Generation of operational transformation rules from examples of model transformations

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. Model transformation by example (MTBE) aims at defining a model transformation according to a set of examples of this transformation. Examples are given in the form of pairs, each having an input model and its corresponding output transformed model, with the transformation traces. The transformation rules are then automatically extracted from the examples. In this paper, we propose a two-step approach to generate the transformation rules. In a first step, transformation patterns are learned from the examples through a classification of the model elements of the examples, and a classification of the transformation links using Formal Concept Analysis. In a second step, those transformation patterns are analysed in order to select the more pertinent ones and to transform them into operational transformation rules written for the Jess rule engine. The generated rules are then executed on examples to evaluate their relevance through classical precision/recall measures.

1 Introduction

Model Transformation is a key component of Model Driven Engineering (MDE). In model-driven development, the involved models are processed by programs as a matter of priority (rather than by hand). To ease the development of such programs handling models, several languages were introduced, e.g. graph transformation languages such as VIATRA [4], declarative or semi-declarative languages like ATL [3], or object-oriented and imperative languages such as Kermeta [25].

Implementing a model transformation requires two distinct skills: model-driven engineering skills (in particular, metamodeling and model-transformation environments), and domain-specific skills, *i.e.*, good knowledge about the specification of the transformation: the input domain, the output domain, and the transformation rules by themselves. While the first skills are possessed by model-driven engineering experts, the second ones are specific to domain experts. Experience shows that domain experts more easily give transformation examples than complete and consistent transformation rules [16]. In this context, Model Transformation By Example (MTBE) [28] has emerged as a convenient way to

let domain experts design transformations by giving an initial set of examples. An example consists of an input model, the corresponding transformed model, and fine-grained mappings between the constructs of both models. From those examples, an MTBE approach learns transformation rules. When those rules are operational, *i.e.*, they are written in a rule language disposing of a rule engine, they form the model transformation.

In this context, we present a Model Transformation By Example approach that goes from examples down to operational transformation rules. The learning mechanism used is based on Relational Concept Analysis (RCA) [12], a variant of Formal Concept Analysis [10]. It results in a hierarchy of non-operational rules called transformation patterns. Such transformation patterns are analyzed and filtered to derive the more relevant ones. The selected transformation patterns are then transformed into concrete and operational transformation rules that can be processed by the Jess rule engine [5]. The learning of the transformation patterns is a previous work from the authors [8], in this paper we introduce the filtering of the obtained transformation patterns, and we explain how to obtain operational rules from the transformation patterns. Finally, since the obtained rules are operational, experiments have been carried out on a case study in order to measure the relevance of the generated rules.

The remainder of this paper is structured as follows. We start by introducing the problem and describing our two-step approach in Section 2. Then, in Section 3, we briefly explain how RCA is used to extract information from examples and to generate transformation patterns. In this section, details are also given on how the obtained transformation patterns are filtered and refined. Section 4 describes the mapping of the transformation patterns into Jess rules. We present an evaluation of the approach and a discussion about the obtained results in Section 5. Section 6 presents the related work. Section 7 concludes the paper and describes future work.

2 Overview of the rules generation and execution

Model-Transformation By Example (MTBE) consists in learning transformation programs/rules from examples. Usually, an example is composed of a source model, the corresponding transformed model, and transformation links between those two models. To illustrate MTBE, let us consider the well-known case of transforming UML class diagrams into relational schemas, used, among others, in [19]. For this transformation, examples are given in the form of: an input UML model (such as the one given in Figure 1), the corresponding transformed relational model (such as the one given in Figure 2), and transformation links making explicit from which elements of the UML model, the elements of the relational model stem from. For instance, a transformation link is given to specify that class `Client` is mapped into table `Client`. A transformation link is equivalent to a link of an execution trace of the expected transformation, *i.e* two elements are related by a transformation link if the information contained in the first element is necessary to build the second one.

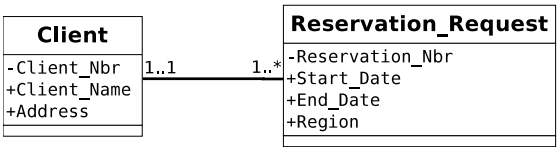

Fig. 1. Example for the UML2R transformation: input model

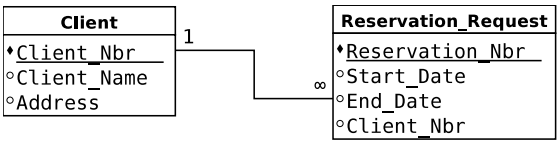

Fig. 2. Example for the UML2R transformation: transformed model

An MTBE process analyzes the examples and learns from them transformation rules such as *a class is transformed into a table*, or *a UML property linked to a class (i.e., an attribute and not a role) is transformed into a column of a table*. This process should produce operational rules, *i.e.*, rules that can be directly executed by a rule engine to transform any source model into a target model.

