Consistent Co-Evolution of Models and Transformations

Angelika Kusel, Jürgen Etzlstorfer, Elisabeth Kapsammer,
Werner Retschitzegger, and Wieland Schwinger
Johannes Kepler University Linz, Austria
[firstname.lastname]@cis.jku.at

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

***Abstract*—Evolving metamodels are in the center of Model-Driven Engineering, necessitating the co-evolution of dependent artifacts like models and transformations. While model co-evolution has been extensively studied, transformation co-evolution has received less attention up to now. Current approaches for transformation co-evolution provide a fixed, restricted set of metamodel (MM) changes, only. Furthermore, composite changes are treated as monolithic units, which may lead to inconsistent co-evolution for overlapping atomic changes and prohibits extensibility. Finally, transformation co-evolution is considered in isolation, possibly inducing inconsistencies between model and transformation co-evolution. To overcome these limitations, we propose a complete set of atomic MM changes being able to describe arbitrary MM evolutions. Reusability and extensibility are supported by means of change composition, ensuring an *intra-artifact consistent* co-evolution. Furthermore, each change provides resolution actions for both, models and transformations, ensuring an *inter-artifact consistent* co-evolution. Based on our conceptual approach, a prototypical implementation is presented.**

I. Introduction

Model-Driven Engineering uses models as first-class artifacts throughout the software life-cycle [1], which conform to their respective abstract syntax being defined in terms of metamodels (MMs). However, like any other software artifact, MMs are subject to constant change, i.e., they evolve, caused by, e.g., changing requirements [2], [8], [16], [21]. Through the evolution of the MM, the conformance between the MM and dependent artifacts, e.g., models and transformations, may be violated, which hinders further processing thereof. Thus, automatic co-evolution is indispensable to re-enable the processing of models and the execution of model transformations. While model co-evolution has been studied extensively (cf. [10] for a survey), transformation co-evolution is currently less understood, although first works exist (cf. e.g., [5], [6], [13]).

However, current approaches for transformation co-evolution provide support for a *fixed, restricted set of MM changes*, only, mostly basing on composite changes such as common refactorings in MMs, but neglecting other MM evolution scenarios. Furthermore, these approaches do not consider the composite nature of refactorings, i.e., composite changes are treated as *monolithic units*. Hence, this hinders reusability and endangers consistency of resolution actions for overlapping atomic changes across different composite changes, i.e., *intra-artifact consistency*. For example, the composite changes ExtractSuperClass and InlineSubClass both require moving of features, which should be handled consistently across them. Finally, current approaches treat transformation co-evolution in isolation, i.e., the same semantics for model co-evolution and transformation co-evolution is not ensured [10], termed as *inter-artifact consistency*. For example, if the composite change ExtractSubClass retypes instances to the new subclass, transformations should still be able to transform them.

This paper presents a *systematic set of composable atomic MM changes* based on our previous works [14], [15] on inspecting potential variation points between two Ecore-based MMs, which is now enhanced with heterogeneities between different versions of a MM. Thus, our change set goes beyond commonly considered atomic changes such as create, delete, or update, which may induce ambiguities during co-evolution [20]. For every atomic change, dedicated resolution actions for models and transformations are defined. Consequently, a set of *reusable* and *composable* atomic MM changes is provided. Composite changes such as refactorings solely compose the resolution actions of atomic changes to provide *extensibility*. Thus, (i) an *intra-artifact consistent* co-evolution is ensured, since resolution actions of atomic changes may be reused across different composite changes and (ii) an *inter-artifact consistent* co-evolution is ensured, since the same co-evolution semantics is applied for models and transformations. Finally, a prototypical implementation of our conceptual approach is presented, which provides resolution actions for Ecore-based models as well as rule-based transformation languages, whereby we focussed on the evolution of the source MM as well as the declarative part of ATL [12] in a first step.

The paper is structured as follows: Section II introduces a running example, whereas Section III presents the set of reusable atomic MM changes on basis of Ecore. In Section IV, resolution actions for each change are presented, while in Section V we report on our prototypical implementation. Finally, in Section VI we discuss related work, before our approach is critically discussed in Section VII.

II. Running Example

This section introduces a simple running example to illustrate the main concepts of our approach. It encompasses an extract of the well-known Class2Relational transformation and demonstrates an evolution of the source MM_0 and its effects on models and transformations.

