# On using Inplace Transformations for Model Co-evolution*

M. Wimmer[1], A. Kusel[2], J. Schoenboeck[1],
W. Retschitzegger[2], W. Schwinger[2], and G. Kappel[1]

[1] Vienna University of Technology, Austria
{lastname}@big.tuwien.ac.at
[2] Johannes Kepler University Linz, Austria
{firstname.lastname}@jku.at

**Abstract.** Metamodel evolution and model co-evolution are considered to be essential ingredients for the successful adoption of model-driven engineering in practice. In this respect, on the one hand, dedicated co-evolution languages have been proposed for migrating models conforming to an initial metamodel to models conforming to a revised metamodel with the drawback of requiring to learn a new language. On the other hand, the employment of dedicated model-to-model transformation languages has been proposed demanding for the specification of rules for copying unchanged elements.
In this paper, we propose to tackle the co-evolution problem from a different viewpoint. Instead of describing the co-evolution of models as a transformation between two metamodels, we employ existing inplace transformation languages. For this, the prerequisite is to represent both language versions within one metamodel which is automatically computed by merging the initial and the revised metamodel. This ensures that the initial as well as the revised model conform to the merged metamodel, enabling the employment of inplace transformations for initializing new metamodel elements. Finally, a check-out transformation is used for eliminating model elements which are no longer covered by the revised metamodel. We demonstrate this idea by using ATL for merging the metamodels and realizing the check-out transformation. Furthermore, we discuss the ATL refinement mode for co-evolving the models.

**Key words:** Model Co-evolution, Metamodel Merging, Inplace Transformations

## 1 Introduction

Metamodel evolution and model co-evolution are considered to be essential ingredients for the successful adoption of Model-Driven Engineering in practice. Especially, when domain-specific modeling languages are employed, the necessity of language adaptations arise to reflect changes in the modeling domain

* This work has been funded by the Austrian Science Fund (FWF) under grant P21374-N13.

as well as in technologies without losing existing models. Thus, (i) dedicated co-evolution languages (e.g., COPE [10]) and (ii) the usage of model-to-model (M2M) transformation languages (cf. [6] for an overview) have been proposed to migrate models conforming to an initial metamodel $MM$ to models conforming to a revised metamodel $MM'$. However, this requires to learn a new language in the first case and to copy the entire model—also the elements which are not affected by metamodel changes—in the second case.

In this paper, we tackle the co-evolution problem from a different viewpoint. Instead of describing the co-evolution of models as transformation between two metamodels, we employ existing inplace transformation languages for this task. For being able to employ inplace transformations, the prerequisite is to represent both language versions within one metamodel which is automatically computed by merging the initial and the revised metamodel. Thereby, it is ensured that the original model as well as the to be evolved model always conform to the merged metamodel. After performing the co-evolution by specifying inplace transformation rules which add or update elements according to the revised metamodel, the co-evolution process is completed by a fully automated check-out transformation eliminating model elements which are no longer covered by the revised metamodel. We demonstrate this idea by using ATL for merging the metamodels as well as for the check-out transformation. Furthermore, we discuss the ATL refinement mode for co-evolving the models. Please note that the focus of this paper is how to support breaking and (non-)resolvable metamodel changes [9] by inplace transformations for reflecting these changes on existing models and not on discussing the possible impacts of metamodel changes.

The remainder of this paper is structured as follows. Section 2 introduces a co-evolution scenario which is used as running example throughout the paper. Section 3 presents (i) the conceptual architecture of our approach, (ii) the metamodel merging algorithm, (iii) the co-evolution rules expressed as inplace transformations, and (iv) how the check-out transformation is derived. Related work is discussed in Section 4, and finally, the paper is concluded with an outlook on future work in Section 5.

## 2 Motivating Example

To elaborate on our co-evolution approach, we introduce in this section an evolution scenario, namely the evolution of a small domain specific modeling language which is inspired by the UML class diagram language. Fig. 1 illustrates the scenario which serves as a running example throughout the rest of the paper. In the upper part of Fig. 1, the initial metamodel $MM$ is illustrated. The metamodel comprises four modeling concepts being `Class`, `Reference`, `ID_Attribute` (identifying attributes), and `Desc_Attribute` (descriptive attributes). A `Class` may extend another `Class` (reference `extends`) and may be marked as being abstract (attribute `isAbstract`). A `Reference` has an `opposite` reference, actually representing a bi-directional association, and comprises an upper multiplicity (attribute `upperMult`) as well as a lower multiplicity (attribute `lowerMult`).

