On Formalizing EMF Modeling Operations with Graph Transformations

Petra Brosch, Sebastian Gabmeyer, Gerti Kappel
Vienna University of Technology, Austria
eMail: {lastname}@big.tuwien.ac.at

Martina Seidl
Johannes Kepler University Linz, Austria
eMail: martina.seidl@jku.at

Abstract

The development of software in accordance with the model-driven engineering paradigm places model transformations at a central position. Desirable yet contradicting properties of model transformations are user-friendliness as offered by by-demonstration approaches and formal conciseness as provided by algebraic graph transformations which is indispensable for verification tasks.

In this paper, we show how to unite the properties of the two different approaches. We employ the state-of-the-art by-demonstration environment Emo *to prototype graph transformations by embedding the operations obtained from* Emo *in the formal framework of graph transformation theory.*

Introduction

Model transformations are the core technology of model-driven engineering (MDE) to convert, transform, generate, and evolve software models. Their area of application ranges from refactoring to translating between different modeling languages to synthesizing program code, to name but a few examples. Due to the central role of model transformations in MDE a rich and diverse landscape of model transformation approaches has emerged over the last years, which follow different, partly orthogonal design objectives in order to effectively specify and realize modifications on models [CH06, MVG06].

As models may be viewed and treated as graphs, the *algebraic graph transformation theory* [Roz97, EEPT06] may be employed to describe model transformations in a formal, declarative, and rule-based fashion. The algebraic graph transformation theory provides a powerful framework, which allows us not only to analyze systems of transformations with respect to their, e.g., confluence and termination properties, but also to define the precise semantics of model transformations as, for example, presented in [BET11].

The focus of *by-demonstration* based model transformation approaches lies on usability. This goal is achieved by providing an easy-to-use environment, where the user *demonstrates* the desired transformation with an example. The approach offers the advantage that the bulk of the transformation specification is automatically generated based on this demonstration; so, the user may focus on fine tuning the derived specification with little to no programming effort.

We aim to combine the two approaches with the intention to develop a by-demonstration environment, which generates executable, algebraic graph transformations. Verification mechanisms inherent in the formal framework of graph transformation can then be used to reason about the properties of the model transformation. The by-demonstration tool EMF Modeling Operations (Emo) is used as a starting point for this task. It was chosen (a) due to its usage of Ecore, a popular implementation of the Essential Mof standard[1], (b) due to the open source availability of its implementation, and (c) for reasons of familiarity with the platform.

In this paper, we formally describe how to express Emo's transformation concepts with algebraic graph transformation theory based on the double-pushout (DPO) approach. Therefore, we first revisit Emo, before we present its formalization illustrated by the Pull-Up Field refactoring. We review related work and conclude with future work.

EMF Modeling Operations

Emo provides a by-demonstration environment to develop in-place model transformation for Ecore-based models [BLS+09]. In the following, we review Emo's features. For the details, we kindly refer to [BLS+09].

Model transformation by demonstration. To specify a model transformation, which in Emo's parlance is called an *operation*, the user constructs the so-called *initial model*. The initial model contains *all but no more than* the necessary elements that need to be present in a model to which the operation shall be applied. Emo stores a snapshot of the initial model once the user has specified all necessary elements. Next, the user demonstrates the intended transformation on the initial model by either adding new elements, or by deleting and modifying existing ones. The model resulting from these modifications is called the *revised model*. Analogous to the initial model, the revised model contains *all but no more than* the necessary elements that need to be present in a model *after* the operation has been applied to it. Emo carries out a state-based comparison between the initial and the revised model. This comparison lists all matching elements (*matches*) and the differences (*diffs*) between the two models. While the diffs list all those elements that have been modified, added, or deleted by the operation, the matches identify all elements that have been preserved. Therefrom, Emo automatically derives an executable operation. Hereby, it generates a so-called *precondition template* for each element in the initial model and a so-called *post-condition template* for each element in the revised model. A *template* stores the type of the model element for which it was generated. Automatically derived OCL conditions describe each of the element's admissible attribute values. Thus, to apply an operation to a model the type of each precondition

[1] `http://www.omg.org/spec/MOF`

template must be matched by some element and each of the template's conditions must be satisfied by a corresponding attribute of the matched element. In the resulting model all postcondition templates need to match a model element of corresponding type with adequate attribute values.

The automatically generated OCL conditions are usually too restrictive. Hence, in a post processing step EMO removes all OCL conditions of those attributes which are initialized to a string literal. This is done before the generated operation is returned back to the user. The user may then refine the templates of the generated operation manually by means of relaxation, enforcement, and deactivation of existing conditions or by introducing additional ones.

