Model-to-Model Transformations By Demonstration

Conference Paper · June 2010

DOI: 10.1007/978-3-642-13688-7_11 · Source: DBLP

3 authors, including:

Philip Langer
TU Wien
45 PUBLICATIONS **352** CITATIONS

Model-to-Model Transformations By Demonstration*

Philip Langer[1], Manuel Wimmer[2], and Gerti Kappel[2]

[1] Department of Telecooperation, Johannes Kepler University Linz, Austria
philip.langer@jku.ac.at
[2] Business Informatics Group, Vienna University of Technology, Austria
{wimmer|gerti}@big.tuwien.ac.at

Abstract. During the last decade several approaches have been proposed for easing the burden of writing model transformation rules manually. Among them are Model Transformation By-Demonstration (MTBD) approaches which record actions performed on example models to derive general operations. A current restriction of MTBD is that until now they are only available for in-place transformations, but not for model-to-model (M2M) transformations.
In this paper, we extend our MTBD approach, which is designed for in-place transformations, to also support M2M transformations. In particular, we propose to demonstrate each transformation rule by modeling a source model fragment and a corresponding target model fragment. From these example pairs, the applied edit operations are computed which are input for a semi-automatic process for deriving the general transformation rules. For showing the applicability of the approach, we developed an Eclipse-based prototype supporting the generation of ATL code out of EMF-based example models.

Key words: model transformations, by-example, by-demonstration

1 Introduction

Model transformations are an essential constituent in Model-driven Engineering (MDE) [6]. Therefore, several approaches have been proposed for easing the burden of writing model transformation rules by hand. One of the most prominent approaches is Model Transformation By-Example (MTBE) [12, 14] which tries to generalize model transformation rules from aligned example models by comparing the structure and the content of the example models. A similar idea is followed by Model Transformation By-Demonstration (MTBD) approaches [3, 13]. MTBD exploits edit operations performed on an example model to gain transformation specifications which are executable on arbitrary models.

* This work has been partly funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) and FFG under grant FIT-IT-819584 and by the Austrian Science Fund (FWF) under grant P21374-N13.

Until now, MTBD approaches are only available for in-place transformations like refactorings. An open challenge is how to adapt MTBD to be applicable for model-to-model (M2M) transformations due to the following hitherto unsolved issues. First, when using MTBD for in-place transformations, the trace model between the initial model (before transformation execution) and the revised model (after transformation execution) comes nearly for free. That trace may either be achieved using an ID-based comparison [3] or by directly recording all performed actions [13]. Unfortunately, these methods cannot be used for M2M transformation examples, because the corresponding elements in the source and target model are independently created and, consequently, have different IDs. Additionally, they are in most cases structurally heterogeneous. Second, state-of-the-art MTBD approaches for in-place transformations allow to specify one composite operation, e.g., a refactoring, performed on the initial model. After generalization, the resulting composite operation is executed on arbitrary models separately from other composite operations. However, in M2M scenarios, the whole target model has to be established based on the source model from scratch by applying a set of different, strongly interdependent transformation rules.

In this paper we tackle the mentioned challenges with a novel MTBD approach for developing M2M transformations. In particular, we elaborate on how the approach presented in [3] for developing in-place transformations is adapted for M2M transformations. To specify a M2M transformation, the user iteratively demonstrates each transformation rule by specifying an example using her preferred editor. Subsequently, the example models are automatically generalized to templates which the user may configure and customize by following a well-defined annotation process. Finally, model transformation rules are automatically derived from these annotated templates. Since the user only gets in touch with templates representing the user-specified examples, she is able to develop general model transformations without requiring in-depth knowledge of the underlying transformation language. Please note that our approach is orthogonal to existing high-level transformation approaches, such as Triple Graph Grammars and QVT Relations, because instead of directly developing the generalized templates, the user first develops concrete examples which are then systematically generalized. For showing the applicability of the approach, we developed an Eclipse-based prototype which supports the generation of ATL code out of EMF-based example models.

The paper is organized as follows. Starting with a motivating example in Section 2, we outline the process of developing M2M transformations by-demonstration in Section 3. Section 4 provides a by-example presentation of the by-demonstration approach. Section 5 discusses related work, and finally, we conclude with an outlook on future work in Section 6.