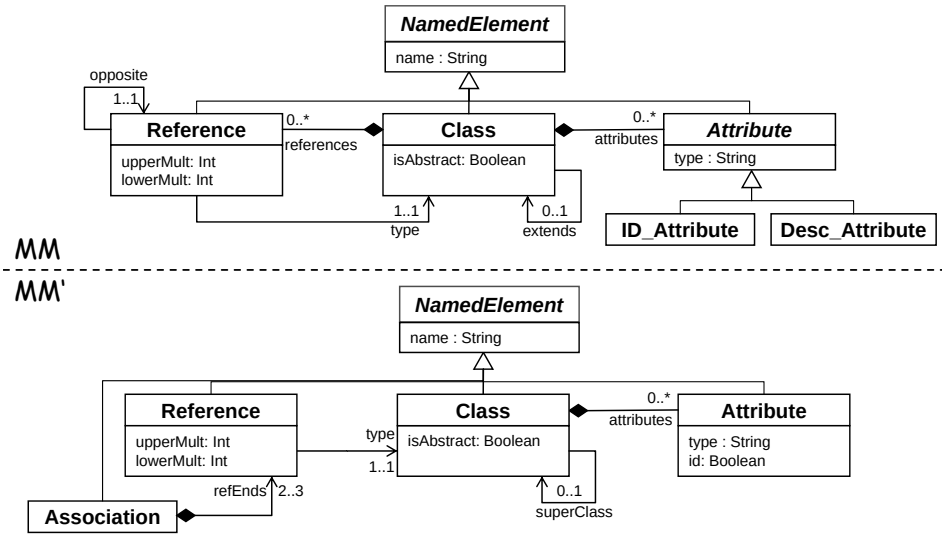


**Fig. 1.** Running Example - Class Diagram: Initial Metamodel ($MM$) and Revised Metamodel ($MM'$).

Imagine that after using the language for several projects, the modelers requested several changes of the language due to their experiences and needs:

- **Change Request 1: Modeling of ternary associations**. The modelers request that they need, in addition to bi-directional associations, the possibility to describe ternary associations.
- **Change Request 2: Change attribute kind dynamically**. The modelers have explored that during establishing models it is often necessary to change the attribute kind from identifying to descriptive and vice versa. This is currently only possible by deleting the attribute and by creating a new one.
- **Change Request 3: Extends reference is too Java-specific**. Some modelers complaint that the inheritance feature of the modeling language should be named `superClass` instead of `extend` to be more platform independent.

The lower part of Fig. 1 illustrates the revised metamodel $MM'$ incorporating the requested changes. For reflecting change request 1, a new class `Association` is introduced which may contain two or three references. The subclasses of the class `Attribute` are deleted and the class `Attribute` is changed to a concrete class. For distinguishing between identifying and descriptive attributes, the class `Attribute` now holds an additional attribute named `id` of type Boolean.

For migrating existing models to the revised metamodel $MM'$, the changes have also to be reflected on the model level. Therefore, four co-evolution rules (please note that change request 2 results in rule 2 and in rule 3) are needed

to transform the existing models conforming to $MM$ to models conforming to $MM'$.

- **Rule 1: Create associations for reference pairs**. For each reference pair, i.e., two references which link each other as opposite references, an association has to be created which contains both references.
- **Rule 2: Convert identifying attributes**. For each identifying attribute, i.e., instance of `ID_Attribute`, an instance of class `Attribute` has to be created whereby the `id` attribute has to be set to true.
- **Rule 3: Convert descriptive attributes**. For each descriptive attribute, i.e., instance of `Desc_Attribute`, also an instance of class `Attribute` has to be created but with the difference that the `id` attribute has to be set to false.
- **Rule 4: Set Class.superClass reference**. Finally, for each class, the new reference `superClass` in $MM'$ has to be set according to the existing `extends` reference in $MM$.

