Learning Model Transformations from Examples using FCA: One for All or All for One? *

Hajer Saada[1], Xavier Dolques[2], Marianne Huchard[1], Clémentine Nebut[1], and Houari Sahraoui[3]

[1] LIRMM, Université de Montpellier 2 et CNRS, Montpellier, France, first.last@lirmm.fr
[2] INRIA, Centre Inria Rennes - Bretagne Atlantique, Campus universitaire de Beaulieu, 35042 Rennes, France, xavier.dolques@inria.fr
[3] DIRO, Université de Montréal, Canada, sahraouh@iro.umontreal.ca

Abstract. In Model-Driven Engineering (MDE), model transformations are basic and primordial entities. An efficient way to assist the definition of these transformations consists in completely or partially learning them. MTBE (Model Transformation By-Example) is an approach that aims at learning a model transformation from a set of examples, i.e. pairs of transformation source and target models. To implement this approach, we use Formal Concept Analysis as a learning mechanism in order to extract executable rules. In this paper, we investigate two learning strategies. In the first strategy, transformation rules are learned independently from each example. Then we gather these rules into a single set of rules. In the second strategy, we learn the set of rules from all the examples. The comparison of the two strategies on the well-known transformation problem of class diagrams to relational schema showed that the rules obtained from the two strategies are interesting. Besides the first one produces rules which are more proper to their examples and apply well compared to the second one which builds more detailed rules but larger and more difficult to analyze and to apply.

1 Introduction

Model Driven Engineering is a recent paradigm emphasizing the use of models in the development process of an application. The models may be graphical or not, but they are all structured conforming to particular models named meta-models. In Model-Driven Development, several kinds of models are successively handled (e.g. requirements, use cases, classes, etc.) and models may be obtained one from each other, in an automated way, thanks to model transformations. A Model Transformation is a program that takes as input a model conforming to a source meta-model and produces as output another model conforming to a target meta-model. Implementing a model transformation requires a strong knowledge about model driven engineering (meta-modeling and model-transformation environments) and about the specification of the transformation: the input domain,

* This work has been supported by project CUTTER ANR-10-BLAN-0219

© Laszlo Szathmary, Uta Priss (Eds.): CLA 2012, pp. 45–56, 2012.
ISBN 978–84–695–5252–0, Universidad de Málaga (Dept. Matemática Aplicada)

the output domain and the transformation rules by themselves. Moreover, the transformation rules are usually defined at the meta-model level, which requires a clear and deep understanding about the abstract syntax and semantic interrelationships between the source and target models [7].

Domain experts generally do not have sufficient skills in model driven engineering. An innovative approach called Model Transformation By Example (MTBE) [12] is proposed to let them design model transformation by giving an initial set of examples. An example consists of an input model, the corresponding transformed model and links explaining which target element(s) one or several source model elements are transformed into. From these examples, transformation rules are deduced using a learning approach.

In this context, we presented a Model Transformation By Example approach that goes from examples to transformation patterns. The proposed learning mechanism is based on Relational Concept Analysis (RCA) [5], one of the extensions of Formal Concept Analysis [3], that considers links between objects in the concept construction. This approach results in a lattice of transformation patterns. Those patterns are filtered to keep the more relevant ones, and are made operational using a rule engine. In this paper, we analyze and compare two strategies for learning the transformation patterns with RCA. In the first one, each example is used alone to learn transformation patterns, and the transformation patterns obtained from all the examples are then gathered. In the second strategy, the examples are first gathered into a single large example, that is then used to learn the transformation patterns. The obtained transformation patterns are inspected and applied to test examples.

The remainder of this paper is structured as follows. Section 2 gives an overview of the approach. Section 3 briefly explains how RCA is used to generate transformation patterns from examples. Section 4 details how the pattern lattices are filtered and refined. Section 5 describes the two learning strategies, and an experimentation to compare them on a case study. Section 6 positions our work w.r.t. related work, and Section 7 concludes the paper and describes future work.

