Co-Transformation of Type and Instance Graphs Supporting Merging of Types and Retyping

Florian Mantz, Yngve Lamo, Gabriele Taentzer

24 pages

Co-Transformation of Type and Instance Graphs Supporting Merging of Types and Retyping [*]

Florian Mantz[1]**, Yngve Lamo**[1]**, Gabriele Taentzer**[21]

[1] fma@hib.no, yla@hib.no
Institutt for data og realfag
Høgskolen i Bergen, Norway
[2] taentzer@informatik.uni-marburg.de
Fachbereich Mathematik und Informatik
Philipps-Universität Marburg, Germany

Abstract: Algebraic graph transformation is a well-known rule-based approach to manipulate graphs that can be applied in several contexts. In this paper we use it in the context of model-driven engineering. Graph transformation rules usually specify changes to only one graph per application, however there are use cases such as model co-evolution where not only a single graph should be manipulated but also related ones. The co-transformation of type graphs together with their instance graphs has shown to be a promising approach to formalize model and meta-model co-evolution. In this paper, we extend our earlier work on co-evolution by allowing transformation rules that have less restrictions so that graph manipulations such as merging of types and retyping of graph elements are allowed.

Keywords: Meta-model evolution, model migration, graph transformation

1 Introduction

Model-driven engineering (MDE) [BCW12] is a software engineering discipline that uses models as the primary artifacts throughout the software development process and adopt model transformation for both, model optimization as well as model and code generation. In particular, MDE is a suitable approach for automating recurring development tasks in many areas of software engineering that require application-specific system designs. A commonly used technique to define modeling languages is meta-modeling. An effective means to describe the application-specific part of a software system are domain-specific modeling languages since they focus on the essential concepts of a domain. In contrast to traditional software development where programming languages rarely change, domain-specific modeling languages, and therefore meta-models, can change frequently: modeling language elements may be, e.g., renamed, extended by additional attributes, merged, or refined by a hierarchy of sub-elements. The evolution of a meta- model requires the consistent migration of all its instance models (See Figure 1) which is still considered to be a research challenge in MDE see e.g. [SRVK10].

Previous work in e.g. [RKPP10, HBJ09, Mig] has mainly focused on the development of useful tools to define and execute model migrations. The relation between meta-model changes
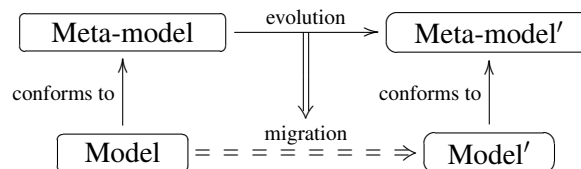
Figure 1: Meta-model evolution and model migration

and model migrations has been studied little on a formal level.

In our work, we focus on a formal setting of meta-model and model co-evolution to study their relations. Models are specified in a suitable category of graphs while model relations are defined by graph morphisms. The type conformance of models to their meta-models is specified by morphisms from instance to type graphs. Graph manipulations can be described by graph transformation [EEPT06] and category theoretical constructs [BW95]. In particular, the proposed approach is based on the co-span double pushout [EHP09] approach which is a variant of the "traditional" double pushout (DPO) approach where transformation rules are co-spans. Equivalence between the two approaches has been shown in [EHP09]. We prefer the alternative DPO approach since migrations can be easier synchronized over an joint-type graph instead of a difference type graph [TML12a]).

The aim of our work is to understand the co-evolution problem better so that future tool support can profit from this formalization. In particular, we are interested under which conditions we can perform different manipulations on type graphs and its instances by categorical constructions so that we get a well-typed and unique result. A resulting graph is considered to be well-typed if there is a suitable morphism to its evolved type graph. A corresponding model migration has to fit to its meta-model evolution such that the resulting model is well-typed over the evolved meta-model. By our formalization, we get formal criteria for consistently defined meta-model evolutions and related model migrations.

In [TML12a] we propose a framework that relates co-span DPO transformations [EHP09] of type graphs and their instance graphs. The framework is defined for weak adhesive categories i.e. it can be applied to different kinds of graphs. The framework clarifies the conditions model migrations need to satisfy and explains how the fully determined part of migration transformations can be derived. Variants of instance graph migrations are still possible and can be specified choosing a proper migration strategy. While our earlier work considers injective rules with injective matching only, we show in this paper that these conditions can be relaxed so that one morphism of the rule may be non-injective. In contrast to [TML12a], it is now possible to merge elements in type graphs and to retype and optionally merge instance nodes and edges accordingly. This is a quite useful feature when it comes to model co-evolution since the context of merged elements is preserved. Note that in the following, we call a co-span transformation of the type graph "evolution", while co-span transformations of instance graphs are considered to be "migrations". An "evolution" transformation together with its related "migration" transformation are considered as "co-transformation". The paper is structured as follows: In Section 2 we motivate merging as a useful co-evolution action, in Section 3 a running example is presented using merging before the theory to support a merge operation is extended in Section 4. Section 5 presents related work while we conclude in Section 6.

2 Basic Co-evolution Actions

Domain-specific modeling languages are effective means to describe domain knowledge on an abstract level. Their models can be used to generate software artifacts such as executable programs. Usually domain-specific modeling languages are defined by use of the concept of meta-modeling where one meta-model serves as language description for the next abstraction level. The most prominent meta-modeling architecture used for this purpose is Meta Object Facility (MOF) [Obj06] standardized by the Object Management Group (OMG) [Obj]. The syntax of MOF-based modeling languages is defined in terms of class diagrams. Elements of the language become classes that are related to other language elements by references. Models of these languages are instances of these class models. For a more intuitive usage of modeling languages, language elements are usually mapped to a different (visual) presentation. However, the concrete presentation of modeling languages is basically syntactic sugar and will be neglected here. Class diagrams and hence meta-models can be naturally formalized by type graphs. Basic object-oriented concepts such as classes, references, attributes, and even, inheritance can be described by graph elements (e.g. using the category AIGraphs [EHP09]): classes are presented as nodes, references as arrows, inheritance relations as special arrows and attributes as arrows that have a data type as target. Hence changes in modeling languages and models can be represented by graph changes. Therefore, we specify meta-models and instance models by type and instance graphs.

In this section, we first present possible basic change actions on type and instance graphs. Afterwards, we assign the required basic change actions to typical meta-model evolutions found in related research and their migrations. We will see that a merge operation is highly desirable and should be supported by a co-evolution approach.

Graph transformation approaches support different kinds of basic change actions. Elements, i.e. nodes and edges may be:

1. Renamed (**N**)
2. Created (**C**)
3. Deleted (**D**)
4. Merged (**M**) or
5. Split (**S**)

Considering type graphs together with their instances, however, more kinds of changes can be distinguished since their typing relations may also be manipulated. A type graph change action may induce a retyping of instances. Elements may be retyped (**T**) for four reasons. Induced retypings are:

1. Types are renamed (T_N).
2. Types are merged (T_M) and instance elements have to be retyped accordingly.

   In addition, graph elements may be optional retyped:

3. A graph element may be retyped to a subtype (T_{sub}) of its current type.
4. A graph element may be retyped to a supertype (T_{sup}) of its current type. This can trigger deletion actions since the supertype need not support all references or attributes of its subtype.

   When a type graph includes attributes, attribute values may be changed in addition by calcu-

lating a new value using a suitable data algebra. Since attribute values are usually considered as a given set of special nodes [EEPT06], the calculation of a new attribute value results into the deletion and creation of arrows: an arrow to an old value is replaced by an arrow to a new one.

Model co-transformation approaches may support a subset of these operations: The most simple change actions are renaming and retyping after type renaming. They do not change the graph structure, hence migration is trivial. In [TML12a], we introduce a co-transformation approach where the manipulation of graphs was restricted to renaming, creation, and deletion of graph elements. In this work, we extend the set of possible change actions to also include merging of graph elements.