978-1-4673-6908-4/15 © 2015 IEEE

MODELS 2015, Ottawa, ON, Canada
Foundations

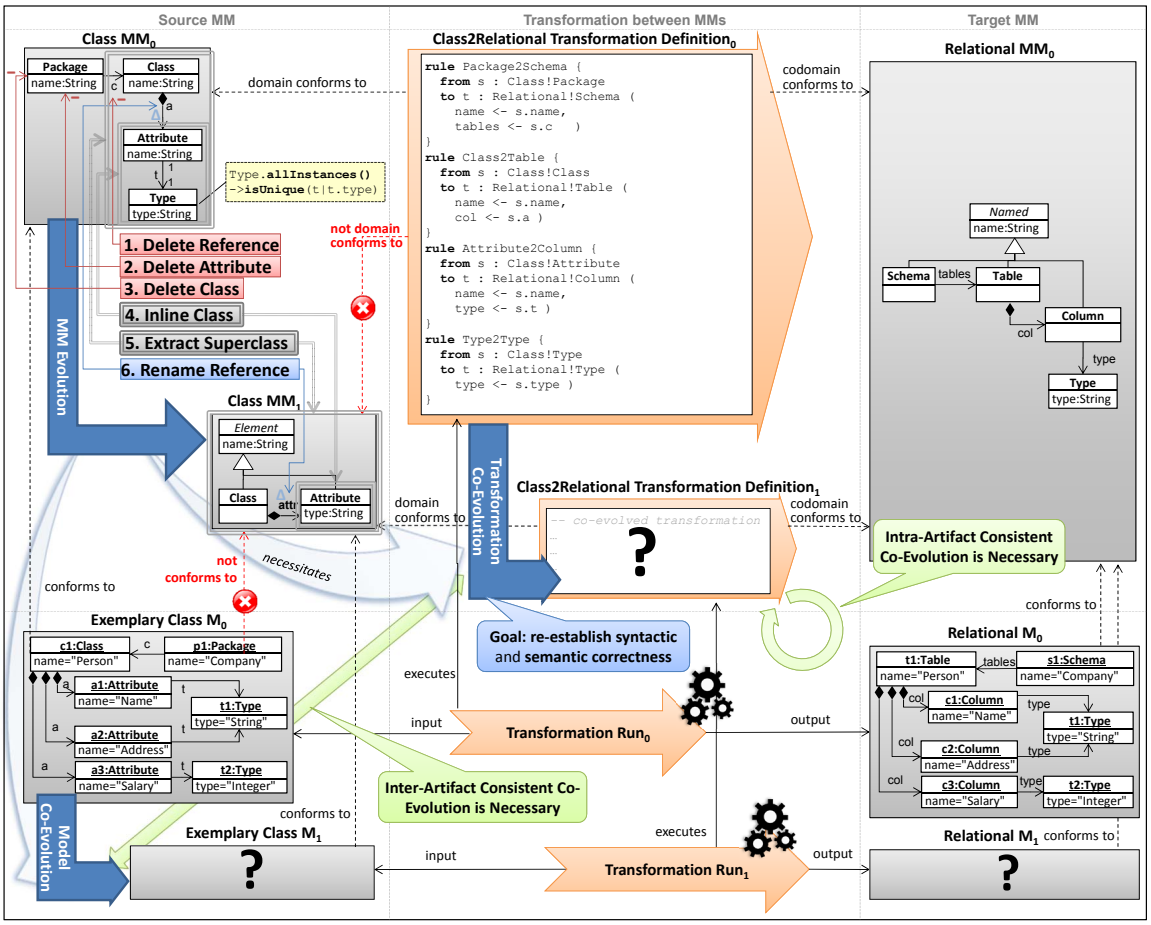


Figure 1. Running Example: Class2Relational

During the evolution of the source MM_0, six changes with diverse impacts on the dependent artifacts have been applied (cf. Fig. 1). First, second, and third the class Package with its features Package.name and Package.c have been deleted. Forth, the composite change Inline Class has been applied, which moved the attribute Type.type to the class Attribute and deleted both, the reference Attribute.t and the class Type. Fifth, the composite change Extract Class has been applied, which created a new common superclass Element for the classes Class and Attribute to hold the common attribute name. Finally and sixth, the reference Class.a has been renamed to Class.attr.

Given a transformation T_0 which maps models M_0 conforming to MM MM_0 to models N conforming to MM NN, the evolution of MM_0 to MM_1 via a fixed repertoire of MM changes gives rise to a co-evolution of T_0 to T_1 as well as M_0 to M_1 by some *resolution actions* to re-establish (i) *syntactical correctness* and (ii) *semantic correctness*, i.e., the original version of the target model Relational M_0 should be preserved as far as possible. Thus, for supporting semantic correctness, it is essential to perform transformation co-evolution consistently with model co-evolution (cf. inter-artifact consistency). Moreover, in doing so, T_1 may be guaranteed to take models M_1 conforming to MM_1 to models N conforming to NN.

Consequently, the information that the values of Attribute.type have been distinct in the original version Relational M_0, as explicated in Fig. 1 by a corresponding OCL constraint, has to be incorporated when co-evolving the Class2Relational transformation definition. Thus, a composable change set allowing to describe arbitrary MM changes with accompanying resolution actions is needed, which is able to not only represent the transformation co-evolution itself, but is also aware of the model co-evolution semantics. In the next section, a change set fulfilling these requirements is presented.

III. A Systematic Set of Atomic MM Changes

In this section, a systematic set of atomic changes is introduced to describe arbitrary MM evolutions and co-evolution semantics for models and transformations, thereby targeting an *inter-artifact consistent* co-evolution. This set builds the basis to build composite changes by reusing the atomic changes to ensure *intra-artifact consistent co-evolution*. For this, we extend our previous work by (i) regarding the declarative part of ATL along with OCL, (ii) considering also models as dependent artifacts instead of transformations only, and (iii) introducing means for composition of resolution actions.

A. Minimal and Complete Change Set Derived from Ecore

To be able to describe arbitrary evolutions, a systematic set of MM changes is needed. Therefore, we build upon our previous works [14], [15], [24] on inspecting potential

variation points between two Ecore-based MMs, which is briefly recapped in the following, before it is enhanced by dealing with heterogeneities between different MM versions. The initial change set has been designed with two criteria in mind, namely (i) *completeness* to allow for any possible MM evolution and (ii) *minimality* to avoid the redundant analysis of changes as may be the case for overlapping atomic changes contained within composite changes, since those changes are a potential source of *intra-artifact inconsistencies* during the co-evolution.

The resulting change set itself comprises three groups of changes, namely (i) *constructive changes*, i.e., changes adding new MM elements, (ii) *destructive changes*, i.e., changes deleting existing MM elements, and (iii) *updative changes*, i.e., changes altering existing MM elements. While the first and the second group have been derived by resorting to all concrete meta-classes, e.g., EClass, since those may be instantiated and, thus, may result in the addition or the deletion of MM elements, the latter group has been obtained by referring to all meta-features, e.g., EClass.abstract, since those may result in a change of an existing MM element (cf. Fig. 2).

B. Extended Change Set to Avoid Co-Evolution Ambiguities

However, not all changes of the above presented complete and minimal change set carry an unambiguous co-evolution semantics, and thus, may entail a heavy user involvement in the co-evolution process, i.e., the user must decide for a specific co-evolution action on a per-artifact basis, which may lead to an *inter-artifact inconsistent co-evolution*. To exemplify this, one might revisit the running example. Thereby, one might recognize that changes of the same type need a different co-evolution semantics, i.e., they are ambiguous. One example thereof is the atomic change Delete Class, which is applied twice: (i) referring to the class Package and (ii) referring to the class Type within the context of the composite change Inline Class (cf. Fig. 3). While for (i), instances of the class Package should be deleted, entailing a real information loss, for (ii) the information contained within Type instances should be moved to Attribute instances.

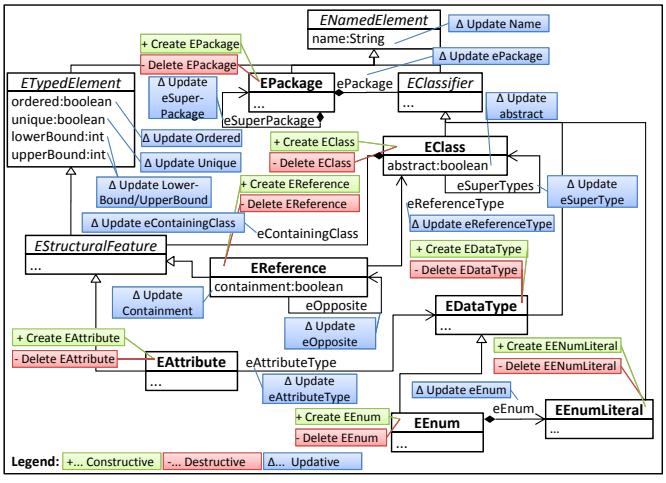


Figure 2. Minimal and Complete Change Set Derived from Ecore

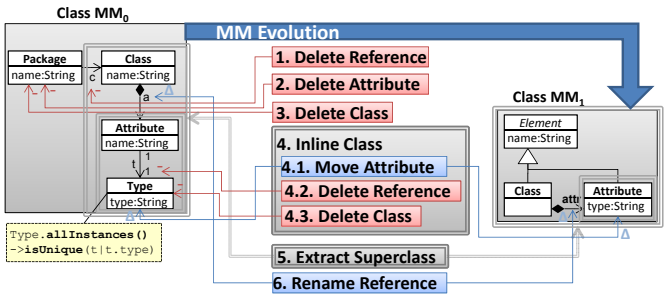


Figure 3. Ambiguous Atomic Changes

Consequently, one specific MM change may exhibit different co-evolution semantics. Thus, we argue to decide for a specific co-evolution semantics already in the MM evolution process and to extend the above change set on basis of a systematic classification of potential MM heterogeneities that might occur during the evolution process. Thus, the resulting extended change set carries unambiguous resolution actions and builds the basis for composition, to avoid *intra-artifact inconsistencies*, stemming from overlapping changes across composite changes. Hence, the question arises, which MM changes may exhibit ambiguous co-evolution semantics.

