refactoring is said to be semantically correct, if the structure of a program or model is changed without changing its observable behavior [26]. For verifying the semantical correctness, regression testing is a common mechanism [7].

Consequently, regression testing may be applied to verify semantical correctness of refactorings of MMs and co-evolved transformations as well. However, there are changes in our complete and minimal set of changes that go beyond refactorings, e.g., destructive changes. Consequently, a more general approach is desirable, which relaxes the strong condition of demanding for the exactly same observable behavior. Thus, we propose to verify semantical correctness by dedicated properties expressed in the PaMoMo language [10], which must be fulfilled by a transformation definition, thereby relaxing the strong condition of exactly same observable behavior and by this, enabling for the verification of semantical correctness for all changes of our complete and minimal set of changes.

5.2 PaMoMo for Verifying Semantical Correctness

Originally developed for testing model transformations, our PaMoMo language provides a visual, declarative, formal specification language to describe correctness requirements for transformations. PaMoMo specifications allow to express *what* a transformation should do, but not how it should be done, thus, the mechanism may be used to specify requirements for semantical correctness as well.

5.2.1 PaMoMo in a Nutshell

A PaMoMo specification consists of declarative visual patterns, which may be *positive* or *negative*. Positive patterns (denoted by a "P") describe necessary conditions to be fulfilled (i.e., the pattern is satisfied by a pair of models, if these contain certain elements) while negative ones (denoted by a "N") state forbidden situations (i.e., the pattern is satisfied, if certain elements are not found). Patterns are composed of two compartments containing object graphs. The left compartment contains objects typed on the source MM, while the objects to the right are typed on the target MM. Objects in the source and target compartments may have attributes that may be assigned either a concrete value or a *variable*. A variable may be assigned to several attributes to ensure equality of their values.

The specified patterns provide a well-defined operational semantics on basis of QVT-Relations [24], which allows to check whether pairs of input models and resulting output models fulfill the specified correctness requirements, which in consequence allows to evaluate the semantical correctness of a transformation definition. For further details about PaMoMo, the interested reader is referred to [10].

5.2.2 PaMoMo for the Running Example

To exemplify this, Figure 4 shows three PaMoMo patterns that specify requirements regarding the semantical correctness for the original Class2Relational transformation. Thereby, the positive pattern PersClass2Table demands that for each Class object that is marked as being persistent (persistent=1) in a given source model, a corresponding Table object with the same name (cf. the bound variable X) must exist in a transformed target model. Furthermore, the negative pattern NonPersClass2Table demands that for each Class object that is not marked as being persistent (persistent=0) in a given source model, no corresponding Table object must exist in a produced target model. Finally, the pattern Attribute2Column demands that for each Attribute object of a persistent Class object in a source model, a corresponding Column object must exist in a transformed target model.

In order to be able to verify the semantical correctness of the co-evolved model transformation, the PaMoMo patterns must be co-evolved as well first. Since PaMoMo patterns are specified by means of object graphs, the co-evolution strategy employed for existing models may be re-used for this task by a dedicated model co-evolution tool such as COPE [13] assuming that the patterns are model fragments of the correpsonding domain MMs, resulting in the PaMoMo patterns shown in Figure 5. One might recognize that deleting changes entail a deletion of the corresponding parts of the patterns, thereby relaxing the requirements for semantical correctness. By this, unbound variables on the target side of the patterns may arise, such as "Y" in the pattern Attribute2Column, which formerly demanded for the assignment of the name of the package to the name of a table. As long as the variable remains unbound, it serves as a wildcard. However, if the user decides to replace the deleted package name, i.e., "Y", by an empty string, then the resolution action has to use the same value, i.e., an empty string, for the transformation definition to produce models that match this pattern, i.e., being semantically correct. The co-evolved patterns may then be used to check, if the co-evolved transformation maintains semantical correctness by checking if pairs of input models and produced output models fulfill the requirements stated by the PaMoMo patterns.

In summary, PaMoMo allows to specify correctness requirements for model transformations, thereby explicating semantical correctness and by this providing a formalism to verify the semantical correctness of a co-evolved model transformation.