However, meta-model evolutions are usually not considered on the level of such atomic changes but on the level of patterns. Some of these meta-model changes have been classified [GKP07] into non-breaking (*NB*), breaking and resolvable (*BR*), and breaking and unresolvable changes (*B¬R*). In this classification, non-breaking changes do not require model migration while breaking and resolvable changes require model migration that can be fully automated. Breaking and unresolvable changes can only be partially automated since they require user interaction. These patterns can be and have to be realized by the basic change actions above. To give some examples of meta-model evolution patterns, consider the following list which is inspired by the one presented by Cicchetti et al. in [CDEP08]:

1. **Add meta-element:** An element e.g. class (node), reference (arrow), or attribute is added to the meta-model. As long as there is no constraint in addition that requires the existence of instance elements, its models do not need to be migrated, but nevertheless can be adapted. For example, default attribute values may be added.
2. **Rename meta-element:** A meta-element i.e. a type is renamed. The model migration is trivial since there is no structural change.
3. **Eliminate meta-element:** A meta-element is deleted and all its direct instances need to be deleted as well.
4. **Push down meta-property:** A meta-property being either a reference or an attribute (arrow), is cloned to all its subclasses (nodes). Migration is required, property instances have to be replaced by instances of the new types. If the superclass had direct instances, these instances loose their "value".
5. **Flatten hierarchy:** A class and its superclass are merged. Their instances need to be retyped.
6. **Move meta-property:** A meta-property e.g. attribute (arrow) is copied from a class A (node) along a reference (arrow) to another class B. Models need to be migrated correspondingly: Property values have to be deleted from the instances of class A, and if existing, copied to all related instances of class B. Values may be duplicated. Instances of elements without an attribute value may get a default value.
7. **Inline meta-class:** Two associated meta-classes A and B are merged. Instances of type A and B have to be retyped. Linked instances of type A and B are merged. Optionally, the reference between A and B and its instances are deleted.
8. **Pull up meta-property:** A meta-property (arrow) of one or more subclasses are represented by a new property (arrow) of the superclass. Instances of this attribute need to be retyped. Direct instances of the superclass may get a new attribute.

Table 1 shows how these changes can be related to the basic model change actions. The classification in Table 1 differs from the one by Ciccetti et al. since they consider the KM3 meta-model [JB06] while we are working with a graph-based formalization. Furthermore, we do not consider constraints by now and summarize equivalent changes like e.g. "Eliminate meta-class" and "Eliminate meta-property" by "Eliminate meta-element". While many patterns and migrations can be expressed by creations and deletions of elements, we will show in the following how our new formalization facilitates the description of these patterns using merging in addition. Note that optional model changes are given in squared brackets. E.g. if a new element is created, it does not have a name but may be (re)named afterwards. Evolution operation "Add meta-element" requires a renaming action since a new type should have a name but its migration operations do not need renaming since instances of a new type may be created without names.

	Type	Pattern name	Evolution operation	Migration operation
1.	*NB*	Add meta-element	C, N	[C],[N]
2.	*BR*	Rename meta-element	N	T_N
3.	*BR*	Eliminate meta-element	D	D
4.	*BR*	Push down meta-property	C,D,N	C,D,[N]
5.	*BR*	Flatten hierarchy	M,N	T_M
6.	*BR*	Move meta-property	C,D,N	C,D,[N]
7.	*BR*	Inline meta-class	M,D,N	M,[D],[N], T_M
8.	*BR*	Pull up meta-property	M,N	T_M,[C],[N]

Table 1: Classification of change patterns

In particular, change actions 5, 7, and 8 in Table 1 above can be easier specified with merging since already existing elements may be retyped. The retyping of elements is always easier than recreating them together with their context. This would be necessary if we could only create and delete. Note that all evolution patterns of Table 1 can be expressed in our proposed formalization. In the next section, we will illustrate merging at an example.

3 Merging Meta-model Elements: An Example

In the following we illustrate an application of the proposed co-transformation framework in the context of meta-model evolution with model migration by a running example. In particular we will see the advantage of a merge operation.

Figure 2a shows the meta-model of SWAL[1], a simple domain-specific modeling language for development of interconnected web pages. The meta-model mainly distinguishes between different kinds of web pages which are connected by hyper references. There are two types of web pages, static and dynamic ones. While instances of static pages result in plain HTML pages after code generation i.e. in pages that contain HTML tags only, instances of dynamic

---

[1] The development of SWAL was initiated by Manuel Wimmer and Philip Langer at the Technische Universität Wien and reimplemented for its use in modeling courses at Philipps-Universität Marburg.

pages result in Java Server Pages (JSP). Note that static pages may contain static HTML forms. In addition, the meta-model defines two special types of dynamic pages, both referring to data entities being further specified in a data model not presented here. An instance of the type *IndexPage* is translated to a JSP handling a list index needed to manage a data entity containing a list of values in a browsable table widget. Each instance of type *DetailsPage* is translated to a JSP showing the full content of a data entity as well as offering the typically CRUD operations like insert and update. Figure 2b shows a SWAL model of a simple address book application. The application starts with a page for address search and prints its results to an address list page. Furthermore, the user can navigate to a page where address details may be updated and stored in a corresponding data entity.
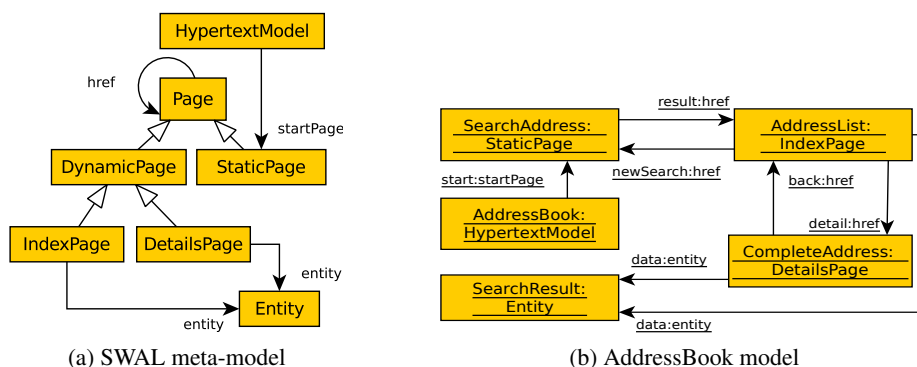

(a) SWAL meta-model   (b) AddressBook model

Figure 2: Meta-model with instance model

The meta-model evolution step: Initially, it was decided that the start page should be a plain HTML page. However, we might realize that the distinction between dynamic pages and static ones is unnecessary and even annoying. For example, in the address book application the search page cannot show previous search parameters because of this decision. The search page is the start page and therefore must be a *StaticPage* which cannot save any information in the session environment. Therefore, we decide to merge class *StaticPage* and *DynamicPage* in the meta-model (see Figure 3).
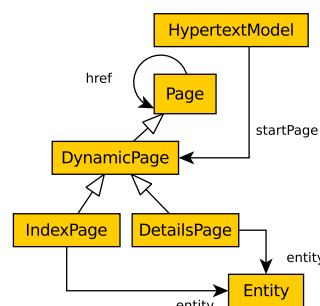

Figure 3: Evolved meta-model

The model migration step: For the model of the address book application, means the evolution step that class *SearchAddress* must be retyped. This can be achieved in two ways:

1. A new instance *SearchAddress* of type *DynamicPage* is created together with its context edges i.e. its incoming and outgoing arrows resembling the context of the old instance *SearchAddress*. Afterwards the old instance *SearchAddress* is deleted together with its context edges. This approach is supported by our old formalization in [TML12a].

2. Instance *SearchAddress* is retyped to *DynamicPage*. The context is not changed. This approach is supported by the formalization presented in this paper.
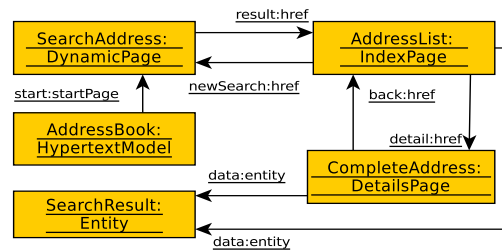

Figure 4: Migrated model

After the introductory example, co-transformations are presented on a formal level in the next section. We will pick up the example and show graph transformations with and without merging of types.

4 Merging of Elements in Theory

In this section, we extend the theory on co-transformations. Co-transformations are defined as coupled co-span transformations related by typing morphisms. First, we relax the definition of co-span transformation given in [TML12a] and hence also implicitly the definition of co-transformation. The relaxed co-span transformations also allow merging of elements. In [TML12a] co-span transformations support the creation and deletion of elements only. Afterwards we show that the main results of [TML12a] hold also in this generalized setting. In particular, we also presented a deduction algorithm for migration transformations in [TML12a] where the left-back face of the co-span transformation (see Figure 5) is required to be a pullback. Here, we also relax this condition so that this face of the double cube only needs to be commuting. This gives us more freedom to define or derive migration rules.

Since the theory should be applicable to different kinds of graphs, the theory is formulated in the context of (weak) adhesive High-Level-Replacement (HLR) categories [EEPT06, LS04]. We find this appropriate since the required concepts needed to formalize models in MDE, like inheritance, can not be described by classical graphs and graph homomorphisms. (Weak) adhesive High-Level-Replacement (HLR) categories are categories that satisfy specific conditions i.e. objects do not need to be classical graphs and morphisms do not need to be graph homomorphisms (for details see Definition 12 and Remark 1 in the appendix). Instead of injective morphisms, we will refer to morphisms in a suitable class of monomorphisms $\mathcal{M}$ in the following. The intuitive idea of such categories is that we have compatible pushout and pullbacks in the sense of van Kampen squares [EEPT06]. To make it easier to understand the proofs, we put the required theorems and definitions of (weak) adhesive HLR categories in the appendix. In the following, we will give the necessary definitions of co-span transformations. Afterwards we will use these definitions to define co-transformations.