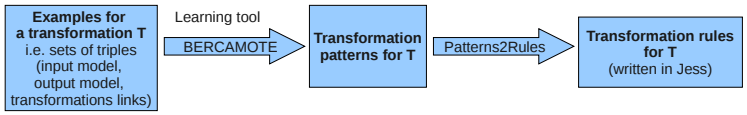

Fig. 3. A two-step approach for MTBE

We propose to generate the operational rules in a two-step approach, as illustrated in Figure 3. The first step is the analysis of examples, that learns transformation patterns using Relational Concept Analysis. This step is supported by the Bercamote tool, that has been introduced in [8]. Each obtained transformation pattern describes a premise in the form of an input model pattern (based on the input metamodel), and a conclusion, in the form of the output model pattern (based on the output metamodel) that should be obtained after the execution of the transformation. The transformation patterns are ordered in a hierarchy. This hierarchy is analyzed to select the more relevant patterns, and sometimes to select in a transformation pattern the more pertinent fragment. We here target model-to-model transformations in which both models represent the same data but in different languages or using different structural constraints *e.g.* a transformation applying design patterns to enforce good structural modeling practices in a language. On the contrary, our MTBE approach is not well-suited to learn transformations in which new values are computed *e.g.* we cannot learn a renaming policy that force to use lowercase for attributes names. Widening the

scope of the transformations that can be learned is possible but would impact on the complexity of the results and the efficiency of the approach.

The main contribution on this paper deals with the second step, that makes the patterns operational. This is done by transforming them into rules that can be executed by a rule engine. To make the transformation patterns operational, we have transformed them into Jess rules and executed them using the Jess Rule engine. This step is detailed in Section 4.

3 A by-example approach to obtain transformation patterns

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. This step has been presented in [8], and is summarized in the beginning of this section, whereas the end of this section is dedicated to the filtering of the obtained transformation patterns.

3.1 Obtaining the transformation patterns

To derive patterns from examples, a data analysis method is used, namely Formal Concept Analysis (FCA) [10] and its extension to relational data, the Relational Concept Analysis (RCA) [12]. 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, RCA computes several connected classifications.

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 transformation pattern hierarchy, *i.e.*, a lattice of patterns, where patterns are organized by inclusion. Fig. 4 shows an excerpt of the obtained pattern hierarchy for the transformation of UML class diagrams into relational models. It contains two transformation patterns (in the two inner boxes). The transformation pattern in the bottom box is more specific than the one in the top box, which is indicated by the inclusion edge between the two boxes. The patterns are automatically named by our tool, they have a prefix beginning by *TPatt* for *transformation pattern*, then we find the number of the pattern, and finally the number of the concept representing the pattern, as generated by our RCA/FCA algorithms.

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_2-Concept_57`, we see that the pattern aims at transforming *properties*. 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 metamodel. 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_2-Concept_57`, `Property` is linked to a `Class` with an edge named `property` and with a cardinality [1..*]. This means that the premise corresponds to a property, and that this property is linked to a class. 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_2-Concept_57`, the conclusion corresponds to a column, linked to a table, linked, in turn, to a primary key.

The transformation pattern `TPatt_2-Concept_57` has been deduced from a set of transformation links that were grouped together because they link a property (connected to a class) to a column (connected to a table, itself connected to a primary key). This pattern is included in the pattern of sub-concept `TPatt_5-Concept_70`. This latter is more specialized because in addition to link the table to a primary key, it also links it to a foreign key.

3.2 Patterns lattice simplification

After obtaining the lattice of transformation patterns, we select in this lattice the useful/relevant patterns or pattern fragments.

