Customizing Model Migrations by Rule Schemes

Florian Mantz
Høgskolen i Bergen
Bergen, Norway
fma@hib.no

Gabriele Taentzer
Philipps-Universität Marburg
Marburg, Germany
taentzer@informatik.uni-marburg.de

Yngve Lamo
Høgskolen i Bergen
Bergen, Norway
yla@hib.no

ABSTRACT

Model-driven engineering (MDE) is a software engineering discipline focusing on models as the primary artifacts in the software development process while programs are mainly generated by means of model-to-code transformations. In particular, modeling languages tailored to specific application domains promise to increase the productivity and quality of software development. Nevertheless due to e.g. evolving requirements, modeling languages and their meta-models evolve which means that existing models have to be migrated correspondingly. In our approach, such co-evolutions are specified as related graph transformations ensuring well-typed model migration results. Based on our earlier work on co-transformations, we now consider the automatic deduction of migration rule schemes from given meta-model evolution rules. Rule schemes form the basis for user customizations on a high abstraction level. A rule scheme deduction algorithm is presented and several customized migration schemes for different co-evolution examples are discussed.

Categories and Subject Descriptors

D.3.2 [**Model Transformation**]: Model Migration

General Terms

Languages

Keywords

Meta-model co-evolution, graph transformation

1. INTRODUCTION

Model-driven engineering [8] (MDE) is a software engineering discipline which raises the level of abstraction by using models as primary artifacts. In particular, domain-specific modeling languages (DSMLs) aim at increasing productivity and quality of software development. Developers can focus on their essential tasks while repetitive and technology-dependent artifacts are automatically generated by transformations specified by experts in these areas. To keep this high level of abstraction, modeling languages have to evolve together with the evolving practice and understanding of target domains. However, this often causes trouble since existing models need to co-evolve with their languages (see Fig. 1).

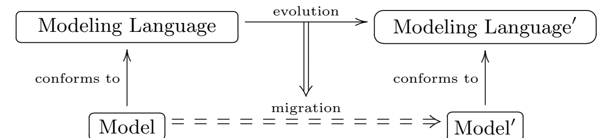

Figure 1: Model co-evolution: Modeling language evolution and model migration

This migration challenge has been studied and different kinds of approaches [27, 3, 13] to (partially) automate the tedious and error-prone process of meta-model co-evolution have been proposed. In this paper, we present an approach that supports the coupled evolution of meta-models and models by coupled rules. The main contribution of this paper is an approach that automatically deduces rule schemes for model migration from meta-model evolution rules. In addition, migration rule schemes can be customized on a high-level of abstraction.

In other approaches that support such coupled evolutions, operators for both i.e. for meta-model evolution and model migration, need to be specified by a tool developer. We try to simplify the second step. In our approach we generate default migration rule schemes automatically from arbitrary meta-model evolution rules using a general heuristic. This has the advantage that, besides standard evolution operators, also individually needed coupled ones can be implemented easier. In this paper we focus on the generation of such rule schemes and refer to our earlier work [20] for details about their application. To our best knowledge, this approach is the only one supporting automatic migration scheme generation yet.

Since desired model migrations are not always completely determined by related meta-model evolution operations, i.e. different migration strategies are possible, we support a form of high-level customization of generated migration rule schemes. In particular, we generate schemes and allow customizations as long as the schemes remain well-formed. In other approaches, languages designers are not guided in the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

IWPSE'13, August 19-20, 2013, Saint Petersburg, Russia
Copyright 2013 ACM 978-1-4503-2311-6/13/08...$15.00
http://dx.doi.org/10.1145/2501543.2501545

process of migration operator specification.

Meta-model evolutions and related model migrations have to be defined consistently. In contrast to other approaches the work presented here ensures such consistency. We employ the formal framework of algebraic graph transformations [6] and consider co-evolutions of models and their meta-models as sequences of coupled graph transformations ensuring well-typed migration results. Well-typedness means that after applying an evolution rule, i.e. a graph transformation rule, to the meta-model and a corresponding migration rule to an instance model, the resulting model can be typed by the changed meta-model even if the transformations are applied independently of each other. Coupled evolution and migration rules means that they are formally related by typing morphisms.

Note that our approach requires model-specific migration rules. We construct such rules by amalgamating basic migration rules coming from model-independent migration schemes. By formalizing model co-evolutions as related graph transformations we have the possibility to statically type check model migration schemes. In addition, generated migration schemes can warn language designers about situations where elements are deleted without replacement causing an information loss.

Approaches either transform in-place, i.e. by changing the existing model or out-place by generating a new one. Models are often migrated using out-place transformations. In this case, well-typed migration results can be ensured trivially by creating instances of types of the evolved meta-model only, like e.g. in [25]. However, we also support in-place transformations here. In-place transformations have the advantage that model elements may not be "forgotten" during the transformations but have to deal with the fact that all elements of a model that need to be migrated, are considered by the migration.

In the rest of this paper, we consider graphs to be synonym with models in abstract syntax. In the next section, we recall the main concepts of co-transformations on graphs. In Section 3, we present an algorithm for automatically deducing migration schemes. Section 4 presents derived migration schemes for some well-known meta-model changes. Some of these migrations need to be customized to get the intended meaning, others are directly provided by the deduction algorithm. Section 5 compares with related work while we conclude in Section 6.

2. CO-TRANSFORMATIONS

In this section, we introduce co-transformations of graphs on an informal level. The interested reader can find the formal definitions of these concepts in [30, 21, 20]. Graph transformation is the rule-based manipulation of graphs. There exists a variety of graph transformation approaches, differing mainly in the kind of transformation rules allowed and the way in which they are applied.

Graphs are often used as an abstract representation of models. When formalizing object-oriented modeling, graph structures are used to represent model and meta-model structures leading to instance and type graphs. A fixed *type graph* TG serves as an abstract representation of a meta-model. As in object-oriented modeling, types can be structured by a generalization relation. Multiplicities and other annotations are not formalized by type graphs, but have to be expressed by additional graph constraints [11, 26]. When considering meta-model conformance, we neglect constraints for now, but we will take them into account in the future. Instance graphs define model structures and have structure-compatible mappings to their type graphs. The attribution of graph vertices and edges can be achieved by using data algebras (for details on typed attributed graphs see [5, 6]). In the following, we use the terms "graph" and "model structure" synonymously.