Definition 1 (Co-span transformation rule)   Let $\mathscr{C}$ be a (weak) adhesive HLR category (see Definition 12 in the appendix). A *co-span transformation rule* $p = L \xrightarrow{l} I \xleftarrow{r} R$ consists of objects L, I and R and two jointly epimorphic morphisms l and r where r is in $\mathscr{M}$.

Note, that l is not required to be in $\mathscr{M}$ anymore which makes merging of elements possible.

Definition 2 (Co-span transformation)

Given a co-span transformation rule $p = L \xrightarrow{l} I \xleftarrow{r} R$ together with an $\mathscr{M}$-morphism $m\colon L \to G$, called *match*, rule p can be applied to G if a co-span double-pushout exists as shown in the diagram on the right. $t : G \xRightarrow{p,m} H$ is called a *co-span transformation*.

$$
\begin{array}{ccccc}
L & \xrightarrow{\ l\ } & I & \xleftarrow{\ r\ } & R \\
{\scriptstyle m}\downarrow & {\scriptstyle (PO1)} & {\scriptstyle i}\downarrow & {\scriptstyle (PO2)} & \downarrow{\scriptstyle m'} \\
G & \xrightarrow{\ g\ } & U & \xleftarrow{\ h\ } & H
\end{array}
$$

A co-span transformation rule is applied to a match m by first constructing the pushout $PO1$ before constructing a pushout complement of $i \circ r$ to make $PO2$ a pushout. A co-span transformation rule is only applicable iff the gluing condition is satisfied since pushout complements are not in general unique. Furthermore co-span transformations are not allowed to introduce dangling edges by deleting nodes. The gluing condition presented here is the same as in [EHP09, TML12a].

Definition 3 ((Co-span) gluing condition)

Given morphism $m\colon L \to G$, let $b\colon B \to L$ be the boundary of m (i.e., the "smallest" morphism such that there is a pushout complement of b and m, for details see Definition 10 in the appendix), then m satisfies the *co-span gluing condition* wrt. rule $p = L \xrightarrow{l} I \xleftarrow{r} R$ if there is a morphism $b'\colon B \to R$ with $r \circ b' = l \circ b$.

$$
\begin{array}{ccccccc}
 & & & {\scriptstyle b'} & & & \\
 & & & {\overset{\frown}{}} & & & \\
 & & & {\scriptstyle (=)} & & & \\
B & \xrightarrow{\ b\ } & L & \xrightarrow{\ l\ } & I & \xleftarrow{\ r\ } & R
\end{array}
$$

Based on co-span transformations (rules), we now define co-transformation (rules).

Definition 4 (Co-transformation rule)

A co-span rule $tp = TL \xrightarrow{tl} TI \xleftarrow{tr} TR$ and a co-span rule $p = L \xrightarrow{l} I \xleftarrow{r} R$ form a *co-transformation rule* (tp, p), if there are graph morphisms $t_L\colon L \to TL, t_I\colon I \to TI$ and $t_R\colon R \to TR$ such that both squares in the diagram on the right commute.

$$
\begin{array}{ccccc}
TL & \xrightarrow{tl} & TI & \xleftarrow{tr} & TR \\
{\scriptstyle t_L}\uparrow & {\scriptstyle (1)} & {\scriptstyle t_I}\uparrow & {\scriptstyle (2)} & \uparrow{\scriptstyle t_R} \\
L & \xrightarrow{\ l\ } & I & \xleftarrow{\ r\ } & R
\end{array}
$$

In such a co-transformation rule (tp,p), rule tp is called an *evolution rule* while rule p is called a *migration rule* wrt. tp. We also say that migration rule p is well-typed wrt. tp.

We now give properties that ensures that corresponding creations and deletions are reflected by the migration rule.

Definition 5 (Reflecting migration rules)   Consider a *co-transformation rule* (tp, p) like in the figure above. A co-transformation rule (tp, p) is called

1. *creation-reflecting* if $TL \xrightarrow{tl} TI \xleftarrow{t_I} I$ is a pushout (left square).
2. *deletion-reflecting* if $I \xleftarrow{r} R \xrightarrow{t_R} TR$ is a pullback (right square).

We also say that the migration rule p is creation-reflecting or deletion-reflecting wrt. tp.

A co-transformation rule applied to a meta-model and a model forms a co-transformation.

Definition 6 (Match of a co-transformation rule)
A match (tm,m) of a co-transformation rule (tp,p) is given by the corresponding matches $tm\colon TL \to TG$ of the evolution rule tp and $m\colon L \to G$ of the migration rule p so that the matches together with the typing morphisms $t_L\colon L \to TL$ and $t_G\colon G \to TG$ construct a commuting square: $tm \circ t_L = t_G \circ m$. If $G \xleftarrow{m} L \xrightarrow{t_L} TL$ is a pullback then the match is called complete.

$$\begin{array}{ccc} TL & \xrightarrow{tm} & TG \\ \uparrow{\scriptstyle t_L} & & \uparrow{\scriptstyle t_G} \\ L & \xrightarrow{m} & G \end{array}$$

Definition 7 (Co-transformation)   A co-transformation (tt,t) is defined by graph transformation $tt\colon TG \overset{tp,m}{\Longrightarrow} TH$ and graph transformation $t\colon G \overset{p,m}{\Longrightarrow} H$ that apply a co-transformation rule (tp,p) via match (tm,m) simultaneously to type graph TG and its instance graph G, so that there are morphisms $t_U\colon U \to TU$ and $t_H\colon H \to TH$ such that all faces of Figure 5 commute. In such a co-transformation (tt,t), the type graph transformation $tt\colon TG \overset{tp,tm}{\Longrightarrow} TH$ is called an *evolution* while the instance graph transformation $t\colon G \overset{p,m}{\Longrightarrow} H$ is called a *migration* wrt. tt.

Figure 5: Co-transformation

Note that $PO2_t$ exists only if the gluing condition is satisfied for p (see Definition 3). The gluing condition has to hold for both transformations of the co-transformation. In specific cases, however, a satisfied gluing condition on the meta-model level implies a satisfied gluing condition on the model level [MTL12].

Proposition 1   *Let (tt,t) with $tt\colon TG \overset{tp,tm}{\Longrightarrow} TH$ and $t\colon G \overset{p,m}{\Longrightarrow} H$ be a co-transformation with migration rule p deduced by the following construction:*

1. *Construct $G \xleftarrow{m} L \xrightarrow{t_L} TL$ as pullback of $G \xrightarrow{t_G} TG \xleftarrow{tm} TL$.*
2. *Complete $L \xrightarrow{t_L} TL \xrightarrow{tl} TI$ by $L \xrightarrow{l} I \xrightarrow{t_I} TI$ to a commuting square.*
3. *Construct $I \xleftarrow{r} R \xrightarrow{t_R} TR$ as pullback of $I \xrightarrow{t_I} TI \xleftarrow{t_r} TR$.*

If tm satisfied the gluing condition wrt. to rule tp, then m satisfies the gluing condition wrt. rule p.

Proof.

If tm satisfies the gluing condition wrt. rule tp, there is a boundary $tb: TB \to TL$ of tm, i.e., there is a morphism $t_{br}: TB \to TR$ with $tr \circ t_{br} = tl \circ tb$. Let $TB \xleftarrow{t_B} B \xrightarrow{b} L$ be the pullback of tb and t_L. In the diagram on the right (1) and (2) are commuting and (3) is a pullback (by assumption 3).

By assumption $tr \circ t_{br} = tl \circ tb$ and since (1) and (2) are commuting, we have $tr \circ t_{br} \circ t_B = t_I \circ l \circ b$. Thus, there is a unique morphism $b_r: B \to R$ with $r \circ b_r = l \circ b$, according to the universal property of pullback (3).

