Model Transformation By-Example: A Survey of the First Wave*

Gerti Kappel[1], Philip Langer[2], Werner Retschitzegger[2], Wieland Schwinger[2], and Manuel Wimmer[1]

[1] Vienna University of Technology, Austria
{gerti,wimmer}@big.tuwien.ac.at
[2] Johannes Kepler University Linz, Austria
{langer,retschitzegger,schwinger}@jku.at

Abstract. Model-Driven Engineering (MDE) places models as first-class artifacts throughout the software lifecycle. In this context, model transformations are crucial for the success of MDE, being comparable in role and importance to compilers for high-level programming languages. Thus, several model transformation approaches have been developed in the last decade, whereby originally most of them are based on the abstract syntax of modeling languages. However, this implementation specific focus makes it difficult for modelers to develop model transformations, because they are familiar with the concrete syntax but not with its computer internal representation.

To tackle this problem, model transformation by-example approaches have been proposed which follow the same fundamental idea as query by-example and programming by-example approaches. Instead of using the computer internal representation of models, examples represented in concrete syntax are used to define transformations. Because different transformation scenarios occur in MDE, different by-example approaches have been developed. This chapter gives an overview on the emerging concepts, techniques, and approaches in this young by-example area.

Keywords: model transformation, by-example, model-driven engineering.

1 Introduction

Model-Driven Engineering (MDE) places models as first-class artifacts throughout the software lifecycle [7,16,38]. In this context, model transformations [39] are crucial for the success of MDE, being comparable in role and importance to compilers for high-level programming languages, for bridging the gap between design and implementation. Thus, several model transformation approaches (cf. [13] for an overview) have been developed in the last decade, whereas most of them are based on the abstract syntax of modeling languages which is defined by

* This work has been partly funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) and FFG under grant FIT-IT-819584.

A. Düsterhöft et al. (Eds.): Thalheim Festschrift, LNCS 7260, pp. 197–215, 2012.

so-called metamodels [26]. Metamodels describe by a limited set of UML class diagram concepts the object structure for computer internally representing and persisting models. However, this implementation specific focus makes it difficult for modelers to develop model transformations, because modelers are mainly familiar with the concrete syntax of the modeling languages (i.e., their notation) and not with their metamodels. This is aggravated by the fact that metamodels may become very large: for instance, the UML 2 metamodel [34] has about 260 metamodel classes [30]. Moreover, some language concepts, which have a particular representation in the concrete syntax, are not even explicitly represented in the metamodel. Instead, these concepts are hidden in the metamodel and may only be derived by using specific combinations of attribute values and links between objects [20]. Thus, they are often hard to discover as illustrated in the example in Fig. 1.

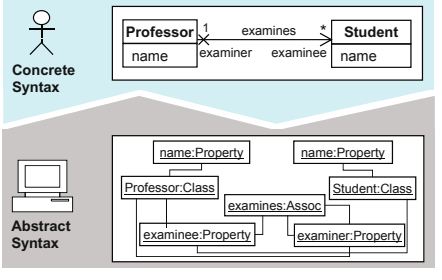


Fig. 1. Gap between user intention and computer internal representation

To tackle this problem, model transformation by-example (MTBE) approaches [45,47] have been proposed which follow the same fundamental idea as *query by-example* developed for querying database systems by giving examples of query results [48] and *programming by-example* for demonstrating actions which are recorded as replayable macros [29]. This means, instead of using the computer internal representation of models, MTBE allows to define transformations using examples represented in concrete syntax. Consequently, the modeler's knowledge about the notation of the modeling language is sufficient.

Because different transformation scenarios occur in MDE [31], different MTBE approaches have been developed in the last years. In general, two kinds of approaches may be distinguished: (i) Approaches following a *demonstration-based* approach, meaning that model transformations are demonstrated in the modeling editor by modifying example models. These modifications are recorded and from the concrete changes, the general transformation is derived which may be replayed on other models as well. (ii) Approaches which follow a *correspondence-based* approach. Instead of demonstrating the transformation in modeling editors, the input model, the output model as well as the correspondences between them have to be given by the user. For both kinds, a multitude of approaches have been proposed during the last years [4,8,14,17,23,27,43,45,47].

2 MDE in a Nutshell

Before MTBE is presented, the prerequisites, i.e., the core techniques of MDE, are explained. First, the essence of modeling language engineering is outlined to illustrate how models are represented in the context of MDE, and subsequently, the main principles and patterns of model transformations are introduced.

2.1 Modeling Language Engineering

Modeling language engineering in MDE comprises at least two components [24]. First, the abstract syntax of a language has to be defined by a metamodel, i.e., a model defining the grammar of the language. Second, to make a language more usable, a mapping of abstract syntax elements to concrete syntax elements (such as rectangles, edges, and labels) has to be provided. In the following, an example-based description of defining a modeling language is given.

Abstract Syntax. Similar as EBNF-based grammars [18] for programming languages, metamodels represent the concepts and their interrelationships of modeling languages. The most widely used formalism to define metamodels in MDE is the Meta Object Facility [33] (MOF), which is a standardized language to define modeling languages based on the core concepts of UML class diagrams (*classes*, *attributes*, and *references*). In the upper part of Fig. 2, an excerpt of the kernel of the UML metamodel is represented in terms of MOF concepts.

The aim of metamodeling lies primarily in defining modeling languages in an object-oriented manner leading also to efficient repository implementations for storing and retrieving models. This means that in a metamodel not necessarily all modeling concepts are represented as first-class citizens. Instead, the concepts are frequently hidden in attributes and in references. We call this phenomenon *concept hiding* (cf. [20] for an in-depth discussion).