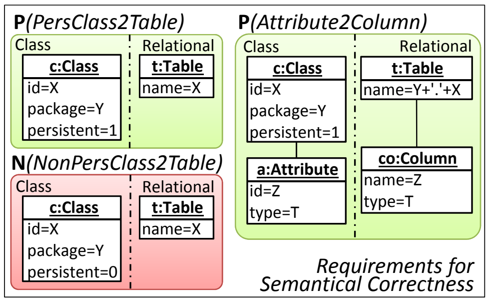

Figure 4: PaMoMo Patterns for Class2Relational

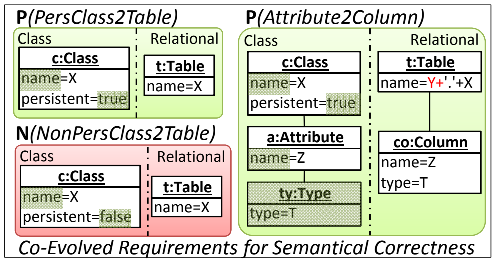

Figure 5: Co-Evolved PaMoMo Patterns

6 Lessons Learned

This section discusses the proposed approach by dedicated lessons learned.

Realization of a Virtual View is Restricted in ATL. As discussed in Section 4, our approach establishes a vir-

Approach	Focus of Work					Supported Changes					Impact Analysis on OCL	Resolution Actions		Implementation
	Co-Evolution of		Technical Space			Classes of Supported Changes			Complete Set	Minimal Set				
	OCL in Model Transformations	OCL Constraints in Metamodels	Ecore	UML	MOF	Constructive	Destructive	Updative				Syntactical Correctness	Verification of Semantical Correctness	
García et al. [8]	~ (ATL)		✓			~	~	~	✗ (23 atomic & composite changes)	✗	~	✓	~	✓
Garcés et al. [7]	✗ (ATL)		✓			~	~	~	✗ (number of atomic & composite changes unkown)	✗	✗	✓	~	✓
Kruse [15]	✗ (ATL)		✓			~	~	~	✗ (16 atomic & composite changes)	✗	✓	✓	~	✓
Hassam et al. [10]		✓			✓	~	~	~	✗ (17 atomic & composite changes)	✗	✓	✓	~	✓
Markovic et al. [19]		✓		✓		✗	✗	~	✗ (6 atomic & composite changes)	✗	✓	✓	~ [1]	✓
Kosiuczenko [14]		✓		✓		~	~	~	✗ (number of atomic & composite changes unkown)	✗	✗	✓	~	✗
Correa et al. [4]		✓		✓		~	✗	~	✗ (number of atomic & composite changes unkown)	✗	✗	✓	✓	✓
Own work	✓ (ATL)		✓			✓	✓	✓	✓ (30 atomic changes)	✓	✓	✓	✓	In progress

Legend: ✓ ... true ✗ ... false ~ ... partially true

Table 3: Comparison of Related Approaches

tual view on MM_1 for resolution such that the input models appear to the transformation in the original structure, i.e., conforming to the source MM_0. Although this conceptual approach is able to restore syntactical and semantical correctness in general, the concrete realization on basis of ATL helpers is restricted, since ATL's view capabilities support virtual features, only, whereas virtual classes are not supported. Thus, changes affecting classes have to be resolved by a "program transformation" realization in ATL, demanding for a hybrid approach.

Breaking Changes Inducing an Information Loss Demand for User Intervention. Changes may be classified according to their potential of inducing an information loss. Thereby, constructive changes, renaming updates, moving updates, relaxing updates as well as constructive updates never induce an information loss. In contrast, destructive changes, restricting updates, and destructive updates always induce an information loss assuming the existence of corresponding instances. Since lost information can not be re-established automatically, those changes with breaking effect on the syntax always demand for user intervention, except when automatic casts are possible as in case of some kinds of restricting updates.

PaMoMo Enables Checking of Semantical Correctness for Arbitrary Changes. As already mentioned, semantical correctness may be specified by dedicated input models and corresponding expected output models, i.e., by specifying the observable behavior of a transformation. Consequently, a co-evolution may be said to be semantics preserving, if the observable behavior has not been changed. However, this methodology is applicable to changes that do not interfere with the observable behavior, i.e., refactorings, only, while all other changes might not be semantics preserving according to this restrictive definition. PaMoMo relaxes this restrictive definition by not specifying the observable behavior by hard-coded pairs of input/output model but instead by stating properties the models must fulfill. Consequently, PaMoMo enables the checking of semantic correctness for arbitrary changes.