Note that in [TML12b], this proposition is shown for a slightly different migration deduction where both back faces in Definition 7 are pullbacks. $\qquad\qquad\Box$

Example 1 (Co-transformation) *Now we pick up the example of the previous section to illustrate the concept of co-transformation. First we consider a meta-model change by creating and deleting elements only, i.e. with co-span rules consisting of injective morphism only following the approach of [TML12a]. Hence, we have to replace elements in the type graph as well as in the instance graph. In the instance graph we have to recreate nodes together with their context edges considering their new types. By using our new approach we will later give a simplified version of this particular transformation.*

*In Figure 6, a co-span graph transformation is shown describing the meta-model evolution step from Figure 2a to Figure 3 using injective rule morphisms i.e. tl and tr are injective. Note that morphisms tm, ti and tm' are always injective since we consider injective matchings only. Mappings are indicated by numbers. Furthermore, all morphisms run between AIGraphs i.e. attributed, typed graphs with node type inheritance (compare category AIGraphs in [HEE09]). The co-span graph transformation in Figure 6 shows how class **StaticPage** as well as reference **startPage** in the SWAL meta-model are deleted and a new reference **startPage** is created. This is done in two steps: First, a second reference **startPage** is created in the meta-model (see TU). Formally, this is done by a pushout construction. Second, the former reference **startPage** and class **StaticPage** are deleted from the meta-model (see TH). This is done by subtracting TI − TR from TU. Formally, we construct a second pushout by its complement which results in a unique graph TH (up to isomorphism) for the case of co-span transformation rules (see [EEPT06]). Note that the additional subclasses with mapping 5 and 6 in the rule of Figure 6 are needed since the formalization requires an AIGraph morphism to be "subtype"-reflective [HEE09]. In addition, we add a further class and another reference to the rule to match class **Page** and reference **href** since we need to consider the full context of **StaticPage** during instance migrations. Since we aim at having meta-model independent rules, however, the required "subtype"-reflective property of AIGraph morphisms makes a rule more complex. It would be better to automatically extend given rules to subtype-reflective ones when being matched to concrete meta-models. Such an extension of rules is easy to automate and straight forward to implement. Since, besides these*

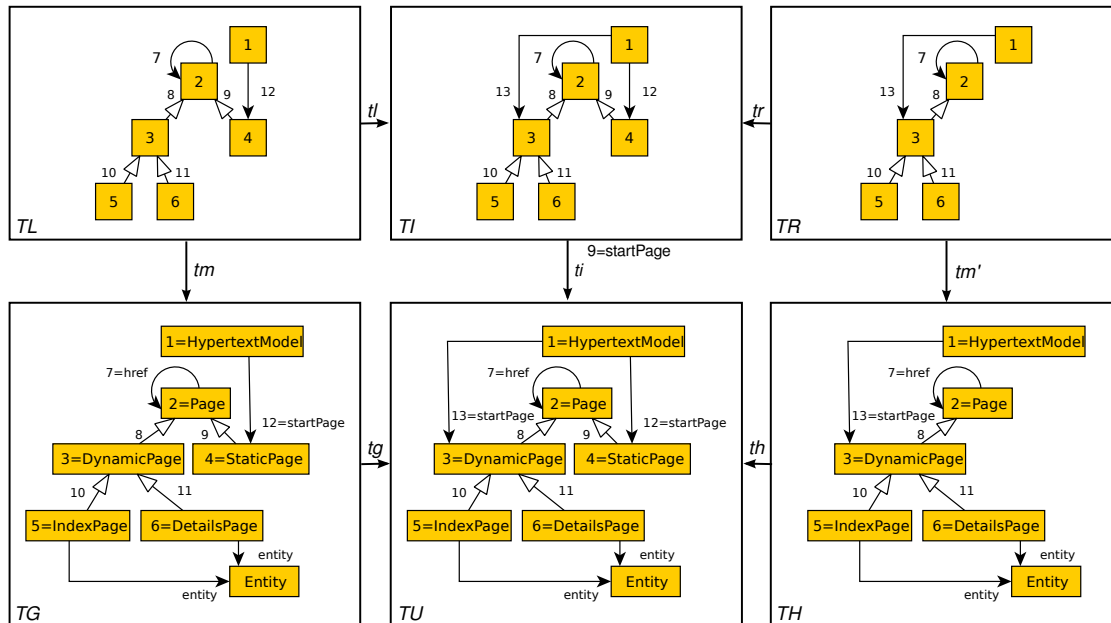*context elements, additional context elements are needed here, this solution for type restructuring is not convincing.*

Figure 6: Evolution co-span transformation *Delete sub-class and create reference*

Figure 7 shows the new version of the meta-model evolution "Merge class". The evolution rule is more compact since the context of the class **StaticPage** e.g. reference **startPage**, does not need to be considered by the rule. The merge is achieved by the non-injective morphisms *tl* and *tg*. The morphisms *th* and *tr* in the right pushout are identity morphisms i.e. nothing is deleted here. Note that the outgoing inheritance arrows from classes **DynamicPage** and **StaticPage** are also merged. This merge of inheritance arrows results from the property of the used category *AIGraphs* (see [HEE09]). Using this revised approach, instance elements only need to be retyped and we do not need to consider their context during the migration.

In *Figure 8* a model migration for the new meta-model evolution operation "Merge class" is shown. The migration rule can be deduced in a similar way as discussed in [TML12a]. This means by implementing a migration strategy that respects the formal properties as discussed in *Proposition 1*. The model migration (shown in *Figure 8*) basically does nothing except that class **SearchAddress** is retyped. Retyping in the instance graph is performed in the first pushout. Hence the graphs in the middle and the right side of the rule are the same. To save space we do not show these graphs in *Figure 8* twice. If we had no retyping facilities available, page **SearchAddress** would have to be recreated with the new type **DynamicPage**. In addition, its context references have to be recreated. Note that the corresponding migration rule would have to be a classical graph transformation rule, while the rule in *Figure 8* is not: There, graphs are not changed but only their typing morphisms. The migration rule contains two pages not being changed. These are included since we derived L by pullback, a construction that ensures that all elements typed by *TL* are matched by the migration rule. The transformations in *Figure 7* and *8*
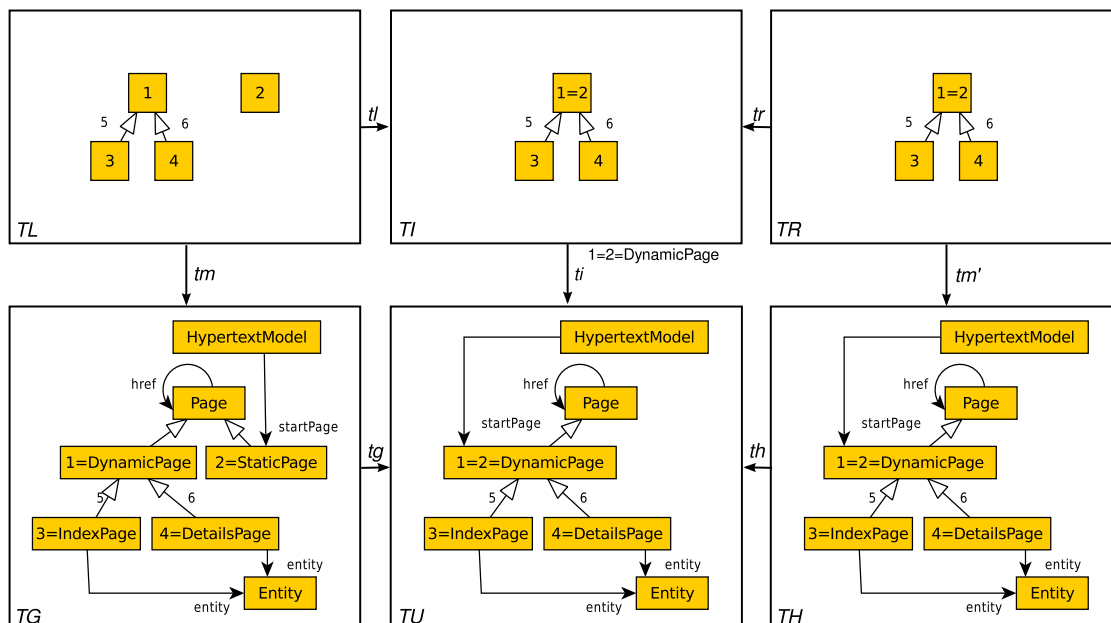
Figure 7: Evolution co-span transformation *Merge class*

together form a co-transformation.
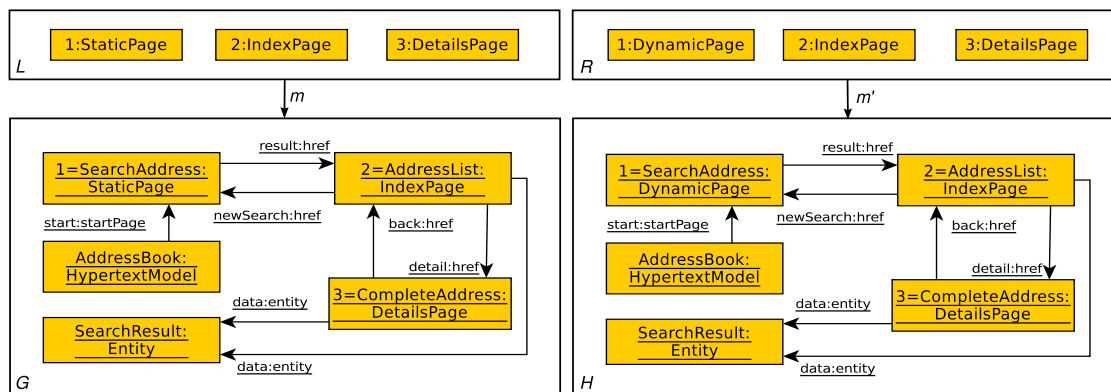
Figure 8: Migration co-span transformation *Merge class*

In the following, we will prove that a match-complete co-transformation always exists.