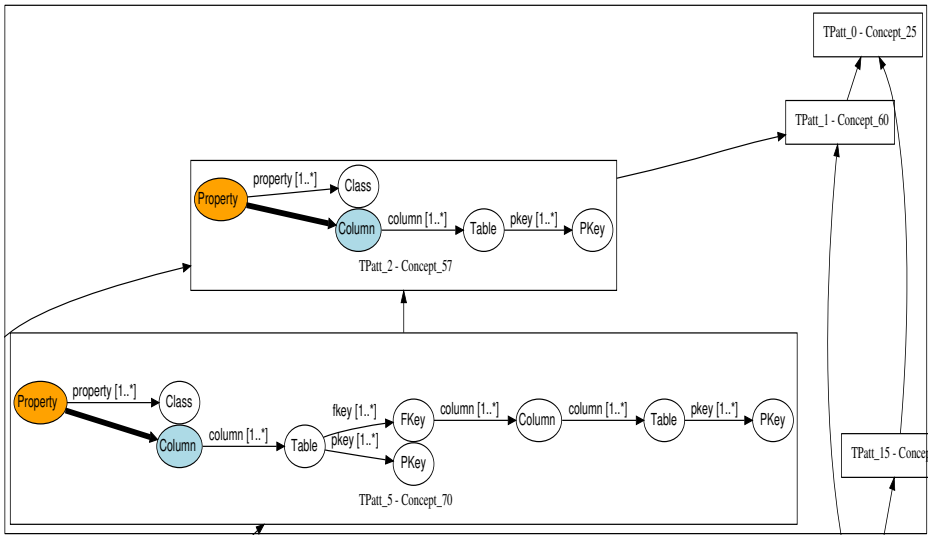

Fig. 4. An excerpt of the obtained hierarchy for the example UML class diagrams to relational models

In the lattice of Figure 4, for instance, concepts `TPatt_0-Concept_25` and `TPatt_1-Concept_60` are empty. They do not contain information about the transformation. They are present in the lattice to link other concepts (representing patterns) not shown in this excerpt. In the final transformation, those empty patterns are automatically removed from the lattice. When an empty concept is removed, we connect all its children with all its parents to keep the order structure of the lattice.

After the lattice pruning, the remaining patterns are analyzed for simplification purpose. 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, if we look at the pattern `TPatt_5-Concept_70` in Figure 4, the important information is that a property linked to a class must be transformed into a column linked to a table. 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 after the first level (key element and its immediate neighbors).

After pruning the patterns according to the depth heuristic, some patterns could become identical. This is the case of patterns `TPatt_2-Concept_57` and `TPatt_5-Concept_70`. For both, only `Property_Class` and `Column_Table` are kept respectively in the premise and conclusion. For redundant patterns, just the top ranked in the lattice is preserved, and all other are automatically removed. For removed concepts, their children are linked to their parents.

4 From transformation patterns to operational rules

This section describes the mapping of transformation patterns into operational rules that can be executed using a rule engine. The rule engine used in our project is the Java Expert System Shell (*Jess*) [13]. In sub-section 4.1, this engine is introduced. Then, in sub-section 4.2, the transformation of patterns into Jess rules is detailed.

4.1 Jess

Jess is a rule engine integrated in the Java platform. Java code can be referred by Jess code [5]. With Jess, we can create Java objects, implement Java interfaces, and call Java objects from its Java scripting environment. Despite this, Jess is mainly a declarative language.

A Jess program is usually composed of *facts* and *rules*. Facts encode data, while rules, activated by pattern matching, encode behavior. [13]. A rule contains conditions, called left-hand-side (LHS), and actions, called right-hand-side (RHS). When the condition part is satisfied, the action part is executed. Conditions mainly test the presence of facts, whereas actions produce facts. Syntactically, a Jess rule is written as follows:
IF< (fact1)(fact2)...(factN) > THEN <(action1)(action2)...(actionM)>

The following example describes a very simple Jess rule which displays the name of each person who has a name.

```
(defrule welcome
  (Person (firstname ?name))
 =>
  (printout t "Hello" ?name "!!!" crlf)
)
```

The conditions in LHS and facts conform to a *template*. A template in Jess is similar to a class in Java. It define a fact type. A template has a name and a set of slots. A fact, *i.e.* a template instance, has specific values for these slots. The example below shows the declaration of *Person* template:

```
(deftemplate Person (slot firstname))
```

This example declares a template named *Person* with a property *firstname*. To instantiate a person fact, we use the command *assert*:

```
(assert (Person (firstname Peter)))
```

4.2 Patterns to Jess rules transformation

In our context of model transformation, facts are model elements and templates are element types defined in the metamodel. A UML class diagram metamodel defines a set of templates such as `Class`, `Attribute`, and `Association`. A specific UML class diagram is described using facts that are instances of these templates such as, `Class Employee`, `Class Position`, and `Association has_position`. Fact `Class Employee` means that the model contains an element "Employee" which is an instance of the type "Class" in the metamodel.

Figure 5 illustrates the steps to follow in order to obtain operational rules from transformation patterns. The transformation process consists of three steps: Meta-model2Templates, Model2Fact, and TransformationPatterns2JessRules.