7 Related Work

Subsequently, related work is surveyed regarding its focus, supported changes, impact analysis on OCL, syntactical and semantical resolution of breaking changes, and availability of a prototypical implementation (cf. Table 3).

Regarding the *focus of co-evolution* in a specific *technical space*, two groups of approaches with respect to their usage of OCL exist. Most closely related, the first group of approaches targets the co-evolution of transformations employing OCL expressions [8, 9, 16], all of them in the *technical space* of Ecore, whereby the co-evolution of the OCL-part is considered particularly by one of them [9], only. More widely related since not focusing on OCL in model transformations but still facing the same problems in OCL co-evolution, the second group concentrates on resolving inconsistent OCL constraints as parts of UML class diagrams [5, 15, 20], and one exception basing on MOF [11].

Considering the *supported changes*, six approaches [5, 8, 9, 11, 15, 16] partially allow for constructive changes, five of those [8, 9, 11, 15, 16] partially consider destructive changes, and updative changes are partially supported by all approaches. Thus, in contrast to our approach, no approach covers a *complete* change set. However, the surveyed approaches additionally consider composite changes, which will be one line of future work as detailed below. By concentrating on composite changes, no approach presents a minimal change set, which is different to our work providing a systematically derived, minimal set of atomic changes.

Regarding the *impact on OCL*, four approaches [9, 11, 16, 20] consider breaking and non-breaking impact on the *syntax*, whereby one of them [9] considers impacts partially, only.

Considering *resolution*, all approaches take care of syntactical correctness, i.e., they syntactically resolve the induced inconsistencies. Regarding semantical correctness, five approaches [8, 9, 11, 15, 16] do not provide any means for automatic verification, but partially rely on refactorings, for which the according resolution actions are defined in literature (cf., e.g., [7]). One approach does not discuss semantics but instead refers to [1], in which the semantics preservation for one refactoring, i.e, Move Attribute, is proven. One approach [5] employs regression testing, being most closely related to our approach.

Finally, six approaches [5, 8, 9, 11, 16, 20] provide an *implementation*, while a sole approach is conceptual, only. A full implementation of the work presented in this paper is in progress, while the proof-of-concept has already been presented in Section 4.

In summary, one may see that the work presented in this paper is unique with respect to completeness and minimality of changes and, consequently, provides an entire examination of changes. Furthermore, resolution actions have been proposed for all breaking changes.

8 Conclusion & Future Work

In this paper, (semi-)automatic resolution actions for the co-evolution of OCL expressions in model transformations in response to MM evolution have been proposed, building upon the work presented in [17]. In the course of this paper, several lines of future work have been identified. First, we plan on investigating composite changes including refactorings, which may affect more than one MM element, therefore resolution actions have to be combined accordingly. Second, we will examine how the ATL helpers realizing the virtual view may have to be adapted in case of changes on the same MM element in series evolution steps. For this, first ideas for optimizing ATL helpers have already been discussed in Section 4. Finally and third, the prototypical proof-of-concept implementation will be extended to a comprehensive tool basing on EMF[4] for impact analysis and co-evolution of OCL expressions in model-transformations.

Acknowledgements

This work has been funded by the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT) grant FFG BRIDGE 832160, 838526, FFG FIT-IT 829598, and by WTZ/ÖAD grant AR18-2013 and UA07-2013.

References

[1] Baar, T. and Marković, S. [2006], A graphical approach to prove the semantic preservation of UML/OCL refactoring rules, *in* 'PSI', Springer, pp. 70–83.

[2] Bézivin, J. [2005], 'On the Unification Power of Models', *SoSym* **4**(2), 171–188.

[3] Büttner, F., Gogolla, M., Hamann, L., Kuhlmann, M. and Lindow, A. [2010], On Better Understanding OCL Collections or An OCL Ordered Set Is Not an OCL Set, *in* 'Models in Software Engineering', Vol. 6002 of *LNCS*, Springer.

[4] Cabot, J. and Teniente, E. [2007], 'Transformation techniques for OCL constraints', *Science of Computer Programming* **68**(3), 179 – 195.

[5] Correa, A. and Werner, C. [2004], Applying Refactoring Techniques to UML/OCL Models, *in* 'UML 2004', Springer, pp. 173–187.

[6] Di Ruscio, D., Iovino, L. and Pierantonio, A. [2012], Evolutionary Togetherness: How to Manage Coupled Evolution in Metamodeling Ecosystems, *in* 'ICGT', Springer, pp. 20–37.