## 3 Co-evolution as Inplace Transformation Problem

In this section, we elaborate on how to apply inplace transformations for co-evolution of models if changes to the metamodel have been made. As we have seen in the previous section, some metamodel elements remained unchanged, some have been deleted, and some have been introduced. Thus, if we want to migrate a model conforming to the initial metamodel to a model conforming to the revised metamodel, we have to copy some model elements, we have to delete some, and some new elements have to be created. However, this viewpoint is inspired from existing approaches which tackle co-evolution by creating completely new models which conform to the revised metamodel from existing ones.

In our opinion, inplace transformations would be a natural choice when the evolution of a model has to be described. However, due to the fact that we have two different metamodels[3], inplace transformations are currently not used for this task. This is quite different to other model evolution scenarios where only one metamodel is used such as describing the simulation/execution of models.

For employing inplace transformations also for co-evolution scenarios, we propose the following process which is shown on a conceptual level in Fig. 2. First, in order to use inplace transformations, we need a special kind of unification of both metamodel versions (cf. Step 1 in Fig. 2). The unification is done in a way that the existing models $M$, which conform to the initial metamodel $MM$, also conform to the unified metamodel $MM \cup MM'$. By this, we do not have to cope with any copy operations for the constant parts of the model (i.e., instances of the unchanged metamodel elements). The unified metamodel is now the key to employ inplace transformations for initializing the newly introduced metamodel elements of *MM'* in the initialization phase (cf. Step 2 in Fig. 2). The output of

[3] Even though, the differences between the metamodels are minimal in most cases, technically we are concerned with two metamodels.

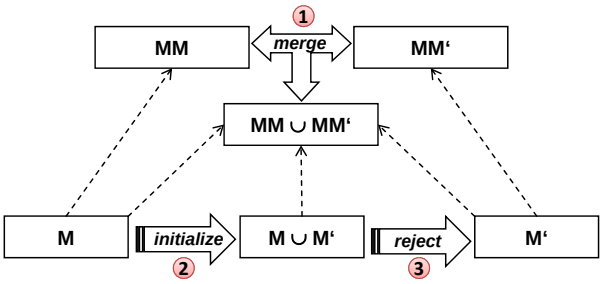


**Fig. 2.** Conceptual Architecture of the Co-evolution Approach

the inplace transformation is the initial model which is extended with instances of the new metamodel elements created by the rules of the inplace transformation. The rejection of old elements (cf. Step 3 in Fig. 2), i.e., elements which are only covered by *MM*, is done by a M2M transformation which is automatically computed by matching the unified metamodel with the revised metamodel and by generating a transformation rule for each match.

In the following we describe the approach in more detail. First, we elaborate on how the two metamodels are merged into one metamodel unifying both language versions. Second, we present how to use inplace transformations for instantiating new model elements which are derived from the existing context. Third, we discuss how to match the merged metamodel with the revised metamodel to derive all necessary transformation rules to check-out a model which solely conforms to the revised metamodel, i.e., all elements which are no longer covered by the metamodel are rejected.

### 3.1 Prerequisite: Metamodel Merging

The prerequisite for employing inplace transformations is that we have a unified metamodel which is capable of representing the commonalities of both metamodel versions as well as their differences. In Fig. 3, the co-evolution of models is illustrated by using set theory. We start with a model which consists of elements which are all covered by the initial metamodel *MM* as well as partly by the revised metamodel *MM'*. From this set, additional elements are computed which are only covered by *MM'*, however, the model still contains elements covered only by *MM*. Finally, these elements are rejected by the check-out transformation which produces a model which only consists of elements covered by *MM'*.

In order to support this process, the unified metamodel must be created as explained in the following. First, only corresponding elements, i.e., classes, attributes, and references, are merged into one element. For example, if two attributes have the same name and the same type, but a different cardinality, they cannot be merged into one element. The reason for this is that if they are merged into one element, we cannot represent the different cardinalities. Therefore, they have to be both incorporated in the unified metamodel by independent elements.