Meta-models2Templates Step 1 consists in generating templates from the meta-models. Each metaclass of the metamodel is transformed into a template with the same name. Each meta-attribute is also transformed into a *slot* keeping the same name. The type of the slot is the type of the meta-attribute. To facilitate the description of relations between the metaclasses, each meta-reference is also transformed into a template. Such a template has two slots respectively containing the name of the source element and the target element of the meta-reference. We suppose that the name of each element is its identifier.

Concretely, since we work with the EMF framework, this step corresponds to the following transformations:

- each *EClass* is transformed into a template with the same name,
- each *EAttribute* is transformed into a *slot* with the same name and whose type is the *EDataType* of the *EAttribute*,

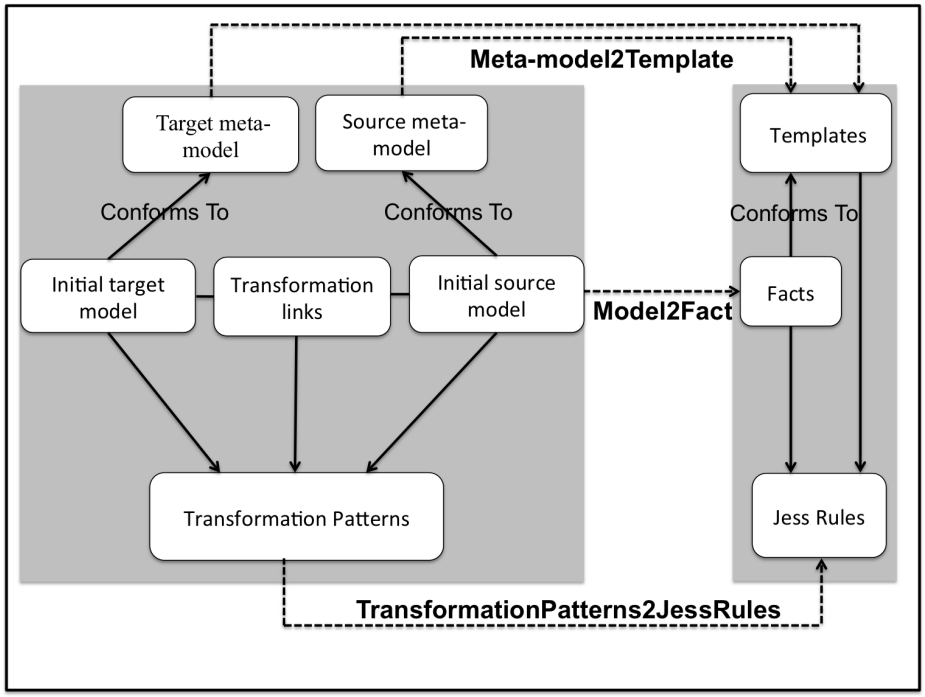

Fig. 5. Transformation Process

– each *EReference* is transformed into a template.

Figure 6 shows the transformation of a partial view of the relational schema meta-model. As indicated by the arrows, the *EClasses* table and column are transformed into templates. The *EAttribute* name is also converted to slot in each template. The *EReference* between table and column is transformed to a template which contains two slots containing the names of source and target elements of the *Ereference*.

Models2Facts Step 2 aims at transformaing models into facts. A model is an instantiation of its meta-model. Accordingly, each instance of a meta-class present in the model is transformed into a fact the same name. The instances of meta-attributes are transformed into slot values of the corresponding template. Each instance of meta-reference between two instances of meta-classes is also transformed into a fact which contains the names of relation elements.

A simple transformation example is presented in Figure 7. The three instances of metaclasses (the table and the two columns) are transformed into three facts. The two instances of meta-relations (from table to column) are transformed into the two facts instanciating the template *RelTabCol*.