2 Motivating Example

To emphasize our motivation for developing a by-demonstration approach for M2M transformations, we introduce a well-known M2M transformation scenario,

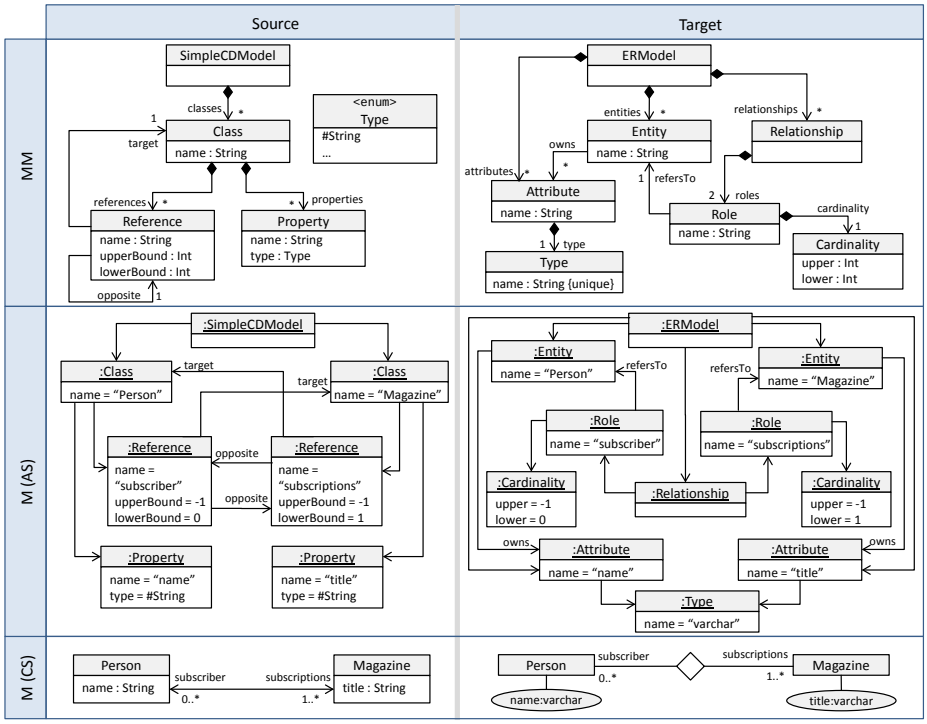

Fig. 1. Motivating example: Source metamodel, target metamodel, source model (abstract syntax), target model (abstract syntax), source model (concrete syntax), and target model (concrete syntax).

namely the transformation from UML Class Diagrams to Entity Relationship (ER) Diagrams. Fig. 1 illustrates the scenario which serves as a running example throughout the rest of the paper. Although the involved modeling languages provide semantically similar modeling concepts, this scenario also exhibits challenging correspondences between metamodel elements.

In the following, the main correspondences between the UML Class Diagram and the ER Diagram are shortly described. Simple one-to-one correspondences exist between the root containers `SimpleCDModel` and `ERModel` as well as between `Class` and `Entity`. However, the example also contains more complex correspondences. In particular, these are the correspondences between (1) the class `Property` and the classes `Attribute` and `Type` as well as (2) between the class `Reference` and the classes `Relationship`, `Role`, and `Cardinality`. In the first case, for each property, an attribute has to be generated. However, only for each distinct value of `Property.type` a type should be generated. When a type already exists with the same name, it should be reused. In the second case, for *every unique pair* of `References` that are marked as opposite of each other a corresponding `Relationship` has to be established containing two `Roles`, which

again contain their `Cardinalities`. With *every unique pair*, we mean that the order in which the references are matched does not matter. For example, if `Reference` r1 and `Reference` r2 are marked as opposite, then the transformation should produce *one* relationship for the match <r1, r2>, instead of creating another one for <r2, r1>. Therefore, we speak about the matching strategy *Set* if the order of the matched elements does not matter, and *Sequence* if the order does matter. On the attribute level, only simple one-to-one correspondences occur. On the reference level, some references can easily be mapped, e.g., `SimpleCDModel.classes` to `ERModel.entities`. However, some references on the target side have to be computed from the context of the source side, because they miss a direct counterpart, e.g., `ERModel.relationship`.

3 M2M Transformation By-Demonstration at a Glance

The design rationale for our by-demonstration approach is as follows. M2M transformations may be seen as a set of operations that are applied to the target model for each occurrence of a pattern of model elements in the source model. Thus, the target model is incrementally built by finding patterns in the source model and by applying the appropriate operations to the target model. Target elements created by these operations might need to be added to and refer to already existing elements, which had been created in prior transformation steps. Therefore, operations mostly have to be applied within a context. To enable the derivation of a transformation rule from examples, we apply the *by-demonstration process* depicted in Fig. 2.

Phase 1: Modeling. The user demonstrates a single transformation rule by adding model elements to the source model and by modeling the desired outcome in the target model. A transformation usually consists of several transformation rules. If a rule does not depend on other rules, no context elements are necessary to illustrate the rule, thus the user creates empty models. But usually, rules depend on other rules, which must have been previously applied forming the context. Thus, they are called *context rules*. Therefore, the user might select a context in which a new rule is demonstrated. If a context rule is selected, the source and target example model contained by the context rule is extended by the user to demonstrate the new context-dependent rule. For ensuring a high reusability of rules as context rules, they should be as small as possible.

