Automatic Inference of Rule-based Specifications of Complex In-place Model Transformations

Timo Kehrer[1], Abdullah Alshanqiti[2], Reiko Heckel[2]

[1] Humboldt-Universität zu Berlin, Germany,
timo.kehrer@informatik.hu-berlin.de
[2] University of Leicester, UK
amma2@leicester.ac.uk, reiko@mcs.le.ac.uk

Abstract. Optimal support for continuous evolution in model-based software development requires tool environments to be customisable to domain-specific modelling languages. An important aspect is the set of change operations available to modify models. In-place model transformations are well-suited for that purpose. However, the specification of transformation rules requires a deep understanding of the language metamodel, limiting it to expert tool developers and language designers. This is at odds with the aim of domain-specific visual modelling environments, which should be customisable by domain experts.
We follow a model transformation by-example approach to mitigate that problem: Users generate transformation rules by creating examples of transformations using standard visual editors as macro recorders. Our ambition is to stick entirely to the concrete visual notation domain experts are familiar with, using rule inference to generalise a set of transformation examples. In contrast to previous approaches to the same problem, our approach supports the inference of complex rule features such as negative application conditions, multi-object patterns and global invariants. We illustrate the functioning of our approach by the inference of a complex and widely used refactoring operation on UML class diagrams.

Keywords: Model-driven development; Model evolution; In-place model transformation; Transformation rules; Model transformation by-example; Graph transformation

1 Introduction

Model-driven engineering (MDE) [10] raises the level of abstraction in software engineering by using models as primary artefacts. In particular, domain-specific modelling languages (DSMLs) can ease the transition between informally sketched requirements or designs and implementations by supporting high-level yet formal representations as a starting point for automation. With models becoming an integral part of the software throughout its lifecycle, the effectiveness of tools managing model evolution is particularly important and has to be customised to the DSML and project- or domain-specific settings. This includes the set of change operations available to modify models of a particular DSML.

In-place model transformations have shown to be well-suited to define model refactorings, see e.g. [24], and other kinds of complex change operations such as recurring and highly schematic editing patterns [28]. That means, language- or project-specific change operations can be specified by model transformation rules which can be used to adapt a variety of MDE tools supporting model evolution; advanced model editors [28], modern refactoring tools, high-level differencing [20] and merging tools [21], or evolution analysis tools [16] being examples of this.

However, generic model transformation techniques and tools supporting in-place transformations are commonly based on the abstract syntax of modelling languages, Henshin [5] and VIATRA2 [7] being examples of this. Thus, the specification of transformation rules requires a deep understanding of the language meta-model and its relation to the visual representation, which makes dedicated model transformation techniques only accessible to expert tool developers and language designers [2,19]. This is at odds with the aim of domain-specific visual modelling environments, which should be designed and customised with the help of domain experts. We believe that the customisation of generic model management tools by complex change operations for creating, manipulating and refactoring domain-specific models can multiply the benefits of visual DSMLs. Our aim is to enable such a customisation by the users of the tool, i.e., strictly at the level of the visual notation without requiring an understanding of the meta-model and without the need for learning a model transformation language.

We follow the general methodology known as *model transformation by-example* (MTBE) [19], where users can describe model transformations using a standard visual editor as a macro recorder. The examples, recorded in their concrete syntax, are internally mapped to their abstract syntax which in turn is generalised into a model transformation rule. In previous MTBE approaches targeting in-place transformations [11,27], this generalisation was not fully automated but required manual post-processing at the level of the abstract syntax, mainly because the initial version of a transformation rule was derived from only a single example. This causes several problems which will be analysed in more detail in Sec. 2. Our ambition is to stick entirely to the concrete visual notation domain experts are familiar with, using inference techniques to automatically generalise a set of examples. Provided enough examples are specified, there is no need to resort to manual adaptations of inferred transformation rules.

In this paper, we focus on the technical problem of inferring general in-place transformation rules from specific examples. Our approach builds upon previous work [3] on the extraction of graph transformation rules describing Java operations from program execution logs, specifically the generalisation from instance-level towards general rules, but extends this solution in a number of ways: (A) The notion of a simple type graph is extended to a type graph with inheritance, along with an adaptation of the graph matching algorithms used for inference to the notion of graph morphism with inheritance. This enables further generalisation of rules, especially for meta-models with deep inheritance hierarchies as, e.g., in the case of the UML. (B) We add support for inferring multi-object patterns to support universally quantified operations over complex structures.

(C) The inference of model transformation rules also requires the creation of negative application conditions as part of the preconditions of these rules, so we support inference from negative examples. (D) We add a further generalisation step for identifying and eventually abstracting from global invariants, i.e., context universally present in all rules. (E) From a tooling point of view, graph transformation rules in our approach describe model transformations rather than object dynamics, so our solution is integrated with the Eclipse Modeling Framework and the model transformation tool Henshin [5].