EMO is implemented as an Eclipse plug-in[2] that offers a graphical interface and is able to integrate any editor generated by the Graphical Modeling Framework (GMF), within which the initial and the revised model may be created and edited. Alternatively, XMI serializations of initial and revised models produced by external tools can be loaded. So the specification of the model transformation can be done in the modeler's favorite editor without any programming effort what fosters user-friendliness. Language independence is given by using Ecore on the metamodel level.

Specification defects and EMO operations. Even EMO operations are not immune to bugs. Consider the following example. Given a metamodel describing a simple UML-like class diagram that consists of named classes and associations. Associations are defined by two association ends and a name. An association end has a multiplicity and may either be a composition end or plain. A constraint on the metamodel states that if an association end is set to a composition end then the other association end must be plain. For instances of this metamodel we define model transformation for the following use case: a modeler wants to select from the set of associations those whose name is set to has and alter their association end to a composition end if its multiplicity is set to one. A modeler employing a by-demonstration approach may start with an initial model consisting of two classes and an association with name has and an association end with multiplicity one. During the demonstration of the transformation the association end with multiplicity one is changed to a composition and an initial transformation is generated.

Figure 1 depicts on the left hand side three models to which the above specified model transformation is applied and displays the results on the right hand side. The application of the transformation to the first two models yields the intended result. The application to the third model surfaces an error in the model transformation as it can be applied twice resulting in a model that violates the well-formedness constraint of the metamodel, which states that only one association end may define a composition. To fix this problem the preconditions of the model transformation need to be strengthened. To detect problems of this kind, verification

[2] `http://www.modelversioning.org/emf-modeling-operations`

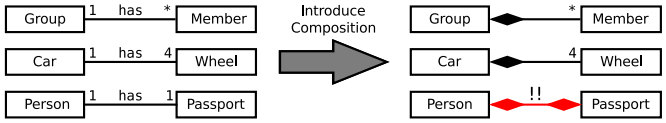

Figure 1: Application of *Introduce Composition on 3 models*

techniques like model checking can be consulted. Therefore, a formal model of EMO's transformation concept is required. In this paper, we provide such a model based on algebraic graph transformations.

Formalizing EMO Operations

In this section we formalize EMO's transformation process and its underlying concepts using the algebraic graph transformation theory which is briefly[3] reviewed in the following.

Algebraic graph transformation theory

The algebraic graph transformation theory offers a set of formal, mathematically founded techniques to manipulate graphs in a declarative and rule-based fashion. Following the notions of term rewriting, a graph rewriting rule consists of a left-hand side (LHS) and a right-hand side (RHS), which are both defined in terms of graphs. A rewriting rule may be applied to a graph, called the *host graph*, if a subgraph matching the rule's LHS is found in this graph. Therefore, the LHS may be viewed as defining the precondition of the rule, while the RHS depicts its postcondition. The modifications performed by a rule are implicitly derived from the differences between the LHS and the RHS. The application of a rule to a host graph is straight forward [Hec06]: After finding an occurrence of the LHS in the host graph, all nodes and edges of the LHS, which are not part of the RHS, are removed, while all nodes and edges of the RHS not part of the LHS are added to the host graph.

A graph $G = (G_N, G_E, src_G, tar_G)$ consists of a set G_N of nodes, a set G_E of edges, and source and target functions $src_G : G_E \to G_N$ and $tar_G : G_E \to G_N$ that map the source and the target of each edge $e \in G_E$ to a node $n \in G_N$.

A mapping between the nodes and edges of two graphs is called a *graph morphism*. A graph morphism $f = (f_N, f_E) : A \to B$ between graphs A and B is defined as tuple (f_N, f_E) where $f_N : A_N \to B_N$ describes the mapping between the nodes of A and B, and $f_E : A_E \to B_E$ describes the mapping between their edges. A graph morphism $f : A \to B$ is *structure-preserving* if the source and the target of each edge is preserved by the morphism, e.g., $src_B \circ f_E(e) = f_N \circ src_A(e)$ for all $e \in A_E$.

A double-pushout (DPO) graph rewriting rule or graph production[4] p is defined as a span of graphs $L \leftarrow K \rightarrow R$,

[3] For more information on the algebraic graph transformation theory the reader is kindly referred to [Roz97] and [EEPT06].

[4] We will use the terms (graph) rewriting rule and (graph) production interchangeably.

which consists of a left-hand side L, a right-hand side R, a so-called *interface graph* K, and morphisms $l : K \to L$ and $r : K \to R$. Here, the interface graph obtains its name from the fact that it consists of all nodes and edges that are present in both L and R, i.e., $K = L \cap R$. The elements of K are thus preserved by the production.