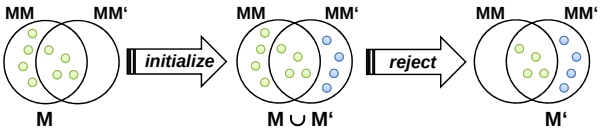


**Fig. 3.** Model Co-evolution from a set-theoretic point of view.

In the following we explain the merge algorithm which is shown as pseudo code in Algorithm 1. Please note that we only consider classes and their features, i.e., attributes and references, in the pseudo code whereas other elements of MOF such as packages are not discussed in the paper.

First of all classes are merged together if they have the same name, since we assume that the class name being unique. If two classes are found with the same name, the merge of these two classes is achieved by introducing a new class which gets the union of the features and superclasses of both classes. Furthermore, the class is only defined as abstract class if both classes are abstract. If no name match is found, the class is directly added to the merged metamodel.

In contrast to classes which have only to correspond concerning their names, features have to correspond totally. This means that they have to be equivalent in all their meta-features such as name, type, multiplicity, unique constraints and so on. Note that if an equally named attribute occurs in the original and in the revised version but they offer, e.g., a different type, both attributes have to be considered in the merged metamodel. If no total match is found, the feature is simply added to the merged metamodel.

Fig. 4 shows the unified metamodel for our running example. Please note that due to readability, elements which are only contained by the initial metamodel are shown in red (for indicating a deletion), elements which are only contained by the revised metamodel are shown in green (for indicating an addition), and elements which are contained by both metamodels are shown in black.

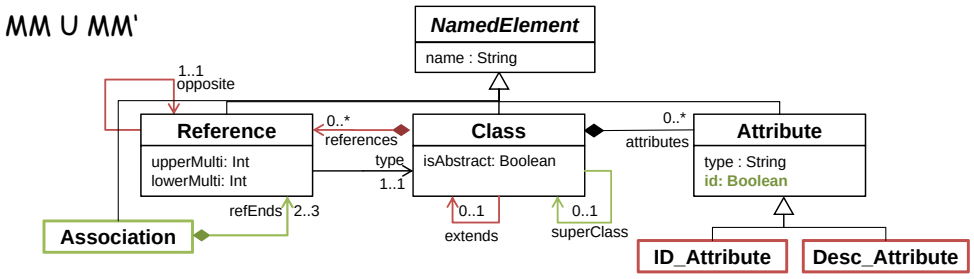


**Fig. 4.** Merged metamodel for the metamodels of Fig. 1

```
Input: initialMM, revisedMM
Output: mergedMM
// create empty mergedMM
1  mergedMM = new MM()
// Add classes to mergedMM
2  for originalClass ∈ Class.allInstancesFrom (initialMM) do
3  |  Class revisedClass = findEquivalentClass (revisedMM, originalClass)
4  |  if revisedClass <> OclUndefined then
5  |  |  Class modifiedClass = new Class ()
6  |  |  modifiedClass.name = originalClass.name
7  |  |  modifiedClass.abstract = originalClass.abstract and
   |  |  revisedClass.abstract
8  |  |  modifiedClass.features =
   |  |  originalClass.features.union(revisedClass.features)
9  |  |  modifiedClass.superClasses =
   |  |  originalClass.superClasses.union(revisedClass.superClasses)
10 |  |  mergedMM.add(modifiedClass.Annotate ('both'))
11 |  end
12 |  else
13 |  |  mergedMM.add(originalClass.Annotate ('old'))
14 |  end
15 end
16 for revisedClass ∈ Class.allInstancesFrom (revisedMM) do
17 |  Class originalClass = findEquivalentClass (initialMM, revisedClass)
18 |  if originalClass = OclUndefined then
19 |  |  mergedMM.add(revisedClass.Annotate ('new'))
20 |  end
21 end
// Add features to mergedMM
22 for originalFeature ∈ Feature.allInstancesFrom (initialMM) do
23 |  Feature revisedFeature = findEquivalentFeature (revisedMM,
   |  originalFeature)
24 |  if revisedFeature <> OclUndefined then
25 |  |  getContainer (originalFeature).add(originalFeature.Annotate
   |  |  ('both'))
26 |  end
27 |  else
28 |  |  getContainer (originalFeature).add(originalFeature.Annotate
   |  |  ('old'))
29 |  end
30 end
31 for revisedFeature ∈ Feature.allInstancesFrom (revisedMM) do
32 |  Feature originalFeature = findEquivalentFeature (initialMM,
   |  revisedFeature)
33 |  if originalFeature = OclUndefined then
34 |  |  getContainer (revisedFeature).add(revisedFeature.Annotate ('new'))
35 |  end
36 end
```