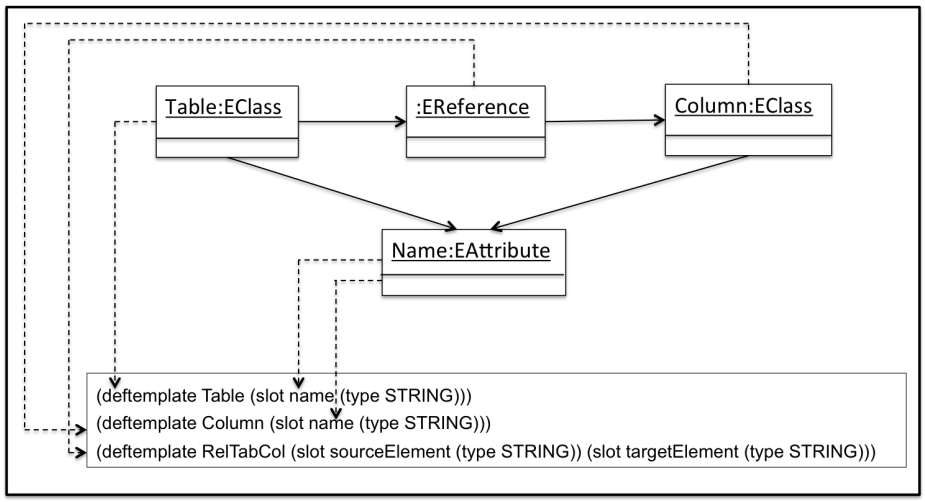

Fig. 6. Transformation of an extract of relational meta-model to Jess

TransformationPatterns2JessRules Step 3 consists in the actual rule generation from transformation patterns. As it can be seen in Figure 8, there is a similarity between transformation-pattern structure and Jess-rule structure. Both of them are composed of two main parts. The premise of a pattern is equivalent to the LHS of a rule. Both describe the situation to find to fire the rule or to apply the transformation pattern. Similarly, the conclusion is equivalent to the RHS. Both are the action to perform or the conclusion to reach when the first part is satisfied.

The premise is a description of a set of source elements. These elements are linked together. Consequently, each element in the premise is transformed into a Jess condition corresponding to the test of the presence of a fact. As the premise elements are not named, we generate a slot name for each element. When more than one element are involved, conditions corresponding to relations are also generated. As relations do not have names, we named it by concatenating the three first letters of the relation elements names.

The conclusion of a transformation pattern is a description of a set of target elements together with their relations. It it similar to the premise. Consequently, each element in the conclusion is transformed into a Jess fact assertion. Names and relations between facts are also generated.

Figure 8 shows the transformation into a Jess rule of an example of transformation pattern. The premise of the transformation pattern is a class linked to a property. The corresponding Jess rule has for LHS four conditions, respectively checking: the existence of a class i, the existence of a property j, the existence of a relation from class to property, and that the existing relation links i to j. The conclusion of the transformation pattern is a table linked to a column.

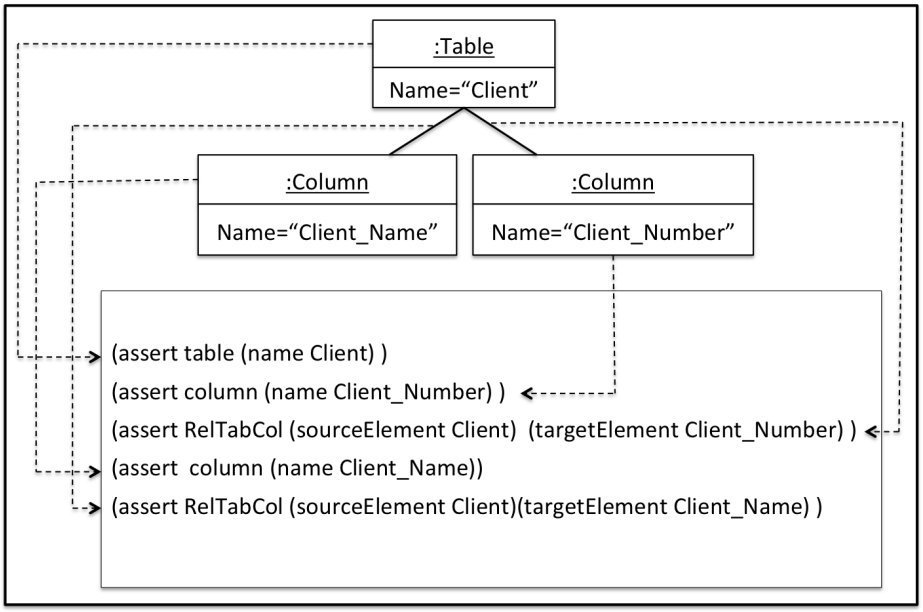

Fig. 7. Transformation of a partial view of relational schema model to Jess

The corresponding RHS of the generated Jess rule contains three fact assertions, respectively stating: a table i, a column j, and a relation from i to j.

5 Case study

This section illustrates the rule generation process using a case study. It also reports on the efficiency of our approach through classical precision/recall measures. Like for testing, we compare the target models produced by our executable rules with the expected models. Precision and recall show to what extent the inferred rules perform the correct transformations.