To answer this question, one has to resort to the effects a specific MM change may entail. In our previous works [14], [15], we classified these effects along the criteria *structural complexity* and *information capacity*. While the former refers to the effects on the number of instantiable types a MM change may entail, the latter refers to the effects on the potential number of valid instances. Actually, changes modifying the structural complexity may exhibit ambiguous co-evolution semantics, since for added as well as removed instantiable types, the concrete co-evolution semantics is unclear. This is since it is unknown whether an *inherent mapping* exists between an added or deleted element in MM_1 and some element in MM_0 (as is the typical case in a refactoring). Thus, a added or deleted element in MM_1 may exhibit some hidden relation (mapping) to elements in MM_0 necessitating different co-evolution semantics. Hence, to get unambiguous changes, one needs to introduce specific changes for constructive and destructive changes to explicate these inherent mappings.

Systematic Set of Mapping Situations. To obtain a systematic set of potential mapping situations, one might combine all core concepts of a MM, i.e., classes (denoted as C), attributes (denoted as A), and references (denoted as R) with potential cardinalities (denoted as N, 1, and 0), as shown by the feature model in Fig. 4. Each constructive or destructive change might actually represent one of the arising combinations, e.g., when referring to the change Delete Class "Package", this would represent a C2C with cardinality 1:0, since the class Package has been deleted without any compensating element, whereas the change Delete Class "Type" would represent a C2C with cardinality N:1, since N=2 classes (classes Type and Attribute) have been merged into one (class Attribute). Hence, specific subchanges for constructive and destructive changes are needed for each potential combination to achieve an extended

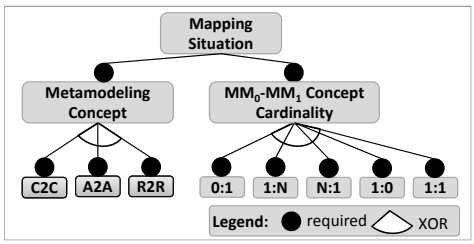


Figure 4. Potential Mapping Situations

change set exhibiting an *unambiguous co-evolution semantics*, as presented in the following. In addition, dedicated changes dealing with inheritance in MMs have been introduced.

Unambiguous Change Set. The consideration of specialized kinds of changes results in the change MM shown in Fig. 5. For each *constructive change* (cf. CreateClass, CreateAttr, and CreateRef), dedicated subchanges for the cardinalities 0:1 and 1:N have been introduced, since a newly created element may represent either a *new element* without any inherent mapping or a *partition* of an existing element. Concerning the latter, we differentiate the cases that either the instances have been projected or selected to obtain partitions following the common partitioning cases well-known from relational database design (cf. vertical versus horizontal partitioning [18]). In case of vertical partitioning, it may further be differentiated, if the new partition holds redundant information (e.g., in case of a 1..1 relationship between the partitions – cf. VClassSplitter), or distinct information, only (e.g., in case of a *..1 relationship between partitions – cf. DistVClassSplitter). In case of classes, a constructive change might also introduce changes concerning inheritance, represented by the classes CreateSubclass or CreateSuperclass in the MM.

Likewise for each *destructive change* (cf. DeleteClass, DeleteAttr, and DeleteRef), dedicated subchanges for the cardinalities 1:0 and N:1 have been introduced, since a deleted element may represent either an element without any compensating element, incurring a real information loss or may represent an element that has been *merged* into another element. Furthermore, again the distinction between vertical and horizontal merging has been introduced, accordingly. In case of classes, dedicated subclasses to deal with inheritance have been introduced (cf. DeleteSubClass and DeleteSuperClass).

For reasons of completeness, *updative changes* are depicted as well, for which more details may be found in [15].

C. Composition of Changes for Intra-Artefact Consistency

The proposed set of changes builds the basis for composite changes by reusing atomic changes, thereby allowing for an *intra-artifact consistent co-evolution* for overlapping atomic changes across composite changes. Fig. 5 depicts exemplary composite changes on top of the atomic changes. For example: the composite change InlineClass composes either the change VClassMerger or DistVClassMerger, depending on whether the instances of MM_0 have been distinct or not, as well as an arbitrary number of MoveFeature changes. Likewise the composite change ExtractClass composes either the change VClassSplitter or DistVClassSplitter, as well as again an arbitrary number of MoveFeature changes. To highlight such overlapping changes across the exemplary shown composite changes, dedicated common superclasses have been introduced (cf. MovingChange, PullingUpChange, and PushingDownChange). Moreover, this set of changes is *extensible*, since new compositions of atomic or existing composite changes may be built on top of them. However, a dedicated composition language is subject to future work.

IV. Inter-Artifact-Consistent Resolution Actions

Based on the systematic classification of changes, resolution actions re-establishing *syntactic correctness* as well as *semantic correctness* in (i) models and (ii) transformations are introduced. Finally, the actual composition and execution order of resolution actions are discussed.

A. Resolution Actions for Models

For each change a resolution action for models has been defined (cf. Table I), basing on existing literature (cf. e.g., [9]). Subsequently, the resolution actions are described along the two criteria *structural complexity & information capacity*.

Changes Increasing Structural Complexity & Information Capacity. In general, no resolution action for changes that increase structural complexity and information capacity (cf. changes with heterogeneity 0:1 in Table I) are required, since no existing MM elements are altered. The only exception is, if a new element is required, since in this case default values for attributes, references, and objects need to be created.

Changes Increasing Structural Complexity. Changes increasing structural complexity, but not influencing information capacity (cf. changes with heterogeneity 1:N as well as CreateSubclass and CreateSuperclass in Table I), demand for specific resolution actions for (i) vertical partitioning, (ii) horizontal partitioning, and (iii) changes in the inheritance hierarchy. For *vertical partitioning*, instances have to be vertically split, whereby one has to consider the relationship between the partitioned classes. In case of an 1..1 relationship (VClassSplitter in Table I), every object needs to be split and results in a new instance of class X, whereas in case of a *..1 relationship (DistVClassSplitter in Table I), new instances of class X arise for distinct values, only. For *horizontal partitioning*, the instance set needs to be split. Finally, changes in the *inheritance hierarchy* again involve a split of the instance set, but this time along the inheritance hierarchy.

Changes Decreasing Structural Complexity & Information Capacity. Changes, decreasing both, structural complexity and information capacity (cf. changes with heterogeneity 1:0 in Table I), entail a deletion of the corresponding instances as shown in Table I, and thus, cause information loss.

Changes Decreasing Structural Complexity. Changes decreasing structural complexity, but not influencing information capacity (cf. changes with heterogeneity N:1, DeleteSubclass and DeleteSuperclass in Table I) require the inverse resolution actions as applied for changes increasing structural capacity.