**Algorithm 1:** Merge Algorithm

We have implemented the merge algorithm as an ATL model-to-model transformation by using only declarative rules. The transformation takes two input models, i.e., the initial metamodel and the revised metamodel, and one output model, i.e., the unified model. The complete ATL code and the example models for testing the code can be found on our project page[4].

*Instantiation conformance.* Because we copy each element of the initial metamodel into the revised metamodel, the existing models conform to the merged metamodel. However, there are some exceptions, namely required features which are coming from the revised metamodel. For example, please imagine that the `superClass` reference in Fig. 4 is mandatory. Of course, the existing models would not contain such links, thus, we would have a violation of the metamodel constraints. However, these violations are only requiring additional elements which are currently not present. All present elements are conforming to a subset of the merged metamodel, i.e., the initial metamodel. What we have explored is that current modeling tools as well as transformation engines ignore violations which require further elements in the model. These issues are only reported by executing additional validation checks, but fortunately the models are loadable and usable in current modeling tools and transformation engines.

### 3.2 Instantiating new Metamodel Elements with Inplace Transformations

After generating a unified metamodel as discussed in the previous subsection, we are now able to instantiate the introduced metamodel elements from the context of the initial model with the help of inplace transformations. Fig. 5 shows the necessary transformation rules on a conceptual level in the AGG graph transformation syntax [15]. Please note that Rule 3 is not shown, because it is analogous to Rule 2. As can be seen in this figure, the inplace transformation rules are very concise and naturally to develop with existing inplace transformation language constructs. In the following we shortly discuss the functionality of each rule.

Rule 1 is used for instantiating the new class `Association`. As can be seen in the LHS of the rule, we match for all reference pairs which are connected via an opposite link. For each match, an association is introduced in the model which is connected to matched references (cf. RHS of Rule 1) if no association already exists which links the matched references. This is ensured by the negative application condition (NAC). The second rule is used for creating an instance of `Attribute` for each instance of `ID_Attribute`. This is done by defining such a new instance on the RHS and connecting this instance to the class which also contains the id attribute. Finally, Rule 4 simple generates for each `extends` link between two classes an additional `superClass` link.

*Implementation with ATL Refinement Mode.* We have implemented the rules shown in Fig. 5 in ATL using the refinement mode. The resulting transformation definition for the co-evolution rules is presented in Listing 1.1. Whereas Rule 4 can be implemented by the provided language constructs and execution

[4] www.modeltransformation.net

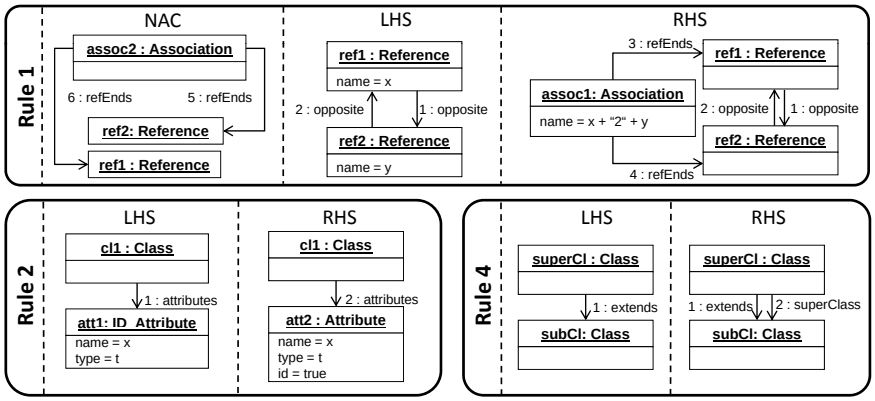


**Fig. 5.** Co-evolution rules expressed as graph transformation

semantics of the ATL refinement mode, the Rules 1-3 can not as detailed in the following.