Our case study concerns the transformation of class diagrams into relational schemas. The rule generation is performed starting from a set of 30 examples of class diagrams and their corresponding relational schemas. Some of them were taken from [16], 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, *i.e.*, 30 examples divided into three groups of 10. For each fold, two groups (20 examples) were used for generating the rules, and the remaining third group was used for testing them. Each fold used a different group for testing. 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

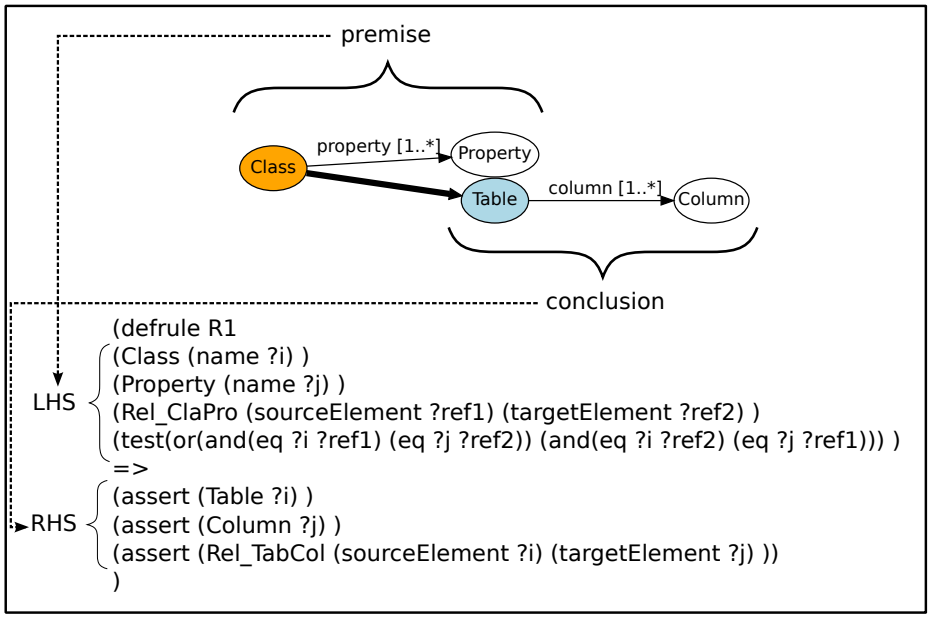

Fig. 8. Example of the transformation of a pattern into Jess

the examples. This comparison allows calculating the precision (Equation 1) and the recall (Equation 2) measures.

We calculate precision and recall separately for each type T of fact (table, column, etc.).

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

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

Table 1 shows precision and recall averages (on all fact types) of the 10 generated transformations for the 3-folds. The precision and recall averages are higher than 0,70 in all cases. Some models were perfectly transformed (precision=1 and recall=1). For the others, the precision and recall could be better than the ones calculated automatically. This is due to 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 a simple table which contains the attributes of general and specific classes. The second transformation method is to transform it into two tables. So, in the case of generalization, two rules are applied and this decreases the precision and the recall. The same problem exists for the aggregation which has also two transformation possibilities (1 or 2 tables).

Examples	Fold1		Examples	Fold2	
	Precision Average	Recall Average		Precision Average	Recall Average
1	1	1	1	0,78	0,79
2	0,77	0,75	2	0,90	0,75
3	0,70	0,75	3	0,85	0,77
4	0,94	0,75	4	0,77	0,79
5	1	1	5	1	0,80
6	1	0,77	6	1	0,77
7	0,88	0,77	7	0,85	0,77
8	1	0,77	8	0,85	0,80
9	0,90	0,77	9	1	0,75
10	0,90	0,85	10	1	0,80

Examples	Fold3	
	Precision Average	Recall Average
1	0,80	0,75
2	1	1
3	1	0,85
4	1	0,80
5	0,77	0,75
6	1	0,77
7	1	1
8	1	0,80
9	0,85	0,77
10	0,88	0,80

Table 1. Result of 3-fold cross validation

Discussion

The study presented in this section is a first evaluation of our approach. This evaluation is a proof-of-concept to check if RCA-based derivation and pattern-to-rule mapping are effective. In this context, the obtained results are very satisfactory. They show that the proposed approach allows to find most of the expected transformation rules and that these rules are executable on actual models.

To help us improving the rule generation process, additional experiments have to be conducted, in particular to study the two following issues:

- First, we used a small number of examples, based on small meta-models. Larger meta-models and more numerous examples have to be considered in the future to draw a better portrait on the strengths and weaknesses of the approach.
- Second, we measured the correctness of the obtained model transformation by comparing elements of the produced and expected models without considering their relations. A better and comprehensive correctness measure should be defined in the future.