A production p transforms a *host* graph G into its *derived* graph H if there exists a *match* of nodes and edges from L to G. A match is thus a graph morphism $m : L \to G$. Once such a match is found the production may be applied to G by constructing two so-called *pushouts* (hence the name). A pushout constructs a graph X as the union of two graphs B and C which are "glued" together along a common subgraph A. That is, X is constructed in such a manner that all elements of B and C, which have a common pre-image in A, are added to X, and all other elements present in either B or C are attached to this common subgraph. When a DPO production is applied to a graph G at a match m, a graph D is derived in such a manner that graph G is a pushout object of L and D with common subgraph K. Then, graph H is derived as a pushout object from graphs R and D with common subgraph K.

One distinguishing feature of graph transformation theory is the duality principle asserting for each construction the existence of an inverse, i.e., a dual construction. The dual of a pushout is a *pullback*. A pullback constructs a graph A as the intersection of those elements of two given graphs B and C which have a common image in X.

To control the admissible structure of graphs a *type graph* may be defined. A graph is called an *instance graph* of a type graph if it conforms to the type graph's structure. A type graph is in fact the graph-theoretic equivalent to a metamodel in MDE. Further extensions are necessary to represent EMF models as graphs and to describe transformations of EMF models with algebraic graph rewriting rules. These are presented in [BET11]:

- *Attributes* are used to associate additional information with nodes and edges. The value of an attribute is represented by a *data node*, which is connected to a node or an edges by means of a node-attribute or an edge-attribute edge, respectively. The valid values of a data node are described by its domain, which is usually specified as a Σ-(term-)algebra, possibly many-sorted and allowing variables (see, e.g., [EEPT06]).

- *Inheritance* relations may be used to establish type hierarchies. A type graph with inheritance is an extension of an attributed type graph and has a set I of inheritance edges and a set A of abstract, non-instantiable nodes. Attributed type graphs with inheritance may reuse the existing theory of attributed type graphs if the inheritance hierarchy is flattened out [dLBE+07].

- *Composition* or *containment* relations allow us to build composite objects from a set of constituting parts such that the life cycle of the parts is bound to the composite objects' life cycle. An attributed type graph with inheritance is extended by a set C of containment edges to support composition. To ensure that a graph transformation is consistent w.r.t. composition additional constraints need to be taken into consideration, e.g., a production may not introduce any containment cycles (see [BET11]).

- *Multiplicities* on edges and nodes may be realized by so-called *graph constraints* as shown in [TR05].

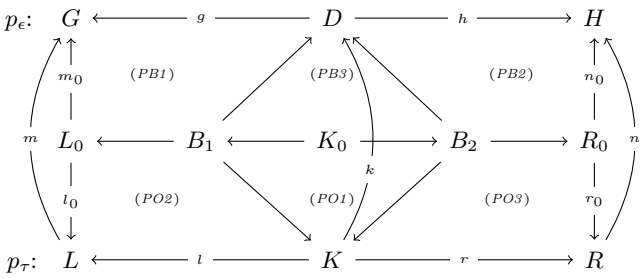

Figure 2: Deriving a template graph production p_τ from p_ϵ

With these extensions it is possible to describe EMF models as attributed, typed graphs with inheritance, composition, and multiplicities. As a result we can specify productions that transform such graphs and we are thus able to describe model transformations of EMF models with algebraic graphs productions as shown in the following.

Derivation of a graph production

In the following we describe the derivation steps performed by Emo to construct an operation from a formal point of view. We show that the steps necessary to derive an operation from an initial model, a revised model, and their differences allows us to construct a DPO graph rewriting rule that precisely captures the transformation semantics of the Emo operation. In what follows we decompose the steps that are performed by Emo to construct an operation from a demonstration. In this way, we obtain a graph rewriting rule of the form $L \leftarrow K \rightarrow R$ where L and R represent the graph-based equivalents of Emo's pre- and postcondition templates. The derivation of a DPO production based on an Emo operation is depicted in Fig. 2.

Algebraic construction of the example production. Emo derives a model transformation on the basis of a state-based comparison between the initial and the revised model. We denote the initial model by G and the revised model by H. We first construct a graph rewriting rule $G \leftarrow D \rightarrow H$ where D denotes the interface graph of G and H. By definition, an interface graph contains all elements which are preserved by the graph rewriting rule. Thus, $D = (D_N, D_E)$ is constructed in such a way that $D_N = G_N \cap H_N$ and $D_E = G_E \cap H_E$. To identify elements which are present

