Softw Syst Model (2009) 8:347–364
DOI 10.1007/s10270-008-0092-1

SPECIAL SECTION PAPER

Model transformation by example using inductive logic programming

Zoltán Balogh · Dániel Varró

Received: 11 July 2007 / Revised: 8 April 2008 / Accepted: 13 May 2008 / Published online: 13 August 2008

Abstract Model transformation by example is a novel approach in model-driven software engineering to derive model transformation rules from an initial prototypical set of interrelated source and target models, which describe critical cases of the model transformation problem in a purely declarative way. In the current paper, we automate this approach using inductive logic programming (Muggleton and Raedt in J Logic Program 19-20:629–679, 1994) which aims at the inductive construction of first-order clausal theories from examples and background knowledge.

Keywords Model transformation · By-example synthesis · Inductive logic programming

1 Introduction

The efficient design of automated model transformations between modeling languages has become a major challenge to model-driven engineering (MDE) by now. Many highly expressive transformation languages and efficient model transformation tools are emerging to support this problem. The evolution trend of model transformation languages is characterized by gradually increasing the abstraction level of such languages to declarative, rule-based formalisms as promoted by the QVT (Queries, Views and Transformations) [23] standard of the OMG.

However, a common deficiency of all these languages and tools is that their transformation language is substantially different from the source and target models they transform. As a consequence, transformation designers need to understand not only the transformation problem, i.e. how to map source models to target models, but significant knowledge is required in the transformation language itself to formalize the solution. Unfortunately, many domain experts, who are specialized in the source and target languages, lack such skills in underlying transformation technologies.

Model transformation by example (MTBE) is a novel approach (first introduced in [34]) to bridge this conceptual gap in transformation design. The essence of the approach is to derive model transformation rules from an initial prototypical set of interrelated source and target models, which describe critical cases of the model transformation problem in a purely declarative way. A main advantage of the approach is that transformation designers use the concepts of the source and target modeling languages for the specification of the transformation, while the implementation, i.e. the actual model transformation rules are generated (semi-) automatically. In our context, (semi-)automatic rule generation means that transformation designers give hints how source and target models can *potentially* be interconnected in the form of a mapping metamodel. Then the actual contextual conditions used in the transformation rules are derived automatically based upon the prototypical source and target model pairs.

The current paper proposes the use of inductive logic programming [22] (ILP) to automate the model transformation

Communicated by Dr. Jean Bezivin.

This work was partially supported by the SENSORIA European project (IST-3-016004). Dániel Varró was also supported by the J. Bolyai Scholarship.

Z. Balogh · D. Varró (✉)
Department of Measurement and Information Systems,
Budapest University of Technology and Economics,
Magyar tudósok krt. 2, 1117 Budapest, Hungary
e-mail: varro@mit.bme.hu

Z. Balogh · D. Varró
OptXware Research and Development Ltd,
Budapest, Hungary
e-mail: zoltan.balogh@optxware.hu

by example approach. ILP can be defined as an intersection of inductive learning and logic programming as it aims at the inductive construction of first-order clausal theories from examples and background knowledge, thus using induction instead of deduction as the basic mode of inference.

As the main practical advantage of this interpretation, we demonstrate that by using existing ILP tools, we can achieve a higher level of automation for MTBE compared to our initial experiences in [34] using relatively small examples.

The current paper extends [34] by providing a systematic approach based on first-order logic foundations instead of an intuitive solution. Compared to [35], the current paper provides significantly more details on the core approach and main algorithms (in Sect. 6). Furthermore, we discuss certain limitations that were revealed by a complex case study for our MTBE approach. Finally, we also sketch a prototype tool chain that has been implemented as an initial tool support for MTBE.

The rest of this paper is structured as follows. In Sect. 2, we present a running example which will be used throughout the paper. Section 3 provides a brief introduction to inductive logic programming. Section 4 gives an overview of the core concepts of the MTBE approach. Section 5 summarizes the inputs required for MTBE. In Sect. 6, which is the central part of the current paper, we give a detailed presentation on how to automate MTBE using inductive logic programming. In Sect. 7, we collected some known limitations of our approach, while Sect. 8 sketches a prototypical tool chain that we used for carrying out our case studies. Finally, Sect. 9 discusses related work and Sect. 10 concludes our paper.

2 Running example: object-relational mapping

As the running example of the current paper, we use a standard object-relational mapping problem where UML class diagrams are mapped into relational database tables. We expect that this case study is relatively simple and well-known to the reader, which allows us to better concentrate on the technicalities of our approach, and enables an easier comparison with existing rule-based solutions to the same problem.

In addition to this simple case study, we also fully investigated a complex model transformation aiming to analyze UML statecharts by Petri net techniques [14]. While we gained significant insight especially in the limitations of MTBE by this transformation, we regard it too complex and technical to demonstrate our approach. Finally, we decided to present the identified limitations (Sect. 7) within the object-relational domain to keep the current paper more focused.