In the algebraic graph transformation approach, rules are roughly expressed by two graphs L and R, where L is the left-hand side of a rule and R is its right-hand side, which are usually overlapping in graph parts. Rule graphs may contain variables for attributes. The left-hand side L represents the pre-conditions of a rule, while the right-hand side R describes its post-conditions. The intersection $L \cap R$ (the graph part that is not changed) and the union $L \cup R$ should both form graphs, i.e., they must be structure-compatible wrt. source, target mappings as well as type-compatible, in order to apply the rule. $L \setminus (L \cap R)$ defines the part that is to be deleted, and $R \setminus (L \cap R)$ defines the part to be created. Furthermore, the application of a graph rule may be restricted by so-called *negative application conditions* (NACs) prohibiting the existence of certain graph patterns in the current instance graph. Graph elements common to L and R or common to L and a NAC, are indicated by equal names or numbers.

Given a rule $r : L \to R$, a *direct graph transformation* $G \overset{r,m}{\Longrightarrow} H$ between two instance graphs G and H is defined by first finding a match m of the left-hand side L of the rule r in graph G such that m is structure- and type-compatible, and satisfies the NACs (i.e., the forbidden graph patterns are not found in G). Attribute variables used in a graph element $e \in L$ are bound to concrete attribute values of graph element $m(e)$ in G. The resulting graph H is usually constructed by first removing all graph elements from G that are in L but not in R and then adding all those new graph elements that are in R but not in L. This kind of graph transformation is formalized by the standard double pushout (DPO) approach as presented in [6].

However, this is not the only possible order to perform graph changes. It is possible to reverse this order which seems to better fit the needs of model co-evolution. By first adding new meta-model elements while keeping the ones to be deleted, the intermediate meta-model can be used for both, continuous typing of migrating models as well as synchronizing required migration changes (see also [30, 22]). Meta-model elements that should be deleted, are removed in the second step. This form of transformation can be formalized by the so-called co-span DPO approach as presented in [7].

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

Note that the name "co-span transformation" comes from the formalization of rules as co-span of graph morphisms $L \xrightarrow{l} I \xleftarrow{r} R$.

A schematic notation of a co-span graph transformation is presented above, the application is done by constructing two pushouts (PO_A and PO_B). Furthermore, we assume that match m is injective. A co-span rule consists of a (not necessarily injective mapping) l and an injective mapping r which are jointly surjective. Hence, merging of graph elements is allowed but not their splitting (beside creating and deleting).

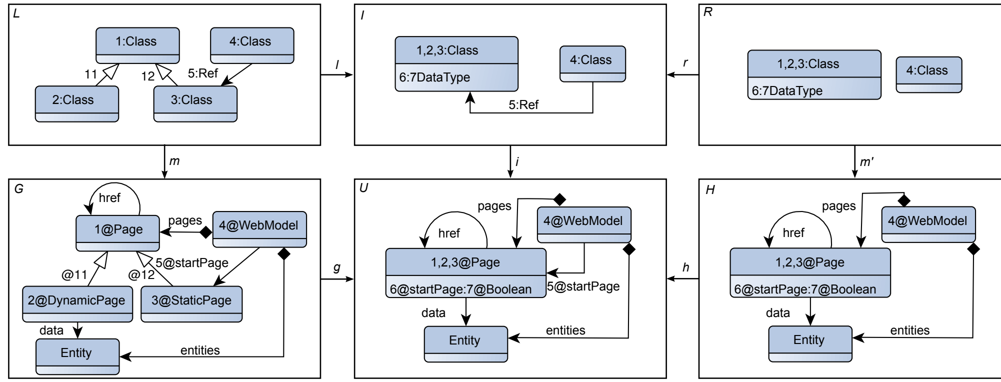

Figure 2: Example: A co-span graph transformation

EXAMPLE 2.1. *Figure 2 shows an example of a co-span graph transformation applying all supported kinds of graph changes. Model graph G in the figure shows a part of the meta-model in Figure 3 for modeling simple web applications. In this language the navigation between web pages can be modeled. They show information about data entities. Pages can either be static or dynamic. The entry page is a static page and marked by a reference "startPage". In addition data entities can be referenced by dynamic pages. Data entities contain the data that should be displayed on a page and which can be modified by simple CRUD operations. Pages, entities and datatypes are contained in class "WebModel". Most of the following examples are concerned with the evolution of similar meta-models.*

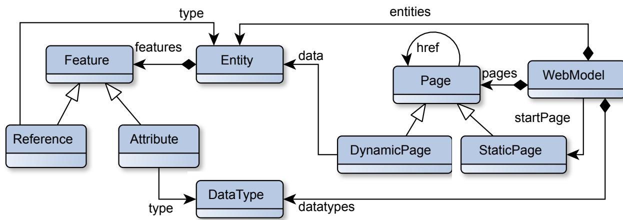

Figure 3: Meta-element for a simple web modeling language

In the presented evolution step (see graph H in Figure 2) all page classes are ***merged****, reference "startPage" is replaced by a boolean flag in class "Page" i.e. reference "startPage" is* ***deleted*** *and flag "startPage"* ***created****.*

While the rule in the upper layer shows all type names, type information is omitted in the lower layer. This is possible since the typing is expressed by a unique graphical representation: Classes are represented as rectangles, references and inheritance relations as different kinds of arrows, attribute names as strings before ":", and data types as strings after ":". All mappings are indicated by numbers. Note that all mappings are structure- and type-compatible. Classes matched by Classes "1" to "3" are merged by the rule since they are all mapped to Class "1,2,3" in I. Furthermore, I defines a new attribute, since attribute "6" with data type "7" is not in L. In addition, the reference matched by "5" is deleted by the rule application since it is in L but not in R. Note that the co-span rule also contains inheritance edges ("11" and "12"). On the formal level, the class mapped by "1,2,3" should contain an inheritance self reference due to the specified merge, however such references are not shown since they implicitly exist for all nodes in the used formalization of graphs (compare [12]).