in both G and H we use the results of the state-based comparison that, besides highlighting the differences, also identifies the elements of the initial model that are still present in the revised model. In this way, the state-based comparison induces a partial function *match* from G to H that identifies the common elements of graphs G and H. We define *match* formally by $match$: $G \to H$ and denote by $match(p)$ the application of *match* to an element $p \in G$ which returns an element $q \in H$ or *undefined* if $match(p)$ is not defined. Note that the state-based comparison does not match two different elements from the initial model to the same element in the revised model. Consequently, no two elements in G, for which *match* is defined, are mapped to the same element in H and the implication $\forall p_1, p_2 \in G.\ p_1 \neq p_2 \implies match(p_1) \neq match(p_2)$ always holds. The interface graph D is constructed such that for each element p' in the set $G' = \{p \in G | match(p) \in H\}$ a unique element $r' \in D$ is created and $g(r) = p'$ and $h(r) = match(p')$ for morphisms $g : D \to G$ and $h : D \to H$. Then, D consists of all elements in $G \cap H$ and morphisms g and h are injective. We now obtain a DPO graph production $p_\epsilon : G \leftarrow D \rightarrow H$, called the *example production*.

Construction of the template graphs. Next, we construct the so-called *template graphs* L_0 and R_0, which host the initial set of pre- and postcondition templates, respectively. Recall that Emo constructs exactly one precondition template for each element in the initial model. Similarly, it constructs exactly one postcondition template for each element in the revised model. This construction induces injective morphisms $m_0 : L_0 \to G$ between the initial model and the precondition template graph, and $n_0 : R_0 \to H$ between the revised model and the postcondition template graph. Note that a template differs from an element contained in either the initial or the revised model only in the range of allowed values which may be assigned to its attributes: While attributes in the initial or revised model may be assigned constants or literal values only, the attributes in the template may contain arbitrary OCL expressions. We use a higher-order, multi-sorted term algebra with variables to represent OCL expressions, which may be assigned to attributes of the a template. Note, however, that if we restrict the assignable values of an attribute in the initial and the revised model to constant and ground terms, i.e., terms without variables, we do not need to introduce two distinct value domains, but simply use the mentioned term algebra.

In fact, we can use the very same term algebra to represent the set of assignable values for the attributes in the initial and the revised model if we restrict the set of assignable values to constant and ground terms, i.e., terms without variables.

The use of a term algebra as our value domain requires two additional morphisms to describe the instantiation and the evaluation of terms. We use an *assignment morphism* to describe the assignment of concrete values to variables of a term, thus producing a ground term. An *evaluation morphism* is then used to interpret the operations of a grounded

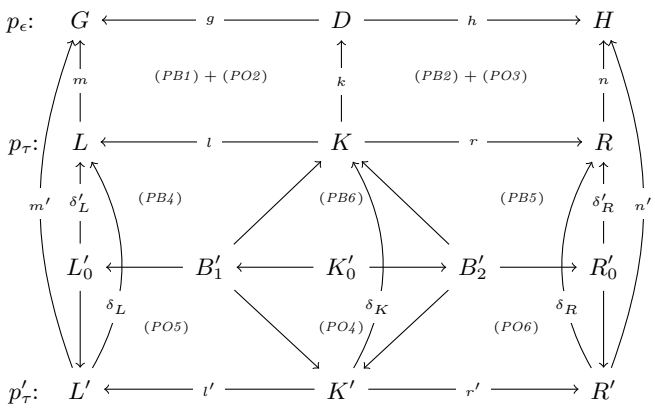

Figure 3: Refining the template graph production p_τ to p'_τ

term which is then evaluated. Roughly spoken, the assignment and the evaluation morphism are equivalent in effect to term unification and term simplification, respectively. For example, suppose a term $s(s(x))$ where x is some integer value, which is *assigned* to $x \mapsto 0$ and results in the ground term $s(s(0))$, and $s(.)$ is interpreted as the successor function; then the term *evaluates* to $s(s(0)) = s(1) = 2$. Note that the assignment and the evaluation morphism establish a mapping between terms assigned to attributes and thus become part of an (attributed) graph morphism that maps nodes, edges, and attributes between two attributed graphs.

Extraction of the template graph production. Finally, we extract a DPO production from the span of graphs $L_0 \to G \leftarrow D \to H \leftarrow R_0$. In a first step, we perform two pullback constructions, $(PB1)$ and $(PB2)$, and obtain the intermediate template graphs B_1 and B_2. The third pullback construction $(PB3)$ creates graph K_0 as the intersection of B_1 and B_2.

The sequence of pushouts $(PO1)$–$(PO3)$ constructs the template graphs L, K, and R. This allows us to define the template graph production $p_\tau : L \leftarrow K \rightarrow R$ which represents the initial, executable operation automatically generated by Emo. However, this construction by itself does not add any real value as up until now p_τ is equivalent to p_ϵ except that the attribute values are no longer restricted to constants and ground terms (this will become clearer after working through the example in the next section).