By instantiating metamodels, models are created. An instantiation is represented by a UML object diagram comprising of *objects* as instances of *classes*, *values* as instances of *attributes*, and *links* as instances of *references* as e.g., depicted in the middle part of Fig. 2. It has to be noted that in contrast to EBNF-based grammars, metamodels do not define the concrete syntax of the languages. Thus, only generic object graphs as depicted in the middle part of Fig. 2 may be created. The concrete syntax has to be defined in addition to the metamodel which is explained next.

Concrete Syntax. The concrete syntax of modeling languages [2] comprises in most cases graphical elements such as ellipse, label, and rectangle, which may be further combined to more complex forms. The actual notation of modeling concepts is defined by a mapping of abstract syntax elements to concrete syntax elements. The mappings may be expressed in triples of the following form:

$$Triple :=< as\_E, cs\_E, const(as\_E)? > \quad (1)$$

The first part *as_E* stands for an element of the *abstract syntax*, the second *cs_E* for an element of the *concrete syntax*, and the last *const(as_E)* stands

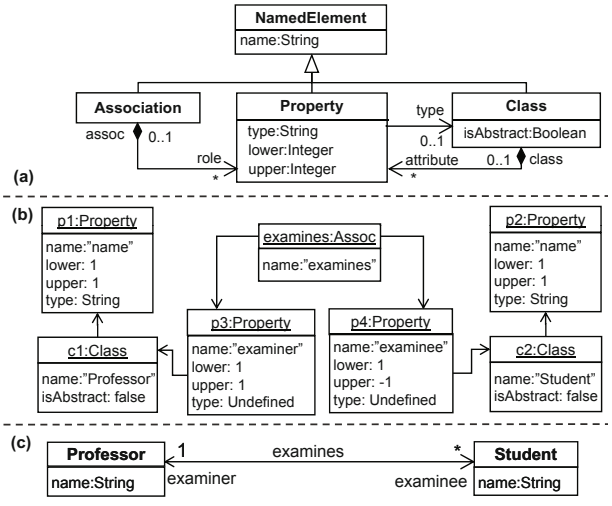


Fig. 2. (Meta)modeling: (a) UML Metamodel Kernel, (b) Example Model in AS, and (c) Example Model in CS

for an optional constraint, mostly defined in the Object Constraint Language (OCL) [36], that defines under which conditions, i.e., links and attribute values of an *as_E* element, this element is represented by a *cs_E* element. In case no constraint is defined, there is a *one-to-one* mapping between an abstract syntax element and a concrete syntax element, i.e., the concept defined in the metamodel is directly represented by one concrete notational element. However, the other case is the more interesting one in the context of MTBE. The presence of a constraint defines a new concept for the notation layer, which is not explicitly represented by one of the metamodel classes. Consequently, when defining model transformations based on the abstract syntax, the constraints for these concepts must be defined by the user. This is a tedious and error-prone task that requires excellent knowledge about the metamodel.

When considering our running example, for instance, the `Class` concept is mapped to `Rectangle` and `Class.name` is mapped to `Label`. By using such mappings, the UML object diagram shown in the middle part of Fig. 2 may be rendered as shown in the lower part of Fig. 2 by graphical modeling editors.

2.2 Transformation Engineering

In general, a model transformation takes a model as input and generates a model as output[1]. Mens et al. [31] distinguish between two kinds of model transformations: (i) *exogenous transformations* a.k.a. model-to-model transformations

[1] Also several input models and output models may be possible, but in the scope of this paper, such settings are not considered.

or out-place transformations, in which the source and target metamodels are *distinct*, e.g., transforming UML class diagrams to relational models, and (ii) *endogenous transformations* a.k.a. in-place transformations, in which the source and target metamodels are the *same*, e.g., a refactoring of a UML class diagram. In the following, we elaborate on these two kinds in more detail.

Exogenous Transformations. Exogenous transformations are used both to exploit the constructive nature of models in terms of *vertical transformations*, thereby changing the level of abstraction and building the bases for code generation, and for *horizontal transformation* of models that are at the same level of abstraction [31]. Horizontal transformations are of specific interest to realize different integration scenarios, e.g., translating a UML class model into an Entity Relationship (ER) model. In vertical and horizontal exogenous transformations, the complete output model has to be built from scratch.

Endogenous Transformations. In contrast to exogenous transformations, endogenous transformation only rewrite the input model to produce the output model. For this, the first step is the identification of model elements to rewrite, and in the second step these elements are updated, added, and deleted. Endogenous transformations are applied for different tasks such as model refactoring, optimization, evolution, and simulation, to name just a few.

Model Transformation Languages. Various model transformation approaches have been proposed in the past decade, mostly based on either a mixture of declarative and imperative concepts, such as ATL [19], ETL [25], and RubyTL [12], or on graph transformations, such as AGG [44] and Fujaba [32], or on relations, such as MTF[2] and TGG [1]. Moreover, the Object Management Group (OMG) has published the model transformation standard QVT [35] which is currently only partly adopted by industry. Summarizing, all approaches describe model transformations by rules using metamodel elements, whereas the rules are executed on the model layer for transforming a source model into a target model. Rules comprise *in-patterns* and *out-patterns*. The in-pattern defines when a rule is actually applicable as well as retrieves the necessary model elements for computing the result of a rule by querying the input model. The out-pattern describes what the effect of a rule is, such as which elements are created, updated, and deleted. All mentioned approaches are based on the abstract syntax of modeling languages only, and the notation of the modeling language is totally neglected.

Defining model transformations by using the abstract syntax of modeling languages comes on the one hand with the benefit of the generic applicability. On the other hand, the creation of such transformations is often complicated and their readability is much lower compared to working with the concrete syntax [3,28,41,45,46]. Therefore, MTBE approaches have been proposed to use the concrete syntax of modeling languages for defining model transformations. In the following two sections, we present the essence of MTBE for endogenous transformations as well as for exogenous transformations.