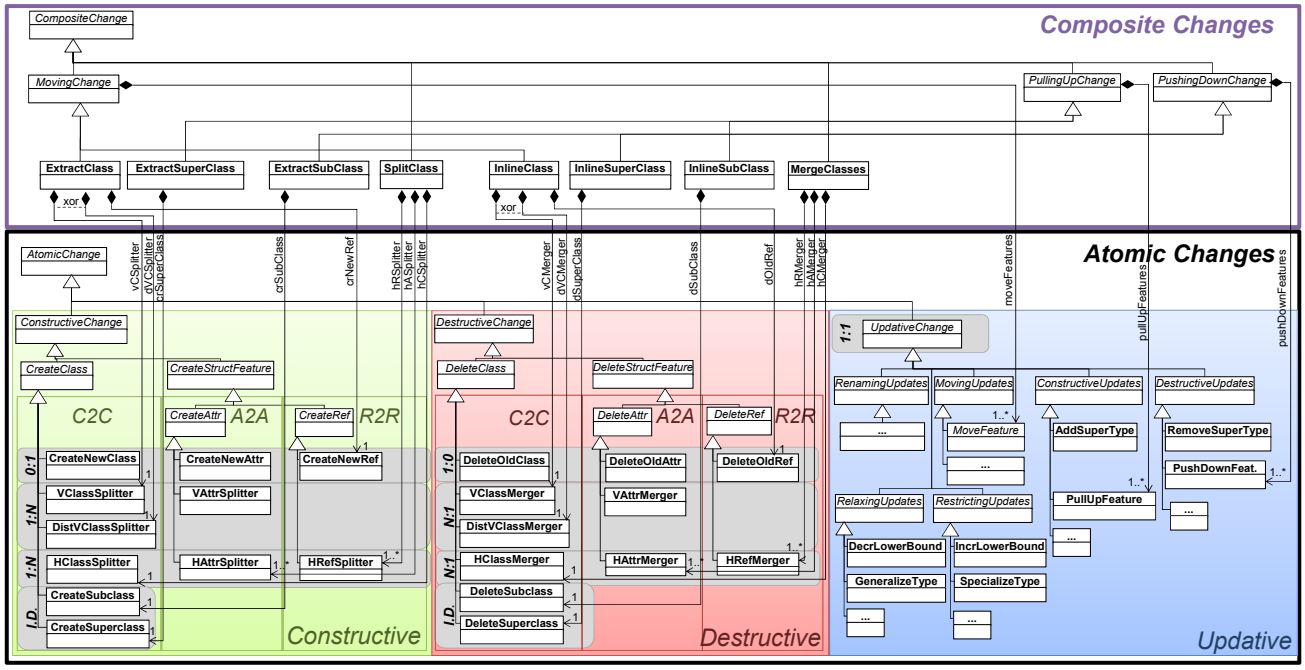


Figure 5. Change Metamodel

B. Resolution Actions for Transformations

Based on the resolution actions for models, this subsection presents resolution actions for rule-based model transformations (cf. Table I, which depicts the resolution semantics on basis of ATL snippets – being a popular rule-based transformation language). For achieving *syntactic & semantic correctness*, we propose a conceptual approach that establishes a virtual view on source MM_1, which 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. For changes on features, this virtual view is realized by ATL helpers (as proposed in [15]), whereas for classes this virtual view is realized by rewriting the ATL transformation itself, since ATL helpers do not allow for the definition of virtual classes. Subsequently, the resolution actions are described groupwise again.

Changes Increasing Structural Complexity & Information Capacity. As for models, no specific resolution actions on transformations are required for these changes (cf. changes with heterogeneity 0:1 in Table I), since no transformation rules or bindings may have targeted newly created elements.

Changes Increasing Structural Complexity. Changes with heterogeneity 1:N in Table I again have to consider the cases of (i) vertical partitioning, (ii) horizontal partitioning, and (iii) changes in the inheritance hierarchy with different resolution actions for classes and features, as detailed in the following.

Concerning the *vertical partitioning* of classes, on rule level no specific resolution is required, since the information of the split instances may be accessed over the introduced reference, i.e., only the feature access has to be co-evolved accordingly. In contrast, the vertical partitioning of attributes demands for a resolution by a corresponding inverse function, which may be realized by a dedicated helper in ATL. For example, if a VAttrSplitter splits a string attribute into two, then an inverse function is needed, which merges those string values again to retain the original value for the transformation.

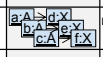
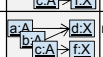
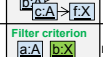
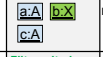
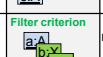
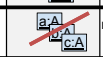
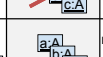
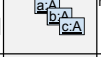
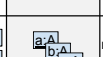
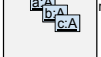
Regarding the *horizontal partitioning* of classes, the resolution must provide dedicated copies of the original rule, whereby the arising rules need to be set to match for instances of correct type (either the original type or the newly introduced type). To achieve this, transformation rules need to be copied and rewritten (by means of a HOT - cf. Section V).

Concerning *inheritance*, no resolution is required for the change CreateSubclass, if the transformation language supports the concept of type substitutability, i.e., if a rule may be applied to all instances of class A, then this rule may also be applied to all instances of all subclasses of A (cf. [23]), which is the case for ATL. In contrast the change CreateSuperclass, may require the introduction of a corresponding filter criterion. However, no resolution is required, if the newly introduced class is abstract, since no instances may exist.

Changes Decreasing Structural Complexity & Information Capacity. As for models, resolution for these changes (cf. changes with heterogeneity 1:0 in Table I) needs to delete all transformation rules matching for the deleted class in T_0 as well as all bindings accessing the deleted features.

Changes Decreasing Structural Complexity. Resolution actions for this type of changes (cf. changes with heterogeneity N:1 as well as DeleteSubclass and DeleteSuperclass in Table I) (i) have to ensure that no rules match anymore for classes that have been deleted during the merge process to achieve *syntactic correctness* and (ii) have to provide means to gain again access to the previously separated information in MM_0 to achieve *semantic correctness*. In case of a *vertical merge* the rule that matched for elements of the deleted class X needs to be deleted and the rule that matches for