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. To our best knowledge, two main tracks have been explored to assist the process of developing a model transformation: using only source and target meta-models linked by the transformation, or using transformation examples.

A first approach is based on meta-model alignment and is inspired by research on ontology alignment and schema alignment. Transformation patterns are then deduced from this alignment. Lopes et al. [22,21] define a two-step process: the alignment algorithm SAMT4MDE computes alignments using a similarity metric on elements with the same type (classes, enumerations, etc.), then the tool MT4MDE generates a model transformation skeleton in ATL language [14]. Del Fabro et Valduriez [6] generate a transformation as a post-processing of a *weaving model*. This weaving model is built using a similarity metric between the elements and propagating similarities thanks to the Similarity Flooding algorithm [23]. Falleri et al. [9] study several configurations for applying Similarity Flooding algorithm in the context of meta-model alignment with the aim of determining which configurations work best. Kappel et al. [15] transpose their meta-models into ontologies and apply COMA++ tool [1]. Alignments on ontologies are brought back to the meta-models.

Meta-model alignment is especially relevant when the source and target meta-models are semantically and structurally closed, *e.g.* when the transformation aims at migrating models from one meta-model version to another, but is inefficient on complex cases. When it can be applied, meta-model alignment reduces significantly the time of the development. Other approaches (MTBE for Model

Transformation Based Example) take advantage of transformation examples to learn transformations in more complex cases. One of their strengths is that transformation examples, written in the concrete syntax, are easier to manipulate than meta-models and their creation can be deferred to domain experts who don't need any programming skill.

The MTBE approach has been initiated by Varró [28]. An alignment between representative source and target example models is manually created. Transformation links are annotated by the transformation rule they illustrate (*e.g. ClassToEntity*). Transformation rules are derived from the transformation links and refined by the developer. Rules are validated on new source and target example models. If they are not satisfactory, the process iterates. The proposal of [28] was extended in [2], by using inductive logics programming (ILP [24]) to derive the transformation rules. ILP is a machine learning technique which derives a logic program from existing knowledge (source and target models), positive examples (pairs of model elements connected by transformation links) and negative examples (pairs of model elements that are not connected by transformation links). Considering only the immediate neighbors of each transformation-link end, the ILP engine infers an hypothesis for each transformation rule.

Wimmer et al. [29] propose a similar work but derive ATL transformation rules from examples written in concrete syntax by taking advantage of the constraints explicitly applied by the transformation from the concrete syntax of a language to its abstract syntax. The main advantage of this solution is to be able to use the concrete syntax to define models and transformation links. However, model editors need to be written in a way that permits to extract constraints and to edit transformation links.

Contributions [2] and [29] generate abstract rules and not executable ones. Although abstract rules could be individually correct, they are not a full-edged transformation program. These rules represent fragments of knowledge and must be arranged in a non-trivial way to perform the actual transformation (execution control). Furthermore, concrete rule languages and engines have their own constraints, which make the implementation of abstract rules not straightforward. In this paper, we produce executable rules and test them on real cases.

The work of Garcia-Magarino et al. [11] is also considered as a variant of MTBE approaches. In their approach, the authors generate transformation rules from meta-models which satisfy some developer constraints.

Another MTBE approach [7,8] uses an extension of the anchorPrompt approach [26] to assist the transformation link discovery, and Relational Concept Analysis to derive commonalities between the source and target meta-models, models and transformation links. Compared to the ILP-based proposal, the RCA-based approach does not use annotations on transformation links and propose a set of transformation patterns organized in a lattice. However, the transformation patterns cannot be directly executed, and this paper proposes to translate them into JESS rules to provide consistency checking and executability.

Model Transformation By Demonstration (MTBD) [20,27], is a similar approach to MTBE. Through direct editing of the source model, users are asked

to demonstrate how the model transformation should be done. The recorded actions are then generalized to produce transformation patterns.

Another track in MTBE consists in using the analogy to perform transformations using examples [17,18,19]. 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. This could be considered as a limitation if the goal is to infer reusable knowledge about transformations.

7 Conclusion

In this paper, we presented an approach that aims at deriving model transformation rules from a set of model transformation examples. A first step of the approach uses a data analysis method, RCA, to learn recurrent transformation patterns. In the second step, the transformation patterns are filtered, refined, and automatically transformed into Jess rules. Those rules constitute the expected transformation. Provided that meta-models and models are written as Jess facts (which is done by automatic transformation), the rules can be executed by the Jess engine to actually transform models. The approach is successfully evaluated on a case study used in previous research work.