2 Learning and executing model transformation rules

A model transformation is a program handling and transforming models. We focus here on transformations that take a model as input and result in another model as output. Model transformations are usually written in languages dedicated to model handling, or generic languages using adequate frameworks to handle models. Those transformations implement transformation rules, expressing for each kind of elements from an input model which kind of elements of the output model they will be transformed into. Model-Transformation By Example (MTBE) consists in learning those transformation rules from examples: instead of programming the model transformation, the user designs examples illustrating the behavior of the transformation, and the transformation rules are automatically learned from those examples.

Usually, an example is composed of a source model, the corresponding transformed model, and transformation links between those two models. To illustrate MTBE, we consider the well-known case of transforming UML class diagrams into entity relationship. An example for this transformation is thus composed of a UML model (the input model), a Relational model (the output model) and transformation links making explicit from which elements of the UML model, the elements of the entity relationship model stem from. Such an example is given in Figure 1. The input UML model is on the l.h.s., and is composed of two classes named `Text` and `Style`, each one owning an attribute (respectively named `Title` and `Name`) and linked with an association named `Has a`. The output model (on the r.h.s. of Figure 1) has two entities `Text` and `Style`, each one described by an attribute, linked with a relation named `Has a`. The dotted lines show some of the transformation links. For instance, there is a transformation link specifying that the class `Text` is mapped into the entity `Text`.

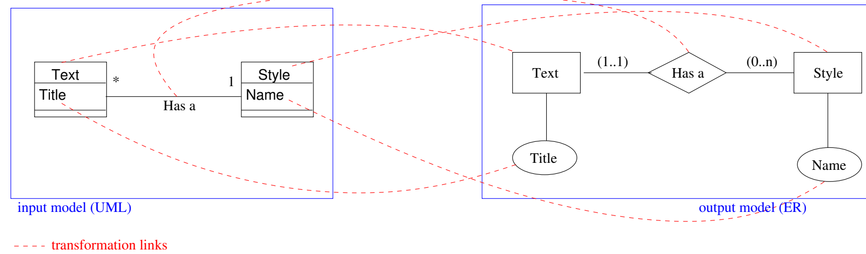

Fig. 1. Example for the UML2ER transformation: input model (l.h.s), output model (r.h.s.) and transformation links (dotted lines)

An MTBE process analyzes the examples and learns from them transformation rules such as *a class is transformed into an entity*, or *a UML property linked to a class (i.e., an attribute and not a role) is transformed into a role of an entity*.

We proposed an MTBE approach in which the learning mechanism relies on Relational Concept Analysis [2]. The abstract learned rules are named *transformation patterns*, they are obtained in a *transformation patterns lattice*, and are then filtered so as to select the more relevant ones. To make those transformation patterns operational so as to be able to execute the learned transformation, we designed a transformation from the transformation patterns to Jess rules [6], Jess being a rule engine [11].

3 RCA and transformation patterns discovery

As stated in Section 2, a key step in our MTBE approach consists in generating transformation patterns. Such patterns describe how a source model element is

transformed into a target model element, within a given source context and a given target context. To derive patterns from examples, a data analysis method is used, namely Formal Concept Analysis (FCA) [3] and its extension to relational data, the Relational Concept Analysis (RCA) [5]. Both Formal and Relational Concept Analysis, also used for data mining problems, group entities described by characteristics into concepts, ordered in a lattice structure. While FCA produces a single classification from one formal context, RCA computes several connected classifications from several formal contexts linked by relational contexts.

Definition 1 (Relational Context Family). *A* Relational Context Family $\mathfrak{R}$ *is a couple* (K, R). K *is a set of* Object-Attribute Contexts $K_i = (O_i, A_i, I_i)$ *where* O_i *is a set of objects,* A_i *is a set of attributes and* $I_i \subseteq O_i \times A_i$. R *is a set of* Object-Object contexts $R_j = (O_k, O_l, I_j, S_j)$ *where* $(O_k, A_k, I_k) \in K$, $(O_l, A_l, I_l) \in K$, $I_j \subseteq O_k \times O_l$, *and* S_j *is a scaling operator,* i.e. *a boolean function taking as parameter an object from* O_k, *a concept extent* $e \subseteq O_l$ *and the binary relation* I_j.

RCA considers a Relational Context Family $\mathfrak{R} = (K, R)$ as input of the lattice building process. This process applies iteratively FCA on each Object-Attribute Context from K extended with the Object-Object contexts of R scaled with the lattices of the previous iteration.