Table I
CO-EVOLUTION TABLE: MODEL AND TRANSFORMATION CO-EVOLUTION

Elem.	Type	Name of Change	MM Evolution: MM_0	MM Evolution: MM_1	Hetero-geneity	Struct. Complex-ity	Informa-tion Capacity	Instance Co-Evolution: M_0	Instance Co-Evolution: M_1	Transformation Co-Evolution: T_0	Transformation Co-Evolution: T_1
Class	Constructive	CreateNewClass	-		0:1	+	+	-	-	-	-
		VClassSplitter			1:N	+	o			rule A2Target { from s : MM!A to …. }	rule A2Target { from s : MM!A to…. }
		DistVClassSplitter			1:N					rule A2Target{ from s : MM!A to…. }	rule A2Target{ from s : MM!A to….}
		HClassSplitter			1:N					rule A2Target{ from s : MM!A to…. }	rule A2Target{ from s : MM!A to….} copy & retype rule A2Target_1{ from s : MM!X to…. }
		CreateSubclass			Inheritance Difference					rule A2Target{ from s : MM!A to…. }	rule A2Target { from s : MM!A to…. }
		CreateSuperclass			Inheritance Difference					rule A2Target{ from s : MM!A to…. }	rule A2Target{ from s : MM!A to…. } rule A2Target { from s : MM!X (filter) to…. }
	Destructive	DeleteOldClass		-	1:0	-	-			rule A2Target{ from s : MM!A to…. }	~~rule A2Target{ from s : MM!A to….}~~
		VClassMerger			N:1		o			rule A2Target{ from s : MM!A to…. } rule B2Target{ from s : MM!B to…. }	rule A2Target{ from s : MM!A to t1 : TMM!A (…), t2 : TMM!B (…) } ~~rule B2Target{ … }~~
		DistVClassMerger			N:1					rule A2Target{ from s : MM!A to…. } rule B2Target{ from s : MM!B to…. }	rule A2Target{ from s : MM!A to…. } ~~rule B2Target{ from s: MM!B to t : TMM!B }~~ unique lazy rule B2Target{ from s : [derived from feature] to t : TMM!B (…) }
		HClassMerger			N:1					rule A2Target{ from s : MM!A to…. } rule B2Target{ from s : MM!B to…. }	rule A2Target{ from s : MM!A (filter) to…. } rule B2Target{ from s : MM!A (filter) to…. }
		DeleteSubclsas			Inheritance Difference					rule A2Target{ from s : MM!A to…. } rule B2Target{ from s : MM!B to…. }	rule A2Target{ from s : MM!A (filter) to…. } rule B2Target{ from s : MM!A (filter) to…. }
		DeleteSuperclass			Inheritance Difference					rule A2Target{ from s : MM!A to… } rule B2Target{ from s : MM!B to… }	rule A2Target{ from s : MM!B (filter) to… } rule B2Target{ from s : MM!B (filter) to… }
Attribute	Constructive	CreateNewAttr			0:1	+	+	-	-	-	-
		VAttrSplitter			1:N	+	o			rule A2Target { from s : MM!A to t : TMM!T (feature <- s.a)}	helper context MM!A def a : String = return mergeFunction(); rule A2Target { from s : MM!A to t : TMM!T (feature <- s.a)}
		HAttrSplitter			1:N					rule A2Target{ from s : MM!A to t: TMM!T(feature <-s.a)}	rule A2Target{ from s : MM!A to t: TMM!T(feature <- s.a)} -- rule created by "HClassSplitter" rule B2Target_1{ from s: MM!B to t : TMM!T(feature <- s.a)}
	Destructive	DeleteOldAttr			1:0	-	-			rule A2Target { from s : MM!A to t : TMM!T (feature <- s.a) }	rule A2Target { from s : MM!A to t : TMM!T (~~feature <- s.a~~) }
		VAttrMerger			N:1		o			rule A2Target{ from s : MM!A to t : TMM!T (feature1 <- s.a, feature2 <- s.b) }	helper context MM!A def a(i) : String = return inverseFunction(i); rule A2Target{ from s : MM!A to t : TMM!T (feature1 <- s.a(1), feature2 <- s.a(2)) }
		HAttrMerger			N:1					rule A2Target { from s : MM!A to t : TMM!T (feature <- s.a)} rule B2Target{ from s : MM!B to t : TMM!T (feature <-s.a)}	rule A2Target { from s : MM!A to t : TMM!T (feature <- s.a)} rule B2Target{ from s : MM!B to t : TMM!T (~~feature <-s.a~~)}
Reference	Constructive	CreateNewRef			0:1	+	+	-	-	-	-
		HRefSplitter			1:N		o			rule A2Target{ from s : MM!A to t : TMM!T (feature <- s.r) }	rule A2Target{ from s : MM!A to t : TMM!T (feature <- s.r->union(s.r2)) }
	Destructive	DeleteOldRef			1:0	-	-			rule A2Target{ from s : MM!A to t : TMM!T (feature <- s.r) }	rule A2Target{ from s : MM!A to t : TMM!T (~~feature <- s.r~~) }
		HRefMerger			N:1		o			rule A2Target{ from s : MM!A to t : TMM!T (feature <- s.r1, feature <- s.r2) }	helper context MM!A def r(i) : String = return splitFunction(i); rule A2Target{ from s : MM!A to t : TMM!T (feature <- s.r(1), feature <- s.r(2)) }

elements of the merged class A needs to produce the target elements of the rule that matches for the class X. Similar to the case of vertical partitioning during model co-evolution, the relationship that existed in MM_0 has to be considered. If a 1..1 relationship existed, the output pattern of the rule that matched for the class X may be added to the migrated rule (cf. VClassMerger). However, if a *..1 relationship existed, it has to be ensured that only for distinct values a target element is produced to be *semantically correct*. Thus, in ATL we may use a unique lazy rule that matches for distinct values only[1] (cf. DistVClassMerger). The remaining resolution actions follow the resolution actions presented for 1:N changes on

[1]Other rule based transformation languages would require to query the trace model to achieve this semantic.

models in order to get access to the before split elements, i.e., retyping of rules and adding of filters to match for the correct set of instances (cf. HClassMerger, DeleteSubclass and DeleteSuperclass in Table I).

C. Composition and Execution of Resolution Actions

After having introduced resolution actions on a conceptual level, this subsection presents the execution thereof as well as the co-evolution of the running example. Since interdependencies between resolution actions may exist, they must be applied in a certain order as detailed below and shown in Fig. 6.

Constructive Actions. First, constructive changes on EClasses are executed, since during co-evolution according objects or transformations rules are created, which might further on act as containers for features or bindings, respectively. In the running example, the composite change ExtractSuperclass contains the atomic change CreateSuperclass, which is applied first (cf. ① in Fig. 6). However, since the extracted class Element is abstract, no resolution is required. Additionally, constructive parts of destructive changes of transformation resolution actions need to be applied, since they might act again as container for subsequent bindings. In the running example, a DistVClassMerger has been applied to ensure unique Type instances, as it is the case for the output model of T_0. Thus, the unique lazy rule gets created, being the creative part of the DistVClassMerger (cf. ⑪ in Fig. 6).

Updative Actions. Although updative changes have not been discussed in this paper due to space limitations, they need to be considered for a comprehensive co-evolution. Thus, we refer to [9] and our work in [15] for a discussion of resolution actions for updatives changes on models and transformations. After having created the according containers, i.e., objects or rules, in a first step, updative changes commonly acting on features may be executed. During model co-evolution, the object c1 is migrated by changing the name of the reference and all Attribute instances get migrated by containing the values of the moved attribute Type.type. For transformation co-evolution, a virtual view on the old MM is provided by means of so-called adapter functions to resolve the changes ReferenceRenamed and MoveAttribute (cf. ② and ③ in Fig. 6).