Refinement of the template graph production. However, since attributes are no longer restricted to constants and ground terms, we are now able to formally describe the refinements that a user may introduce by means of relaxation, enforcement, deletion, or addition of conditions.

A refinement of a template graph A is a *refined graph* A' such that there exists an injective morphism $\delta : A' \to A$, called the *refinement morphism*. The identity morphism $id_A : A \to A$ is a trivial refinement morphism. The refinement of a template graph production $p_\tau : L \leftarrow K \rightarrow R$ results in a refined production $p'_\tau : L' \leftarrow K' \rightarrow R'$ if there exist

injective refinement morphisms $\delta_L : L' \to L$, $\delta_K : K' \to K$, and $\delta_R : R \to R'$.

A refinement p'_τ of p_τ is constructed according to Fig. 3. In fact, the construction of a refinement is similar[5] to the previously explained derivation of p_τ from p_ϵ. Starting from a template graph production $p_\tau : L \leftarrow K \rightarrow R$ the user introduces and provides a set of refinements to either L, R, or both. These refinements are represented by graphs L'_0 and R'_0, which are injectively mapped to L and R by $\delta'_L : L'_0 \to L$ and $\delta'_R : R'_0 \to R$. With a sequence $(PB4)$–$(PB6)$ of pullbacks we construct the intermediate graphs B'_1, B'_2, and K'_0. Finally, the subsequently performed pushouts $(PO4)$–$(PO6)$ construct the graphs of the refined template graph production $p'_\tau : L' \leftarrow K' \rightarrow R'$.

Due to the compositionality of pushouts and by requiring refinement morphisms to be injective we thus ensure that no refinement results in a refined template graph production $p''_\tau : L'' \leftarrow K'' \rightarrow R''$ for which there exists either (a) no morphism $m'' : L'' \to G$, or (b) no morphism $n'' : R'' \to H$, or (c) both. In other words, a refinement must not result in a production that is unable to transform the initial model G to the revised model H. The restriction to injective morphisms ensures the soundness of this refinement construction.

The Pull-Up Field Refactoring

For the purpose of this example (see Fig. 4) we will use a simple metamodel/type graph[6] that describes the valid structure of class diagrams. The metamodel consists of Class and Attribute types. Classes may be organized in inheritance hierarchies. An inheritance relationship is visualized as an open arrow pointing from the subclass to the superclass. A class contains zero or more attributes which is realized as a composition relation and visualized by a diamond-shaped arrow head starting at the container element. Classes and attributes are identified by their name. Moreover, attributes have a type.

The model transformation we want to implement is a simplified version of the Pull-Up Field refactoring (*PUFR*) which works as follows: Given a class hierarchy with a set SUB of classes all having in common (a) an attribute ATT of equivalent name and type and (b) a superclass SUP, pull ATT up to SUP and delete all occurrences of ATT from the classes in SUB. For the sake of simplicity we will restrict our implementation of the *PUFR* to two subclasses, and we will not check if SUP already contains an attribute whose name is equivalent to that of ATT, but whose type differs from that of ATT (in which case the *PUFR* should not be applied).

The demonstration of the Pull-Up Field refactoring within Emo is straight forward. First, the initial model (cf. with graph G in Fig. 4) is created with three classes c1, c2, and c3, named "Person", "Student", and "Teacher", of which the first is the superclass of the latter two. Classes "Student" and "Teacher" both have an attribute a1 and a2, named "Address" of type String. Now, we delete attribute a2 and the composition relation between class c2 and attribute a1. Attribute a1 is now pulled up by creating a composition relation between c1 and a1 (cf. with graph H in Fig. 4). Upon completion of these modifications Emo performs the state-based comparison between the initial and the revised model, which identifies the preserved elements as shown in graph D in Fig. 4. The automatic generation of the pre- and post-condition templates results in intermediate graphs L_0 and R_0. Note that Emo's post-processing step removes all string-valued attributes and replaces them by variables. Variable values are assigned dynamically during the matching phase and thus act as a wildcard expression, which is indicated by the '$*$'.

Next, we refine the automatically generated operation which will allow us to pull up fields of arbitrary, but equivalent type, i.e., the set of fields eligible for pull-up is no longer restricted to fields with type String. This is achieved by introducing a variable for the type value of attribute a1 and requiring the type value of attribute a2 to be equivalent to that of attribute a1 with the expression a2.type = a1.type. Further, we need to constraint the name value of attribute a2 to be equivalent to the name of attribute a1 with the expression a2.name = a1.name. We include these refinements into graph L'_0[7] and leave graph R unchanged, hence, $R = R'_0$. Then, the final template graph production $p'_\tau : L' \leftarrow K' \rightarrow R'$ (displayed in the last row of Fig. 4) is derived from the span $L'_0 \to L \leftarrow K \rightarrow R \leftarrow R'_0$.