Initialization step At the first step, FCA is applied on each Object-Attribute Context $K_i = (O_i, A_i, I_i)$ to produce a lattice L_{0i}. The output of this step is a Concept Lattice Family $\mathfrak{L}_0$.

Step n+1 At step n+1, from each context $R_j = (O_k, O_l, I_j, S_j)$ from R and the lattice L_{nl} we compute an Object-Attribute Context $C_{(n+1)j} = (O_k, \{j\} \times L_{nl}, J_{(n+1)j})$ where $J_{(n+1)j} = \{(o, (j, c)) | o \in O_k \land c \in L_{nl} \land S_j(o, extent(c), I_j)\}$. Then each context $K_i = (O_i, A_i, I_i)$ from K is extended to obtain $K_{(n+1)i} = (O_i, A_{(n+1)i}, I_{(n+1)i})$ where $A_{(n+1)i} = A_i \bigcup_{\{p,q | \exists R_q = (O_i, O_p, I_q, S_q) \in R\}} \{q\} \times L_{np}$ and $I_{(n+1)i} = I_i \bigcup_{\{q | \exists R_q = (O_i, O_p, I_q, S_q) \in R\}} J_{(n+1)q}$. The lattice $L_{(n+1)i}$ is then obtained by applying FCA on $K_{(n+1)i}$.

The process stops when an iteration does not add any new concept and we consider the last lattice family obtained as the output of the process.

We use RCA to classify: the source model elements, the target model elements and the transformation links. Which means that every one of them will be modelled as an Object-Attribute context in RCA. Those contexts will be linked by Object-Object contexts modelled after the following relations. Source and target model elements are classified using their metaclasses and relations. The transformation link classification relies on model element classifications and groups links that have similarities in their source and target ends: similar elements in similar contexts. From the transformation link classification, we derive a lattice of transformation patterns. Figure 2 shows an excerpt of the obtained pattern lattice for the transformation of UML class diagrams into relational models. The

transformation patterns are represented by rectangles, and are named with the prefix TPatt, the number of the transformation pattern, and then the number of the corresponding concept.

In each concept representing a transformation pattern, we have two types in two ellipses connected by a bold edge. The source ellipse of the bold edge represents the type T_s of the element to transform by the pattern. It can be seen as the main type of the premise. For instance, in Concept `TPatt_17-Concept_67`, we see that the pattern aims at transforming *generalizations* (note that in the UML meta-model, there is a meta-class named Generalization, which represents an inheritance relationship, and which is linked to two classes (in fact: classifiers, `Classifier` being a superclass of the meta-class `Class`) : the specific one and the general one). This main type of the premise is linked, with non-bold edges, to the environment that an element of type T_s must have in order to be transformed by the pattern. Those edges are named according to the relation-role names between the type T_s and its environment in the meta-model. Those edges also have a cardinality defining the cardinality of the environment. Such an environment corresponds to the rest of the premise. For instance, in Concept `TPatt_17-Concept_67`, `Generalization` is linked to a specific `Class` and a general `Class` with a cardinality [1..*], meaning that the Generalization must have a specific and a general classes. The target ellipse of the bold edge represents the main type T_t of the conclusion of the pattern, *i.e.*, a T_s will be transformed into a T_t (with a specific environment). For example, in the transformation pattern `TPatt_17-Concept_67`, the conclusion corresponds to a role, connected to a relationship, an entity, and zero or one cardinalities. Note that the conclusion of this pattern is quite long: this will be discussed in the next section.

4 Patterns lattices simplification

The obtained lattice of transformation patterns has to be filtered to keep only the useful/relevant patterns or pattern fragments.

First, the empty concepts are removed. They do not contain information about the transformation. They are present in the lattice to link other concepts (representing patterns). We only keep the Bottom and Top to maintain the order structure. For the same reason, when an empty concept is removed, its children are connected with the Top concept.