Future work includes transforming the obtained Jess facts (after rule application) to produce models conforming to the initial meta-models. Furthermore, we plan to work on rule execution control to select the rule to apply when we have more than one rule for the same source element.

References

1. Aumueller, D., Do, H.H., Massmann, S., Rahm, E.: Schema and ontology matching with coma++. In: Özcan, F. (ed.) SIGMOD Conference. pp. 906–908. ACM (2005)
2. Balogh, Z., Varró, D.: Model transformation by example using inductive logic programming. Software and Systems Modeling 8(3), 347–364 (2009)
3. Bézivin, J., Dupé, G., Jouault, F., Pitette, G., Rougui, J.E.: First experiments with the atl model transformation language: Transforming xslt into xquery. In: OOPSLA 2003 Workshop (2003)
4. Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D.: Viatra: Visual automated transformations for formal verification and validation of uml models. In: Proceedings of the 17th IEEE international conference on Automated software engineering. IEEE Computer Society (2002)
5. Daniele, L.M.: Towards a Rule-based Approach for Context-Aware Applications. Ph.D. thesis, University of Twente The Netherlands (May 2006)

6. Del Fabro, M.D., Valduriez, P.: Semi-automatic model integration using matching transformation and weaving models. In: International Conference SAC'07. pp. 963–970. ACM (2007)
7. Dolques, X., Dogui, A., Falleri, J.R., Huchard, M., Nebut, C., Pfister, F.: Easing model transformation learning with automatically aligned examples. In: 7th European Conference, ECMFA 2011. pp. 189–204 (2011), http://www.ecmfa-2011.org/
8. 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)
9. Falleri, J.R., Huchard, M., Lafourcade, M., Nebut, C.: Meta-model Matching for Automatic Model Transformation Generation. In: MODELS'08, LNCS 5301. pp. 326–340. Springer (2008)
10. Ganter, B., Wille, R.: Formal Concept Analysis, Mathematical Foundations. Springer (1999)
11. 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)
12. 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)
13. Jess rule engine, http://herzberg.ca.sandia.gov/jess
14. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.M. (ed.) MoDELS Satellite Events. pp. 128–138. Springer (2005)
15. 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: Proceedings of MoDELS 2006. pp. 528–542 (2006)
16. Kessentini, M.: Transformation by Example. Ph.D. thesis, University of Montreal (2010)
17. Kessentini, M., Sahraoui, H., Boukadoum, M.: Model Transformation as an Optimization Problem. In: MODELS'08, LNCS 5301. pp. 159–173. Springer (2008)
18. 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)
19. Kessentini, M., Sahraoui, H., Boukadoum, M., Ben Omar, O.: Model transformation by example : a search-based approach. Software and Systems Modeling Journal (2010), (à paraître)
20. Langer, P., Wimmer, M., Kappel, G.: Model-to-model transformations by demonstration. In: ICMT. pp. 153–167 (2010)
21. Lopes, D., Hammoudi, S., Abdelouahab, Z.: Schema matching in the context of model driven engineering: From theory to practice. In: Sobh, T., Elleithy, K. (eds.) Advances in Systems, Computing Sciences and Software Engineering. pp. 219–227. Springer (2006)
22. Lopes, D., Hammoudi, S., Bézivin, J., Jouault, F.: Generating transformation definition from mapping specification: Application to web service platform. In: CAiSE'05, LNCS 3520. pp. 309–325 (2005)
23. Melnik, S., Garcia-Molina, H., Rahm, E.: Similarity flooding : A versatile graph matching algorithm and its application to schema matching. In: ICDE. pp. 117–128. IEEE Computer Society (2002)

24. Muggleton, S., De Raedt, L.: Inductive logic programming: Theory and methods. Journal of Logic Programming 19/20, 629–679 (1994)
25. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In: Briand, L., Kent, S. (eds.) Proceedings of MODELS/UML'2005 (2005)
26. Noy, N.F., Musen, M.A.: Anchor-prompt: Using non-local context for semantic matching. In: Proc. of the Workshop on Ontologies and Information Sharing at IJCAI-2001. pp. 63–70. Seattle (USA) (2001)
27. Sun, Y., White, J., Gray, J.: Model transformation by demonstration. In: MoDELS. pp. 712–726 (2009)
28. Varró, D.: Model transformation by example. In: Proc. MODELS 2006, LNCS 4199. pp. 410–424. Springer (2006)
29. Wimmer, M., Strommer, M., Kargl, H., Kramler, G.: Towards model transformation generation by-example. In: HICSS. p. 285 (2007)