The advantage of having a graph rewriting representation of the *PUFR* model transformation becomes obvious if we want to analyze, e.g., its confluence properties with other graph rewriting rules, for example, a Move Field refactoring. In [MTR05], Mens et al. perform such an analysis of common object-oriented refactorings represented as graph rewriting rules with the critical pair analysis feature of Agg.

Related Work

In the following, we first compare out derivation and refinement procedure to the minimal rule extraction technique. Next, we discuss related model transformation by-demonstration (MTBD) approaches. And finally, we survey approaches supporting the specification of graph transformations.

Extraction of a minimal rule. In [BHE08] Bisztray et al. present the minimal rule extraction technique that, given an injective span $G \leftarrow D \rightarrow H$ of graphs, produces the *minimal* graph production $p_\mu : L \leftarrow K \rightarrow R$ that transforms G

[5] Intuitively, the similarity between the construction of a derivation and the construction of a refinement seems only natural if the derivation of p_τ from p_ϵ is viewed as the first refinement step.

[6] In this section we will use the terms metamodel and type graph, model and graph, and (graph) production and model transformation interchangeably.

[7] Note that due to space constraints graphs L'_0, B'_0, K'_0, B'_2, and R'_0 are shown collapsed, yet contain the same information as graphs L'_0, K' (for graphs B'_0, K'_0, B'_2), and R', respectively.