[2] `http://www.alphaworks.ibm.com/tech/mtf`

3 MTBE for Endogenous Transformations

For endogenous transformations, two dedicated by-example approaches [8,43] have been proposed in the last years that can be seen as a special kind of MTBE called *Model Transformation By Demonstration* (MTBD). MTBD exploits the *edit operations* demonstrated on an example model in order to obtain transformation specifications that are also applicable to other models. Interestingly, no *correspondence-based* approach has been proposed for endogenous transformations, so far. In the following, we present the common process of both MTBD approaches for specifying an endogenous transformation by demonstration and show how this process is applied to a concrete model refactoring example. Finally, we conclude this section by elaborating on the peculiarities of both MTBD approaches.

3.1 Process

In general, the MTBD process consists of two phases: (1) demonstrating the edit operations and (2) the configuration and generation of the general transformation. This process is illustrated in Fig. 3, which is explained in the following step-by-step.

Phase 1: Modeling. In the first step, the user creates a model in the concrete syntax of the modeling language in her familiar modeling environment. This model comprises all model elements which are required to apply the transformation. The output of this step is called the *initial model*. In the second step, the user performs the complete transformation on this initial model by applying all necessary atomic operations, again in the concrete syntax. The output of this step is the *revised model* and a *change model* containing all changes that have been applied to the initial model during the demonstration. This change model together with the initial model and the revised model is the input for the second phase of the transformation specification process.

Phase 2: Configuration & Generation. In the second phase, an initial version of the transformation's *pre-* and *postconditions* is inferred by analyzing the initial model and the revised model, respectively. These automatically generated conditions from the example might not entirely express the intended pre- and postconditions of the transformation. Therefore, they only act as a basis for accelerating the transformation specification process and can be refined by the user in the next step and additional conditions may be added. After the revision of the conditions is finished, the *transformation* is generated from the change model and the revised pre- and postconditions.

3.2 Example

For exemplifying the presented MTBD process, a transformation for a simplified UML Class Diagram refactoring, namely "Extract Class" [15], is used. The aim of the refactoring is to create a new class and move the relevant attributes from an existing class into the new class.

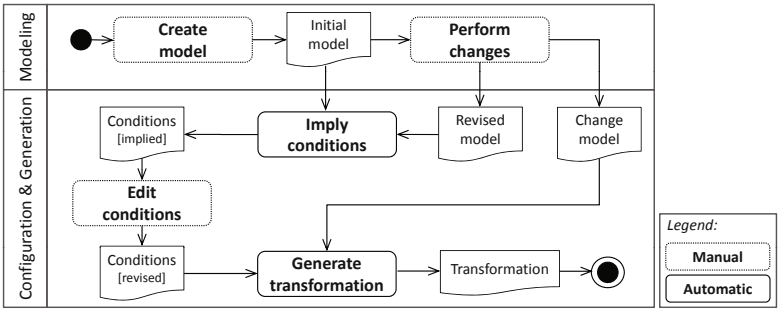


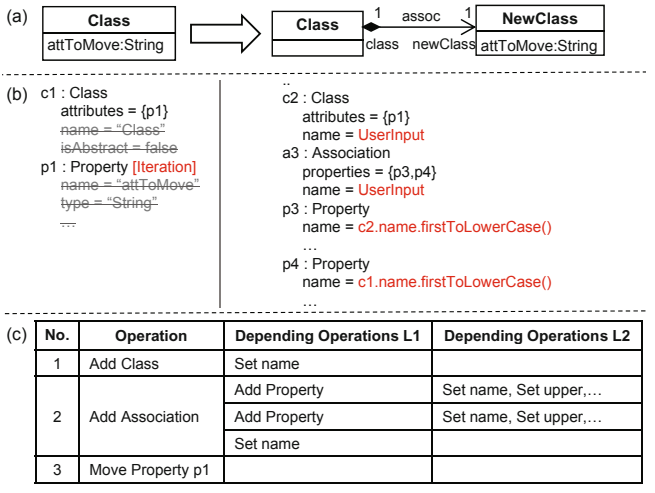
Fig. 3. MTBE Process for Endogenous Transformations

The transformation is demonstrated in Fig. 4(a). The user models the initial situation by introducing one class containing one attribute. Then, the user demonstrates the transformation by introducing another class and an association with two roles and, finally, the user moves the attribute to the newly introduced class. From this demonstration, the change model shown in Fig. 4(c) is obtained. For readability purposes, the changes are structured according to the container hierarchy of the model elements.

In addition to the change model, pre- and postconditions are derived from the initial and revised models, respectively. Subsequently, the user may fine-tune the inferred conditions by activating, deactivating, or modifying conditions as is depicted in Fig. 4(b). On the precondition side of our example, the name of a class and whether it is abstract or not does not matter. Furthermore, the transformation should be agnostic of the name and type of the attribute. Thus, these derived preconditions are deactivated. On the postcondition side, the user may introduce also some annotations for specifying how certain values in the resulting model should be obtained. In this context, a value may either be computed from values in the initial model, or by querying the user for an input value. In our example, the name of the newly introduced class is not derivable from the initial model and therefore has to be specified as *user input* before executing the transformation. The same is true for the association name. However, the role names should be derived from the existing model context. This is specified by additional expressions. In particular, the property name should be equal to the name of the adjacent class but the first letter has to be converted to lower case to fulfill common modeling conventions.

Besides fine-tuning the conditions, current MTBD approaches allow for annotating repetitions of certain edit operations. By this, the transformation may be configured to equally transform multiple model elements that fulfill the transformation's preconditions. In this example, such a mechanism is quite useful, because the transformation may then be capable of moving an arbitrary number of attributes from the original class to the newly introduced class.