However, whereas in the model co-evolution literature no specific order is enforced, a certain execution order for transformation co-evolution is required to ensure that the combination of adapters result in a valid OCL expression. Thereby, four combinations are possible. A rename and a move may be combined with either (i) no additional type change, (ii) a change of a collection type, (iii) a type switch from a collection type to a single value typed element, or (iv) a switch from a single value typed element to a collection type. Listing 1 shows an EBNF, which describes the composition of adapters updative resolution actions in ATL transformations.

Listing 1. Composition of Adapters

```
1  EmulatedView = HelperSignature "="
2   NoTypeSwitch |  CollectionTypeSwitch |
3   CollectionToSingleValueSwitch |
4   SingleValueToCollectionSwitch;
5  HelperSignature =
6   "helper context ≪featureV0.eContainingClass≫
7   def : ≪featureV0.name≫() : ≪featureV0.type≫";
8  Renaming =  ".≪featureV0.name≫" |
9              ".≪featureV1.name≫";
10 Moving = "" | ".≪reference≫";
11 NoTypeSwitch = "self" [Moving] Renaming;
12 CollectionTypeSwitch =  NoTypeSwitch (".asSet()" |
13   ".asBag()" | ".asSequence()" | ".asOrderedSet()");
14 CollectionToSingleValueSwitch =
15  NoTypeSwitch "->any()";
16 SingleValueToCollectionSwitch =
17  "≪featureV0.type≫{" NoTypeSwitch "}";
```

Destructive Actions. Finally resolution actions for destructive changes need to be applied, whereby features are deleted before classes. As can be seen in Figure 6 ⑦-⑨, according features or bindings are deleted before objects and according transformations rules get deleted (cf. Figure 6 ⑩-⑪).

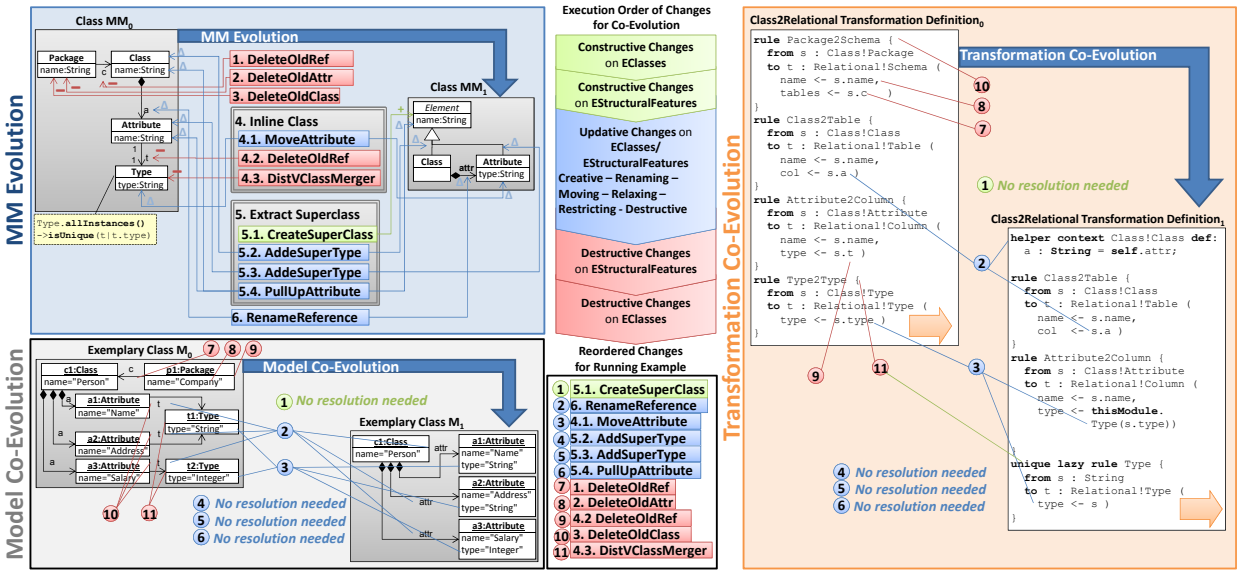


Figure 6. Application of Resolution Actions

V. A Toolchain for Consistent Co-Evolution

After having introduced a systematic set of MM changes and according resolution actions in the previous sections, this section presents our proof-of-concept prototype (cf. Fig. 7).

A. Change Detection

Our approach requires an Ecore-based MM, a conforming model as well as an ATL transformation, which uses the MM as source MM of the transformation (cf. ① in Fig. 7). In order to specify the evolution of the source MM, we follow an operation-based approach (cf., e.g., [8]). Thus, all atomic changes are recorded by the editor (cf. ② in Fig. 7) and represented as a change model conforming to our change MM in a first step. These changes are also visualized in the Change History view (cf. ③ in Fig. 7), which allows the user to resolve ambiguities by selecting an according change in a second step. If we consider our running example, the change Delete Class Type in position 5 in Figure 7 needs to be switched to an instance of VClassMerger instead of DeleteOldClass, resulting in an altered change model that reflects the desired semantics for co-evolution (cf. ④ in Fig. 7). Please note that manual disambiguation is only needed in case of atomic changes, since for composite changes the intended semantics is predefined in terms of a specific change combination. Since it is unlikely to model every desired composite change upfront, the presented mechanism is still necessary and therefore we consider the addition of new composite changes which are not yet predefined as future work. The resulting change model builds the basis for automatic model and transformation co-evolution.

B. Consistent Co-Evolution of Models and Transformations

Model Co-Evolution. Since numerous approaches for model co-evolution have been already developed, we try to reuse them in our prototype. We selected the model co-evolution language Epsilon Flock [19], since it provides (i) the necessary expressivity to define our changes and (ii) a concrete textual DSL which eases the implementation of the required transformation. In this respect, a model-to-text transformation implemented in XTend is applied, which uses the change model as input and produces an according Epsilon Flock script (cf. Listing 2 and ⑤ in Fig. 7). After executing this script, the co-evolved model M_1 is obtained (cf. ⑥ in Fig. 7).

Listing 2. Model Migration Script

```
1 migrate Class{ migrated.attr = original.a.equivalent();}
2 migrate Attribute{ migrated.type = original.t.type;}
```

Transformation Co-Evolution. In order to achieve transformation co-evolution, a Higher Order Transformation (HOT) [22] is provided (cf. ⑦ in Fig. 7), which takes the altered change model (cf. ④ in Fig. 7) and the transformation T_0 as input. After executing the HOT, the co-evolved transformation T_1 is obtained (cf. ⑧ in Fig. 7).

C. Validating Semantic Correctness

As discussed previously, resolution actions must ensure *syntactic & semantic correctness*. While the former may be verified by EMF for models or by the ATL compiler for transformations, the verification of the latter is more challenging, especially for transformations. For verifying the semantic correctness of transformations, regression testing is a common mechanism [4]. Thus, we verify semantic correctness by properties expressed in our PaMoMo language [7], which must be fulfilled by a transformation, thereby relaxing the strong condition of generating the exactly same output model (due to destructive changes) and by this, enabling for the verification of semantic correctness for the presented changes. Another means to verify correctness could be dedicated verification conditions derived from OCL pre- and postconditions of the change set. However, this is currently left to future work.