The paper is structured as follows: Sec. 2 motivates our approach introducing a complex and widely used refactoring operation on UML class diagrams as a running example. Our approach to inferring complex in-place transformation rules is presented in Sec. 3, its integration with an MDE environment is briefly outlined in Sec. 4. We demonstrate the functioning of our approach by showcasing the inference of a transformation rule for our running example in Sec. 5. Related work is discussed in Sec. 6 and Sec. 7 concludes the paper.

2 Problem Analysis and Motivating Example

In this section, we analyse the main reasons which demand manual post-processing if transformation rules are derived from a single example. These problems apply to all in-place MTBE approaches proposed in the literature (see Sec. 6).

Consider the well-known object-oriented refactoring operation pullUpAttribute as a running example. It replaces all common attributes, i.e. attributes having the same name and type, in a set of subclasses by a single attribute of this name and type in the superclass. This refactoring operation has been adopted for UML class diagrams and formally specified using in-place model transformation techniques [5]. It may be demonstrated by providing an original and a changed UML class diagram like the ones shown on top of Fig. 2. The corresponding abstract syntax representations of these models, basically attributed graphs typed over the UML meta-model [26], are shown at the bottom of Fig. 2. The relevant yet slightly simplified excerpt of the UML meta-model is shown in Fig. 1. Provided there is a proper strategy for identifying the corresponding elements in the model versions before and after the sample transformation, the graphs shown in Fig. 2 can be considered as pre- and postconditions which may be conveyed in a model transformation rule following graph transformation concepts. Pre- and post graphs represent the left-hand side (LHS) and the right-hand side (RHS) of this rule, respectively. In Fig. 2, correspondences between LHS and RHS nodes

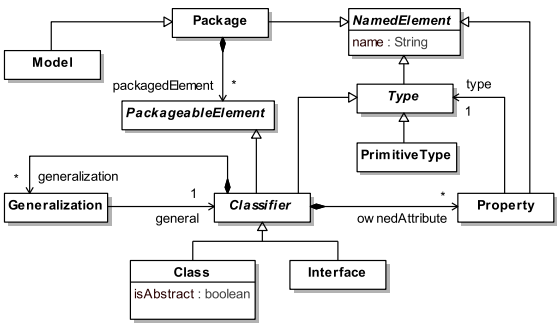

Fig. 1. Excerpt of the UML meta-model.

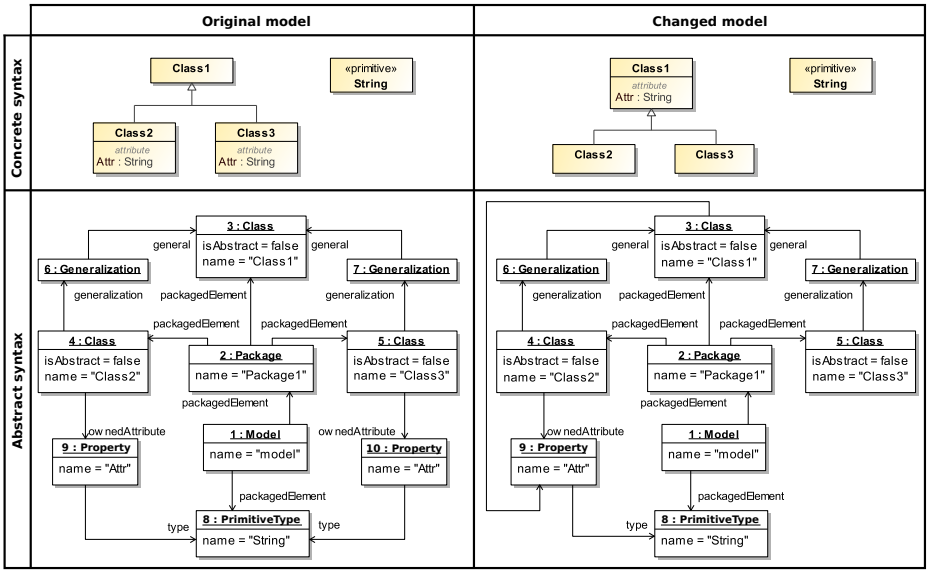

Fig. 2. Transformation pullUpAttribute demonstrated by a single example: Original (left) and changed model (right) demonstrating the transformation; concrete syntax (top) and abstract syntax (bottom).

are indicated by node identifiers. Corresponding edges are given implicitly if they have the same type and their source and target nodes are corresponding.

In general, transformation rules derived from a single example are neither *correct*, i.e. they may allow transformation behaviour which leads to a wrong result, nor *complete*, i.e. there may be situations in which the derived rule achieves only a partial effect or is not applicable at all. In particular, preconditions derived from a single example are usually too restrictive, while the postconditions are typically too weak. In the remainder of this section, we discuss the main issues in more detail, illustrating them by means of our running example of Fig. 2. Issues (1) , (2), (3) and (4) refer to completeness, while issue (5) pertains correctness.

(1) Overspecified structural context. Transformation rules derived from a single example are typically affected by overspecified structural context. In our example, the UML classes *"Class1"*, *"Class2"* and *"Class3"* as well as the definition of the UML primitive type called *"String"* do not necessarily have to be contained in the same package. Moreover, a derived rule would contain context which does not prevent a more general usage of the rule but which can be considered as unnecessary context which might affect the performance of a model transformation. In the UML, for example, a node of type *Model* is always the top-most container of a model. This node will show up as unnecessary context in every transformation rule obtained from an example.