Co-transformations are defined based on co-span transformations. Figure 4 presents a schematic notation of a co-transformation. While the top faces of the double cube form a co-span transformation that transforms type graph TG to type graph TH (representing meta-models), the bottom faces form a co-span transformation of instance graph G to graph H (representing models). All type and instance graphs are related by corresponding typing morphisms so that we get a double cube. Since all faces of the cube commute, such co-transformations ensure a correct typing. In previous work [30, 21], we studied under which conditions such co-transformations can be constructed. Given a type graph transformation representing a meta-model evolution step, there are suitable instance graph transformations representing model migrations such that a type and an instance graph transformation together form a co-transformation. In Figure 7, an example of the essential graphs in co-transformations is shown for the case of a non-deleting co-transformation. The upper part of the figure shows a graph transformation representing a meta-model evolution while the lower part of the figure shows a graph transformation representing a migration (more details in Section 4).

3. DERIVING MIGRATION SCHEMES

In co-transformations, migration rules (specified by instance graph rules) have to match all elements to be deleted or retyped to ensure well-typed migration results and thus, are model-specific. If e.g. an attribute is moved between meta-model classes, this movement has to be reflected in models as often as there are instances of this attribute. Obviously, two different models do not need to have the same number of attribute instances. Hence, corresponding migration rules would be different.

In the following, we are heading towards a model-independent specification of model migrations. For a given meta-model evolution rule, we generate a migration rule

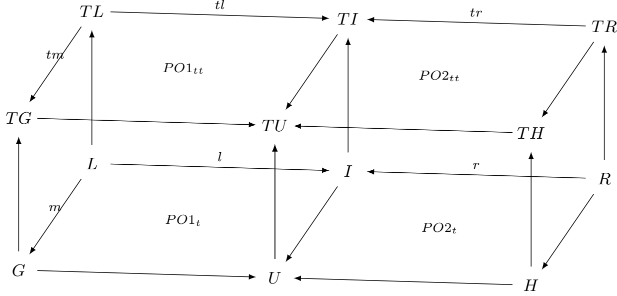

Figure 4: Co-transformation

scheme consisting of rules for simple changes that can be used to construct individual migration rules for all affected models. In particular, we focus on the left-hand sides of migration rules being responsible for creating and merging model elements and leaving space for customizations. The right-hand sides are constructed by a standard construction that projects out the maximal model part of I that can be typed by TR (representing the evolved meta-model). To generate migration rules from arbitrary evolution rules, we use a heuristic: We assume that a meta-model change needs to be repeated as often as the specified pattern appears on the instance level. Elements in a connected graph are considered to be related, hence derived migration rules deal with connected graphs only. Generally, we assume that related parts of a model are replaced by related parts and claim that it does not make sense to replace unrelated model elements in one step, in general.

Algorithm 1 shows pseudo code for generating a model-independent scheme of non-deleting migration rules. To apply a migration scheme to a model, all migration rules are intended to be matched as often as possible, therefore we call them multi-rules. We produce a copy of a multi-rule for each of its match and amalgamate these copies to a model-specific migration rule.

In Algorithm 1, the "largest" left-hand sides are generated first (line 7) by method GENERATEALLMAXIMALLHS. This method structurally copy the LHS of the evolution rule to the migration rule LHSs such that they are as large as possible but still connected. Furthermore, the method ensures that each type is instantiated at least once. Due to inheritance and the fact that the left-hand side of TL is not required to be a connected graph, there may be more than one "largest" left-hand side. Afterwards, additional left-hand sides are generated by computing all connected subgraphs of the largest ones. This is done by method GENERATEALLPOSSIBLESUBGRAPHS (lines 10-12). In addition, separate left-hand-sides are added for all loop edges (having the same source and target vertex) by method GENERATERULESFORLOOPSINTGRAPH (line 15). These are added since an instance of a loop edge in the meta-model can type also non-loops in models. In the next step (lines 18-33), right-hand sides are generated by taking a copy of each left-hand side, retyping it if necessary, and extending it with instances of newly introduced types (specified by the meta-model evolution rule). Note that elements can be glued by morphism *tl* of the meta-model evolution rule. This gluing is reflected by gluing corresponding instance elements in a migration rule (line 21). The extension of right-hand-sides is done by method EXTENDRHS (line 24). This method takes an instance graph I and all new types of TI without TL. This difference specifies all types newly introduced by the meta-model evolution rule. The method extends graph I by instances of new element types as long as graph I stays connected. This method ensures that there is at most one instance per type. Rules saved in the migration scheme (lines 29-32) are those that either create new elements or match elements that later needs to be deleted due to the fact that their type will be deleted. In the final step, we (partially) order these rules such that rules with larger left-hand sides and more specialized types have higher priorities and are matched first since they describe more specific migration cases. Method SORTBY_SUBGRAPHINCL_TYPESPEC (line 34) does this.

Algorithm 1 Generate Migration Scheme

```
 1: function generateMigrationScheme(eRule:Rule)
 2:     // eRule = TL → TI ← TR
 3:     val mRules := new List[NonDeletingRule]()
 4:     val lhsMRules := new Set[Graph]()
 5:
 6:     //Generate maximal LHSs:
 7:     lhsMRules+ =generateAllMaximalLHS(TL)
 8:
 9:     //Generate subgraph LHSs
10:     for all L : Graph in lhsMRules do
11:         lhsMRules+ =generateAllPossibleSubgraphs(L)
12:     end for
13:
14:     //Generate LHSs for loop edges in TL: (unfold)
15:     lhsMRules+ =generateRulesForLoopsInTGraph(TL)
16:
17:     //Generate RHS for each LHS in lhsMRules:
18:     for all L : Graph in lhsMRules do
19:         val I : Graph = copy(L)
20:         retype I to TI //by use of morphisms L → TL → TI
21:         glue elements in I analog to the gluing of TL → TI
22:
23:         //Extend I by connected instances of new types:
24:         extendRHS(I, (TI without TL)))
25:         //map elements of L to their copies in I
26:         val l : Morphism = map(L,I)
27:
28:         //Add multi-rule:
29:         val r = new NonDeletingRule(L − l → I)
30:         if r contains elements to create, merge or delete then
31:             mRules+ = r
32:         end if
33:     end for
34:     return sortBy_SubgraphIncl_TypeSpec(mRules)
35: end function
```