PaMoMo in a Nutshell. PaMoMo provides a visual, declarative, formal specification language to describe correctness requirements for transformations (cf. [7] for details). A PaMoMo specification consists of declarative visual patterns, which may be *positive*, i.e., describing necessary conditions to occur (denoted by a "P"), or *negative*, which state forbidden situations (denoted by an "N"). Patterns are composed of two compartments containing object graphs typed on the source MM (left compartment) or target MM (right compartment). 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 [17], 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 semantic correctness of a transformation definition.

PaMoMo for the Running Example To be able to verify the semantic correctness of the co-evolved model transformation, the PaMoMo patterns defined for the transformation T_0 must be co-evolved as well first. Since PaMoMo patterns are specified by means of object graphs, the resolution strategy employed for existing models may be re-used to evolve the patterns (cf. Fig. 8). The first positive pattern states that for every Class instance in the source model an equally named Table instance must exist in the target model. The second pattern states, that for every Attribute instance contained in a Class, the equally named Table must contain a column that is equally named as the Attribute. Additionally, the Column instance must refer to a Type instance, whose attribute type is set to the value of Attribute.type. The co-evolved patterns may then be used to check if the co-evolved transformation maintains semantic correctness by checking if pairs of the co-evolved input models and produced output models fulfill the requirements stated by the PaMoMo patterns. After having executed the patterns, a report is provided, stating if a semantically correct co-evolution has been achieved, as is the case for our running example (cf. ⑨ in Fig. 7).

VI. Related Work

Few approaches for the (semi-)automatic co-evolution of model transformations in response to MM evolution have been

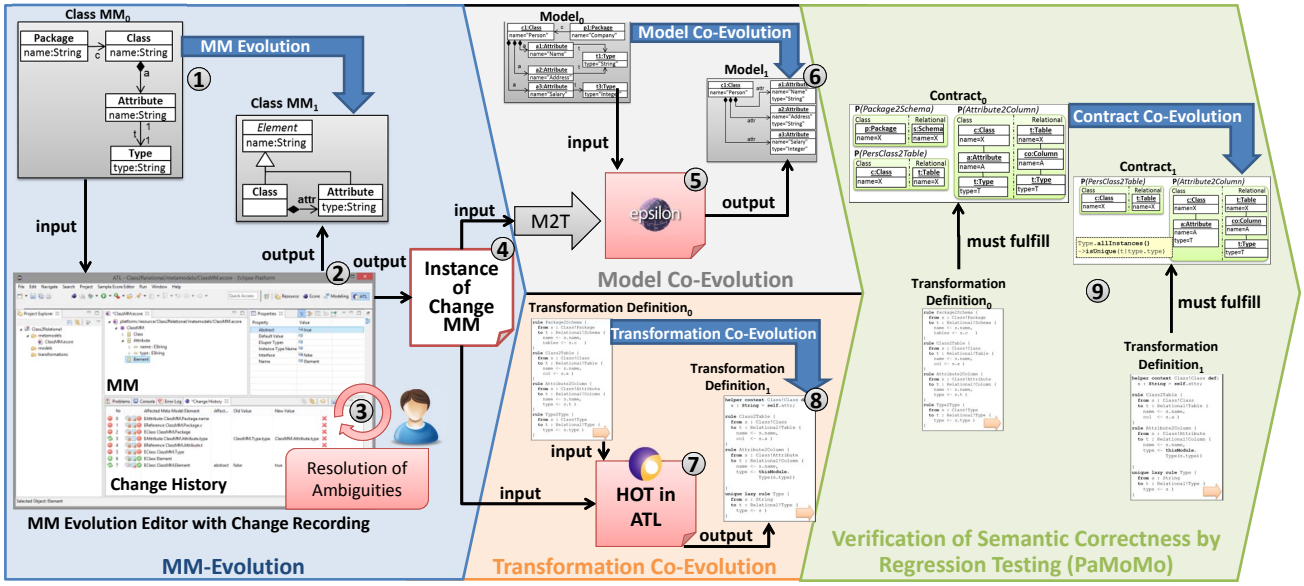


Figure 7. Exemplary Toolchain

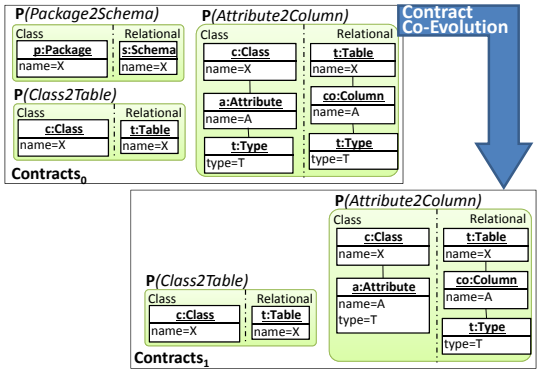


Figure 8. Co-Evolved PaMoMo Patterns

proposed in the last years. In [13], a fixed set of changes and corresponding resolution actions for the evolution of MM and the co-evolution of transformations is presented, comprising 14 atomic and 2 composite changes. Due to this limited set, only a limited number of MM evolutions is possible. Furthermore, resolution actions for composite changes are monolithic, i.e., atomic changes are not reused for building more complex changes, in contrast to our approach which is both extensible with respect to the set of composite changes and resolution actions for atomic changes can be reused. Moreover, the dependency to model co-evolution is not explicated, therefore, inter-artifact inconsistencies may arise. Garces et al. [5] use transformation chains to emulate the functionality of the evolved transformations in response to metamodel evolution, i.e., the original transformation is not rewritten, but specific adaptation transformations that emulate a particular change are chained into a transformation. In contrast to our approach, adaptation transformations have to be redefined for new MM evolution scenarios, whereas in our approach resolution actions for a complete set of atomic changes are predefined and can be reused. Furthermore, inter-artifact consistency is not considered since the approach focuses solely on transformation co-evolution. Garcia et al. [6] present an approach for co-evolution of transformations by proposing resolution actions realized by means of a HOT for a fixed set of atomic and composite changes, thereby rewriting the original transformation. The approach is able to automatically co-evolve transformations except for constructive changes, for which a code skeleton is generated, only. However, resolution actions for atomic changes can not be reused and the set of changes is not extensible. Additionally, model co-evolution is not considered, which may lead to inter-artifact inconsistencies during co-evolution. In [3] the authors propose a DSL for defining resolution actions for different kinds of artifacts, which allows to build libraries of these resolution actions. However, those actions have to be defined by the user, whereas our approach comes with predefined resolution actions for all atomic changes that can be assembled to composite changes.

In summary, our approach is unique with respect to the possibility to extend the change set by composing atomic changes and by this reusing accompanying resolution actions, thereby ensuring intra-artifact consistency. Furthermore, another unique aspect is its exploitation of model co-evolution semantics in order to ensure inter-artifact consistency.

VII. Lessons Learned and Critical Discussion

In this section, we report on lessons learned, which reflect the approach critically, and discuss points for future work.