(2) Literal interpretation of attribute values. Attribute values derived from a single example are interpreted in a strictly literal way. For instance, a transfor-

mation rule derived from our example would be only applicable to classes named *"Class1"*, *"Class2"* and *"Class3"*, and each of these classes would be required to be a non-abstract class.

(3) Overspecialised typing of nodes. A transformation rule derived from our example of Fig. 2 may be applied to pull up attributes having a UML primitive type. However, it should be also capable of handling attributes having a complex type such as UML class or interface instead of a primitive type. The only relevant condition is that all attributes share the same type.

(4) Incomplete specification of transformation. A single example is often not capable of capturing the entire behaviour of the intended operation. In particular, if the operation involves multi-object patterns, it can not be demonstrated by a single example. For instance, we may pull up an arbitrary number of attributes shared among a set of subclasses. This desired transformation behaviour should be specified by the transformation rule.

(5) Missing negative application conditions. In the same vein as preconditions derived from a single example often require too much context, they lack conditions on context that must not occur. This may lead to incorrect transformation behaviour. Negative application conditions are not captured at all if we derive a transformation rule from positive transformation example(s) only. For instance, the common superclass which serves as the new container for the attributes to be pulled up must not already have an attribute with the same name.

3 Inference of Transformation Rules

We work with model transformation rules based on graph transformation concepts following the DPO approach [13], supporting typed attributed directed multi graphs with node-type inheritance. An attributed type graph TG, visually represented as a class diagram, defines the DSML meta-model. An *object graph* over TG is a graph G equipped with a homomorphism (a structure-preserving mapping) $G \to TG$ that assigns every element in G its type in TG. In accordance with the notion of E-graph [13], data occurring in graphs are represented by value nodes linked to objects by attribute edges. Apart from equations and assignments over attributes, variables and constants we do not infer complex data conditions or operations. Therefore, we do not require access to the full algebraic structure (operations and axioms) of the data types used for attribution.

Our aim is to derive rules of the form $r : L \Rightarrow R$ (formally spans of graph inclusions $L \leftarrow K \rightarrow R$) with graphs L and R, called the *left-* and *right-hand side* of the rule, expressing its pre- and postconditions and thus its effect in a declarative way: $L \backslash R$, $L \cap R$ $(= K)$ and $R \backslash L$ represent the elements to be deleted, preserved and created by the rule. In addition, a basic rule r may be equipped with a set NAC_r of negative application conditions of the form $NAC(x)$, where $x : L \rightarrow X$ is a type-preserving monomorphism. The left-hand side of a rule

r can have several matches ("occurrences") in a graph G, a match m being a type-compatible monomorphism $m : L \to G$ (types of rule graph elements may be more general than those of their corresponding model graph elements). Given a graph G, a rule is applicable at a match m if all negative application conditions are satisfied, i.e. for all $NAC(x_i) \in NAC_r$ with $x_i : L \to X_i$, there does not exist a type-compatible monomorphism $p : X_i \to G$ with $p \circ x_i = m$. This means that there is no NAC graph to which the occurrence $m(L)$ of the left-hand side can be extended. Finally, we use the concepts of rule schemes and amalgamation as a concise way to specify transformations of recurring model patterns. A rule scheme $RS = (r_k, M)$ consists of a kernel rule r_k and a set $M = \{r_i \mid 1 \leq i \leq n\}$ of multi-rules with $r_k \subseteq r_i$ for all $1 \leq i \leq n$. The idea is that each multi-rule specifies one multi-object pattern and its transformation. A rule scheme is applied as follows: The kernel rule is applied once, just like a "normal" rule. The match of this application is used as a common partial match for all multi-rules, which are matched as often as possible. We will refer to rule schemes as *rules with multi-object patterns*. Such a rule distinguishes a set $MOP = \{P_i \mid 1 \leq i \leq n\}$ with $P_i = r_i \setminus r_k$ ($L_i \setminus L_k$, $K_i \setminus K_k$ and $R_i \setminus R_k$) representing the specification of a multi-object pattern transformation. Note that, in general, multi-object patterns are just graph fragments, i.e. partial graphs and not graphs.

Fig. 3 describes our inference process starting from a set of instance-level transformations, i.e. positive and negative examples demonstrating a dedicated transformation. We assume here that all examples refer to the same change operation to be specified by an in-place transformation. Positive examples are pairs of pre and post graphs of transformations, so-called *rule instances*. Negative examples are individual graphs, referred to as *NAC graph instances*, to which the rules to be derived shall not be applicable. From sets of rule instances and NAC graph instances, general rules with negative application conditions and multi-object patterns can be inferred as follows. First, we combine the given instances into higher-level rules by (1) classifying them by effect and (2) abstracting from non-essential context. Then, we (3) derive negative application conditions using NAC graph instances. Next, we (4) further generalise by identifying complex object structures being treated in a uniform way by a set of generalised rules, but with different cardinalities. The result is a set of generalised rules with NACs and multi-object structures. Taking this set of generalised rules as input, we (5) further raise the level of abstraction by extracting context universally present as global invariant (not shown in Figure 3). Each of the above mentioned steps will be discussed in the remainder of this section.