**No Type Changes.** For Rule 2 and 3, we explored that it is not possible to write the ATL rules as concise as it is shown in Fig. 5. For example, for creating an `Attribute` for each `ID_Attribute`, we started with a declarative rule consisting of a `from` block for matching the `ID_Attributes` and a `to` block for generating the `Attributes`. However, when executing the described rule, we ended up again with `ID_Attributes` in the output model, only. It was neither possible to upcast the existing instances nor was it possible to generate new attribute instances. Thus, we modified the `to` block for dealing with the existing id attributes by the first target pattern (cf. line 19) and an additional target pattern for creating new attributes (cf. line 20).

**No Queryable Execution State.** Another issue we encountered was the unsupported query of the transformation state which is in particular necessary for implementing the NAC of Rule 1. As we have mentioned before, we have to check when generating an association, if an association has been already created for a reference pair (each pair is matched twice because of binding a particular reference to variable `ref1` in one match and to variable `ref2` in another). This check was not possible for us to implement as additional guard clause due to the fact that only the initial input model can be matched.

**No Imperative Code.** For overcoming the restriction mentioned above, we tried to solve this issue by storing the already matched `References` in a global helper. Nevertheless, it was not possible to add new elements to the global helper during transformation, since no imperative language constructs, i.e., calls in the `do` block, are supported in the refining mode.

### 3.3 Getting rid of out-dated model elements

After initializing the newly introduced metamodel elements of the revised metamodel, it is now time to modify the extended model in order to conform solely

to the revised metamodel (cf. reject arrow in Fig. 2 and 3). This step is automatically achieved in our approach by comparing the unified metamodel with the revised metamodel by using a matching transformation inspired by [7]. Because the merge described in Subsection 3.1 is based on one-to-one correspondences, it is now again possible to use exact one-to-one correspondences as matches between the unified and the revised metamodel. Thus, we only have to compare both metamodels with a state-based comparison and generate for each class match (defined by equal class names) a declarative ATL rule and for each feature match (defined by equal qualified feature names) an assignment statement. The resulting M2M transformation is so to say a projection transformation, i.e., only a subset of the extended model is transformed to the new model. The matching transformation can be found again on our project web site.

**Listing 1.1.** Co-evolution rules expressed in ATL

```
module CoEvolution;
create OUT : MM refining IN : MM;

--Rule 1
rule GenerateAssociations {

    from ref1 : MM!Reference,
      ref2 : MM!Reference (ref1.opposite = ref2) -- missing NAC

    to dummy : MM!Reference,
      assoc : MM!Association (refEnds <- Set{ref1,ref2})
}

--Rule 2
rule GenerateAttributes {

  from id_att : MM!ID_Attribute

  to dummy : MM!ID_Attribute,
      att : MM!Attribute (
              name <- id_att.name,
              type <- id_att.type,
              id <- true,
              class <- id_att.class
      )
}

--Rule 3 is analogous to Rule 2

--Rule 4
rule setSuperClassRefs {

  from c1 : MM!Class

  to c2 : MM!Class (
    superClass <- c1.extends
  )
}
```

## 4 Related Work

Co-evolution has been subject for research since the introduction of the first object-oriented database systems [2], consequently a huge amount of approaches

for dealing with this issue have been proposed (cf. [14] for a survey). In this section we therefore focus on most closely related approaches, only, i.e., approaches dedicated to reflecting changes of metamodels on the model layer in the field of model-driven engineering.

Garces et al. [8] propose a set of heuristics to automatically compute equivalences and differences between two metamodel versions in order to adopt models to their evolving metamodels and thus follow a matching approach to co-evolution. The computed equivalences and differences are stored in a so-called matching model, acting as input for a higher-order transformation [16], producing an executable adaptation transformation.

Cicchetti et al. [5] present a similar approach to Garces et al., i.e., the approach is again based on a metamodel difference representation, which acts as input for a higher-order transformation. Moreover, the computed differences are classified into (i) non-breaking changes, (ii) breaking and resolvable changes, and (iii) breaking and unresolvable changes, further structuring the higher-order transformation.