Secondly, we noticed that some patterns contain a deep premise or conclusion, *i.e.*, a long chain of linked objects. After observing many patterns of this type for many transformation problems, we found that after a certain depth, the linked elements are not useful. For instance, in the pattern `TPatt_17-Concept_67` in Figure 2, the important information is that a generalization linked to two classes (specific and general) must be transformed into a role linked to a relationship, an entity and a cardinality. The other elements are details specific to some examples, that are not relevant to the transformation. Starting from this observation, we implemented a simplification heuristic that prunes the premises and conclusions

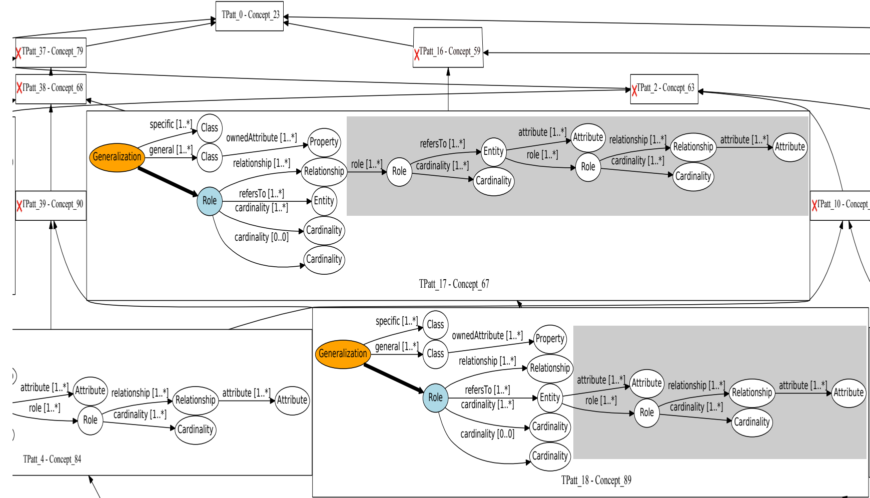

Fig. 2. An excerpt of the obtained hierarchy for the example UML class diagrams to entity-relationship model. Grey boxes indicate the part of the patterns that will be removed during the filtering phase, red crosses indicate the patterns that will be removed.

after the first level (key element and its immediate neighbors). In Figure 2, the grey boxes indicate the part of the patterns that will be removed.

After pruning the patterns according to the depth heuristic, some patterns could become identical. This is the case of patterns `TPatt_17-Concept_67` and `TPatt_18-Concept_89`). In Figure 2, we see the kept part of those two patterns (that is not in grey rectangles) is identical. For those redundant patterns, only the highest in the lattice is preserved, and all others removed. For removed concepts, their children are linked to their parents. Note that by doing so, we may lose the lattice structure.

5 Experimentation

In this section we experimentally compare transformation patterns obtained from distinct examples to transformation patterns obtained by the union of all examples. Our case study concerns the transformation of class diagrams into relational schema. The rule generation is performed starting from a set of 30 examples of class diagrams and their corresponding relational models. Some of them were taken from [7], the others were collected from different sources on the Internet. We ensured by manual inspection that all the examples conform to valid transformations. To take the best from the examples, a 3-fold cross validation was performed. We divide the j ($j \in 1..30$) examples into three groups of 10. For each fold i ($i \in 1..3$), we use two strategies to produce transformation rules:

- In the first one, we use the experimentation of [11] which consists of using two groups (20 examples) separately for generating 20 pattern lattices (denoted l_{ij}). The l_{ij} lattices are analyzed and simplified, as explained in Section 4, to select automatically the relevant transformation patterns. Then, we transform them into operational rules written for Jess. The remaining third group is used for testing them. Testing consists in executing the generated rules on the source models of the testing examples and in comparing the obtained target models with those provided in the examples.
- In the second one, we gather two groups (20 examples) for generating only one lattice of patterns (denoted L_i). L_i is analyzed and simplified to select automatically the relevant patterns. Those patterns are then transformed into operational rules. The remaining third group is used for testing them.

The goal is to compare in each fold i the results obtained from the two strategies. First, we compare the lattices generated from examples (lij), and the lattice generated from the union of those examples (Li). Then, we compare the results of executing the rules obtained from each strategy on the source models provided in the testing examples.