Classification by effect and derivation of shared context. For each rule instance, we generate a *minimal rule* that is able to perform the rule instance's effect in a minimal context. It is obtained from an instance by cutting all context not needed to achieve the observed changes. The result is a classification of rule instances by effect: All instances with an isomorphic minimal rule have the same effect, but possibly different preconditions. Minimal rule construction has been formalised in [9] and implemented (without considering node-type inheritance)

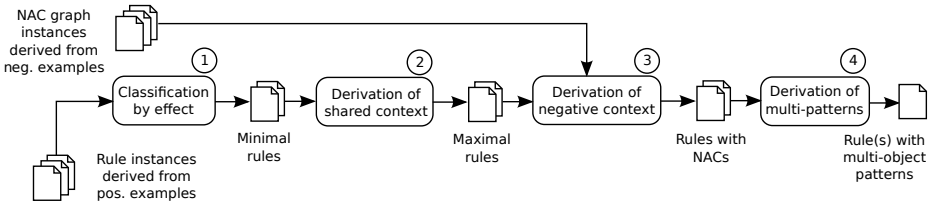

Fig. 3. Overview of the rule inference process.

in [4]. Formally, given a rule instance $i : G \Rightarrow H$, its minimal rule is the smallest rule $L \Rightarrow R$ such that $L \subseteq G$, $R \subseteq H$ with $G \setminus H = L \setminus R$ and $H \setminus G = R \setminus L$.

Here, we extend the minimal rule construction by exploiting node-type inheritance. Two minimal rules may be merged if they only differ in the types of their context nodes and these types share a common super type to which the respective nodes can be generalised. More formally, given two minimal rules $r_1 : L_1 \Rightarrow R_1$ and $r_2 : L_2 \Rightarrow R_2$, they may be merged to a single rule $r : L \Rightarrow R$ if there are type-compatible isomorphisms $L_1 \rightarrow L, L_2 \rightarrow L$ and $R_1 \rightarrow R, R_2 \rightarrow R$ such that $L \setminus R = L_1 \setminus R_1 = L_2 \setminus R_2$ and $R \setminus L = R_1 \setminus L_1 = R_2 \setminus L_2$. That means r, r_1 and r_2 specify the same effect while the types of context nodes in $L \cap R$ may be more general than those in $L_1 \cap R_1$ and $L_2 \cap R_2$, respectively.

Sets of rule instances classified by a minimal rule are generalised by one so called *maximal rule* which extends the minimal rule by context that is present in all instances, essentially the intersection of all its instances' preconditions [3].

For a rule instance obtained from our example shown in Fig. 2, a minimal rule would only contain the nodes of type *Class* and *Property* along with their connecting edges. All other graph elements are context elements not required to achieve the transformation effect. Since there is only one rule instance, they would be part of the maximal rule. Note that if we provide another example in which we replace the primitive type of the UML attribute being pulled up by a UML complex type such as *Class* or *Interface*, we still obtain a single minimal (and maximal) rule, however, with the *Property* node (representing the UML attribute) typed over the more general type *Type*.

Derivation of negative context. Providing negative examples enables the derivation of negative application conditions for maximal rules. The idea is to associate a negative example to a positive one, thereby implying which model elements prevent a transformation rule from being applicable. In other words, we assume a NAC graph instance to be a supergraph of a rule instance's pre graph.

More formally, for each positive example yielding a rule instance $i : G \Rightarrow H$, we may add several sets of negative examples, say NEG_x. Each negative example represents a NAC graph instance $N \supseteq G$, i.e., an extension of the rule instance's left-hand side by forbidden context. A NAC graph X is obtained from each of the sets NEG_x as the intersection of all NAC graphs $N_i \in NEG_x$, analogously to the construction of maximal rules. The obtained condition is $NAC(x)$ with $x : G \rightarrow X$. It is translated into a condition $NAC(x')$ over the maximal rule $r_{max} : L_{max} \Rightarrow R_{max}$ generalising rule instance i, where $x' : L_{max} \rightarrow X'$ is

obtained by restricting x to $L_{max} \subseteq G$. After inferring all negative application conditions for the same maximal rule, duplicate NACs are eliminated.

Please note that, since we treat attributes and their values as attribute edges and value nodes, the NAC inference procedure includes the derivation of negative conditions over attributes. In our example of Fig. 2 we can add a negative example in which the superclass already contains an attribute named *"Attr"*.