An excerpt of the generated transformation is depicted in Listing 1.1. For simplicity, we just assume that the input parameters of the `extractClass` operation are specified by the user, e.g., by selecting the elements in the modeling



(c)

No.	Operation	Depending Operations L1	Depending Operations L2
1	Add Class	Set name	
2	Add Association	Add Property	Set name, Set upper,...
		Add Property	Set name, Set upper,...
		Set name	
3	Move Property p1		

Fig. 4. Example for Endogenous Transformations: (a) Demonstration, (b) Revised Conditions, and (c) Change Model

editor. Another scenario would be that the preconditions are employed to match all occurrences of the initial situation for a given model.

Listing 1.1. Generated Refactoring Code

```
1  method extractClass(String className, String attName,
2          Class c, Collection<Property> props){
3
4      //Check precondition
5      assert c.attributes -> includesAll(props);
6
7      //Create additional class
8      Class newClass = new Class();
9      newClass.setName(className);
10
11     //Shift Attributes into new Class
12     Iterator iter = props.iterator();
13     while (iter.hasNext()){
14         Property p = iter.hasNext();
15         c.attributes().remove(p);
16         newClass.attributes().add(p);
17     }
18     ... //Create additional elements and link them properly
19 }
```

The first statement is to verify that the preconditions are fulfilled by the given input elements. In this example, only one precondition has to be checked, namely if the selected attributes are all included in the feature `attributes` of the selected class. After checking the precondition, a new class is created and each attribute contained in the collection `props` is moved to the new class. Assuming that the user configured the transformation to support extracting multiple attributes at once, all changes applied to the attribute in the demonstration are repeated in a loop. Afterwards, the additional elements, particularly the association and the

roles, have to be created and properly linked. Due to space limitations, this is not shown in the listing.

3.3 Existing Approaches

To the best of our knowledge, two MTBE approaches dedicated to endogenous transformations exist in literature. In the following we compare both approaches by highlighting their differences.

Brosch et al. [8,9] were the first to propose a "by-demonstration" approach to specify endogenous transformations. With this approach, endogenous transformations for any Ecore-based modeling language can be specified. Moreover, to be also independent from the used modeling editor, a *state-based model comparison* is employed to derive the atomic changes that have been performed on the initial model during the demonstration. The inherent imprecision of state-based model comparison is overcome by annotating unique identifiers to each model element before the user starts to demonstrate the transformation. By this, also element moves and intensively modified elements are supported. For expressing the pre- and postconditions, *OCL* constraints are employed. In the postconditions, users may also specify how attribute values in the target model shall be derived from values in the initial model. Repetitions of certain changes are realized by the notion of so-called *iterations*. Iterations are attached to precondition elements (representing model elements in the initial model) and indicate that each model element that fulfills these preconditions shall be transformed equally to the respective initial model element in the demonstration.

In the approach by Sun et al. [43], the changes applied during the demonstration are *recorded* and not derived by a subsequent comparison. After the demonstration, an inference engine generates a general transformation pattern which comprises the transformation's preconditions and its sequence of operations. This pattern may also be refined by the user in terms of adding preconditions and attribute value computations. In contrast to Brosch et al., *Groovy*[3]—a script language for the JVM—is employed to express these conditions and computations. In a more recent publication [42], Sun et al. extended this step so that users may also identify and annotate *generic operations*, which corresponds to the concept of iterations in [8]. However, these annotations are directly attached to the change model instead of to the preconditions as in [8].

4 MTBE for Exogenous Transformations

Various MTBE approaches [4,14,17,23,27,45,47], dedicated to exogenous transformations, have been proposed. Except [27] which is a *demonstration-based* approach, all others are based on *correspondences*. Thus, in the following, we discuss the general process of specifying exogenous transformations by-example based on correspondences, and subsequently, we present an instantiation of this

[3] `http://groovy.codehaus.org`

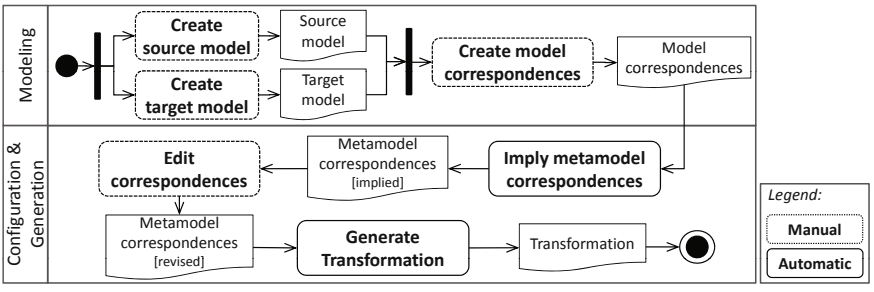


Fig. 5. MTBE Process for Exogenous Transformations

process for transforming UML Class Diagrams to ER Diagrams [11]. Finally, we conclude this section by elaborating on the peculiarities of current MTBE approaches.

4.1 Process

The main idea of MTBE for exogenous transformations is the semi-automatic generation of transformations from so-called correspondences between source and target model pairs. The underlying process for deriving exogenous model transformations from model pairs is depicted in Fig. 5. This process, which is largely the same for all existing approaches, consists of five steps grouped in two phases.

Phase 1: Modeling. In the first step, the user specifies semantically equivalent model pairs. Each pair consists of a source model and a corresponding target model. The user may decide whether she specifies a single model pair covering all important concepts of the modeling languages, or several model pairs whereby each pair focuses on one particular aspect. In the second step, the user has to align the source model and the target model by defining correspondences between source model elements and corresponding target model elements. For defining these correspondences, a correspondence language has to be available. One important requirement is that the correspondences may be established using the concrete syntax of the modeling languages. Hence, the modeling environment must be capable of visualizing the source and target models as well as the correspondences in one diagram or at least in one dedicated view.