The resulting migration rule scheme may be *customized to special needs*: Well-typed multi-rules may be added or deleted. In addition, multi-rules may be customized by adding or deleting elements while keeping the rule well-typed over the given evolution rule. This way, new meta-model elements may be reflected in various way in migration rules. Furthermore, so-called kernel rules may be added. They are used to relate multi-rules. They may identify new elements to be added only once for a group of multi-rule matches. Of course, kernel rules need to be typed by a given meta-model evolution rule. Furthermore, kernel rules need to fulfill the following well-formedness condition: If a kernel rule relates the left-hand sides of two multi-rules, it also has to relate their

right-hand sides in a compatible way. Finally, the priority of multi-rules may be changed to adapt the migration strategy. However, it may happen that rules become unreachable that way.

In the following, we show how a migration rule scheme can be applied to an individual model: For this purpose, an individual migration rule is generated for a model, the rule can be applied as usual. A *model-specific migration rule* is constructed as follows: Multi-rules are matched to a given model as often as possible obeying relations by kernel rules if existing. Multi-rules $l_i : L_i \to I_i$ for $1 \leq i \leq n$ are matched one after the other (along their priorities). A new match is not allowed to fully cover an already matched model part. Therefore, the priorities of multi-rules are important. If priorities are set in a "wrong" order, more special or larger multi-rules cannot be applied. The left part of the model-specific migration rule $l : L \to I$ is constructed by taking a copy of each multi-rule for each match and by gluing those elements that are related due to their matches. In addition, new elements introduced by multi-rules are glued along their relating kernel rules $l_{ij} : L_{ij} \to I_{ij}$ for $1 \leq i < j \leq n$ if existing. Furthermore, we extend L by all unmatched elements that are typed by TL and I by their retyped counterparts typed by TI. The right part of the model-specific migration rule $r : I \leftarrow R$ is constructed by deleting the elements of $R = I$ that cannot be typed by TR anymore. A schematic illustration is presented in Figure 5. It extends the double cube in Figure 4. (For the formal underpinning of this diagram we refer to [20].)

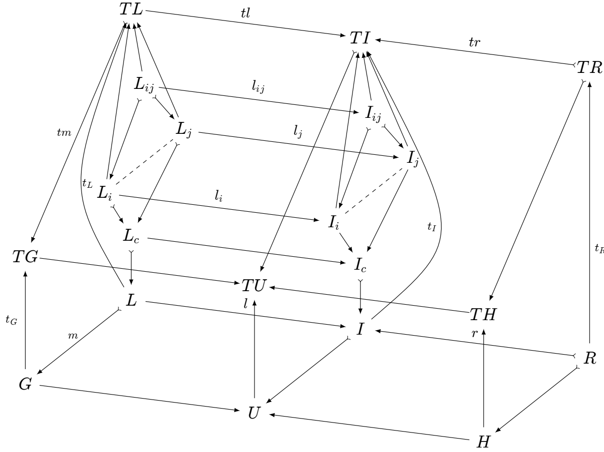

Figure 5: Co-transformation using a migration scheme

4. CUSTOMIZING MIGRATION SCHEMES

The literature on meta-model co-evolution contains several case studies. In particular, the evolution history of open source project have been studied and recurring evolution steps have been observed. An outcome of this work is that there exist several catalogs and list of such evolution steps such as presented in [14]. In this section we will use some prominent examples of such steps to illustrate our approach.

Table 1 on the right shows some typical changes that are listed in related research [14, 3]. Usually these changes are classified into *non-breaking*, *breaking and resolvable*, and *breaking and unresolvable* ones [10]. However, not taking additional constraints into account, we can state that all additions are non-breaking, while all deletions, renames and merges are breaking and resolvable. Breaking and unresolvable changes actually require that there is an additional non-solvable constraint satisfaction problem.

Table 1: Selected meta-model changes

	Name
1.	Rename meta-element
2.	Delete meta-element
3.	Add meta-element
4.	Move meta-property
5.	Pull up meta-property
6.	Push down meta-property
7.	Flatten hierarchy
8.	Inline meta-class

4.1 Rename meta-element.

A meta-element is renamed and model elements need to be retyped. This change is trivial since the type as well as instance graph structure do not change. Therefore, such evolutions can be considered as identity transformations where the name attribute of the meta-element changes and models are not subject of change. Model updates in some modeling frameworks, such as EMF, are due to the fact that type names are often used to encode the typing morphism. However, the only reason why this change is named here is that it is a prominent example. But since their corresponding model migrations are mostly trivial, we do not further elaborate on them.

4.2 Delete meta-element.

Another trivial meta-model change is "Delete meta-element". Therefore, we also do not elaborate on this kind of changes. The derived migration scheme establishes model migration rules with left morphisms being identity morphisms $id := L \to (I = L)$. The right-hand side R is determined by deleting all elements from I that cannot be typed anymore. Customization of the rule scheme is usually not required. More elaborated migration schemes are possible however, if the meta-model evolution rule is extended in TL (and TI). For example, deleted elements may be replaced by new elements of new or existing types. The case is discussed in the following.