Derivation of multi-object patterns. To derive rules with multi-object patterns from generalised rules, we have to discover sets of rule patterns that have the same structure and transformation behaviour, and thus can be represented by a single multi-object pattern. To that end, we first introduce a notion of *rule pattern* and their *equivalence*. Let $r_{max} : L_{max} \Rightarrow R_{max}$ be a maximal rule derived in step (2). Let further $P = (FL, FR)$ be a pair of graph fragments with $FL \subseteq L_{max}$ and $FR \subseteq R_{max}$, and let $BL \subseteq L_{max}$ and $BR \subseteq R_{max}$ be the smallest graphs completing FL and FR to graphs; we refer to BL and BP as boundary graphs of FL and FR, respectively. P is a rule pattern in r_{max} if $BL \Rightarrow BR$ is a subrule of r_{max}. Two rule patterns P_1 and P_2 in r_{max} are equivalent if (i) their boundary graphs are isomorphic, i.e. there are type-preserving isomorphisms $BL_1 \to BL_2$ and $BR_1 \to BR_2$; and (ii) they overlap in the completing nodes of their boundary graphs, i.e. $(BL_1 \setminus FL_1) = (BL_2 \setminus FL_2)$ and $(BR_1 \setminus FR_1) = (BR_2 \setminus FR_2)$.

Assume that we extend our example of Fig. 2 such that we have three subclasses with a common UML attribute being pulled up. Fig. 4 illustrates the maximal rule derived from this example, omitting nodes of types *Package* and *Model*, edge types and attributes. It contains two equivalent patterns, the respective graph fragments are completed to boundary graphs by the same nodes.

We derive rules with multi-object patterns in two steps. First, we merge equivalent rule patterns in maximal rules: For each maximal rule m in a set of maximal rules, and each non-trivial equivalence class of rule patterns in m, one pattern is chosen as the representative for that class and added to the set MOP of multi-object patterns for m, while all other patterns of that class are deleted. The resulting set of rules with multi-object patterns is MOR. Second, we combine isomorphic rules: A maximal set of structurally equivalent rules in

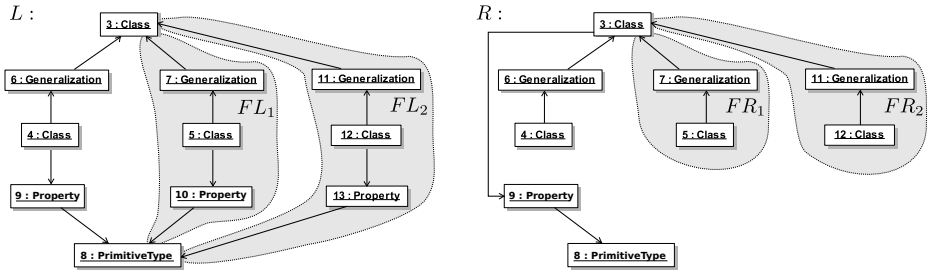

Fig. 4. Maximal rule illustrating the occurrence of two equivalent rule patterns $P_1 = (FL_1, FR_1)$ and $P_2 = (FL_2, FR_2)$.

MOR forms an isomorphism class. For each such class we derive a single rule by selecting a representative one.

Derivation of universal context. To extract and cut context universally present in all generalised rules, we employ a similar proceeding as for the derivation of maximal rules. That is, we compare the preconditions of all maximal rules to identify structures that are universally present. Universal context presented as global invariant can reduce the size of rules, make them more concise and readable. In case of the UML, for instance, we could spot the universal presence of a *Model* node serving as a container for all model elements.

4 Integration with an MDE Development Environment

Our approach is open to be used in the context of any modelling environment or visual editor that can be used for specifying example transformations, and any model transformation engine for executing the inferred transformation rules.

As a proof of concept, we implemented the inference approach presented in Sec. 3 based on the Eclipse Modeling Framework (EMF) and the rule-based model transformation language and tool Henshin, available from the accompanying website for this paper [1]. Since EMF and Henshin employ the usual object-oriented representation with attributes being inlined as properties of objects, attributes are transformed into our conceptual representation based on attribute edges and value nodes. The general idea of exporting attribute edges and value nodes occurring in generalised rules is to derive equations over attributes from attribute edges, e.g., if two attributes a, b point to the same value node, an equation $a = b$ is added. In Henshin, an internal variable, say x, has to be defined and assigned to attributes, i.e., we have $a = x$ and $b = x$ for the attributes a and b in the example above.

To be independent of the visual editor being used to specify example transformations, we follow a state-based approach to transformation recording [19]. To reliably identify the corresponding model elements in the original and the changed model of an example, we assume a model matcher [22] for a given DSML to be readily available.

5 Case-based Evaluation

In this section, we illustrate the applicability of our approach by means of the example of Sec. 2. We show how a suitable rule can be inferred using our approach and tool. For depicting transformations, we use the visual UML editor Papyrus (v. 1.0.2), which is based on an EMF-based implementation of the UML Superstructure Specification (v. 2.5) [26]. We generally ignore meta-model elements defined by the UML Superstructure for which there is no visual representation in the diagram notation supported by Papyrus. Since Papyrus attaches universally unique identifiers (UUIDs) to model elements, we employ the UUID-based

matching facility of the model comparison tool EMF Compare for deriving corresponding elements in the original and the changed model of an example.

Sec. 5.1 outlines the examples defined for demonstrating the refactoring operation pullUpAttribute. We concentrate on the rationale of each of the examples w.r.t. the desired effect on the generalised rule(s), thereby emphasising the issues presented in Sec. 2. We discuss threats to validity in Sec. 5.2.