Phase 2: Configuration & Generation. After finishing the mapping task, a dedicated reasoning algorithm is employed to automatically derive *metamodel correspondences* from the model correspondences. How the reasoning is actually performed is explained in more detail based on an example in Subsection 4.2. The automatically derived metamodel correspondences might not always reflect the intended mappings. Thus, the user may revise some metamodel correspondences or add further constraints and computations. Note that this step is not foreseen in all MTBE approaches, because it may be argued that this is contradicting with the general by-example idea of abstracting from the metamodels.

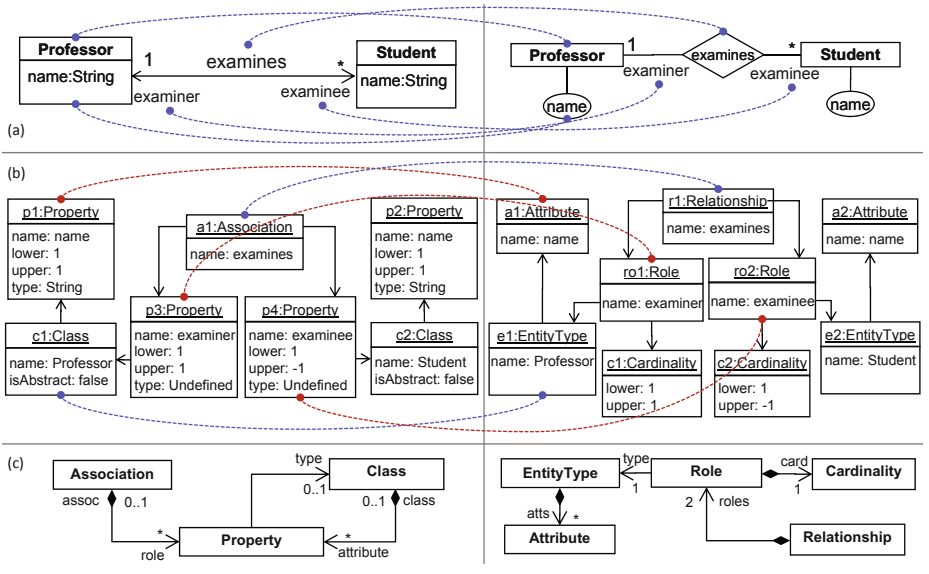


Fig. 6. Example for Exogenous Transformations: (a) Correspondences in concrete syntax, (b) Correspondences in abstract syntax, and (c) Metamodels

Nevertheless, it seems to be more user-friendly to allow the modification of the metamodel correspondences in contrast to modifying the generated model transformation code at the end of the generation process. Finally, a code generator takes the metamodel correspondences as input and generates executable model transformation code.

4.2 Example

For exemplifying the presented MTBE process, we now apply it to specify the transformation of the core concepts of UML class diagrams into ER diagrams. As modeling domain, a simple university information system is used. The user starts with creating the source model comprising the UML classes *Professor* and *Student* as well as a one-to-many association between them as depicted in the upper left area of Fig. 6. Subsequently, the corresponding ER diagram, depicted in the upper right area of Fig. 6, is created. In this figure, both models are represented in the concrete syntax as well as in the abstract syntax in terms of UML object diagrams. After both models are established, the correspondence model is created which consists of simple one-to-one mappings. These mappings are depicted as dashed lines in Fig. 6(a) and (b) between the source and target model elements.

In the next step, a reasoning algorithm now analyzes the model elements in the source and target models, i.e., objects, attribute values, and links, as well as the correspondences between them in order to derive metamodel correspondences. In

the following, we discuss inferring metamodel correspondences between classes, attributes, and references.

Class correspondences. For detecting class correspondences, the reasoning algorithm first checks whether a certain object type in the source model is always mapped to the same object type in the target model. In this case, a *full equivalence mapping* between the respective classes is generated. In our example, a full equivalence mapping between objects of type `Class` and objects of type `EntityType` is inferred. However, `Properties` in the source model are mapped to different object types, namely `Attributes` and `Roles`, depending on their attribute values and links. For such cases, an additional mapping kind is used, namely *conditional equivalence mapping*. The conditions of such a mapping are derived by analyzing the links and values of the involved objects to find a discriminator for splitting the source objects into distinct sets having an unambiguous mapping to target objects. One appropriate heuristic for finding such a discriminator is to examine the container links of these objects. By this, the algorithm may deduce the constraints `property.class != null` to find an unambiguous mapping to `Attributes` and the condition `property.assoc != null` for `Roles`. Finally, also unmapped objects such as the `Cardinality` objects have to be considered. In our example, these objects have to be generated along with their container objects of type `Role`. Thus, the mapping for `Roles` has to be extended to a *one-to-two conditional equivalence mapping*. By this, a `Role` object and a properly linked `Cardinality` object is created for each `Property` in the source model.

Attribute correspondences. Generally, attributes in metamodels may be distinguished in *ontological attributes* and *linguistic attributes* [20]. Ontological attributes represent semantics of the real-world domain. Values have to be explicitly given by the user. Examples for ontological attributes are `Class.name` or `Attribute.name`. In order to find correspondences between ontological attributes, heuristics have to be used which compare the attribute values, for instance, based on edit distance metrics. In our example, we may conclude that `Class.name` should be mapped to `EntityType.name` because the values of the `name` attributes are equivalent for each `Class`/`EntityType` object pair. In contrast, linguistic attributes are used for the reification of modeling concepts such as `Class.isAbstract`. Usually these attributes have predefined, restricted value ranges in the language definition. When dealing with linguistic attributes in the context of MTBE, similar heuristics based on string matching as for ontological attributes may be used. However, the probability for accidentally matching wrong pairs and for ambiguities is much higher. Consider for instance the mapping between the property `p3` and the role `ro1` without taking into account other mappings. Then, we cannot decide if the attribute `Property.lower` is mapped to `Role.cardinality.lower` or to `Role.cardinality.upper` by solely looking at the example. Here, the problem is that we do not have unique values which help us finding the metamodel correspondences. This may be improved by using matching techniques on the metamodel level for finding similarities between attribute names. An alternative solution used in this example is to define an

additional mapping between the property `p4` and the role `ro2` where we have unique values for the lower and upper attributes.