4.3 Add meta-element.

"Add meta-element" is often classified as a "non-breaking" change since models do not need to be migrated. Often it is desired however, that new instances of new elements are also added to models in certain ways. On the left of Figure 6, a meta-model evolution rule is shown (in dashed boxes) that creates a new class with two references pointing to existing ones. Note that this evolution rule is non-deleting. Therefore, the figure shows the left part $TL \to TI$ only. The mappings within meta-model evolution rules are given by names to facilitate their understanding as well as the typing of migration rules. For this example rule, the derivation strategy generates exactly two migration rules $r_1 : L_1 \to I_1$ and $r_2 : L_2 \to I_2$ shown in the middle of Figure 6. We have two "largest" multi-rules since TL is unconnected. Applications of

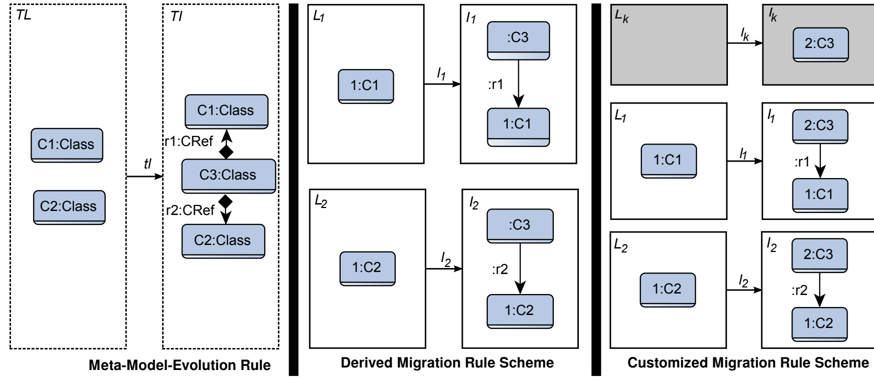

Figure 6: Add meta-element: meta-model evolution rule and migration rule schemes

the derived migration scheme create new referenced instances typed by "C3" for each instance of type "C1" respectively "C2". Sometimes, this migration scheme is already useful as it is. In other cases, the language designer might want to customize it. For example, if class "C3" is meant to be a container class, it should be created only once. In this case, the language designer has to extend the derived migration scheme by a kernel rule as shown on the right of Figure 6 (where the kernel rule has a gray background). The new class "C3" is mapped to class "C3" in both multi-rules. Hence, all the new instances of "C3" added to a model by multi-rule applications are glued so that only one instance of "C3" is created finally. In this customization as well as in all following ones, we do not change the initial rule priorities. In total, only one customization has to be done here.

Figure 7 shows the main graphs of a complete co-transformation (see Figure 4). A container class "WebModel" is added to a meta-model describing web pages. While in the top of the figure, a meta-model evolution step is shown, the bottom part of the figure contains the migration of a concrete model. In the meta-model evolution step, a new container class "WebModel" is introduced containing all pages and entities. Therefore, multi-rule r_1 is matched three times and multi-rule r_2 two times creating five references and five "WebModel" instances in total. However, these five "WebModel" instances are glued to one according to the single kernel match leading to the individual migration transformation presented rule $L \rightarrow R$. (Note that the kernel rule is matched only once since its LHS is empty.)

4.4 Move meta-property.

The top row of Figure 8 shows a meta-model evolution rule that moves an attribute from class "C1" to an associated class "C2". An example where such a rule can be applied is presented on the right of this figure. A security level is moved from class "Server" to class "Service". Below the meta-model evolution rule, the migration scheme is shown consisting of three multi-rules. The first multi-rule copies exactly the behavior of the evolution rule to the instance level (focusing on the left part $L \rightarrow I$). The second derived multi-rule has a left-hand side with LHS=RHS and is matching elements that need to be deleted. This means the original second rule can be used to match all attribute values that have to be deleted from isolated instances of class "C1" i.e. in the example server instances. The rule is marked by an exclamation sign. A tool can and should mark such rules since they specify a

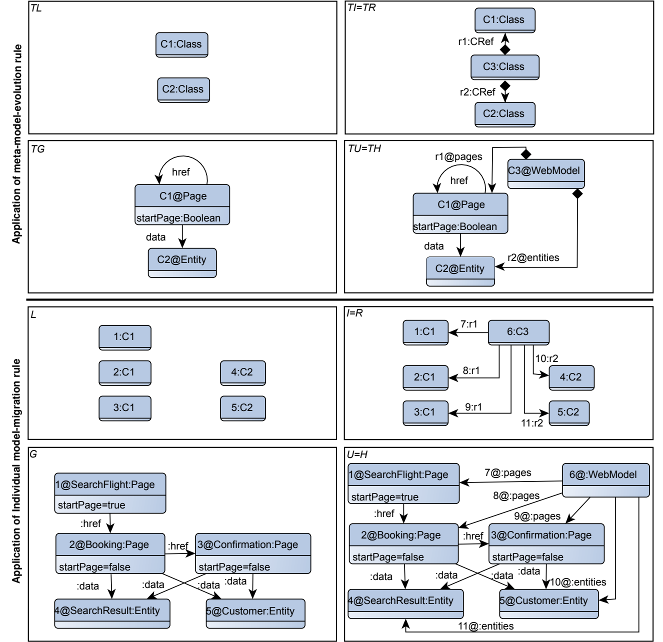

Figure 7: Add meta-element: examples for a co-transformation

real information loss. Figure 8 already shows a customized migration rule scheme and not the derived one. Multi-rule 2 has been customized. The right-hand-side of multi-rule 2 is extended by a referenced class of type "C2" as shown in the figure. It looks similar to multi-rule 1 but differs in class "C2" that is not mapped, but newly created here. In an example model, the security value can e.g. be transfered to a new default service "remote-login". Multi-rule 3 has been added. It can be used to assign a default security level to each service not being connected to a server.

4.5 Pull up meta-property.

The well-known refactoring "Pull up property" is often listed as a recurring meta-model evolution change. In Figure 9, a meta-model evolution rule is presented that pulls up a reference of two subclasses to their super class. The meta-model evolution step is specified by mapping the old references to a new one from the super class. This is allowed in the used formalization of models (see [7]). It is a kind of