5.1 Solving the Motivating Example

To infer a general transformation rule for the refactoring pullUpAttribute, we start by providing an initial example ($Example_1$) corresponding to the one presented in Fig. 2. W.r.t. to the state-of-the-art in generating in-place transformation rules by-example (cf. Sec. 6), the rule derived from this example serves as a baseline for a qualitative evaluation of our approach. To infer a general transformation rule, we proceed by adding further examples, each of them addressing one of the issues discussed in Sec. 2.

(1) Overspecified structural context. We add another transformation example ($Example_2$) to avoid overspecified structural context, where we abstain from using a dedicated container of type *Package*, which thus gets eliminated during maximal rule construction. Moreover, the element of type *Model* is identified as universal context over all examples and gets eliminated, too. The resulting Henshin rule is shown in Fig. 5 (left). In the Henshin notation, the LHS and RHS of a rule are merged into a single graph. The LHS comprises all elements stereotyped by delete and preserve, while the RHS contains all elements annotated by preserve and create. Due to space limitations, we present only a subset of the generated Henshin rule comprising those rule elements being typed over the visible excerpt of the UML meta-model shown in Fig. 1. The complete rule can be found on the supplementary website for this paper [1].

(2) Interpretation of attribute values. As a side-effect of providing a second example, rule variables $x1$ to $x6$ are inferred denoting equalities of attribute values. For instance, all classes occurring in the transformation rule inferred from $Example_1$ and $Example_2$ have to be abstract, indicated by variable $x4$ (see Fig. 5 (left)). To handle attributes properly, we add a third transformation example ($Example_3$) in which attributes irrelevant for the transformation are assigned values differing from the ones used in $Example_1$. We rename all classes, turn these classes into concrete classes, and change the primitive type from *"String"* to *"int"*. As indicated by annotation (2) in Fig. 5, we get rid of all variables except of $x5$, which denotes equality of names of the *Property* to be pulled up. This is on purpose, i.e., we learn that all attributes to be pulled up must have the same name.

(3) Overspecialized typing of nodes. The transformation rule pullUpAttribute shall also be applicable to attributes having a complex type instead of a primitive type. Thus, we add another example ($Example_4$) which slightly modifies our first one

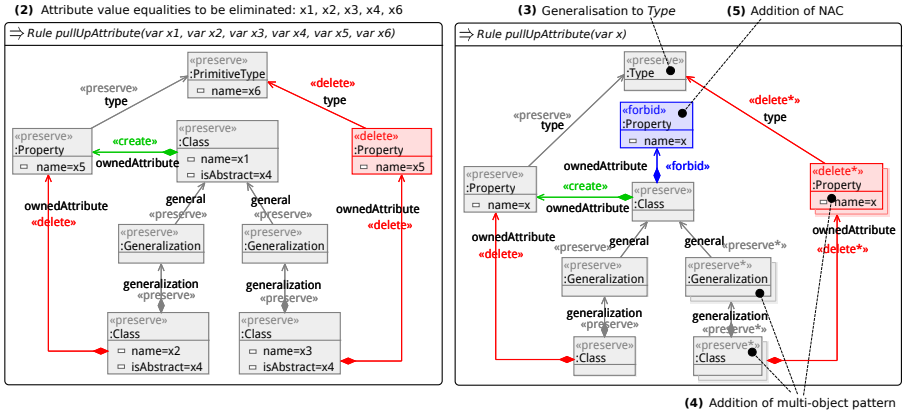

Fig. 5. Inferred Henshin rules after providing additional examples of step 1 (left), and steps 2 to 5 (right).

such that the primitive type called *"String"* of all attributes *"Attr"* is exchanged by a UML class representing a complex type. The minimal rule which results from $Example_1$ to $Example_3$ may be merged with the minimal rule obtained for $Example_4$ since (i) the UML node types *Class* and *PrimitiveType* share the same supertype *Type* (cf. Fig. 1), and (ii) the supertype replacement is permitted in this context. The effect of adding $Example_4$ is illustrated in the inferred rule shown in Fig. 5 (right).

Note that $Example_2$ to $Example_4$ could be combined to a single example achieving the same effect on the generalised transformation rule.

(4) Incomplete specification of transformation. Since, in general, arbitrarily many attributes may be pulled up to a common superclass, we give another example ($Example_5$) which differs from $Example_1$ of Fig. 2 in that we have three subclasses containing an attribute named *"Attr"* typed by the UML primitive type called *"String"*. In the inferred rule, we get the specification of the transformation of a multi-object pattern as shown in Fig. 5 (right).