Theorem 1 (existence of match-complete co-transformation)   *In any (weak) adhesive HLR category: given a co-span graph transformation* $tt : TG \overset{tp,tm}{\Longrightarrow} TH$ *(describing the evolution of type graph TG, see Figure 5) and a graph G typed by TG with typing morphism* $t_G : G \to TG$ *then there exists a co-span graph transformation* $t : G \overset{p,m}{\Longrightarrow} H$ *(describing a corresponding model migration) such that:*

1. *migration rule p is deletion-reflecting wrt. evolution rule tp*
2. (tt,t) *forms a match-complete co-transformation*

Proof.

- **First**, we show that, given an evolution transformation $tt : TG \overset{tp,tm}{\Longrightarrow} TH$ with evolution rule tp and match $tm : TL \to TG$, we can always construct a deletion-reflecting migration rule p that is match-complete applicable to a given graph G with type graph TG: A match-complete left-hand-side L of migration rule p can be deduced by letting $G \overset{m}{\longleftarrow} L \overset{t_L}{\longrightarrow} TL$ be a pullback of $G \overset{t_G}{\longrightarrow} TG \overset{tm}{\longleftarrow} TL$. Since tm is a $\mathcal{M}$-morphism such a pullback always exists (see Remark 1 in the appendix). The rest of the rule we get by constructing a commutative square using $l = id$, $I = L$ and $t_I = tl \circ t_L$ in the left back face, as well as a pullback $I \overset{r}{\longleftarrow} R \overset{t_R}{\longrightarrow} TR$ of co-span $I \overset{t_I}{\longrightarrow} TI \overset{t_r}{\longleftarrow} TR$ in the right back face. Note that we only need the left back face to be commuting, it does not need to be a pullback as required in [TML12a].

- **Second**, we show that the migration rule p is applicable so that (tt,t) forms a co-transformation. We now consider the **left cube** of Figure 5. Pushout $PO1_t$ of rule t can be directly constructed, however we have to show that there is a morphism $t_U : U \to TU$ such that the cube commutes. This we get by the pushout property of $PO1_t$ (see Figure 9):
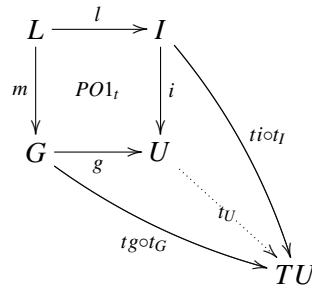
$$
\begin{array}{llll}
ti \circ t_I \circ l & = & ti \circ tl \circ t_L & \quad \text{back face is commuting: } t_I \circ l = tl \circ t_L \\
& = & tg \circ tm \circ t_L & \quad \text{top face is a pushout: } ti \circ tl = tg \circ tm \\
& = & tg \circ t_G \circ m & \quad \text{left face is a pullback: } tm \circ t_L = t_G \circ m
\end{array}
$$

Due to the universal property of $PO1_t$, there exists a unique morphism t_U wrt. $tg \circ t_G$ and $ti \circ t_I$ such that the front and the right faces of the left cube commute. Note the existence of $PO1_t$ is ensured by properties of (weak) adhesive categories (see Remark 1 in the appendix): pushouts along $\mathcal{M}$-morphisms always exist and morphism $m \in \mathcal{M}$ since $tm \in \mathcal{M}$ and $\mathcal{M}$-morphisms are stable under pullback and the left face of the cube is a pullback.

Additionally, we show that the right face of the left cube in Figure 5 is a pullback since it is required in the next part of the proof. In (weak) adhesive categories pushouts along $\mathcal{M}$-morphisms are also pullbacks (see again Remark 1 in the appendix) and match $tm \in \mathcal{M}$, we

get that the top face of the cube is also a pullback. Furthermore, we know that $m \in \mathscr{M}$ (see above). This means that the bottom face is also a pullback since it is also a pushout along an $\mathscr{M}$-morphism. In addition, we know that ti and i are $\mathscr{M}$-morphisms because tm and m are $\mathscr{M}$-morphisms and $\mathscr{M}$-morphisms are also stable under pushout in weak adhesive categories (see Remark 1 in the appendix). Now consider Figure 10: we can compose the left (1) and the top face (2) of the left cube in Figure 5 to a pullback (1+2). By special pushout-pullback decomposition (see Figure 11 and Theorem 2 in the appendix) we get that the right face (4) of the left cube is a pullback since (3) is a pushout and $m, i, ti \in \mathscr{M}$.

$$\begin{array}{ccccc}
L & \xrightarrow{t_L} & TL & \xrightarrow{tl} & TI \\
{\scriptstyle m}\downarrow & (1) & {\scriptstyle tm}\downarrow & (2) & \downarrow{\scriptstyle ti} \\
G & \xrightarrow{t_G} & TG & \xrightarrow{tg} & TU
\end{array} \qquad \begin{array}{ccccc}
L & \xrightarrow{l} & I & \xrightarrow{t_I} & TI \\
{\scriptstyle m}\downarrow & (3) & {\scriptstyle i}\downarrow & (4) & \downarrow{\scriptstyle ti} \\
G & \xrightarrow{g} & U & \xrightarrow{t_U} & TU
\end{array}$$

Figure 10: left face and top face of the left cube in Figure 5

Figure 11: bottom face and right face of the left cube in Figure 5

Now, we consider the **right cube** of Figure 5 and show that we can construct this cube with a pushout in the bottom face. The front face of the cube can be directly constructed as pullback of co-span $U \xrightarrow{t_U} TU \xleftarrow{th} TH$. It remains to show that a morphism $m' : R \to H$ exists such that the cube commutes and the bottom face is a pushout. The existence of an unique m' is shown by the pullback property (see Figure 12) of the front face:

$$\begin{array}{ccc}
TU & \xleftarrow{\quad th \quad} & TH \\
{\scriptstyle t_U}\uparrow & PB & \uparrow{\scriptstyle t_H} \\
U & \xleftarrow{h} & H
\end{array}$$

Figure 12: front face

$$\begin{aligned}
th \circ tm' \circ t_R &= ti \circ tr \circ t_R \\
&= ti \circ t_I \circ r \\
&= t_U \circ i \circ r
\end{aligned} \qquad \begin{array}{l}
\text{top face is a pushout: } th \circ tm' = ti \circ tr \\
\text{back face is a pullback: } tr \circ t_R = t_I \circ r \\
\text{left face is commuting: } ti \circ t_I = t_U \circ i
\end{array}$$

Note that the left face[2] of the right cube commutes as shown above. This means it exists an unique m' wrt. $tm' \circ t_R$ and $i \circ r$ due to the universal property of pullbacks, moreover the right face and hence the whole cube commutes.

---

[2] The left face of the cube is the middle face of the cube in Figure 5 i.e. the right face of the left cube.

It remains to show that the bottom face of the right cube is a pushout $(PO2_t)$. This follows by the (weak) van Kampen property of the category. $tr \in \mathcal{M}$ by assumption, hence, the top face of the right cube is a pushout along **an** $\mathcal{M}$-morphism and therefore also a pullback. Because $\mathcal{M}$-morphisms are also stable under pullback (see Remark 1) and $ti \in \mathcal{M}$ (see above) we get $tm' \in \mathcal{M}$. Hence, the top face of the cube is a pushout along **two** $\mathcal{M}$-morphisms. Since the back and the front face of the cube are pullbacks by construction and we also showed that the left face of the cube is a pullback, we only need to show that the right face of the cube is also a pullback to apply the van Kampen property. Consider Figure 13, we know that we can compose the left (1) and back (2) face of the right cube in Figure 5 to a pullback (1+2) by pullback composition. By pullback decomposition (see Figure 14) follows that the right face (4) of the right cube in Figure 5 is also a pullback since the front face (3) is pullback. Hence the bottom face is a pushout by the van Kampen property.

Figure 13: left face and back face of the right cube in Figure 5

Figure 14: front face and right face of the right cube in Figure 5

Note that the evolution and migration given in the example in Section 3 form a match-complete, deletion-reflecting co-transformation. See Figure 7 together with Figure 8.