Phase 2: Generalization. Added elements are identified and the illustrated transformation scenario is generalized. To determine the new elements if the demonstrated rule is context-dependent, we conduct a comparison between the revised models (source and target) to the respective models of the selected context rules. If the rule is context-free, all elements are considered as new. The new elements in the source model act as "trigger elements" which trigger to create the detected new elements in the target model. The most obvious way to identify the new elements is to record user interactions within the modeling environment. However, this would demand an intervention in the modeling environment, and due to the multitude of modeling environments, we refrain from

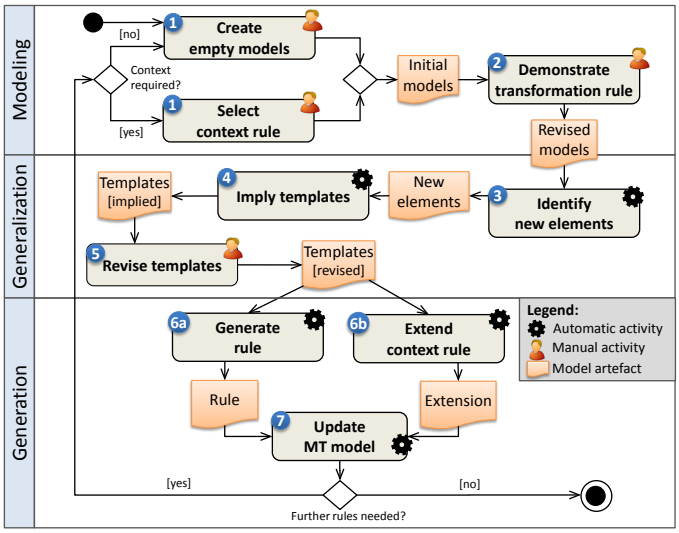

Fig. 2. By-Demonstration Process.

this possibility. Instead, we apply a state-based comparison to determine the executed operations after modeling the context models and the extended models. This allows the use of any editor without depending on editor-specific extensions. After the new elements are identified, we automatically imply templates for each model element in the respective models. A template is a generalized description of a model element. It contains conditions, which have to be fulfilled to identify an arbitrary model element as a valid match for a template. The automatic, metamodel-agnostic condition generation is done by creating a new condition for each feature and its value of a specific model element. After an automatic default configuration like the deactivation of conditions restricting to empty feature values is applied, the user may refine templates by adding or modifying certain conditions during the REMA process.

Attribute values in the target model usually depend on values in the source model. Therefore, we search for similar values in the source and the target model's elements and, for each detected similarity, automatically derive suggestions for the user to create attribute value correspondences. Accepted correspondences are incorporated by adding them to the condition in the target template. Attribute correspondence conditions bind a feature value or a combination of feature values in the source template model to a feature in the target template model. Unambiguous correspondences are automatically added, but the user might adjust the conditions using the semi-automatic REMA process by **r**elaxing, **e**nforcing, **m**odifying or **a**ugmenting templates.

We distinguish between usual templates and inherited templates. Usual templates represent model elements that have been newly created in the current

demonstration. Inherited templates represent context elements that have either been already introduced in a context rule, or that conform to a template in a context rule, i.e., they are processed by the context rule.

Phase 3: Generation. After the revision of the templates, transformation rules are generated by a higher-order transformation. In particular, for the demonstrated scenario, a new rule has to be generated and attached to the transformation model. Furthermore, in case a context-dependent scenarios has been illustrated, the context rules have to be extended with further reference bindings to the newly introduced elements referenced by the context elements.

4 M2M Transformation By-Demonstration in Action

In the previous section, we illustrated the by-demonstration process from a generic point of view. In this section, we show how this process is adopted from a user's point of view. In particular, we discuss each iteration necessary to solve the motivating example of Section 2. To support the user in the demonstration process, we implemented a prototype presented on our project homepage[1].

4.1 Iteration 1: Class Diagram to Entity Relationship Diagram

In Iteration 1, a context-free object-to-object correspondence is illustrated to create for each `SimpleCDModel` instance an `ERModel` instance.

Step 1: Create empty models. The user creates a context-free rule by specifying an empty source model and an empty target model. These models are extended in the following steps.

Step 2: Demonstrate transformation rule. To illustrate the transformation of `SimpleCDModels` to `ERModels`, the user just has to add these elements to the empty models as shown in Fig. 3(a).

Step 3: Identifying new elements. Since the demonstrated rule is context-free, all model elements are considered as new.