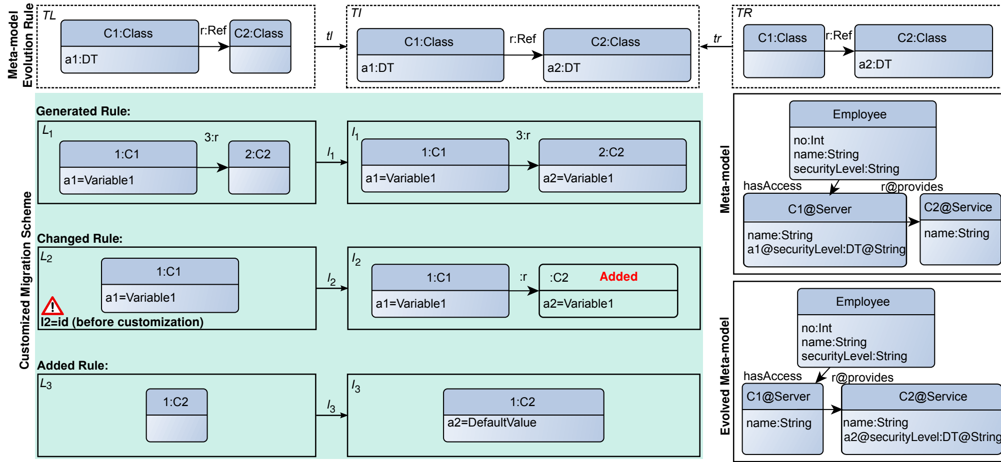

Figure 8: Move meta-property

merge operation and requires only the retyping of instances of $r1$ and $r2$. Our derived migration scheme is the empty one. The required retyping of model elements is achieved by the standard construction. However, e.g. a rule may be added that extends direct instances of superclass "SC", which were not allowed to have a reference of type "r1,r2" before the meta-model change. Such instances may get a reference to a new or existing class. An example for a meta-model change is shown on the right of Figure 9.

4.6 Push down meta-property.

In contrast to "Pull up meta-property", the opposite change "Push down meta-property" cannot be presented by a non-deleting rule. Therefore model migration is not solved by retyping here. This is due to the fact that references cannot be mapped to references of subclasses (see [7]). This time, references have to be replaced by new ones. See the evolution rule at the top of Figure 10. The default migration is reflected by rules r_1, r_2 and r_3. The deletion in right-hand sides is not shown since it is standard (deleting all references of type "r3" in this case). Note that one rule is again marked by an exclamation sign warning against information loss. However, this time the information loss is hard to avoid. One possibility is to retype direct instances of class "SC" to a subtype. This can be a customization.

4.7 Flatten hierarchy.

Figure 11 shows a meta-model evolution rule for "Flatten hierarchy". Subclasses are merged into their superclass. The derived customization scheme is also in this case empty.

4.8 Inline meta-class.

Figure 12 shows a meta-model evolution rule similar to the flatten hierarchy rule but without inheritance. This time, a class and an associated class are merged. Interestingly, the derived migration scheme is not empty. Instead, the derived migration scheme specifies also the merge of instances that are related by a corresponding link. If this is not the intended migration, a customization can be e.g. to delete multi-rule r_1 so that instances are retyped only. In the figure, no customizations is applied.

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 [19, 17, 23, 28]. Especially database schema evolution has been a subject of research in the last decades. 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. [13]). Current co-evolution approaches can be classified into *manual specification, operator based and meta-model matching* approaches [24].

Manual specification approaches like [27, 25] consider two meta-model versions as given and 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 automatically copied are those that have unchanged or compatibly changed types. New elements in the meta-model are basically not considered during model migration and therefore have to be taken into account in manually defined migration specifications. The reuse of migration knowledge in different meta-models is usually not supported. In [27] Sprinkle et al. propose a manual specification approach that has been implemented in the model change language (MCL) [18]. MCL allows the domain designers to express model migrations in graphical syntax on a high abstraction level similar as our proposed language. However, MCL supports only a small set of language primitives that allow a limited number of migrations only, addressing changes to specific model elements locally. While such migration definitions for a concrete meta-model are easy to understand, they are not reusable, in contrast to ours. Rose et al. presents their tool Epsilon Flock in [25]. It is also a manual specification approach that targets to the migration of models only but has a textual syntax. Similarly to MCL, rules for unchanged or slightly unchanged 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 approach, Epsilon Flock rules are not type checked. Moreover, there is no warning if elements are deleted without replacement. This occurs if a migration script is not valid according to the typing morphism or incompletely defined.