5.1 lij vs L_i

Compared to the first strategy, which produces small size lattices (from each lij we have about 9 patterns before simplification and 4 patterns after simplification), the second one produces large ones (from each L_i we have about 100 patterns before simplification and 50 patterns after simplification). Although the lattices L_i are larger and more difficult to analyze, they have more specific and complete transformation patterns compared to lij which are simple to analyze but contain transformation patterns that are proper to their examples. A single pattern of L_i can combine several patterns that exist in l_{ij}.

Figure 3 shows examples of different patterns obtained from l_{1j}. For instance, in the pattern of Fig. 3(a), a transformation link is given to specify that a *class* linked to an *aggregation* is mapped into a *table* linked to *primary foreign key.* Pattern of Fig. 3(b) shows that a class linked to a *property* is transformed into a table linked to a *column.* In the last pattern of Fig. 3(c), the transformation specifies that a class linked to a property and a *generalization* are transformed into a table linked to a column and a *foreign key.*

If we compare these patterns with the pattern of lattice L_1 in Figure 4, we note that the information contained in the three patterns exist in the pattern of Figure 4. It is more complete. It combines all the informations of transformation existing in Fig. 3(a), Fig. 3(b) and Fig. 3(c).

So, if we combine various examples together, the generated lattice contains patterns which are more specific and combine different information. But, if we test each example separately, the obtained lattice contains less information. In addition, L_i contains all the patterns needed to transform a class diagram to a relational schema. The lattices l_{ij} contain just the transformation pattern

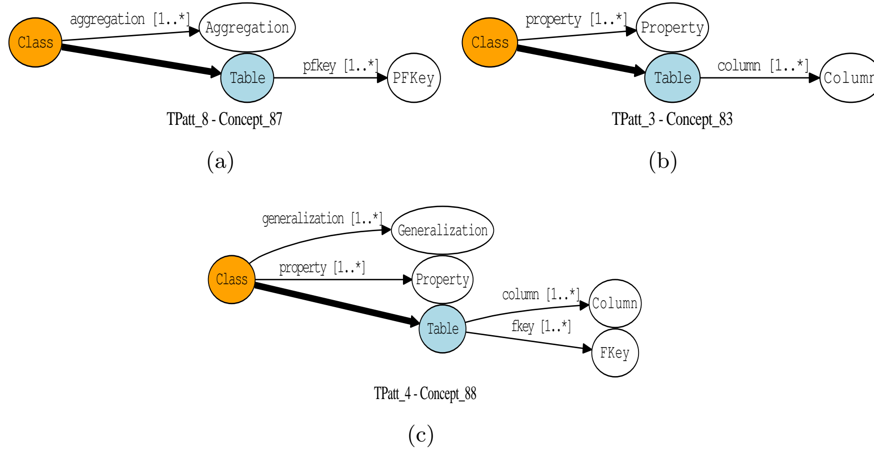

Fig. 3. Examples of transformation patterns extracted from lattices L_{l1}

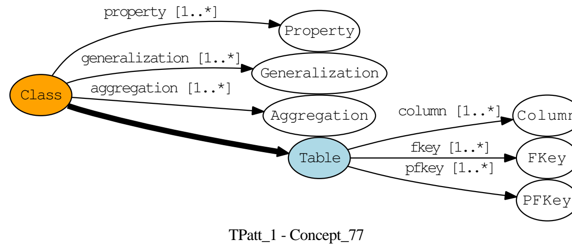

Fig. 4. Example of transformation pattern extracted from lattice L_1

proper to the transformation examples used. So, we need to merge several transformation examples to obtain all transformation rules of a class diagram into a relational schema.

Furthermore, in each fold, a L_i lattice contains about 50 transformation patterns and the union of l_{ij} produces about 40 ones (4 transformation patterns * 20 minus the redundant ones which exist). If we examine the patterns as an expert, we note that L_i contains about 12 relevant transformation patterns which are useful to the transformation. But they are less detailed and not applicable for all examples types. On the other side, the union of l_{ij} contains about 10 relevant transformation patterns. Those patterns are easy to read and to apply because each one contains a piece of information of the transformation compared to L_i'rules which combine several pieces of information in the same pattern.