Step 4: Imply templates. The example models are generalized by automatically implying templates for each model element. The goal of creating these templates is to generically describe model elements. With the help of source templates, we are able to verify if arbitrary model elements should be transformed equally to the illustrated model elements. Target templates indicate which properties and values should be set in the target elements according to the source model. In this example the templates `Template_L_1_1` and `Template_R_1_1` (cf. *specific templates* in Fig. 3(a)) are implied for the respective elements in the example models. The `L` in the template name indicates the *left* side and `R` the right side. The first digit in the template name indicates the rule it has been introduced. The second digit enumerates the templates. Since both elements do not contain any classes or entities, for both templates a condition is created which constrains these features to be empty, e.g., cf. `classes = {}`. After all templates

[1] http://www.modelversioning.org/m2m-operations

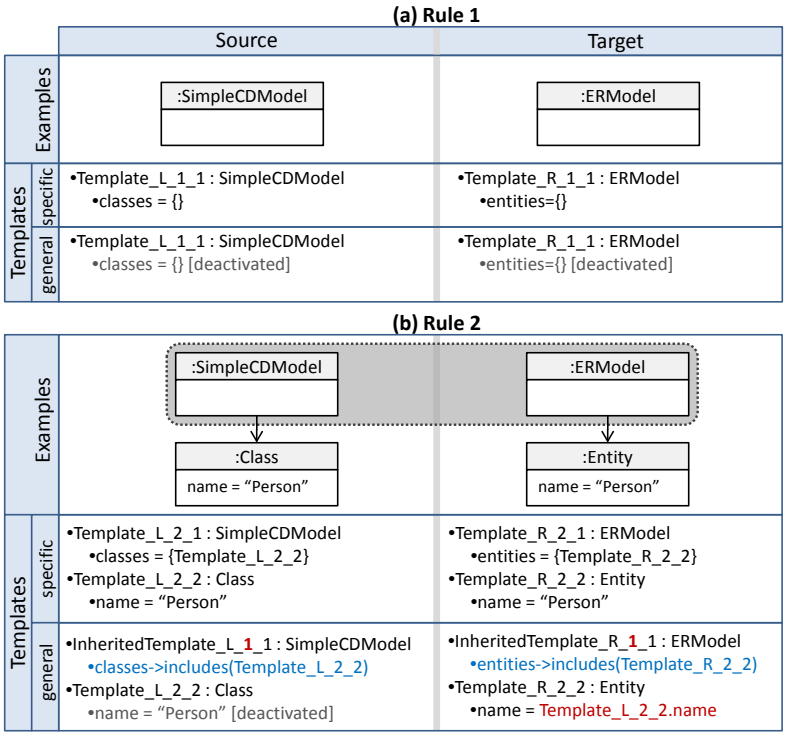

Fig. 3. (a) Rule 1: Class Diagram to ER Diagram. (b) Rule 2: Class to Entity.

are initially generated, they are automatically pre-configured and generalized. This is done by deactivating all conditions in the source templates by default. Solely, source template conditions referring to object values that are represented by other source templates are left to be active. Consequently, source templates only restrict the type of the elements and their dependencies to other source templates. Additionally, conditions in the target templates are deactivated if the features are not set (cf. *general templates* in Fig. 3(a)). This reflects an open world assumption. Only aspects are restricted which are explicitly modeled.

Step 5: Revise templates. Since the templates and their contained conditions are automatically implied, they might not always reflect the user's intention. Therefore, the user may adjust the generated templates and conditions using the *REMA* process. She may **r**elax currently active conditions, **e**nforce currently inactive conditions, or **m**odify existing conditions. Additionally, templates may be **a**ugmented by adding annotations. Using these techniques, the user might for instance tighten source templates by enforcing (reactivating) or modifying certain conditions to restrict the execution of a transformation rule. However, in this iteration none of these is necessary.

Step 6-7: Generation. The revised templates are transformed into ATL transformations. Basically, the source templates are transformed into the `from` block and the target templates into the `to` block of an ATL rule. Additional conditions in source templates are used as *guards* and attribute correspondences are set accordingly via bindings. The generated ATL rule for this iteration is shown in List. 1.1 (line 4-6, 11). Step 6b is not applicable for context-free rules, since no context rule has to be extended.

4.2 Iteration 2: Class to Entity

In this iteration, the transformation of `Classes` to `Entities` is demonstrated. This rule requires a one-to-one object correspondence, a value-to-value correspondence, and a context—the created target elements have to be added to an `ERModel` instantiated in the previous iteration. The example models, the implied templates, and the generalized templates are depicted in Fig. 3(b).

Step 1: Select context rule. `Classes` and `Entities` are always contained by `SimpleCDModels` and `ERModels`, respectively. Thus, the user has to select the transformation rule of Iteration 1 to be the context of the rule created in this iteration. When a context is selected, a copy of the context rule's example models is created and opened in diagram editors in order to be extended.

Step 2: Demonstrate transformation rule. The user extends the loaded context models to illustrate the transformation of a `Class` to an `Entity`. An instance of both model elements have to be added in the respective models. To allow a subsequent automatic detection of attribute value correspondences, the user should use exactly the same values for which a correspondence exists. Consequently, the class is named equally to the entity (cf. Fig. 3(b)).

Step 3: Identifying new elements. New elements are identified automatically by comparing the current source model to the source model of the context rule as well as the current target model to the context rule's target model. Thus, the class and the entity are marked as new elements.