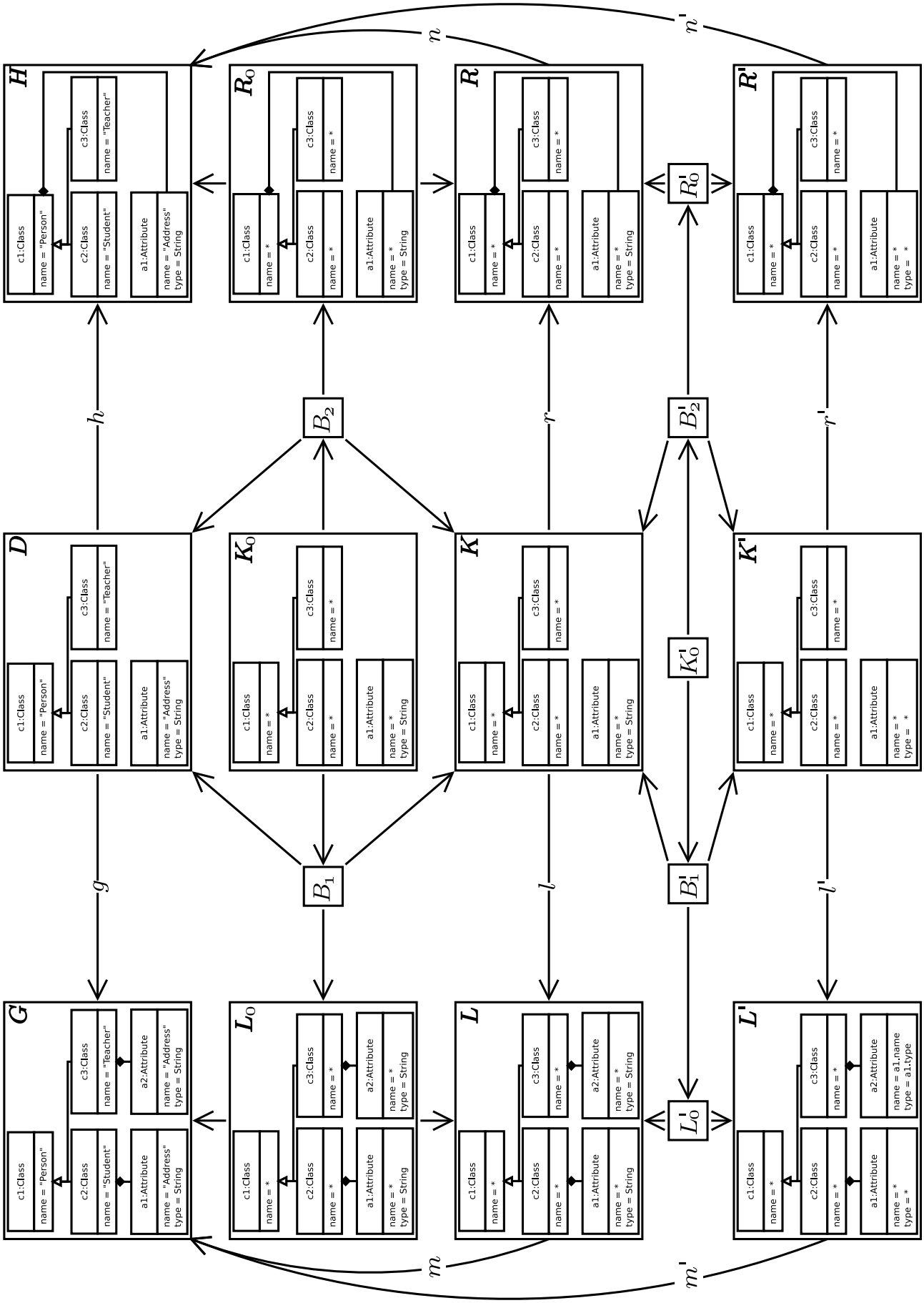

Figure 4: Derivation and refinement of a graph transformation for the Pull-Up Field refactoring

to H. They employ this technique to efficiently verify architectural refactorings, i.e., given two versions G and H of a graph, to decide if the graph H is still semantically correct. Instead of verifying the entire graph, which might turn out to be expensive, only the set of changes is verified to ascertain the validity of the graph H. With the minimal rule extraction technique the minimal set of modifications required to transform graph G to graph H is extracted automatically into a graph production. The construction used to extract the set of graphs equivalent to L_0, B_1, B_2, and R_0 in Fig. 2 is called an *initial pushout* (see [EEPT06]). Note that this technique may only capture the actual modifications, but does not guarantee that the extracted production captures all necessary pre- and postconditions.

Although their extraction technique appears similar to our derivation and refinement construction, a subtle, yet important difference may be observed. While our approach can assume that the initial and the revised graph contain all but no more than the necessary elements (as those are provided by the user), the minimal rule extraction technique does not have this information. Hence, in case of the minimal extracted rule, it might become necessary to add additional elements to the automatically extracted rule. In case of our rule derivation and refinement technique no additional elements are allowed to be added due to the restriction of the refinement construction to ensure its soundness.

Model transformation By-Demonstration. Since graphs and models are referred to visual languages, graph rewriting systems are classified as visual programming languages [BMTS99]. The history of visual programming languages is as old as computers with graphics displays. Early representatives provided basic icon rewriting facilities by matching identical occurrences of a LHS pattern and replacing it with a copy of the RHS pattern. This straightforward procedure advanced the technique of programming by-demonstration, as for example done in [SCS94].

More recently, the language independent by-demonstration approaches Emo [BLS+09] and MT-Scribe [SWG08] have been presented independently. In contrast to Emo, the transformations of MT-Scribe are not derived by a state-based comparison of initial and revised model, but by recording the changes. We conjecture that our work can also be used for characterizing MT-Scribe providing a canonical form for directly comparing the concepts of Emo and MT-Scribe. Closely related to MTBD are so-called model transformation by-example (MTBE) approaches as presented in [BV09, WSKK07]. In contrast to MTBD where the transformation is demonstrated once, MTBE derives the transformation iteratively from a set of examples. Based on inductive logic, Balogh and Varró [BV09] introduced a technique to derive graph transformation rules from a set of user-defined mappings between source and target models, whereas Wimmer et al. [WSKK07] generate ATL rules.

Specification of graph transformations. Several model transformation tools use graph transformations as the underlying theoretical framework. In the following, we shortly discuss a non-exhaustive selection of recent approaches. AGG [Tae04] implements the algebraic graph transformation approach in a visual environment, i.e., the transformations are directly performed on the abstract syntax graph of a model. Baar and Whittle [BW07] use the concrete graphical syntax of modeling languages for QVT. In order to directly use a modeling language as pattern language, certain changes to the metamodel are necessary. However, these changes render existing graphical editors useless. VIATRA2 [VB07] offers a model transformation framework supporting the entire life cycle of model transformations. In order to control the execution of the transformations, VIATRA2 combines graph transformation with abstract state machines. Further, the graph patterns can be represented in the concrete syntax of the source and target language. Henshin [ABJ+10] provides a single-panel editor offering an integrated view of the LHS and the RHS rule. Also the Fujaba [GZ02] environment represents both sides of a rule in one model and uses the concept of storyboards to schedule the execution of rules. The Visual Modeling and Transformation System (VMTS) [LLMC05] offers not only a framework for designing visual languages, but also a model transformation tool for the metamodels of the thereby specified languages following the DPO approach. For details, we kindly refer to [TEG+05]. Whereas several approaches exist to visually specify graph transformations not only in the abstract, but also in the concrete syntax of the used modeling language, no direct MTBD approach has been realized so far which directly relies on graph transformations.

Conclusion

The requirements on modern model transformation approaches are manifold, including user-friendliness for efficient specification and application of such transformations as well as formal conciseness for verification purposes. In this paper, we showed how the by-demonstration approach Emo and graph transformations complement one another in order to fulfill both of these requirements.