5.2 lij's rules execution vs L_i's rules execution

In this section, we compare the result of executing the rules obtained from the two strategies, which are transformed into Jess rules, on the source models provided in the testing examples. This comparison allows calculating the recall (Equation 1) and the precision (Equation 2) measures for each T. T represents the type of elements in the target meta-model (table, column, foreign key...)

$$R(T) = \frac{number\ of\ T\ with\ correct\ transformation}{total\ number\ of\ initial\ T} \quad (1)$$

$$P(T) = \frac{number\ of\ T\ with\ correct\ transformation}{total\ number\ of\ generated\ T} \quad (2)$$

Table 1 shows precision and recall averages on all element types of the 10 generated transformations for the 3-folds. As mentioned in [11], the precision and recall averages are higher than 0.7 in the first strategy. Some models were perfectly transformed (precision=1 and recall=1). Precision and recall decrease in the case of elements which have more than one transformation possibility. For example, if we have a generalization between two classes, we can transform it into two tables or into a simple table which contains the attributes of general and specific classes. In this case, two rules are applied on the same example and this affects the performance results.

In the second strategy, precision and recall averages are low (less than 0.5) in the 3-folds. This is due to the fact that the generated rules are very large and contain different informations from different examples. Thus, the premises of the rules can not be matched for most of the examples because the examples are simple and do not contain all the transformation cases that have been learned. So, the Jess rule engine does not apply a part of the rule premise when it is executed on an example, it searches for each example its corresponding rule and this decreases the precision and the recall.

5.3 Discussion

The study presented in this section is a comparison of two strategies for generating transformation rules using RCA. The first consists to generate from each example its rule lattice and the second consists to gather all the examples and generate only one rule lattice. Each one has its advantages and disadvantages:

- The first strategy produces simple and small transformation patterns which are easy to analyze and to manipulate, but they are proper to their examples. On the other side, the second one produces larger patterns but they are more specific and more complete. They combine different information about the transformation. An analysis on those patterns shows that the two strategies have the same number of relevant patterns (about 12 transformation patterns). The relevant ones of the first strategy are simple and applicable for each example. On the contrary, the relevant patterns of the second strategy are larger and mainly applicable for larger examples.

Examples	Fold1			
	Recall Average		Precision Average	
	First Strategy	Second Strategy	First Strategy	Second Strategy
1	1	0.5	1	0.5
2	0.77	0.45	0.75	0.43
3	0.70	0.5	0.75	0.43
4	0.94	0.43	0.75	0.32
5	1	0.45	1	0.43
6	1	0.5	0.77	0.23
7	0.88	0.43	0.77	0.40
8	1	0.6	0.77	0.43
9	0.90	0.5	0.77	0.44
10	0.90	0.5	0.85	0.45

Examples	Fold2			
	Recall Average		Precision Average	
	First Strategy	Second Strategy	First Strategy	Second Strategy
1	0.78	0.5	0.79	0.4
2	0.90	0.45	0.75	0.31
3	0.85	0.45	0.77	0.43
4	0.77	0.43	0.79	0.40
5	1	0.5	0.80	0.34
6	1	0.43	0.77	0.47
7	0.85	0.4	0.77	0.37
8	0.85	0.45	0.80	0.43
9	1	0.5	0.75	0.34
10	1	0.5	0.80	0.33

Examples	Fold3			
	Recall Average		Precision Average	
	First Strategy	Second Strategy	First Strategy	Second Strategy
1	0.80	0.49	0.75	0.33
2	1	0.44	1	0.34
3	1	0.5	0.85	0.44
4	1	0.45	0.80	0.44
5	0.77	0.40	0.75	0.35
6	1	0.5	0.77	0.40
7	1	0.4	1	0.33
8	1	0.33	0.80	0.23
9	0.85	0.35	0.77	0.3
10	0.88	0.4	0.80	0.39

Table 1. Result of 3-fold cross validation

– The execution of the rules of the first strategy gives good results in our experiment. The rule engine searches and finds for each example the set of rules to apply. On the contrary the rules of the second strategy are more large and contain more information. Thus the rule engine does not find a rule to apply for the simple examples. We can work again on the obtained rules of L_i to execute them on all types of examples (for example by separating into smaller pieces), but as we found good results with the union of l_{ij}, it is not a promising track. We obtained a non-intuitive result: before the experimentation, we thought the best rules would be obtained with the second strategy.