Corollary 1 (unique typing of co-transformation) *Given a deletion-reflecting co-transformation rule (tp, p) with an applicable and complete match (tm, m): the migration t is uniquely typed by evolution tt (up to isomorphism), i.e. t_U and t_H are uniquely determined (see Figure 5).*

Proof. The typing morphisms of the left face and the back faces of the cube in Figure 5 are given, moreover the double cube commutes as shown in Theorem 1. Furthermore it has been already shown in the proof of Theorem 1 that the right cube in Figure 5 fulfills the van Kampen property. Hence the right front face of the cube in Figure 5 is a pullback. It remains to prove that t_U and t_H are unique. The uniqueness of t_H follows directly from the fact that $U \xleftarrow{h} H \xrightarrow{t_H} TH$ is a pullback. However, t_H is only unique with respect to morphism $th \colon TH \to TU$ and morphism $t_U \colon U \to TU$. The uniqueness of morphism t_U follows by the fact that the left cube is commuting and by the universal pushout property of the bottom face $PO1_t$ (see Figure 9 and proof of Theorem 1). $\square$

5   Related Work

Co-evolution of structures has been considered in several areas of computer science such as for database schemes, grammars, and meta-models [Li99, Läm01, PJ07, SK04]. Especially database schema evolution has been a subject of research in the last decades.

5.1 Comparison to database schema evolution

However, the challenge of schema evolution differs slightly from meta-model evolution with model migration for various reasons. To mention a few: (1) While models usually are held in the main memory, database tables often cannot. Hence schema evolution is often considered on the level of implementation focusing on the efficiency of the migration. (2) Data should be retrieved from databases via queries. This means that the structure of stored information has to be considered when formulating a query, but the query response is one joint relation even containing duplicate entries. Hence, research considering schema evolution is often concerned with the relation between database queries and database schemes [CMHZ09, CMDZ10]. In MDE, structural information usually has to be reflected in generated artifacts. (3) The basic constraints of relational databases are only a few, i.e. primary key constraint, foreign key constraint and typing [CMDZ10]. In modeling, different constraints may be needed e.g., to specify business rules. (4) In relational models, retyping of tuples (i.e. tables) is slightly different since there is no concept of inheritance. However, merging of types has been also considered in database schema evolution. For example in [Bré96, PK97], list operators for the merging of types in object-oriented databases are presented. The mentioned approaches provide basic merge operators based on textual implementation descriptions using retyping, creation, and deletion of elements. In object-oriented database schema evolution, merging of types is considered on the level of set theory [AP03]. Furthermore, merging of types has also been considered in the context of relational databases [CMHZ09] and ontology evolution [NK04]. (5) Only two layers are considered in database schema evolution i.e., the schema definition and the actual tables with data entries, while modeling hierarchies may have several layers. Our formalization is stepwise applicable to more than two layers.

5.2 Related work to co-evolution of models and meta-models

Recently, research activities have started to consider meta-model evolution and to investigate the transfer of schema evolution concepts to meta-model evolution (see e.g. [HBJ09]). Herrmannsdoerfer et al. provide an overview of approaches in [HVW10] that considers the coupled evolution of meta-models and models. In particular, the model migration challenge has been studied. Several approaches to (partially) automate the tedious and error-prone process of model co-evolution have been proposed. Generally, our work differs from other work on model co-evolution in the sense that we focus on formal criteria rather than tool development. Our focus is a formal framework supporting the specification of meta-model evolutions and related model migrations that guarantee modeling language consistency. In [RHW+10], co-evolution approaches are classified into *meta-model matching, manual specification and operator-based* approaches. In the following, we consider related work wrt. these approaches.

Meta-model matching approaches consider two versions of meta-models as input. An evolution history [CDEP08, RHW+10] i.e. a sequence of semantic evolution steps, is (semi-)auto–matically derived from the difference of two meta-model versions. Afterwards, all detected meta-model evolution steps are (semi-)automatically mapped to predefined migration functions. It can happen that some changes need to be manually handled by user-defined migration scripts.

Meta-model differencing and the automatic detection of meta-model evolution operations has been considered by e.g. Ciccetti et al. in [CDEP08] and is also in the focus of the Atlas Matching Language (AML) [RHW+10]. Considering current modeling frameworks however, the detection of inter-related changes is a topic of on-going research [RKPP10, KKT11]. I.e. fully satisfying solutions are not yet available.

In [CDEP08, RHW+10], a predefined set of meta-model evolution operations can be detected. In both approaches, a migration script is generated using the textual model transformation language ATL (Atlas Transformation Language) [Atl]. If a meta-model change does not correspond to a pre-defined operation, a new matching strategy and a new migration definition have to be implemented. The customization of model migration scripts has to be done on the level of such ATL scripts. There are no formal criteria ensuring that migration scripts are consistently defined wrt. the meta-model evolutions detected. Furthermore, it is also not ensured that a migration is complete, e.g. covers the whole model, as in our approach. Merging of model elements is not supported by ATL and hence has to be simulated by "create" and "delete" change actions.

The tool EMFMigrate [Mig] is currently under development picking up ideas from [CDEP08]. It also aims at migrating other depending artifacts such as ATL transformations. EMFMigrate employs out-place transformations for migration. Migration rules can be assembled to reusable libraries as well as customized by overriding or refinement. However, there is no support to check that migration rules are defined consistently to a meta-model change. To the best knowledge of the authors, merging of elements is not natively supported and has to be implemented by "create" and "delete" change actions.

Manual specification approaches like [SK04, RKPP10] follow a different idea. Instead of detecting semantic changes, these approaches migrate models by copying as much as possible from a previous model version to a new one, according to the types of the evolved meta-model. Elements are automatically copied if they have not changed or have compatibly changed types. New elements in the meta-model are basically not considered during model migration and have to be taken into account in manual migration scripts. In such approaches, well-typed migration results are ensured trivially by performing out-place transformations that create instances of evolved meta-models only. In-place transformations have the advantage that model elements may not be "forgotten" during model migrations but have to deal with the fact that all model elements need to be migrated. Our formalization abstracts from in-place and out-place transformations i.e., can be used as formal basis for both.

In [SK04] Sprinkle et al. propose a manual specification approach that has been implemented in the model change language (MCL) [LBN+13]. MCL allows the domain designers to express model migrations in graphical syntax on a high abstraction level. However, MCL supports only a small set of language primitives allowing not all kinds of migrations. They address changes to specific model elements locally. Thereby, model migrations rules map changed meta-model elements to their corresponding counter elements. MCL migrates models using out-place transformations. Model migration rules for unchanged types are not required to be defined explicitly, elements of such types are automatically copied. MCL does not support a real merge operation. If types have been merged, MCL allows to recreate instances of such types by instances having the same merged type.

Rose et al. presents their tool Epsilon Flock in [RKPP10]. It offers also a manual specification approach that targets to the migration of models only. In contrast to MCL, textual migration scripts have to be written in Epsilon Flock. Similarly to MCL, rules for unchanged or slightly changed meta-model elements do not need to be defined. Such model elements are automatically copied to a new model conforming to the new meta-model if they pass a conformance test. In contrast to our formal framework, Epsilon Flock rules are not type checked. If a migration script is not valid according to the typing morphism, elements may be forgotten without any warning. Reuse of migration knowledge is supported by locally defined migration functions only. Flock does not support merging of model elements, however retyping of model elements by recreating them with a different type is supported.

Operator-based approaches evolve meta-models using pre-defined operators. The evolution history is tracked as a sequence of changes. Usually, a library of coupled evolution-migration operators is supported, similarly to database schema migration (see e.g. [HBJ09, CMHZ09]). Hence, traced changes are automatically bound to migration procedures. However, researchers in MDE realized that a fixed set of reusable coupled evolution-migration operations is not enough for model co-evolution [HBJ09]. Therefore, current approaches allow to extend model migration transformations by manually written code using a general purpose or transformation language.

In this paper, we propose a formalization fitting to an operator-based approach that supports the coupled evolution of meta-models and models where each coupled operator is given by two related graph transformations. However, such operators may also be used in a meta-model matching approach if a sequence of applied meta-model evolution rules can be derived.

Cope/Edapt [HBJ09] is a meta-model evolution tool for EMF that have been used in industrial projects. As in our formal framework, Cope allows the coupled evolution of meta-models and models by operators. Coupled operations are implemented according to a textual specification in Groovy/Java. The tool provides a rich library of useful coupled operators which can be applied if the required preconditions are satisfied. If an evolution operation is missing or the migration is not the desired one, the migration operation has to be implemented in Groovy/Java without any support to ensure well-defined migration rules. Merging of types is partly supported by predefined operators that implements merging by "create" and "delete" actions. However, a real merge operation is not supported. Edapt supports in-place transformations. Even though the tool implements an operator-based approach, it can also be used in a meta-model matching manner.

5.3 Formal approaches to model co-evolution

König et al. present a formal framework to data migration in [KLS11] based on categorical constructions. It can also be used as formal basis for model migration due to meta-model evolution. In contrast to our framework, their framework does not support reusable evolution operations yet and use a fixed construction for migration while we allow to implement different migration strategies. However, the approach in [KLS11] supports merging of elements as well as splitting.