Wachsmuth [17] proposes to combine ideas from object-oriented refactoring and grammar adaptation to provide the basis for automatic metamodel evolution. In this respect, metamodel relations are defined, building the basis for the definition of semantics preservation and instance preservation. Moreover, a set of transformations based on QVT Relations [12] are proposed, which are accordingly classified into refactoring, construction, and destruction transformations.

In [1, 11] a co-evolution approach for models using the Model Change Language (MCL) is presented. MCL is declarative and graphical language supporting a set of co-evolution idioms. The evolver defines relationships between elements of the metamodel versions. There are different kind of relationships starting from simple one-to-one mappings between classes to more complex mappings for typing objects to new subclasses or for changing the containment hierarchy. More complex co-evolution rules have to be defined in the context of the defined mappings as constraints and actions in terms of C++ code. For unchanged parts of the metamodel no mappings have to be defined due to the usage of name equivalences. It has to be mentioned that the provided idioms only cover syntactical co-evolution concerns but not semantical concerns and that the co-evolution is achieved again by a model-to-model transformation.

Herrmannsdoerfer et al. propose in [10] COPE, an integrated approach to specify the coupled evolution of metamodels and models in order to reduce the migration effort. In this respect, the co-evolution of metamodels and corresponding models is realized by a set of so-called coupled transactions, composing a whole co-evolution problem of modular transformations. The coupled transactions are further divided into custom coupled transactions and reusable coupled transactions, whereby as the name reveals reusable coupled transactions are predefined and have not to be specified by the user, thereby reducing the migration effort. Moreover, the authors argue to provide the needed expressivity by custom coupled transactions expressed by primitives embedded into the Turing-complete language Groovy.

Summarizing, our approach is different to the mentioned approaches, because we tackle co-evolution of models with existing inplace transformation languages instead of using domain-specific languages or M2M transformation languages. By using our specialized metamodel merging algorithm, we do not have to copy elements which are resistant to the metamodel changes as is also supported by COPE. However, our approach allows the automatic elimination of old model elements which are no longer covered by the revised metamodel, which in contrast has to be manually developed with COPE. The first four approaches use M2M transformation languages which are partly supported by automatic derivation of transformations. However, the non-automatically derived parts have to be manually defined which seems to be more challenging for the user in comparison to using inplace transformations, because s/he has to reason on how elements look like in the source model, how elements are represented in the target model, and how they are transformed by analyzing the trace information enforcing the user to work with three models. In contrast, in our approach, only one model is necessary for defining the co-evolution rules by using our unified metamodel in combination with inplace transformations. For computing the check-out transformation for generating the finally co-evolved models, we are using a similar approach as Graces et al. [8] for producing the adaptation transformation, but with the difference that we do not have to rely on heuristics. This is due to the fact that we only have one-to-one correspondences which are unambiguously derivable. Finally, currently we do not support the automatic generation of co-evolution rules for breaking and resolvable changes as is supported by the first two mentioned approaches. We see this support as an orthogonal concern and as subject to future work.

## 5 Conclusion and Future Work

In this paper we have reported on our first experiments using inplace transformations for model co-evolution. Our experiences indicates that co-evolution of models may be easily described with current inplace transformation languages. However, the ATL refinement mode currently has some limitations compared to the M2M mode which complicate the definition of co-evolution rules. An improvement for defining co-evolution rules would be to allow for imperative parts in the ATL refinement mode which are possible in M2M ATL transformations and to match also on intermediate transformation results.

For applying inplace transformations, we provide an automatic merge of two metamodel versions into a unified metamodel and by matching transformations we are able to generate check-out transformations, so the user can focus on describing the co-evolution rules in her/his inplace transformation language of choice. In particular, the user only has to instantiate new metamodel elements and modify existing elements if needed, but s/he is not concerned with copying constant elements or deleting no longer supported elements. One co-evolution scenario has not been mentioned, namely the elimination of instances which are in principle covered by the revised metamodel. Consider the example that ab-

stract classes should no longer be supported, meaning that only concrete classes are migrated. This kind of elimination is a selection and not a projection automatically achieved by the check-out transformation. Thus, selections have to be defined by the user again by employing inplace transformation rules in step 2 of our approach. We are currently investigating such scenarios in ongoing work.