[7] Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D. [1999], *Refactoring: improving the design of existing code*, Addison-Wesley.

[8] Garcés, K., Vara, J., Jouault, F. and Marcos, E. [2013], 'Adapting transformations to metamodel changes via external transformation composition', *SoSym* pp. 1–18.

[9] García, J., Diaz, O. and Azanza, M. [2013], Model Transformation Co-evolution: A Semi-automatic Approach, *in* 'SLE', Springer, pp. 144–163.

[10] Guerra, E., de Lara, J., Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J. and Schwinger, W. [2012], 'Automated verification of model transformations based on visual contracts', *Journal of Automated Softw. Eng.* **20**(1), 5–46.

[11] Hassam, K., Sadou, S., Gloahec, V. L. and Fleurquin, R. [2011], Assistance System for OCL Constraints Adaptation during Metamodel Evolution, *in* 'SMR', IEEE, pp. 151–160.

[4] http://eclipse.org/modeling/emf/

[12] Herrmannsdoeorfer, M. and Wachsmuth, G. [2014], Coupled Evolution of Software Metamodels and Models, *in* 'Evolving Software Systems', Springer, pp. 33–63.

[13] Herrmannsdoerfer, M., Benz, S. and Juergens, E. [2009], COPE - Automating Coupled Evolution of Metamodels and Models, *in* 'ECOOP', Springer, pp. 52–76.

[14] Jouault, F., Allilaire, F., Bézivin, J. and Kurtev, I. [2008], 'ATL: A model transformation tool', *Science of Computer Programming* **72**(12), 31–39.

[15] Kosiuczenko, P. [2009], 'Redesign of UML class diagrams: a formal approach', *SoSym* **8**(2), 165–183.

[16] Kruse, S. [2011], On the Use of Operators for the Co-Evolution of Metamodels and Transformations, *in* 'Int. Workshop on Models and Evolution'.

[17] Kusel, A., Etzlstorfer, J., Kapsammer, E., Retschitzegger, W., Schönböck, J., Schwinger, W. and Wimmer, M. [2014], A Systematic Taxonomy of Metamodel Evolution Impacts on OCL Expressions, *in* 'submitted for Int. Workshop for Models and Evolution @ MODELS 2014'.

[18] Lenzerini, M. [2002], Data integration: A theoretical perspective, *in* 'PODS', ACM, pp. 233–246.

[19] Lientz, B. P., Swanson, E. B. and Tompkins, G. E. [1978], 'Characteristics of application software maintenance', *CACM* **21**(6), 466–471.

[20] Markovic, S. and Baar, T. [2008], 'Refactoring OCL annotated UML class diagrams', *SoSym* **7**(1), 25–47.

[21] Méndez, D., Etien, A., Muller, A. and Casallas, R. [2010], Towards Transformation Migration After Metamodel Evolution, *in* 'Int. Workshop on Models and Evolution'.

[22] Miller, R. J., Ioannidis, Y. E. and Ramakrishnan, R. [1993], The use of information capacity in schema integration and translation, *in* 'VLDB', Vol. 93, Morgan Kaufmann, pp. 120–133.

[23] Object Management Group [2011*a*], 'Meta Object Facility (MOF) 2 Core Specification', www.omg.org/spec/MOF/2.4.1.

[24] Object Management Group [2011*b*], 'Meta Object Facility (MOF) Query/View/Transformation (QVT)', http://www.omg.org/spec/QVT/1.1.

[25] Object Management Group [2014], 'OMG Object Contraint Language (OCL)', http://www.omg.org/spec/OCL/2.4.

[26] Opdyke, W. F. [1992], Refactoring object-oriented frameworks, PhD thesis, University of Illinois at Urbana-Champaign.

[27] Rossi, M. and Brinkkemper, S. [1996], 'Complexity Metrics for Systems Development Methods and Techniques', *Information Systems* **21**(2), 209–227.

[28] Visser, E. [2001], 'A survey of rewriting strategies in program transformation systems', *Electronic Notes in Theoretical Computer Science* **57**, 109–143.

[29] Wimmer, M., Martínez, S., Jouault, F. and Cabot, J. [2011], 'A Catalog of Refactoring for Model-to-Model Transformations', *JOT* **11**(2), 1–40.