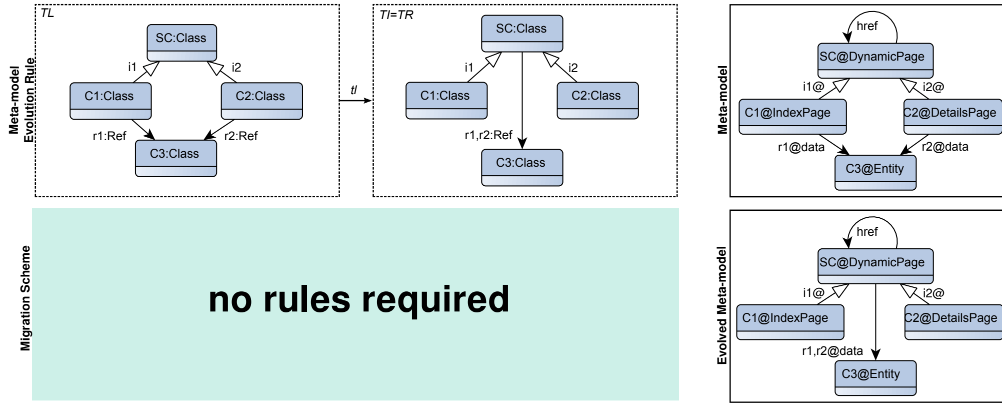

Figure 9: Pull up meta-property

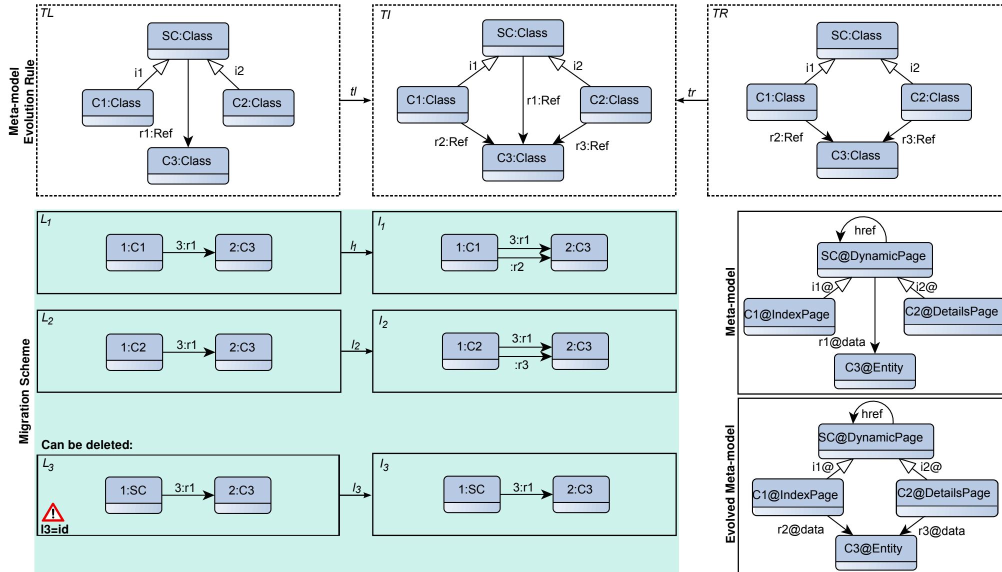

Figure 10: Push down meta-property

In *operator-based* approaches, a meta-model is evolved 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 as for database schema migration (see e.g. [13, 4]). However, researchers in MDE realized that a fixed set of reusable coupled evolution-migration operations is not enough for meta-model co-evolution [13]. Therefore, current approaches allow to extend model migration transformations by manually written code using a general purpose or a special transformation language. Cope/Edapt [13] is a meta-model evolution tool for EMF that have been used in industrial projects. As our approach, Cope/Edapt 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 already 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 as Groovy/Java program without any support to ensure well-defined migration rules. In contrast, we support the specification of custom evolution rules coupled to generated migration schemes that can be adapted. As our approach, Cope/Edapt supports in-place transformations.

Meta-model matching approaches consider two versions of a meta-model as given. An evolution history [3, 9, 24, 15] 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 operations. Meta-model differencing and detecting meta-model evolution operations automatically has been considered by Ciccetti et al. in [3] and is also in the focus of the Atlas Matching Language (AML) [9, 24]. In both approaches, a migration script in ATL (Atlas Transformation Language) is generated. Ciccetti et al. focus in [3] on ordering a set of detected meta-model evolution operations so that sequence of migration operations can be applied. Garces et al. focus in [9] on heuristics to detect applied meta-model evolution operations by inspecting simple changes. In [15] Kehrer et al. present an algorithm to deduce edit operations from a

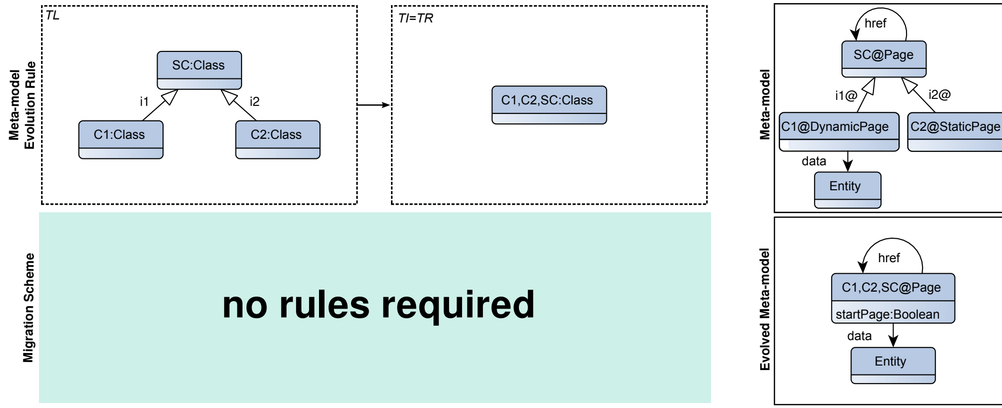

Figure 11: Flatten hierarchy

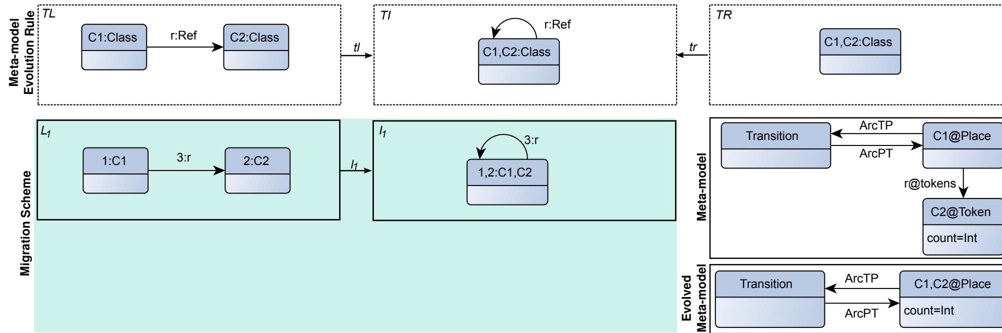

Figure 12: Inline meta-class

given set of model changes. This algorithm is implemented on the basis of the Eclipse Modeling Framework.

König et al. present in [16] a formal approach for induced data migration after data model refactoring. It can also be used for model migration. In contrast to our approach, their approach migrates data fully automatically. However, model migrations cannot always be fully automatically determined. Different migration variants may be possible.

The concept of amalgamation in graph transformation was introduced in the 1980s in [2]. In [1], the classical concept of amalgamated graph transformations [2, 29] is extended to multi-amalgamation and applied to model transformation. Here we use amalgamated graph transformation differently to the traditional approach where kernel-rules are matched first and the complete amalgamated match does not cover all elements of specific types.

6. CONCLUSION

To support the continuous evolution of domain-specific modeling languages, an approach for automatic deduction of model migration schemes from meta-model evolution rules is presented. The deduction algorithm implements a general heuristic ensuring that resulting migration rule schemes fit to their meta-model evolution rules. The automatically deduced migration rule schemes are able to migrate all models of a given modeling language so that resulting models are well-typed. In addition, rule schemes may be customized to special needs. It is allowed to change rules as well as to create and delete them as long as the resulting scheme still fits to its meta-model evolution rule and do not specify conflicts. All main concepts of customizable migration rule schemes and their automatic deduction are formally defined based on graph transformation ensuring well-definedness of deduced schemes. A number of well-known meta-model evolution operations are considered as examples including a discussion of suitable customizations for each one. It turns out that (1) meta-model evolution rules may be individual, (2) automatically deduced migration schemes are meaningful in considered cases, and (3) intended customizations can be expressed within migration rule schemes. In the future, systematic case studies should be performed to get a clearer picture of the potentials and limitations of this approach. Furthermore we are working on a prototypical implementation. In addition it could be interesting to combine our approach with a meta-model differencing technique.