For future work we see many possible directions to further validate and improve our proposed approach. First, by tracking and analyzing the metamodel edit operations, we aim at generating skeleton transformations for the needed co-evolution rules similar to the migration framework supported by Ruby on Rails[5]. In this respect, we plan to consider move and rename operations in our metamodel merge for automatically generating co-evolution rules. For this we plan to employ dedicated model comparison approaches such as [3, 13]. Furthermore, we have to apply our approach to more complex metamodel evolution scenarios to validate if we are able to support all possible metamodel changes [4]. Finally, we plan some student user studies, because in our model engineering course students always struggle with metamodeling examples due to unloadable example models which are important to verify the developed metamodels. However, to allow for a prototypical, incremental, and iterative metamodel development process, co-evolution of models must be easily accomplished.

## References


1. D. Balasubramanian, C. vanBuskirk, G. Karsai, A. Narayanan, S. Neema, B. Ness, and F. Shi. Evolving paradigms and models in multi-paradigm modeling. Technical Report ISIS-08-91, Institute for Software Integrated Systems, Nashville, 11/2008 2008.
2. J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth. Semantics and implementation of schema evolution in object-oriented databases. *SIGMOD Record*, 16(3):311–322, 1987.
3. C. Brun and A. Pierantonio. Model Differences in the Eclipse Modeling Framework. *UPGRADE, The European Journal for the Informatics Professional*, 2008.
4. E. Burger and B. Gruschko. A Change Metamodel for the Evolution of MOF-Based Metamodels. In *Proceedings of Modellierung 2010*, volume 161 of *LNI*, pages 285–300. GI, 2010.
5. A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio. Automating Co-evolution in Model-Driven Engineering. In *Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference (EDOC'08)*, pages 222–231. IEEE Computer Society, 2008.
6. K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006.
7. M. D. D. Fabro and P. Valduriez. Towards the efficient development of model transformations using model weaving and matching transformations. *Software and System Modeling*, 8(3):305–324, 2009.
8. K. Garcés, F. Jouault, P. Cointe, and J. Bézivin. Managing Model Adaptation by Precise Detection of Metamodel Changes. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'09)*, pages 34–49. Springer-Verlag, 2009.


[5] http://api.rubyonrails.org/classes/ActiveRecord/Migration.html

9. B. Gruschko, D. Kolovos, and R. Paige. Towards synchronizing models with evolving metamodels. In *Proceedings of the International Workshop on Model-Driven Software Evolution @ ECSMR*, 2007.
10. M. Herrmannsdoerfer, S. Benz, and E. Juergens. COPE - Automating Coupled Evolution of Metamodels and Models. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP'09)*, pages 52–76. Springer-Verlag, 2009.
11. A. Narayanan, T. Levendovszky, D. Balasubramanian, and G. Karsai. Automatic domain model migration to manage metamodel evolution. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS'09)*, volume 5795 of *LNCS*, pages 706–711. Springer, 2009.
12. Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification. www.omg.org/docs/ptc/07-07-07.pdf, 2007.
13. D. Ohst, M. Welle, and U. Kelter. Differences between versions of UML diagrams. *SIGSOFT Softw. Eng. Notes*, 28(5):227–236, 2003.
14. J. F. Roddick. Schema evolution in database systems - an annotated bibliography. *SIGMOD Record*, 21(4):35–40, 1992.
15. G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *Proceedings of the 2nd International Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE'03)*, volume 3062 of *LNCS*, pages 446–453. Springer, 2003.
16. M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the Use of Higher-Order Model Transformations. In *Proceedings of 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'09)*, pages 18–33. Springer-Verlag, 2009.
17. G. Wachsmuth. Metamodel adaptation and model co-adaptation. In *Proceedings of the 21rd European Conference on Object-Oriented Programming (ECOOP'07)*, pages 600–624. Springer-Verlag, 2007.