Composable Resolution Actions of Changes Ensure Intra-Artifact Consistency. One unique key characteristic of our approach is that resolution actions for composite changes are not defined as monolithic units, but as a composition of resolution actions of the contained atomic changes. By this, an *intra-artifact-consistent* co-evolution is achieved, since for a specific atomic change only a single definition of a co-evolution action exists, which is reused, accordingly. Consequently, code duplication is avoided and by this, the threat of potential inconsistencies between redundant resolution code.

Consistent Co-Evolution Semantics Assure Inter-Artifact Consistency. Consistent co-evolution semantics for both, models and transformations, assure *inter-artifact consistency*, since both kinds of artifacts are co-evolved following the same co-evolution semantics. However, the actual translation of these semantics into resolution actions is defined in the code generators, i.e., through our XTend code for generating model resolution actions and the HOT for generating resolution actions for ATL transformations. Therefore, supporting new kinds of artifacts, besides models and transformations, requires the implementation of new code generators that realize the needed artifact-specific resolution actions.

Atomic Changes Foster Reusability. To achieve empirical evidence on the reusability of atomic changes across composite changes, we conducted a small empirical study on the well-known catalogue of change operations of Herrmannsdoerfer et al. [9]. Thereby, we took a sample of 12 composite changes and decomposed them into atomic changes in such a way that the achieved co-evolution semantics corresponds to the one proposed in the catalogue. This resulted in a total set of 35 atomic changes, of which 23 are unique. Thus, 12 atomic changes have been reused, yielding to a substantial reuse factor of 34.3 %. Furthermore, of the 23 unique changes, 9 atomic changes have been reused among different composite changes, yielding to a reuse factor among the atomic changes of 39.1 %, thus, one may conclude that our proposed set of atomic changes really fosters reusability.

Extending Composite Change Set to Gain Evolution/Co-Evolution Library. While some exemplary composite changes have been shown in Fig. 5, our approach is not limited to them. Therefore, the definition of new composite changes or implementing a set of proposed composite changes (cf., e.g., [9], [11]) may result in a comprehensive evolution/co-evolution library, easing co-evolution by picking changes from a comprehensive set of already defined composite changes with distinct co-evolution semantics for dependent artifacts. However, a dedicated composition language would be needed which should be integrated into the proposed toolchain, which is one direction for future work. To allow for such a language, we are currently working on a more formal specification of applicability conditions for the resolution actions in terms of OCL in order to be able to validate composition. Additionally, to provide a comprehensive library, additional transformation languages should be incorporated. Thus, we are currently investigating on specifying resolution actions for our changes for graph-transformation languages (e.g., TGGs). An additional point of future work is to consider multi-metamodel co-evolution, i.e., the evolution of source and target MM of a transformation, since we currently focused on evolving the source MM, only.

ACKNOWLEDGMENTS

This work has been funded by the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT) grant FFG FIT-IT 829598, FFG BRIDGE 838526 and 832160, FFG COIN 845947, Austrian Science Fund (FWF) 28187-N31, OeAD Marietta Blau grant ICM-2014-08519, OeAD grant AR18/2013, UA07/2013, AR10/2015, and by ERC grant PIOF-GA-2012-328378.

REFERENCES

[1] Bézivin, J.: On the Unification Power of Models. SoSym 4(2) (2005)
[2] Cicchetti, A., Di Ruscio, D., Pierantonio, A.: Managing Dependent Changes in Coupled Evolution. In: Proc. of the ICMT. Springer (2009)
[3] Di Ruscio, D., Iovino, L., Pierantonio, A.: Evolutionary Togetherness: How to Manage Coupled Evolution in Metamodeling Ecosystems. In: Proc. of the ICGT. Springer (2012)
[4] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
[5] Garcés, K., Vara, J., Jouault, F., Marcos, E.: Adapting transformations to MM changes via external transformation composition. SoSym (2013)
[6] García, J., Diaz, O., Azanza, M.: Model Transformation Co-evolution: A Semi-automatic Approach. In: SLE. Springer (2013)
[7] Guerra, E., de Lara, J., Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W.: Automated verification of model transformations based on visual contracts. Journal of Automated Softw. Eng. 20(1) (2012)
[8] Herrmannsdoerfer, M.: COPE A Workbench for the Coupled Evolution of Metamodels and Models. In: SLE. LNCS, vol. 6563. Springer (2011)
[9] Herrmannsdoerfer, M., Vermolen, S.D., Wachsmuth, G.: An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models. In: SLE, LNCS, vol. 6563. Springer (2011)
[10] Herrmannsdoerfer, M., Wachsmuth, G.: Coupled Evolution of Software Metamodels and Models. In: Evolv. Softw. Systems. Springer (2014)
[11] Iovino, L., Di Ruscio, D., Pierantonio, A.: Metamodel Refactorings Catalog. `http://www.metamodelrefactoring.org/` last accessed 07-05-2015 (2012)
[12] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1–2) (2008)
[13] Kruse, S.: On the Use of Operators for the Co-Evolution of Metamodels and Transformations. In: ME Workshop @ MODELS (2011)
[14] Kusel, A., Etzlstorfer, J., Kapsammer, E., Retschitzegger, W., Schönböck, J., Schwinger, W., Wimmer, M.: A Systematic Taxonomy of Metamodel Evolution Impacts on OCL Expressions. In: Int. Workshop on Models and Evolution @ MODELS (2014)
[15] Kusel, A., Etzlstorfer, J., Kapsammer, E., Retschitzegger, W., Schönböck, J., Schwinger, W., Wimmer, M.: Systematic Co-Evolution of OCL Expressions. In: Proc. of the 11th APCCM (2015)
[16] Mantz, F., Taentzer, G., Lamo, Y.: Well-formed Model Co-evolution with Customizable Model Migration. Electronic Communications of the EASST 58 (2013)
[17] Object Management Group: Meta Object Facility (MOF) Query/View/-Transformation (QVT). http://www.omg.org/spec/QVT/1.1 (2011)
[18] Ozsu, M. Tamer, V.P.: Principles of Distributed Database Systems. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edn. (2007)
[19] Rose, L., Kolovos, D., Paige, R., Polack, F.: Model Migration with Epsilon Flock. In: Proc. of the ICMT. Springer (2010)
[20] Rose, L., Paige, R., Kolovos, D., Polack, F.: An Analysis of Approaches to Model Migration. In: Proc. Joint MoDSE-MCCM Workshop (2009)
[21] Schönböck, J., Kusel, A., Etzlstorfer, J., Kapsammer, E., Schwinger, W., Wimmer, M., Wischenbart, M.: CARE – A Constraint-Based Approach for Re-Establishing Conformance-Relationships. In: Proc. of the APCCM (2014)
[22] Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the Use of Higher-Order Model Transformations. In: Proc. of ECMDA-FA'09. Springer (2009)
[23] Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W., Kolovos, D., Paige, R., Lauder, M., Schürr, A., Wagelaar, D.: Surveying Rule Inheritance in Model-to-Model Transformation Languages. JOT 11(2) (2012)
[24] Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W.: Towards an Expressivity Benchmark for Mappings based on a Systematic Classification of Heterogeneities. In: Proc. of the Int. Workshop on Model-Driven Interoperability (2010)