Step 4: Imply templates. Like in the previous iteration, for each element in the example models, a template is implied and a condition for each feature is added to the template (cf. *specific templates* in Fig. 3(b)). In contrast to the previous iteration, the current rule depends on a context, i.e., it includes context elements to be processed by the context rule. For that reason, templates which represent a context model element are replaced during the generalization mechanism with *InheritedTemplate*s pointing to the respective template contained by the context rule (cf. *general templates* in Fig. 3(b)). The first digit of the template name indicates the context rule in which the elements have been introduced, e.g., `InheritedTemplate_R_1_1` represents the ERModel introduced in Iteration 1. Note that this inherited template is refined in this iteration by an additional condition (`entities->includes(Template_R_2_2)`. This condition indicates that the created entity has to be added to the feature `ERModel.entitites`. The conditions of the source templates are again deactivated by default. Additionally, for setting attribute correspondences, for each value in the target model,

a corresponding value in the source model is searched. If an unambiguous correspondence is detected, the target value is automatically restricted to be the value of the source element's attribute by replacing the value assignment (`name = ''Person''`) with a template reference (`name = Template_L_2_2.name`).

Step 5: Revise templates. Like in Iteration 1, no user adjustments are necessary due to the accurate default implications.

Step 6-7: Generation. After the generalization phase, the current rule is transformed into an ATL rule as shown in List. 1.1 (line 13-16, 18). As mentioned in Step 4, we refined `InheritedTemplate_R_1_1` with a new condition. This condition preserves the relationship of the context element `ERModel` to the newly added `Entity`. Hence, an assignment of generated `Entities` to the feature `entities` is added to the context rule (cf. line 7 in List. 1.1).

4.3 Iteration 3: Property to Attribute

Now the transformation of `Properties` to `Attributes` is demonstrated. Properties are contained by classes whereas attributes are directly contained by the root model element. Entities only incorporate a reference to the attributes they own. Moreover, in class diagrams, the property type is expressed using an attribute. In contrast, the type of an attribute in `ERModels` is represented by an additional instance. Thus, we need to specify a one-to-many object correspondence as well as two value-to-value correspondences.

Step 1: Select context rule. This transformation rule has to be illustrated within the context of Rule 1 and Rule 2, because Attributes are referenced by ERModels as well as by Entities.

Step 2: Demonstrate transformation rule. In the source model the user adds a property to the class created in Iteration 2. Correspondingly, an attribute with the same name is appended to the entity (cf. Fig. 4(a)). Corresponding to the type of the property, an instance of `Type` has to be created in the target model and linked to the attribute.

Step 3: Identifying new elements. As in the previous iterations, the new elements are identified properly using the state-based comparison.

Step 4: Imply templates. For each model element, a template is implied. Model elements which have already been created in previous iterations are represented by inherited templates. As in the previous iteration, the both inherited templates in the target template model are refined by additional conditions (e.g., `attributes->includes(Template_R_3_3)`), because the attribute is referenced by the entity and contained by the ERModel. The value-to-value correspondence regarding the attribute `name` is detected and annotated automatically (`name = Template_L_3_3.name`).

Step 5: Revise templates. In contrast to the previous iterations, the user now has to apply two augmentations in the REMA process. First, type has to be reused since it is not intended to add a new type each time an attribute is created. Instead, `Type` instances have to be reused whenever a type already exists with the same name. This is done by annotating the corresponding template with the *reuseObject* operator and providing the `name` feature as discriminator for reuse.