Reference correspondences. For deriving reference correspondences, the afore calculated class correspondences are of paramount importance since they serve as anchors for reasoning about corresponding links. For example, consider the reference `atts` in the ER metamodel between `EntityType` and `Attribute`. For finding the corresponding reference in the UML metamodel, we have to reason about the previously derived class correspondences. First, the `Attribute` class in the ER metamodel is mapped to the `Property` class of the UML metamodel. Furthermore, when looking at the example models, each `Attribute` is contained by an `EntityType` and each `Property` is contained by a `Class`. Luckily, the `EntityType` class is accordingly mapped to the `Class` class on the metamodel level, so that we can conclude that whenever transforming a `Property` into an ER `Attribute`, a link between the created `Attribute` and the `EntityType` previously generated for the `Class` containing the aforementioned `Property` is generated. Thus, there should be a correspondence between the reference `atts` in the ER metamodel and the reference `attribute` in the UML metamodel.

After the metamodel correspondences have been derived automatically, MTBE approaches usually allow the user to verify and adapt the generated correspondences. For our running example however, this is not required. The next task is to automatically translate the correspondences into executable transformation code. Listing 1.2 depicts the transformation required for our running example in imperative OCL [10]. For each metamodel correspondence, a transformation rule is generated which queries the source model and generates the corresponding target model elements. Inside each rule, the attribute and reference correspondences are translated to assignments. Please note that current transformation engines are able to schedule rules automatically and to build an implicit trace model between the source and target model. Based on this trace model, assignments such as `e.atts = c.attribute` (cf. line 2 in Listing 1.2) are automatically resolved. In particular, not the UML attributes (*c.attribute*) are assigned to the `EntityType`, but the ER attributes generated from these UML attributes are resolved by applying the trace model. These features of transformation languages and their encompassing engines drastically ease the transformation code generation from correspondences.

Listing 1.2. Generated Transformation Code

```
1  rule 1: Class.allInstances() -> foreach (c |
2      create Entity e (e.name = c.name, e.atts = c.attribute);
3  rule 2: Property.allInstances()
4      -> select(p | p.class <> OclUndefined) -> foreach( p |
5      create Attribute a (a.name = p.name));
6  rule 3: Property.allInstances() -> select(p |
7      p.assoc <> OclUndefined) -> foreach( p |
8      create Role r (r.name = p.name, r.cardinality = c),
9      create Cardinality c (c.upper = p.upper, c.lower = p.lower,
10     r.type = p.type));
11 rule 4: Association.allInstances() -> foreach(a |
12     create Relationship r (r.name = a.name, r.roles = a.role));
```

4.3 Existing Approaches

We now compare existing approaches by highlighting their commonalities and differences. Mostly all approaches define the input for deriving exogenous transformations as a triple comprising an input model, a semantically equivalent output model as well as correspondences between these two models. These models have to be built by the user, preferably using the concrete syntax as is, e.g., supported by [47], but most approaches do not provide dedicated support for defining the correspondences in graphical modeling editors.

Langer et al. [27] presented, in contrast to the correspondence-based approaches, a demonstration-based approach which allows to demonstrate transformation rules incrementally by giving for each rule an input model fragment and a corresponding output model fragment so that the correspondences between the fragments can be automatically inferred and do not have to be manually specified in advance.

Subsequently, reasoning techniques such as specific rules again implemented as model transformations [17,27,45,47], inductive logic [4], and relational concept analysis [14] are used to derive model transformation code. Current approaches support the generation of graph transformation rules [4,45] or ATL code [17,27,47].

All approaches aim for *semi-automated* transformation generation meaning that the generated transformations are intended to be further refined by the user. This is especially required for transformations involving global model queries and attribute calculations such as aggregation functions, which have to be manually added. Furthermore, it is recommended to iteratively develop the transformations, i.e., after generating the transformations from initial examples, the examples must be adjusted or the transformation rules must be adapted in case the actual generated output model is not fully equivalent to the expected output model. However, in many cases it is not obvious whether to adapt *the aligned examples* or the *generated transformations*. Furthermore, adjusting the examples might be a tedious process requiring a large number of transformation examples to assure the quality of the inferred rules. In this context, self-tuning transformations have been introduced [22,23]. Self-tuning transformations employ the examples as training instances in an iterative process for further improving the quality of the transformation. The goal is to minimize the differences between the actual output model produced by the transformation and the expected output model given by the user by using the differences to adapt the transformation over several iterations. Of course, adapting the transformation is a computation intensive problem leading to very large search spaces. While in [22] domain-specific search space pruning tailored to EMF-based models is used, a generic meta-heuristic based approach is used in [23] to avoid an exhaustive search.

5 Lessons Learned and Future Challenges

In this section, some lessons learned from applying and developing MTBE approaches during the last 5 years are summarized.