(5) Missing negative application conditions. Attributes may only be pulled up if the superclass does not already contain an attribute of the same name. To that end, we provide two negative examples ($Negative_1$ and $Negative_2$) referring to $Example_1$. The basic idea is to copy the original model of the positive example $Example_1$ and to add an attribute named *"Attr1"* to the common superclass *"Class1"*. To demonstrate that it is only the name of this additional attribute which prevents the refactoring from being applied successfully while we do not care about its type, we provide two negative examples in which the additional name attribute has different types. As a result, the inferred rule is equipped with a negative application condition as shown in Fig. 5, indicated by the stereotype forbid in the Henshin notation. This rule represents the final result of our example-driven process to specifying and inferring a general rule for refactoring pullUpAttribute. It corresponds to the transformation rule

PUAExecuteRule presented in [5] for the same refactoring operation but a slightly simplified variant of the UML meta-model.

5.2 Threats to Validity

In this section, we evaluated the applicability of the proposed approach by investigating a complex in-place transformation rule whose inference from a set of examples is faced with several challenges. Although our study shows very promising results, we are aware of some threats to validity giving rise to a critical discussion.

As every case-based evaluation, our experiment suffers from a lack of comprehensiveness, which affects the external validity of our results. We only considered one meta-model and a single transformation rule typed over this meta-model. We mitigate this threat by choosing a comprehensive meta-model exposing several pitfalls for the specification of transformation, and a non-trivial transformation rule which can be considered as representative for a considerable amount of refactoring operations. However, there may be change operations which can only be expressed using more sophisticated model transformation features than those supported by our approach. The most severe limitation pertains the handling of complex constraints on attributes. Ignoring the algebraic structure of data types used for graph attribution limits our approach to the inference of simple equations and assignments over attributes, variables and constants.

Another threat to validity is that we defined the transformation examples ourselves. We are familiar with the UML meta-model, the inference process for learning transformation rules, and the fact that we exploit persistent identifiers attached by Papyrus to identify corresponding model elements. These internal details are typically not known by domain experts who would use our approach. Thus, they might produce examples leading to transformation rules which are both incorrect and incomplete, examples that are inconsistent or contradict each other, or they might simply need significantly more examples to eventually infer proper transformation rules. Such issues and requirements for future research must be analysed in an empirical user study, which we leave for future work.

6 Related Work

The main objective of our work is to provide a technique supporting domain experts in the specification of complex change operations for creating, manipulating and refactoring domain-specific models. In the model transformation community, there are two main lines of research which basically pursue the same goal, *model-transformation by-example* and *model transformation based on concrete syntax*, which we will review in that order as our approach falls into the former category. Finally, we briefly review related *approaches from other domains*.

Model transformation by-example. Since specifying model transformations from scratch using the abstract syntax of a DSML is a difficult and error-prone task,

learning them from existing transformation examples is highly desirable and has motivated a plethora of work in this field, see the pre-2012 approaches surveyed in [19] and, more recently, in [6].

The majority of existing approaches, e.g. [6,8,15,23], target model-to-model transformations, often referred to as exogenous transformations [25] since source and target models typically correspond to different meta-models. Such transformations are well-suited to capture vertical translations, which build the basis for model refinement and code generation, and horizontal translations, which are of specific importance in model-based integration and migration scenarios. Model-to-model transformation scenarios are inherently different from the model evolution scenarios addressed by our approach. Their main problem is to adequately infer semantically equivalent modelling concepts in the source and target domain of a transformation, e.g., in UML class models and Entity Relationship models. This is not an issue at all in case of in-place transformations which are inherently endogenous [25].

To the best of our knowledge, which is in accordance with the surveys presented in [6,19], only two dedicated approaches addressing the specification of in-place transformations by-example have been proposed in the literature [11,27]. They have in common with our approach the idea of using standard visual editors to demonstrate model transformations. However, transformation rules are derived from a single example, which merely corresponds to the first step of our inference process. They do not support to generalise a set of examples including advanced features such as the inference of multi-object patterns, negative application conditions, abstraction from universal context and super type generalisation as offered by our approach.

Model transformation based on concrete syntax. An alternative approach to improve the usability of transformation languages is to enable modellers to specify model transformations based on the concrete syntax of a DSML.

One class of approaches is based on the idea of turning a DSML into a domain-specific *transformation* language. Hölldobler et al. [18] assume a DSML to be specified using a context-free grammar and present a generative approach to systematically derive a textual domain-specific transformation language from that grammar. Consequently, the approach is limited to textual DSMLs.

In contrast, Grønmo [17] and Acreţoaie et al. [2] address concrete syntax-based transformation of graphical models. Similar to [18], the approach presented in [17] is generative in the sense that transformation rule editors are generated from the DSML meta-model which must be equipped with a mapping to the set of symbols of the DSML concrete syntax. However, this means domain experts cannot define the transformation rules using their favourite editor. This problem is avoided in [2] which follows a generic approach in which transformation rules are specified using a readily available host language model editor. The only assumption is that this editor provides a mechanism to annotate models with certain transformation actions. The approach is similar to ours in the sense that it aims at achieving transparency w.r.t. the DSML, the modelling environment and the underlying transformation language at the same time. However, domain

experts still have to learn a dedicated transformation language called VMTL to be used for annotating transformation actions.

Approaches from other domains. In other areas, rule inference approaches have been suggested to address problems of mining business processes [12] and learning (bio-)chemical reaction rules [14,29]. Although related in the aim of discovering rules, the challenges vary based on the nature of the graphs considered, e.g., directed, attributed or undirected graphs, the availability of typing or identity information, as well as in the scalability requirements in terms of the size and number of the examples to be considered.