7. ACKNOWLEDGMENTS

This work was partially funded by NFR project 194521 (FORMGRID).

8. REFERENCES

[1] E. Biermann, H. Ehrig, C. Ermel, U. Golas, and G. Taentzer. Parallel Independence of Amalgamated Graph Transformations Applied to Model Transformation. In *Graph Transformations and*

Model-Driven Engineering, volume 5765 of *LNCS*, pages 121–140. Springer, 2010.

[2] P. Boehm, H.-R. Fonio, and A. Habel. Amalgamation of graph transformations: a synchronization mechanism. *J. Comput. Syst. Sci.*, 34(2-3):377–408, June 1987.

[3] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating Co-evolution in Model-Driven Engineering. In *EDOC 2008*, pages 222–231. IEEE, 2008.

[4] C. Curino, H. J. Moon, M. Ham, and C. Zaniolo. The PRISM Workwench: Database Schema Evolution without Tears. In Y. E. Ioannidis, D. L. Lee, and R. T. Ng, editors, *ICDE 1999*, pages 1523–1526. ICDE 1999, 2009.

[5] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Fundamental Theory for Typed Attributed Graphs and Graph Transformation based on Adhesive HLR Categories. *Fundam. Inform.*, 74(1):31–61, 2006.

[6] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, March 2006.

[7] H. Ehrig, F. Hermann, and U. Prange. Cospan DPO Approach: An Alternative for DPO Graph Transformation. *EATCS Bulletin*, 98:139–149, 2009.

[8] M. Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[9] K. Garcés, F. Jouault, P. Cointe, and J. Bézivin. Managing model adaptation by precise detection of metamodel changes. In R. F. Paige, A. Hartman, and A. Rensink, editors, *ECMDA-FA 2009*, volume 5562 of *LNCS*, pages 34–49. Springer, 2009.

[10] B. Gruschko, D. Kolovos, and R. Paige. Towards Synchronizing Models with Evolving Metamodels. In D. Tamzalit, editor, *MoDSE 2007*, March 2007.

[11] A. Habel and K.-H. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *MSCS*, 19(2):245–296, 2009.

[12] F. Hermann, H. Ehrig, and C. Ermel. Transformation of Type Graphs with Inheritance for Ensuring Security in E-Government Networks. In M. Chechik and M. Wirsing, editors, *FASE 2009*, volume 5503 of *LNCS*, pages 325–339. Springer, 2009.

[13] M. Herrmannsdoerfer, S. Benz, and E. Jürgens. COPE - Automating Coupled Evolution of Metamodels and Models. In S. Drossopoulou, editor, *ECOOP 2009*, volume 5653 of *LNCS*, pages 52–76. Springer, 2009.

[14] M. Herrmannsdoerfer, S. Vermolen, and G. Wachsmuth. An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models. In B. A. Malloy, S. Staab, and M. van den Brand, editors, *SLE 2010*, volume 6563 of *LNCS*, pages 163–182. Springer, 2010.

[15] T. Kehrer, U. Kelter, and 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.

[16] H. König, M. Löwe, and C. Schulz. Model Transformation and Induced Instance Migration: A Universal Framework. In A. da Silva Simão and C. Morgan, editors, *SBMF 2011*, volume 7021 of *LNCS*, pages 1–15. Springer, 2011.

[17] R. Lämmel. Grammar Adaptation. In *FME*, pages 550–570, 2001.

[18] T. Levendovszky, D. Balasubramanian, A. Narayanan, F. Shi, C. Buskirk, and G. Karsai. A semi-formal description of migrating domain-specific models with evolving domains. *Software and Systems Modeling*, pages 1–17, January 2013.

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

[20] F. Mantz, G. Taentzer, and Y. Lamo. Well-formed Model Co-evolution with Customizable Model Migration. *ECEASST*, page (accepted paper).

[21] F. Mantz, G. Taentzer, and Y. Lamo. Co-Transformation of Type and Instance Graphs Supporting Merging of Types with Retyping. In *GCM 2012*, pages 47–58, September 2012. `gcm2012.imag.fr/proceedingsGCM2012.pdf`.

[22] B. Meyers, M. Wimmer, A. Cicchetti, and J. Sprinkle. A generic in-place transformation-based approach to structured model co-evolution. *ECEASST*, 42, 2011.

[23] M. Pizka and E. Juergens. Automating Language Evolution. In *TASE 2007*, pages 305–315, Washington, DC, USA, 2007. IEEE Computer Society.

[24] L. Rose, M. Herrmannsdoerfer, J. R. Williams, D. Kolovos, K. Garcés, R. F. Paige, and F. A. C. Polack. A Comparison of Model Migration Tools. In D. C. Petriu, N. Rouquette, and Ø. Haugen, editors, *MoDELS 2010*, volume 6394 of *LNCS*, pages 61–75. Springer, 2010.

[25] L. Rose, D. Kolovos, R. F. Paige, and F. A. C. Polack. Model Migration with Epsilon Flock. In L. Tratt and M. Gogolla, editors, *ICMT 2010*, volume 6142 of *LNCS*, pages 184–198. Springer, 2010.

[26] A. Rutle, A. Rossini, Y. Lamo, and U. Wolter. A formal approach to the specification and transformation of constraints in MDE. *JLAP*, 81(4):422–457, 2012.

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

[28] J. Sprinkle and G. Karsai. A domain-specific visual language for domain model evolution. *J. Vis. Lang. Comput.*, 15(3-4):291–307, 2004.

[29] G. Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems*. PhD thesis, TU Berlin, 1996.

[30] G. Taentzer, F. Mantz, and Y. Lamo. Co-Transformation of Graphs and Type Graphs With Application to Model Co-Evolution. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *ICGT 2012*, volume 7562 of *LNCS*, pages 326–340. Springer, 2012.