		MTBE Technique	
		Correspondences	***Demonstrations***
Transformation Scenario	***Endogenous Transformations***		[8,42]
	Exogenous Transformations	[4,14,17,23,45,47]	[27]

Fig. 7. Classification of MTBE approaches by whether they support exogenous or endogenous transformations and whether they are correspondence or demonstration-based

Different Transformation Scenarios/Different MTBE Techniques. When categorizing MTBE approaches w.r.t. transformation scenarios and MTBE techniques (cf. Fig 7), the following discriminators are derivable. Approaches using correspondences are exclusively but intensively applied for deriving exogenous transformations. Surprisingly, not a single work considers to apply correspondences for endogenous transformations. In contrast, demonstration-based approaches originally have been proposed for endogenous transformations and only one work discusses the application of demonstrations to derive exogenous transformations.

Challenge: *What are the commonalities and differences of correspondence-based and demonstration-based approaches?*

MTBE as Enabler for Test-driven Transformation Development. A significant advantage of MTBE is the existence of examples. Besides serving as input for the derivation of a model transformation during the MTBE process, the example models may also be used to *test the generated transformation*. By applying the inferred transformation again to the source model, the obtained target model may be compared to the target model specified during the MTBE process. If any differences are found in the comparison, either the transformation or the target model is obviously wrong. In this sense, MTBE inherently implements the idea of *test-driven development* [5]. An interesting direction for future work in this area is to automatically suggest corrections to the transformation based on the detected differences between the target example model and the actual transformation result.

Challenge: *Which logic and machine learning techniques can be employed for optimizing the quality of derived transformations in reasonable time with a small amount of examples?*

MTBE outperforms Metamodel Matching. With the rise of the Semantic Web and the emerging abundance of ontologies, several automated matching approaches and tools have been proposed (cf. [37,40] for an overview). The typical output of such tools are correspondences mostly computed based on schema information, e.g., name and structure similarity. In experiments, we have reused ontology matching tools for matching metamodels by beforehand transforming

metamodels into corresponding ontologies. However, the quality of the produced correspondences is on average significantly lower compared to MTBE approaches [21]. The reasons for this are twofold. First, structural heterogeneities between metamodels and the mismatch between the terminology used for different modeling languages makes it hard to reason about correspondences solely on the metamodel level. Second, there is no automated evaluation of the quality of correspondences based on the model level, because the matching approaches are not bound to a specific integration scenario [6], such as transformation, merge, or search. Finally, we also learned that the preparation phase required for using MTBE approaches, i.e., building the example models, is less work than the comprehensive reworking phase, i.e, validating and correcting the correspondences, required for metamodel matching approaches.

Challenge: *More empirical studies on MTBE approaches for identifying the strengths and weaknesses of existing approaches are required.*

Multifaceted Usage of Examples. Another benefit of specifying endogenous model transformations by demonstration is to reuse the developed transformation specifications for *detecting applications of the transformation*. Since such a specification developed using an MTBD approach comprises the transformation's preconditions, postconditions, and its change pattern, a dedicated detection mechanism may triage arbitrary model differences for the transformation's change pattern and, given the pattern could be found, validate its pre- and postconditions to reveal an application of the transformation. This is especially useful if these transformations implement model refactorings because this knowledge gains valuable information on the evolution of a model and is of paramount importance for various application domains such as model co-evolution, model versioning, and model repository mining.

Challenge: *How may the developed examples and derived transformations be employed for supporting different model management tasks in MDE?*

6 Resume

More than 30 papers have been published in the first 5 years and more and more research groups start working in this area. MDE and by-example approaches both aim to ease the development of software systems. However, both stand on orthogonal dimensions. MDE aims to abstract from the implementation level of software systems such as particular technology platforms and programming languages by using platform independent modeling techniques. In contrast, by-example approaches aim to ease the development of systems by using examples instead of directly developing generalized programs. We believe that combining both paradigms seems to be promising and would have a major impact on end-user programming or better say end-user modeling.

References

1. Amelunxen, C., Königs, A., Rötschke, T., Schürr, A.: MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 361–375. Springer, Heidelberg (2006)
2. Atkinson, C., Kühne, T.: Model-Driven Development: A Metamodeling Foundation. IEEE Software 20(5), 36–41 (2003)
3. Baar, T., Whittle, J.: On the Usage of Concrete Syntax in Model Transformation Rules. In: Virbitskaite, I., Voronkov, A. (eds.) PSI 2006. LNCS, vol. 4378, pp. 84–97. Springer, Heidelberg (2007)
4. Balogh, Z., Varró, D.: Model transformation by example using inductive logic programming. Software and System Modeling 8(3), 347–364 (2009)
5. Beck, K.: Test Driven Development: By Example. Addison-Wesley (2002)
6. Bernstein, P.A., Melnik, S.: Model management 2.0: manipulating richer mappings. In: Int. Conf. on Management of Data (SIGMOD 2007), pp. 1–12. ACM (2007)
7. Bézivin, J.: On the unification power of models. Software and System Modeling 4(2), 171–188 (2005)
8. 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: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 271–285. Springer, Heidelberg (2009)
9. Brosch, P., Langer, P., Seidl, M., Wimmer, M.: Towards End-User Adaptable Model Versioning: The By-Example Operation Recorder. In: Proc. of CVSM 2009 @ ICSE 2009. IEEE (2009)
10. Cabot, J.: From Declarative to Imperative UML/OCL Operation Specifications. In: Parent, C., Schewe, K.-D., Storey, V.C., Thalheim, B. (eds.) ER 2007. LNCS, vol. 4801, pp. 198–213. Springer, Heidelberg (2007)
11. Chen, P.P.S.: The Entity-Relationship Model—Toward a Unified View of Data. ACM Transactions on Database Systems 1, 9–36 (1976)
12. Cuadrado, J.S., Molina, J.G., Tortosa, M.M.: RubyTL: A Practical, Extensible Transformation Language. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 158–172. Springer, Heidelberg (2006)
13. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3), 621–646 (2006)
14. Dolques, X., Huchard, M., Nebut, C.: From transformation traces to transformation rules: Assisting Model Driven Engineering approach with Formal Concept Analysis. In: 17th Int. Conf. on Conceptual Structures (ICCS 2009), vol. 483, pp. 15–29. CEUR-WS (2009)
15. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston (1999)
16. France, R., Rumpe, B.: Model-driven Development of Complex Software: A Research Roadmap. In: 29th Int. Conf. on Software Engineering (ICSE 2007) - Future of Software Engineering, pp. 37–54 (2007)
17. 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: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 52–66. Springer, Heidelberg (2009)
18. ISO/IEC: 14977:1996(E) Information technology – Syntactic metalanguage – Extended BNF, International standard (1996)

19. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Sci. Comput. Program 72(1-2), 31–39 (2008)
20. Kappel, G., Kapsammer, E., Kargl, H., Kramler, G., Reiter, T., Retschitzegger, W., Schwinger, W., Wimmer, M.: Lifting Metamodels to Ontologies: A Step to the Semantic Integration of Modeling Languages. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 528–542. Springer, Heidelberg (2006)
21. Kappel, G., Kargl, H., Kramler, G., Schauerhuber, A., Seidl, M., Strommer, M., Wimmer, M.: Matching Metamodels with Semantic Systems - An Experience Report. In: Workshop Proceedings of Datenbanksysteme in Business, Technologie und Web, BTW 2007 (2007)
22. Kargl, H., Wimmer, M., Seidl, M., Kappel, G.: SmartMatcher: Improving Automatically Generated Transformations. Datenbank-Spektrum 29, 42–52 (2009)
23. Kessentini, M., Sahraoui, H.A., Boukadoum, M.: Model Transformation as an Optimization Problem. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 159–173. Springer, Heidelberg (2008)
24. Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Addison-Wesley (2008)
25. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The Epsilon Transformation Language. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 46–60. Springer, Heidelberg (2008)
26. Kühne, T.: Matters of (Meta-)Modeling. Software and System Modeling 5(4), 369–385 (2006)
27. Langer, P., Wimmer, M., Kappel, G.: Model-to-Model Transformations By Demonstration. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 153–167. Springer, Heidelberg (2010)
28. de Lara, J., Vangheluwe, H.: AToM^3: A Tool for Multi-formalism and Metamodelling. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 174–188. Springer, Heidelberg (2002)
29. Lieberman, H.: Your wish is my command: programming by example. Morgan Kaufmann Publishers Inc. (2001)
30. Ma, H., Shao, W.-Z., Zhang, L., Ma, Z.-Y., Jiang, Y.-B.: Applying OO Metrics to Assess UML Meta-models. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (eds.) UML 2004. LNCS, vol. 3273, pp. 12–26. Springer, Heidelberg (2004)
31. Mens, T., Gorp, P.V.: A Taxonomy of Model Transformation. Electr. Notes Theor. Comput. Sci. 152, 125–142 (2006)
32. Nickel, U., Niere, J., Zündorf, A.: The FUJABA environment. In: Int. Conf. on Software Engineering (ICSE 2000), pp. 742–745 (2000)
33. Object Management Group (OMG): Meta Object Facility, Version 2.0 (2006), http://www.omg.org/spec/MOF/2.0/PDF/
34. Object Management Group (OMG): Unified Modeling Language Superstructure Specification, Version 2.1.2 (2007), http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF
35. OMG, O.: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Final Adopted Specification (November 2005)
36. OMG, O.: OCL Specification Version 2.0 (June 2005), http://www.omg.org/docs/ptc/05-06-06.pdf
37. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. VLDB Journal 10(4), 334–350 (2001)
38. Schmidt, D.C.: Model-Driven Engineering. IEEE Computer 39(2), 25–31 (2006)

39. Sendall, S., Kozaczynski, W.: Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Software 20, 42–45 (2003)
40. Shvaiko, P., Euzenat, J.: A Survey of Schema-Based Matching Approaches. In: Spaccapietra, S. (ed.) Journal on Data Semantics IV. LNCS, vol. 3730, pp. 146–171. Springer, Heidelberg (2005)
41. Strommer, M., Wimmer, M.: A Framework for Model Transformation By-Example: Concepts and Tool Support. In: 46th Int. Conf. on Objects, Components, Models and Patterns (TOOLS 2008). LNBIP, vol. 11, pp. 372–391. Springer, Heidelberg (2008)
42. Sun, Y., Gray, J., White, J.: MT-Scribe: an end-user approach to automate software model evolution. In: 33rd Int. Conf. on Software Engineering (ICSE 2011), pp. 980–982. ACM (2011)
43. Sun, Y., White, J., Gray, J.: Model Transformation by Demonstration. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 712–726. Springer, Heidelberg (2009)
44. Taentzer, G.: AGG: A Graph Transformation Environment for Modeling and Validation of Software. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 446–453. Springer, Heidelberg (2004)
45. Varró, D.: Model Transformation by Example. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 410–424. Springer, Heidelberg (2006)
46. Whittle, J., Moreira, A., Araújo, J., Jayaraman, P.K., Elkhodary, A.M., Rabbi, R.: An Expressive Aspect Composition Language for UML State Diagrams. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 514–528. Springer, Heidelberg (2007)
47. Wimmer, M., Strommer, M., Kargl, H., Kramler, G.: Towards Model Transformation Generation By-Example. In: 40th Hawaiian Int. Conf. on Systems Science (HICSS 2007). IEEE Computer Society (2007)
48. Zloof, M.M.: Query-by-Example: the Invocation and Definition of Tables and Forms. In: Int. Conf. on Very Large Data Bases (VLDB 1975), pp. 1–24. ACM (1975)