In [LBN$^+$13] MCL is presented as semi-formal approach. MCL rules are formalized by classical DPO graph transformation rules consisting of injective morphisms only where the left rule morphism is the identity. Hence only the creation of model elements is supported but not merging. Furthermore theorems are presented concerning termination and confluence of MCL trans-

formations specified by a given set of MCL rules. MCL transformations always terminate but are not always confluent. However, confluence is decidable in MCL. In our framework we deduce each migration step from a meta-model evolution step, hence our migrations trivially terminate.

To the best of our knowledge, merging of types and related retyping of instances have not been considered in the context of algebraic graph transformations before.

6 Conclusion

After having introduced co-span co-transformations that allow creation and deletion of elements in type and instance graphs, this approach is extended by co-evolution rules that allow to specify also merging of types with corresponding retyping and optionally merging of model elements. By applying this extension to model co-evolution, meta-model evolutions as well as model migrations can be specified easier since the merging of elements does not have to be emulated by creation and deletion actions but is supported in a natural way. In Theorem 1 we show that it is always possible to construct a match-complete and deletion-reflecting co-transformation for a given type graph evolution and an instance graph. In addition, we relaxed the conditions for such a construction so that we have more freedom when we define or derive migration rules. Corollary 1 states that the application of a deletion-reflecting co-transformation always results in a unique typing.

The usefulness of a merge operation has been motivated by discussing change patterns that are easier formulated with merging. In addition, a running example has been presented which shows a typical co-evolution case where merging of types is useful. Basic change actions on (meta-)model elements are *renaming*, *creating*, *deleting*, *merging*, and *splitting* as well as four different retype change actions. We support all these change actions in our formalization except of splitting, subtyping, and supertyping. It is up to future work to extend our formalization to these remaining change actions. One approach to support element splitting can be to switch from the (co-span) DPO approach to the (co-span) Sesqui-Pushout approach [CHHK06].

While the presented formalization considers co-transformations on a theoretical level, providing criteria for consistent model migrations, we plan to develop suitable tools in future. Furthermore, we also plan to take meta-model constraints into account.

Bibliography

[AP03]   R. Alhajj, F. Polat. Rule-based schema evolution in object-oriented databases. *Knowledge-Based Systems* 16(1):47–57, 2003.
DOI 10.1016/S0950-7051(02)00051-5

[Atl]   Atlas Transformation Language. User Guide. http://wiki.eclipse.org/ATL/User_Guide.

[BCW12]   M. Brambilla, J. Cabot, M. Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.
DOI 10.2200/S00441ED1V01Y201208SWE001

[Bré96]   P. Bréche. Advanced Primitives for Changing Schemas of Object Databases. In Constantopoulos et al. (eds.), *CAiSE 1996: Proceedings of the 8th International Conference on Advanced Information Systems Engineering*. Lecture Notes in Computer Science 1080, pp. 476–495. May 1996.
DOI 10.1007/3-540-61292-0˙26

[BW95]   M. Barr, C. Wells. *Category Theory for Computing Science (2nd Edition)*. Prentice Hall, 1995.

[CDEP08]   A. Cicchetti, D. Di Ruscio, R. Eramo, A. Pierantonio. Automating Co-evolution in Model-Driven Engineering. In *Proceedings of EDOC 2008: 12th International IEEE Enterprise Distributed Object Computing Conference*. Pp. 222–231. IEEE Computer Society, 2008.
DOI 10.1109/EDOC.2008.44

[CHHK06]   A. Corradini, T. Heindel, F. Hermann, B. König. Sesqui-Pushout Rewriting. In Corradini et al. (eds.), *Proceedings of ICGT 2006: 3nd International Conference on Graph Transformations*. Lecture Notes in Computer Science 4178, pp. 30–45. Springer, September 2006.
DOI 10.1007/11841883˙4

[CMDZ10]   C. Curino, H. J. Moon, A. Deutsch, C. Zaniolo. Update Rewriting and Integrity Constraint Maintenance in a Schema Evolution Support System: PRISM++. *Proceedings of VLDB 2010: 36th International Conference on Very Large Database Endowment* 4(2):117–128, 2010.

[CMHZ09]   C. Curino, H. J. Moon, M. Ham, C. Zaniolo. The PRISM Workwench: Database Schema Evolution without Tears. In Ioannidis et al. (eds.), *Proceedings of ICDE 1999: 25th International Conference on Data Engineering*. Pp. 1523–1526. Proceedings of ICDE 1999: 25th International Conference on Data Engineering, 2009.

[EEPT06]   H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, March 2006.
DOI 10.1007/3-540-31188-2

[EHP09]   H. Ehrig, F. Hermann, U. Prange. Cospan DPO Approach: An Alternative for DPO Graph Transformation. *EATCS Bulletin* 98:139–149, 2009.
http://tfs.cs.tu-berlin.de/publikationen/Papers09/EHP09.pdf

[Erm06]   C. Ermel. *Simulation and Animation of Visual Languages based on Typed Algebraic Graph Transformation*. PhD thesis, Fakultät IV (Elektrotechnik und Informatik), Technische Universität Berlin, Germany, July 2006.

[GKP07]   B. Gruschko, D. Kolovos, R. Paige. Towards Synchronizing Models with Evolving Metamodels. In Tamzalit (ed.), *Proceedings of MoDSE 2007: 1st International Workshop on Model-Driven Software Evolution*. March 2007.

[HBJ09]   M. Herrmannsdoerfer, S. Benz, E. Jürgens. COPE - Automating Coupled Evolution of Metamodels and Models. In Drossopoulou (ed.), *Proceedings of ECOOP 2009: 23rd European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science 5653, pp. 52–76. Springer, 2009.
DOI 10.1007/978-3-642-03013-0˙4

[HEE09]   F. Hermann, H. Ehrig, C. Ermel. Transformation of Type Graphs with Inheritance for Ensuring Security in E-Government Networks. In *Fundamental Approaches to Software Engineering, 12th Int. Conference, FASE 2009*. Lecture Notes in Computer Science 5503, pp. 325–339. Springer, 2009. Long version as TR 2008-07 at TU Berlin.

[HVW10]   M. Herrmannsdoerfer, S. Vermolen, G. Wachsmuth. An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models. In Malloy et al. (eds.), *Proceedings of SLE 2010: 3rd International Conference on Software Language Engineering*. Lecture Notes in Computer Science 6563, pp. 163–182. Springer, 2010.
DOI 10.1007/978-3-642-19440-5˙10

[JB06]   F. Jouault, J. Bézivin. KM3: A DSL for Metamodel Specification. In Gorrieri and Wehrheim (eds.), *Proceedings of FMOODS 2006: 8th International Conference on Formal Methods for Open Object-Based Distributed Systems*. Lecture Notes in Computer Science 4037, pp. 171–185. Springer, June 2006.
DOI 10.1007/11768869˙14

[KKT11]   T. Kehrer, U. Kelter, G. Taentzer. A rule-based approach to the semantic lifting of model differences in the context of model versioning. In *ASE 2011, Lawrence, KS, USA, November 6-10, 2011*. IEEE, 2011.

[KLS11]   H. König, M. Löwe, C. Schulz. Model Transformation and Induced Instance Migration: A Universal Framework. In Silva Simão and Morgan (eds.), *Proceedings of SBMF 2011: 14th Brazilian Symposium on Formal Methods, Foundations and Applications*. Lecture Notes in Computer Science 7021, pp. 1–15. Springer, 2011.
DOI 10.1007/978-3-642-25032-3˙1

[Läm01]   R. Lämmel. Grammar Adaptation. In Oliveira and Zave (eds.), *Proceedings of FME 2001: Formal Methods for Increasing Software Productivity: 1st International Symposium of Formal Methods Europe*. Lecture Notes in Computer Science 2021, pp. 550–570. Springer, 2001.
DOI 10.1007/3-540-45251-6˙32

[LBN+13]   T. Levendovszky, D. Balasubramanian, A. Narayanan, F. Shi, C. Buskirk, G. Karsai. A semi-formal description of migrating domain-specific models with evolving domains. *Software and Systems Modeling*, pp. 1–17, January 2013.
DOI 10.1007/s10270-012-0313-5

[Li99]   X. Li. A Survey of Schema Evolution in Object-Oriented Databases. In *TOOLS*. Pp. 362–371. IEEE Computer Society, 1999.

[LS04]   S. Lack, P. Sobocinski. Adhesive Categories. In Walukiewicz (ed.), *Proceedings of FoSSaCS 2004: 7th Foundations of Software Science and Computation Structures.* Lecture Notes in Computer Science 2987, pp. 273–288. 2004. DOI 10.1007/978-3-540-24727-2˙20

[Mig]   E. Migrate. Project Web Site. http://www.emfmigrate.org.