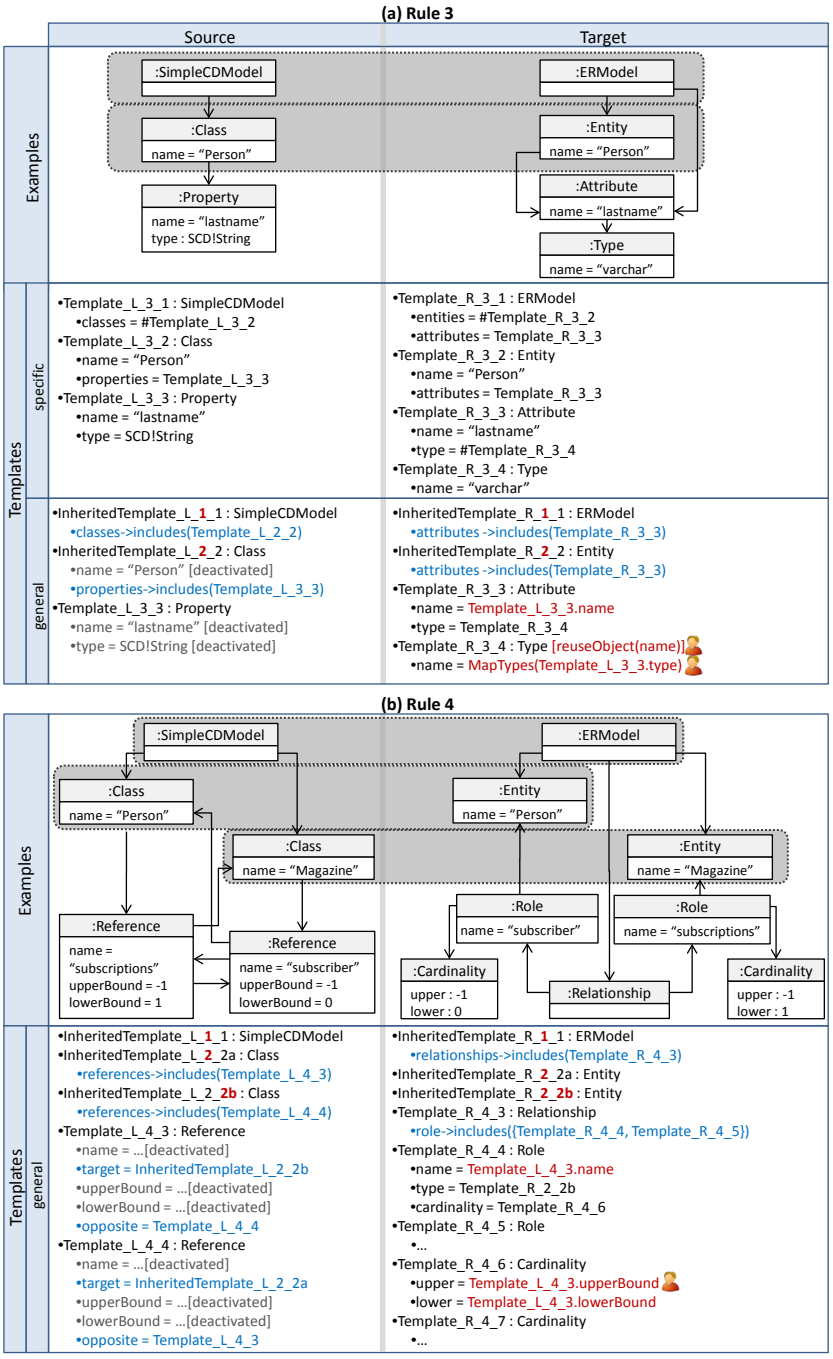

Fig. 4. (a) Rule 3: Property to Attribute. (b) Rule 4: Reference to Relationship.

Second, the literals of the `Type` enumeration of the Class Diagram have to be converted to String values in ER Diagrams. To enable such static value-to-value conversions, the user may set up a mapping table. In the template conditions, this table is used by calling its name (cf. `name = MapTypes(Template_L_3_3)`).

Step 6-7: Generation. A matched rule is created to generate attributes from properties (cf. line 25-33 in List. 1.1). A lazy rule and a helper is generated for creating types if necessary (cf. line 35-40, 23). Furthermore, a helper for the mapping table is generated (cf. line 20-21). Both ATL rules created for the used context rules are extended by new feature bindings (cf. line 8, 17).

4.4 Iteration 4: Reference to Relationship

The last iteration demonstrates the transformation of `References` to `Relationships`. For this, a many-to-many object correspondence is needed, since two references marked as opposite are transformed into a relationship with two `Roles` comprising `Cardinalities` (cf. Fig. 4(b)). As in the previous iteration, the context rule 2 has to be used. Furthermore, tuples of reference instances have to be processed only once by applying the *Set* matching strategy (cf. Section 2). In our experience, this is the intuitive matching strategy for multiple query patterns matching for the same type, thus it is used as default.

Step 1: Select context rule. The transformation of `References` into `Relationships` is in the context of Rule 1 and of Rule 2.

Step 2: Demonstrate transformation rule. A new class named "Magazine" and two new references ("subscriber" and "subscriptions") are added to the source model. In the target model, the user adds an Entity "Magazine", a relationship, two roles, and cardinalities for each role. All values in the target are consciously set according to the corresponding values in the source model.

Step 3: Identifying new elements. Beside the `Class` and `Entity` "Person", which has been directly added in the context rule, the user also added a second `Class` and correspondingly a second `Entity` named "Magazine". To correctly identify these two "Magazine" elements to be context elements and not new elements, we match each added element in the example against all context templates. Since the left and right "Magazine" elements are matching the corresponding context templates, they are considered as context elements.

Step 4: Imply templates. For each model element, a template or an inherited template is created. Since there are now two instances of `Class` and of `Entity` we enumerate the corresponding inherited templates with a letter (`a` and `b`). The `InheritedTemplate_R_1_1` (representing a `ERModel`) is extended for this rule by an additional condition specifying the reference to the added relationship. All attribute value correspondences are detected and annotated automatically. Solely, for the upper bounds of the `Cardinalities` only suggestions can be made since the values of these features cannot be unambiguously mapped.