In fact, we exploited the theoretical framework of graph transformations to conceptually align the operations created with the by-demonstration environment Emo to algebraic graph productions. Yet, our approach supports the genericity of modern graph rewriting systems which are not bound to any particular modeling language and facilitates the reuse of existing modeling editors without any extensions.

We provide a prototypical implementation of our approach, named emo2ggx[8], which transforms Emo operations to the ggx input format of the Agg tool suite. In a first step, this allows us to test Emo operations within Agg and analyze, for example, the confluence properties of multiple operations with Agg's critical pair analyzer.

[8] http://www.modelevolution.org/prototypes/emo2ggx

In future work, we plan to integrate more advanced concepts of Emo like iterations and negative application conditions in the presented formal framework. Furthermore, this work yields the bases for further improvements of by-demonstration approaches with respect to configurability and verification of the derived transformations. Techniques like critical pair analysis can be used to analyze if the application of two Emo operations interfere and potential problems can be detected before they actually occur.

Acknowledgments This work was partially funded by the Vienna Science and Technology Fund through project ICT10-018 and by the fFORTE WIT Program of the Vienna University of Technology and the Austrian Federal Ministry of Science and Research.

References

[ABJ+10] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *Proc. of MODELS 2010*. 2010.

[BET11] E. Biermann, C. Ermel, and G. Taentzer. Formal foundation of consistent EMF model transformations by algebraic graph transformation. *Software and Systems Modeling*, pages 1–24, 2011.

[BHE08] D. Bisztray, R. Heckel, and H. Ehrig. Verification of Architectural Refactorings: Rule Extraction and Tool Support. *ECEASST*, 16, 2008.

[BLS+09] P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger. An Example Is Worth a Thousand Words: Composite Operation Modeling By-Example. In *Proc. of MODELS 2009*, 2009.

[BMTS99] R. Bardohl, M. Minas, G. Taentzer, and A. Schürr. Application of Graph Transformation to Visual Languages. In *Handb. of Graph Gram. and Comp. by Graph Trans.* 1999.

[BV09] Z. Balogh and D. Varró. Model transformation by example using inductive logic programming. *SoSym*, 8(3):347–364, 2009.

[BW07] T. Baar and J. Whittle. On the Usage of Concrete Syntax in Model Transformation Rules. In *Perspectives of Systems Informatics*, volume 4378 of *LNCS*, pages 84–97. Springer, 2007.

[CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[dLBE+07] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Attributed graph transformation with node type inheritance. *Theoret. Comp. Science*, 376(3):139–163, 2007.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fund. of Alg. Graph Trans.* Springer, 2006.

[GZ02] L. Geiger and A. Zündorf. Graph Based Debugging with Fujaba. *ENTCS*, 72(2):112, 2002.

[Hec06] R. Heckel. Graph Transformation in a Nutshell. *ENTCS*, 148(1):187–198, 2006.

[LLMC05] T. Levendovszky, L. Lengyel, G. Mezei, and H. Charaf. A Systematic Approach to Metamodeling Env. and Model Transformation Systems in VMTS. *ENTCS*, 127(1):65–75, 2005.

[MTR05] T. Mens, G. Taentzer, and O. Runge. Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. *ENTCS*, 127(3):113–128, 2005.

[MVG06] T. Mens and P. Van Gorp. A taxonomy of model transformation. *ENTCS*, 152:125–142, 2006.

[Roz97] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I* . World Scientific Pub., 1997.

[SCS94] D. C. Smith, A. Cypher, and J. Spohrer. KidSim: Programming Agents without a Programming Language. *Com. ACM*, 37(7):54–67, 1994.

[SWG08] Y. Sun, J. White, and J. Gray. Model Transformation by Demonstration. In *Proc. MoDELS 2008*. 2008.

[Tae04] G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *Proc. of AGTIVE*, 2004.

[TEG+05] G. Taentzer, K. Ehrig, E. Guerra, J. De Lara, L. Lengyel, T. Levendovszky, U. Prange, D. Varro, and S. Varro-Gyapay. Model transformation by graph transformation: A comparative study. In *Proc. Workshop Model Transformation in Practice at MODELS'05*, 2005.

[TR05] G. Taentzer and A. Rensink. Ensuring Structural Constraints in Graph-Based Models with Type Inh. In *Fund. App. Soft. Eng.* 2005.

[VB07] D. Varró and A. Balogh. The model transformation language of the VIATRA2 framework. *Sci. Comput. Program.*, 68(3):214–234, 2007.

[WSKK07] M. Wimmer, M. Strommer, H. Kargl, and G. Kramler. Towards Model Transformation Generation By-Example. *Hawaii International Conference on System Sciences*, 2007.