[MTL12]   F. Mantz, G. Taentzer, Y. Lamo. Co-Transformation of Type and Instance Graphs Supporting Merging of Types with Retyping: Long Version. Technical report, Department of Mathematics and Computer Science, University of Marburg, Germany, September 2012. http://www.uni-marburg.de/fb12/forschung/berichte/berichteinformtk.

[NK04]   N. F. Noy, M. C. A. Klein. Ontology Evolution: Not the Same as Schema Evolution. *Knowledge-Based Systems* 6(4):428–440, 2004.

[Obj]   Object Management Group. Web site. http://www.omg.org.

[Obj06]   Object Management Group. Meta-Object Facility Specification. January 2006. http://www.omg.org/spec/MOF/2.0/.

[PJ07]   M. Pizka, E. Juergens. Automating Language Evolution. In *TASE '07: Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering.* Pp. 305–315. IEEE Computer Society, Washington, DC, USA, 2007. DOI 10.1109/TASE.2007.13

[PK97]   A. Pons, R. K. Keller. Schema Evolution in Object Databases by Catalogs. In *IDEAS 1997: Proceedings of the 1st International Database Engineering and Applications Symposium.* Pp. 368–378. IEEE Computer Society, August 1997.

[RHW$^+$10] L. Rose, M. Herrmannsdoerfer, J. R. Williams, D. Kolovos, K. Garcés, R. F. Paige, F. A. C. Polack. A Comparison of Model Migration Tools. In Petriu et al. (eds.), *Proceedings of MoDELS 2010: 13th International Conference on Model Driven Engineering Languages and Systems.* Lecture Notes in Computer Science 6394, pp. 61–75. Springer, 2010. DOI 10.1007/978-3-642-16145-2˙5

[RKPP10]   L. Rose, D. Kolovos, R. F. Paige, F. A. C. Polack. Model Migration with Epsilon Flock. In Tratt and Gogolla (eds.), *Proceedings of ICMT 2010: 3rd International Conference on Theory and Practice of Model Transformation.* Lecture Notes in Computer Science 6142, pp. 184–198. Springer, 2010. DOI 10.1007/978-3-642-13688-7˙13

[SK04]   J. Sprinkle, G. Karsai. A Domain-Specific Visual Language for Domain Model Evolution. *Journal of Visual Languages and Computing* 15(3–4):291–307, 2004. DOI 10.1016/j.jvlc.2004.01.006

[SRVK10] J. Sprinkle, B. Rumpe, H. Vangheluwe, G. Karsai. Metamodelling - State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems*. Lecture Notes in Computer Science 6100, pp. 57–76. Springer, 2010.

[TML12a] G. Taentzer, F. Mantz, Y. Lamo. Co-Transformation of Graphs and Type Graphs With Application to Model Co-Evolution. In Ehrig et al. (eds.), *Proceedings of ICGT 2012: 6nd International Conference on Graph Transformations*. Lecture Notes in Computer Science 7562, pp. 326–340. Springer, 2012.

[TML12b] G. Taentzer, F. Mantz, Y. Lamo. Co-Transformation of Graphs and Type Graphs with Application to Model Co-Evolution: Long Version. Technical report, Dep. of Mathematics and Computer Science, University of Marburg, Germany, 2012. http://www.uni-marburg.de/fb12/forschung/berichte/berichteinformtk.

Appendix

Definition 8 (Pushout complement)

Let $a\colon A \to B$ and $b\colon B \to D$ be two morphisms, morphisms $c\colon A \to C$ and $d\colon C \to D$ are called a *pushout complement* of a and b if b and d are the pushout of a and c.

$$
\begin{array}{ccc}
B & \xrightarrow{b} & D \\
a\uparrow & & \uparrow d \\
A & \xrightarrow{c} & C
\end{array}
$$

Definition 9 (Pullback complement)

Let $a\colon A \to B$ and $b\colon B \to D$ be two morphisms, morphisms $c\colon A \to C$ and $d\colon C \to D$ are called a *pullback complement* of a and b if a and c are the pullback of b and d.

$$
\begin{array}{ccc}
B & \xrightarrow{b} & D \\
a\uparrow & & \uparrow d \\
A & \xrightarrow{c} & C
\end{array}
$$

Definition 10 (Initial pushout)   Given a morphism $f\colon A \to A'$ in a (weak) adhesive HLR category, a morphism $b\colon B \to A$ with $b \in \mathcal{M}$ is called the **boundary** over f if there is a pushout complement of f and b such that (1) is a pushout which is initially over f. Initially of (1) over f means, that for every pushout (2) with $b' \in \mathcal{M}$ there exist unique morphisms $b^*\colon B \to D$ and $c^*\colon C \to E$ with $b^*, c^* \in \mathcal{M}$ such that $b' \circ b^* = b$, $c' \circ c^* = c$ and (3) is a pushout. B is then called the boundary object *and* and C the context wrt f.

$$
\begin{array}{ccc}
B & \xrightarrow{b} & A \\
\downarrow & (1) & \downarrow f \\
C & \xrightarrow{c} & A'
\end{array}
\qquad
\begin{array}{ccccc}
B & \xrightarrow{b^*} & D & \xrightarrow{b'} & A \\
\downarrow & (3) & \downarrow & (2) & \downarrow f \\
C & \xrightarrow{c^*} & E & \xrightarrow{c'} & A'
\end{array}
$$

Definition 11 (Van Kampen square)   A pushout (1) is a van Kampen square if, for any commutative cube (2) with (1) in the top and where the back faces are pullbacks, the following statement holds: the bottom face is a pushout iff the front faces are pullbacks:
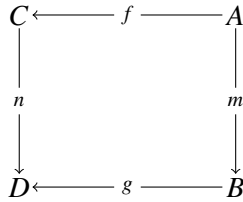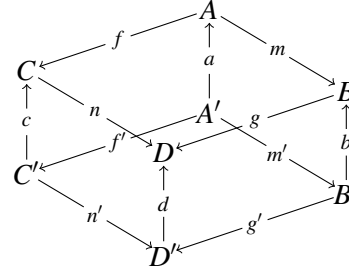
Figure 15: (1)

Figure 16: (2)

For graph transformations (weak) *Adhesive High-Level Replacement (HLR) categories* are considered as suitable framework. Therefore we recall the definition of Adhesive HLR category from [EEPT06]:

Definition 12 (Adhesive HLR category) A category C with a morphism class $\mathcal{M}$ is called an adhesive HLR category if:

1. $\mathcal{M}$ is a class of monomorphisms closed under isomorphisms, composition ($f : A \to B \in \mathcal{M}$, $g : B \to C \in \mathcal{M} \Rightarrow g \circ f \in \mathcal{M}$), and decomposition ($g \circ f \in \mathcal{M}, g \in \mathcal{M} \Rightarrow f \in \mathcal{M}$).
2. C has pushouts and pullbacks along $\mathcal{M}$-morphisms, and $\mathcal{M}$-morphisms are closed under pushouts and pullbacks.
3. Pushouts in C along $\mathcal{M}$-morphisms are VK squares.

Remark 1

1. *Recall Remark 4.10 from [EEPT06]: A pushout along an $\mathcal{M}$-morphism is a pushout where at least one of the given morphisms is in $\mathcal{M}$. Pushouts are closed under $\mathcal{M}$-morphisms if, for a given pushout (1) (see Figure 15), $m \in \mathcal{M}$ implies that $n \in \mathcal{M}$. Analogously, pullbacks are closed under $\mathcal{M}$-morphisms if, for a pullback (1), $n \in \mathcal{M}$ implies that $m \in \mathcal{M}$.*
2. **Weak** *adhesive HLR categories are adhesive HLR categories where the third condition is replaced by a stricter one:*

    3. *Pushouts in C along $\mathcal{M}$-morphisms are weak VK squares, i.e. the VK square property holds for all commutative cubes with $m \in \mathcal{M}$ **and** $f \in \mathcal{M}$ or $b, c, d \in \mathcal{M}$.*

3. *Note that already in weak adhesive HLR categories pushouts along **one** $\mathcal{M}$-morphism are pullbacks (compare Theorem 4.26 in [EEPT06]). This means if $f \in \mathcal{M}$ **or** $m \in \mathcal{M}$ in Figure 16 the pushout is also a pullback.*

Theorem 2 (Special pushout/pullback composition and decomposition)

Given the commutative diagram to the right with square (1) being a pushout and square (1+2) being a pullback, and f_1, f_2, f_3 being monomorphisms. Then, squares (1) and (2) are pullbacks.
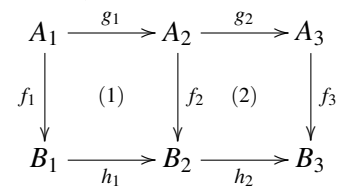
Remark 2 Proof: See Property A.8 in [Erm06].