Step 5: Revise templates. The current transformation rule should only be executed for reference pairs which are marked as `opposite` of each other. As already mentioned before, all source template conditions are deactivated, except for those, which refer to an object value that is represented by another

template. Consequently, also the two conditions restricting the two references to be `opposite` of each other remains active which is the intended configuration. Also the aforementioned difficulty regarding the matching strategy of these two instances of `Reference` is solved by using the matching strategy *Set* which is adopted by default. With this strategy, every combination of the reference instances irrespectively of their order is "consumed" during the transformation. Consequently, two references that refer to each other are only processed once by the resulting rule. If this is not intended, the user may annotate the templates to use the matching strategy *Sequence*.

Step 6-7: Generation. To realize the aforementioned *Set* matching strategy we generate a *unique lazy rule* with a guard expression (cf. line 42-56 in List. 1.1) which is called by the root rule to create and to add the relationships to the feature `ERModel.relationships` (cf. line 9-10).

5 Related Work

Varró [14] and Wimmer et al. [15] have been the first proposing to develop M2M transformation by-example. Both used input models, corresponding output models, and the alignments between them to derive general transformation rules. In [1], Balogh and Varró extended their MTBE approach by leveraging the power of inductive logic programming. As before, the input of their approach are one or more aligned source and target model pairs which are translated to Prolog clauses. These clauses are fed into an inductive logic programming engine that induces inference rules which are translated into model transformation rules. If these rules do not entirely represent the intended transformation, the user has to refine either the generated rules directly or she has to specify additional model pairs and start another induction iteration. That approach might require less user interaction compared to our approach, but we follow a different conceptual aim. By our demonstration approach we are aiming at a very interactive approach. In particular, the user is guided to demonstrate and configure each transformation rule iteratively. To ease that interaction for the user, in each iteration the user may focus only on one part of the potentially huge transformation until the current rule is correctly specified. We believe, this is a natural way of dividing and conquering the whole transformation.

For omitting to manually define alignments between source and target models, two further by-example approaches have been proposed. (1) García-Magariño et al. [7] propose to develop M2M transformations by annotating the source metamodel and the target metamodel with additional information, which is required to derive transformations based on given example models. Because the approach of García-Magariño et al. uses a predefined algorithm to derive the transformations purely automatically, the user has no possibility to influence the generalization process, which is in our point of view a must for developing model transformation in practice. The implication of this limitation is that most attribute correspondences cannot be detected as well as configurations such as reusing existing objects for aggregation or determining the matching strategy

such as Sequence or Set cannot be considered during the generalization process. The only possibility for the user is to adapt and extend the generated ATL code, which is more difficult compared to providing such configurations in our proposed template language. (2) Kessentini et al. [10] interpret M2M transformations as an optimization problem. Therefore, Kessentini et al. propose to apply an adapted version of a particle swarm optimization algorithm to find an optimal solution for the transformation problem, which is described by multiple source and target model pairs. However, as it is the case with most artificial intelligence approaches, only an approximation of the optimal solution can be found. This may be enough for some scenarios, e.g., searching for model elements in a model repository where the user has to select one of the best matches, for others, e.g., model exchange between different modeling tools, carefully-engineered model transformations are necessary [2]. Such scenarios are not supported by Kessentini et al., because the transformation logic is only implicitly available in the trained optimization algorithm, which is not adaptable.

Finally, a complementary approach for generating model transformations automatically is metamodel matching. Two dedicated approaches [4, 5] have been proposed for computing correspondences between metamodels which are input for generating model transformations. We have experimented with technologies for ontology matching by transforming metamodels into ontologies [9]. However, we have experienced [8] that in a setting where (1) metamodels use different terminology for naming metamodel elements and (2) the structures of metamodels are very heterogeneous, it is sometimes impossible for the matching algorithms to find the correct correspondences. However, we have to mention that a hybrid approach, i.e., combining a matching approach with a by-example approach, seems to be very promising for gaining the benefits of both worlds. We consider this topic as subject to future work.

6 Conclusions and Future Work

The presented by-demonstration approach provides a novel contribution to the field of MTBD for developing carefully-engineered M2M transformations. Our approach is metamodel-independent and does not rely on the editors used to illustrate the transformation scenarios. The automatic generalization technique uses default implications which are proven useful in most of the cases. However, the user may still fine-tune the derived rules using the REMA process without touching the automatically generated ATL code.

Up to now, we support four operators that may be used to annotate templates, namely *reuseObject*, *MapTypes*, as well as two different matching strategies *Set* and *Sequence*. In future work, we will evaluate our approach in further scenarios to determine if further operators are required. For instance, we will elaborate on *selector templates* to support complex user-defined selections of source elements, e.g., a recursive selection of elements. Negative application conditions (NAC) as well as many-to-one attribute correspondences are currently only supported by manually editing conditions. In future work, we plan

to imply NACs from user-specified examples and incorporate techniques from instance-based ontology matching [11] to allow automatic detection of many-to-one attribute correspondences. Furthermore, we intend to add features easing large transformation development like rule inheritance and debugging support. By employing user-specified test scenarios and comparing their transformation result to the desired result, we can backtrack the differences to the relevant rule and directly point the user to the template specification causing the difference. Finally, we will elaborate if it is possible to derive Triple Graph Grammar definitions from example models by using our REMA process. By this, we want to conduct if the approach is generic enough to generate also transformation rules for other transformation languages which might follow different trace models and execution semantics.