7 Conclusion

In this paper, we presented a novel approach to model transformation by-example which enables the inference of transformation rules which are formally considered as graph transformation rules and obtained by generalising over a set of rule instances. Our approach supports the inference of complex rule features such as negative application conditions, multi-object patterns and global invariants. The approach is supported by a tool integrated with the Eclipse Modeling Framework and the model transformation tool and language Henshin. Our case-based evaluation illustrates the applicability of our approach and shows that, in principle, the inference of complex transformation rules is possible by providing only a few example transformations. In this paper we focused on the technical approach to rule inference. We leave a wider empirical evaluation, e.g., of the manual effort to provide examples for a larger set of transformations for future work.

Acknowledgments

The work of the first author was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future – Managed Software Evolution.

References

1. Accompanying material for this paper (2017), `http://icmt.mtrproject.uk`
2. Acreţoaie, V., Störrle, H., Strüber, D.: VMTL: a language for end-user model transformation. Software & Systems Modeling pp. 1–29 (2016)
3. Alshanqiti, A., Heckel, R.: Extracting visual contracts from Java programs. In: ASE (2015)
4. Alshanqiti, A., Heckel, R., Khan, T.: Learning minimal and maximal rules from observations of graph transformations. ECEASST 58 (2013)
5. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place emf model transformations. In: MoDELS (2010)
6. Baki, I., Sahraoui, H.: Multi-step learning and adaptive search for learning complex model transformations from examples. TOSEM 25(3) (2016)

7. Balogh, A., Varró, D.: Advanced model transformation language constructs in the VIATRA2 framework. In: ACM Symposium on Applied Computing (2006)
8. Balogh, Z., Varró, D.: Model transformation by example using inductive logic programming. SoSym 8(3), 347–364 (2009)
9. Bisztray, D., Heckel, R., Ehrig, H.: Verification of architectural refactorings: Rule extraction and tool support. ECEASST 16 (2009)
10. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Morgan & Claypool Publishers (2012)
11. Brosch, P., Langer, P., Seidl, M., Wieland, K., Wimmer, M., Kappel, G., Retschitzegger, W., Schwinger, W.: An example is worth a thousand words: Composite operation modeling by-example. In: MoDELS (2009)
12. Bruggink, H.: Towards process mining with graph transformation systems. In: Graph Transformation, LNCS, vol. 8571 (2014)
13. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer (2006)
14. Flamm, C., Merkle, D., Stadler, P.F., Thorsen, U.: Automatic inference of graph transformation rules using the cyclic nature of chemical reactions. In: ICGT (2016)
15. García-Magariño, I., Gómez-Sanz, J.J., Fuentes-Fernández, R.: Model transformation by-example: An algorithm for generating many-to-many transformation rules in several model transformation languages. In: ICMT (2009)
16. Getir, S., Rindt, M., Kehrer, T.: A generic framework for analyzing model coevolution. In: ME@MoDELS (2014)
17. Grønmo, R.: Using concrete syntax in graph-based model transformations. Ph.D. thesis, University of Oslo (2009)
18. Hölldobler, K., Rumpe, B., Weisemoller, I.: Systematically deriving domain-specific transformation languages. In: MoDELS (2015)
19. Kappel, G., Langer, P., Retschitzegger, W., Schwinger, W., Wimmer, M.: Model transformation by-example: a survey of the first wave. In: Conceptual Modelling and its Theoretical Foundations. Springer (2012)
20. Kehrer, T., Kelter, U., Ohrndorf, M., Sollbach, T.: Understanding model evolution through semantically lifting model differences with silift. In: ICSM (2012)
21. Kehrer, T., Kelter, U., Reuling, D.: Workspace updates of visual models. In: ASE (2014)
22. Kolovos, D.S., Di Ruscio, D., Pierantonio, A., Paige, R.F.: Different models for model matching: An analysis of approaches to support model differencing. In: CVSM@ICSE (2009)
23. Kühne, T., Hamann, H., Arifulina, S., Engels, G.: Patterns for constructing mutation operators: Limiting the search space in a software engineering application. In: European Conf. on Genetic Programming (2016)
24. Mens, T.: On the use of graph transformations for model refactoring. In: Generative and transformational techniques in software engineering. Springer (2006)
25. Mens, T., Van Gorp, P.: A taxonomy of model transformation. ENTCS 152 (2006)
26. Object Management Group: Uml 2.5 superstructure specification. OMG Document Number: formal/15-03-01 (2015)
27. Sun, Y., White, J., Gray, J.: Model transformation by demonstration. In: MoDELS (2009)
28. Taentzer, G., Crema, A., Schmutzler, R., Ermel, C.: Generating domain-specific model editors with complex editing commands. In: AGTIVE (2008)
29. You, C.h., Holder, L.B., Cook, D.J.: Learning patterns in the dynamics of biological networks. In: Intl. Conf. on Knowledge Discovery and Data Mining (2009)