Although the example used is a classical one, it is a good example of typical model transformations that we aim to learn. To confirm what is the best strategy to produce transformation rules, additional experiments have to be conducted with other model transformation kinds. Besides, the obtained result depends on the models on which we execute the rules. If we use larger models, the second strategy may have better results.

6 Related Work

Writing model transformations requires time and specific skills: the transformation developer needs to master the transformation language and both transformation source and target meta-models. Model Transformation by Example is a recent field of research that intends to use models as artifacts of development of the transformation.

Most of the research works consider all the examples at once. In [1], the authors use inductive logics programming to derive transformation rules, and although they consider an iterative process where examples are added to complete the derived transformation, they don't consider the examples independently. The same remark applies to [13], whose work uses the constraints explicitly applied by the transformation from the concrete syntax of a language to its abstract syntax and for [4] who proposes an algorithm to produce many to many transformation rules.

Another track in MTBE consists in using the analogy to perform transformations using examples [8,9,10]. The provided examples are decomposed into transformation blocks linking fragments of source models to fragments of target models. When a new source model has to be transformed, its elements are compared to those in the example source fragments to select the similar ones. Blocks corresponding to the selected fragments, coming from different examples, are composed to propose a suitable transformation. Fragment selection and composition are performed through a meta-heuristic algorithm. Compared to the above-mentioned approaches, the analogy-based MTBE does not produce rules. Here the examples are considered differently as they are added separately and not as a whole, influencing incrementally the system.

7 Conclusion

In this paper, we studied an approach for inferring model transformations (composed of transformation rules) from transformation examples. We compare two strategies for applying this approach: inferring the rules from the example taken separately (then gathering the rules), or inferring the rules from the gathering of the examples. Although we thought the second strategy would produce better rules (more detailed), it appeared that the rules (less detailed) produced by the first approach execute better. Future work includes learning rules whose premise and conclusion have several main elements and design heuristics to determine the best rule to apply when several rules are candidate.

References

1. Balogh, Z., Varró, D.: Model transformation by example using inductive logic programming. Software and Systems Modeling 8(3), 347–364 (2009)
2. Dolques, X., Huchard, M., Nebut, C.: From transformation traces to transformation rules: Assisting model driven engineering approach with formal concept analysis. In: Supplementary Proceedings of ICCS'09. pp. 15–29 (2009)
3. Ganter, B., Wille, R.: Formal Concept Analysis, Mathematical Foundations. Springer (1999)
4. 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. pp. 52–66 (2009)
5. Huchard, M., Hacène, M.R., Roume, C., Valtchev, P.: Relational concept discovery in structured datasets. Ann. Math. Artif. Intell. 49(1-4), 39–76 (2007)
6. Jess rule engine, http://herzberg.ca.sandia.gov/jess
7. Kessentini, M.: Transformation by Example. Ph.D. thesis, University of Montreal (2010)
8. Kessentini, M., Sahraoui, H., Boukadoum, M.: Model Transformation as an Optimization Problem. In: MODELS'08, LNCS 5301. pp. 159–173. Springer (2008)
9. Kessentini, M., Sahraoui, H., Boukadoum, M.: Méta-modélisation de la transformation de modèles par l'exemple : approche méta-heuristiques. In: Carré, B., Zendra, O. (eds.) LMO'09: Langages et Modèles à Objets. pp. 75–90. Cepaduès, Nancy (mars 2009)
10. Kessentini, M., Sahraoui, H., Boukadoum, M., Omar, B.O.: Model transformation by example : a search-based approach. Software and Systems Modeling Journal (2010), (To appear)
11. Saada, H., Dolques, X., Huchard, M., Nebut, C., Sahraoui, H.: Generation of operational transformation rules from examples of model transformations. In: to appear in proc. of MODELS'12 (Sept 2012)
12. Varró, D.: Model transformation by example. In: Proc. MODELS 2006, LNCS 4199. pp. 410–424. Springer (2006)
13. Wimmer, M., Strommer, M., Kargl, H., Kramler, G.: Towards model transformation generation by-example. In: HICSS. p. 285 (2007)