References

1. Z. Balogh and D. Varró. Model transformation by example using inductive logic programming. *Software and Systems Modeling*, 8(3):347–364, 2009.
2. P. A. Bernstein and S. Melnik. Model Management 2.0: Manipulating Richer Mappings. In *SIGMOD'07*, pages 1–12, 2007.
3. P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger. An Example Is Worth a Thousand Words: Composite Operation Modeling By-Example. In *MoDELS'09*, pages 271–285, 2009.
4. M. D. D. Fabro and P. Valduriez. Semi-automatic model integration using matching transformations and weaving models. In *SAC'07*, pages 963–970, 2007.
5. J.-R. Falleri, M. Huchard, M. Lafourcade, and C. Nebut. Metamodel matching for automatic model transformation generation. In *MoDELS'08*, pages 326–340, 2008.
6. R. France and B. Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *FOSE'07*, pages 37–54, 2007.
7. I. García-Magariño, J. J. Gómez-Sanz, and R. Fuentes-Fernández. Model Transformation By-Example: An Algorithm for Generating Many-to-Many Transformation Rules in Several Model Transformation Languages. In *ICMT'09*, pages 52–66, 2009.
8. G. Kappel, H. Kargl, G. Kramler, A. Schauerhuber, M. Seidl, M. Strommer, and M. Wimmer. Matching metamodels with semantic systems - an experience report. In *Workshop Proceedings of BTW'07*, pages 38–52, 2007.
9. G. Kappel, T. Reiter, H. Kargl, G. Kramler, E. Kapsammer, W. Retschitzegger, W. Schwinger, and M. Wimmer. Lifting metamodels to ontologies - a step to the semantic integration of modeling languages. In *MoDELS'06*, pages 528–542, 2006.
10. M. Kessentini, H. Sahraoui, and M. Boukadoum. Model Transformation as an Optimization Problem. In *MoDELS'08*, pages 159–173, 2008.
11. E. Rahm and P. A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *The VLDB Journal*, 10(4):334–350, 2001.
12. M. Strommer and M. Wimmer. A Framework for Model Transformation By-Example: Concepts and Tool Support. In *TOOLS'08*, pages 372–391, 2008.
13. Y. Sun, J. White, and J. Gray. Model transformation by demonstration. In *MoDELS'09*, pages 712–726, 2009.
14. D. Varró. Model Transformation by Example. In *MoDELS'06*, pages 410–424, 2006.
15. M. Wimmer, M. Strommer, H. Kargl, and G. Kramler. Towards Model Transformation Generation By-Example. In *HICSS'07*, pages 285–286, 2007.

Listing 1.1. Generated ATL Code

```
module CD2ER;
create OUT : ER from IN : CD;

rule GenerateERModel {
  from cdmodel : CD!SimpleCDModel
  to ermodel : ER!ERModel (
    entities <- cdmodel.classes,
    attributes <- cdmodel.classes ->
      collect(e| e.properties) -> flatten(),
    relationships <- cdmodel.classes -> collect(x| x.references) ->
      flatten() -> collect(x| thisModule.GenerateRelationship(
      x, x.opposite)))
}
-------------------------------------------------------------------------
rule GenerateEntity {
  from class : CD!Class
  to entity : ER!Entity (
    name <- class.name,
    attributes <- class.properties)
}
-------------------------------------------------------------------------
helper def : mapTypes(x : CD!Types) : ER!Types =
    Map{(#String,'varchar'), ...)}.get(x);

helper def : seenERTypes : Set(ER!Type) = Set{};

rule GenerateAttribute {
  from property : CD!Property
  to attribute : ER!Attribute (
    name <- property.name,
    type <-
      if(thisModule.seenERTypes -> exists(e|
                e.name = thisModule.mapTypes(property.type)))
      then thisModule.seenERTypes -> any(e|
                e.name = thisModule.mapTypes(property.type))
      else thisModule.CreateType(property.type) endif)
}

lazy rule CreateType {
  from cdType : CD!Types
  to erType : ER!Type (
    name <- thisModule.mapTypes(cdType))
  do{thisModule.seenERTypes <- thisModule.seenERTypes ->
        including(erType);}
}
-------------------------------------------------------------------------
unique lazy rule GenerateRelationship {
  from reference1 : CD!Reference,
     reference2 : CD!Reference (reference1.opposite = reference2)
  to relationship1 : ER!Relationship (
      roles <- Set{role1, role2}),
    role1 : ER!Role(
      name <- reference1.name,
      refersTo <- reference1.target,
      cardinality <- cardinality1),
    role2 : ER!Role(...),
    cardinality1 : ER!Cardinality(
      upper <- reference1.upperBound,
      lower <- reference1.lowerBound,
    cardinality2 : ER!Cardinality(...)
}
```