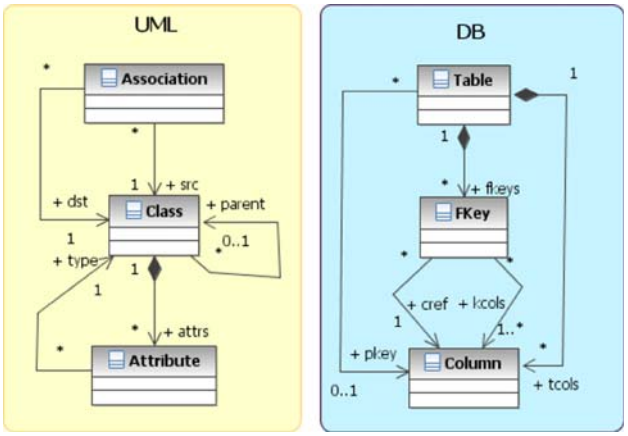

Fig. 1 Source and target metamodels of the example

Source and target metamodels

The source and target languages (UML and relational databases, respectively) are captured by their respective metamodels in Fig. 1. To avoid mixing the notions of UML class diagrams and metamodels, we will refer to the concepts of the metamodel using nodes and edges (more precisely, node types and edge types) for classes and associations, respectively. Since model transformation rules will be captured by graph transformation [28], this terminology is compliant with the notions of our model transformation domain.

UML class diagrams consist of class nodes arranged into an inheritance hierarchy (by *parent* edges). Classes contain attribute nodes (*attrs*), which are typed over classes (*type*). Directed edges are leading from a source (*src*) class to a destination (*dst*) class.

Relational databases consist of table nodes, which are composed of column nodes by *tcols* edges. Each table has a single primary key column (*pkey*). Foreign key (*FKey*) constraints can be assigned to tables (*fkeys*). Such a key refers to columns (*cref*) of another table, and it is related to the columns of local referring table by *kcols* edges.

Informal transformation rules

The main guidelines of (this variant of) the object-relational mapping can be summarized as follows:

- Each top-level UML class (i.e. a top-most class in the inheritance tree) is projected into a database table. Two additional columns are derived automatically for each top-level class: one for storing a unique identifier (primary key), and one for storing the type information of instances.
- Each attribute of a UML class will appear as a column in the table related to the top-level ancestor of the class. For the sake of simplicity, the type of an attribute is

restricted to user-defined classes. The structural consistency of storing only valid object instances in columns is maintained by foreign key constraints.
– Each UML association is projected into a table with two columns pointing to the tables related to the source and the target classes of the association by foreign key constraints.

Obviously, our aim is to semi-automatically derive transformation rules in correspondance with the informal guidelines above by only relying on a prototypical set of interrelated source and target models serving as the specification of the transformation.

3 An overview of inductive logic programming

Inductive logic programming [22] (ILP) aims at inductively constructing first-order clausal hypotheses from examples and background knowledge. In case of ILP, induction is typically interpreted as abduction combined with justification. *Abduction* is the process of hypothesis formation from some facts while *justification* (or confirmation) denotes the degree of belief in a certain hypothesis given a certain amount of evidence.

Formally, the problem of inductive inference can be defined as follows. Given some (a priori) background knowledge B together with a set of positive facts E^+ and negative facts E^-, find a hypothesis H such that the following conditions hold:

Prior satisfiability. All $e \in E^-$ are false in $\mathcal{M}^+(B)$ where $\mathcal{M}^+(B)$ denotes the minimal Herbrand model of B (denoted as $B \wedge E^- \not\models \top$).
Posterior satisfiability (consistency). All $e \in E^-$ are false in $\mathcal{M}^+(B \wedge H)$ (denoted as $B \wedge H \wedge E^- \not\models \top$).
Prior necessity. E^+ is not a consequence of B, i.e. some $e \in E^+$ are false in $\mathcal{M}^+(B)$ (denoted as $B \not\models E^+$).
Posterior sufficiency (completeness). E^+ is a consequence of B and H, i.e. all $e \in E^-$ are true in $\mathcal{M}^+(B \wedge H)$ (denoted as $B \wedge H \models E^+$).

A generic ILP algorithm [22] keeps track of a queue of candidate hypotheses. It repeatedly selects (and removes) a hypothesis from the queue, and expands that hypothesis using inference rules. The expanded hypothesis is then added to the candidate queue, which may be pruned to discard unpromising hypotheses from further consideration.

Many existing ILP implementations like Aleph [33] that we used for our experiments are closely related to Prolog, and the following restrictions are quite typical:

– B is restricted to definite clauses where the conjunction of (positive or negative) body clauses implies the head, formally

$$Head :- Body_1, Body_2, \ldots Body_n$$

– E^+ *and* E^- *are restricted to ground facts.*

Further language and search restrictions can be defined in Aleph by using mode, type and determination declarations. Moreover, we can also ask in Aleph to find all negative constraints, i.e. Prolog clauses of the form *false :-* $Body_1, Body_2, \ldots, Body_n$. More details on negative constraints will be given in Sect. 6.3.

As a demonstrative example, let us consider some traditional family relationship. The background knowledge B may contain the following clauses:

```
grandparent(X,Y) :- father(X,Z),
                    parent(Z,Y).
father(george,mary).
mother(mary,daniel).
mother(mary,greg).
```

Some positive examples E^+ can be given as follows:

```
grandfather(george,daniel).
grandfather(george,greg).
```

Finally, some negative facts E^- are also listed:

```
grandfather(daniel,george).
grandfather(mary,daniel).
```

Believing B, and faced with the facts E^+ and E^-, Aleph is able to set up the following hypothesis H by default.

```
grandfather(X,Y) :- father(X,Z),
                    mother(Z,Y).
```

By default settings, Aleph will set up a hypothesis only for clauses used in the positive and negative examples. However, there is some (limited) support for abductive learning [1] when Aleph will be able to derive hypothesis not only for the clause covered by positive and negative examples:

```
parent(X,Y) :- mother(X,Y).
```

Our results presented in the current paper rely exclusively on Aleph, which is considered to be a powerful inductive logic programming engine. In the future, we plan to investigate alternate ILP engines such as Progol [25], for instance to reveal if tool-specific limitations can be overcome this way.

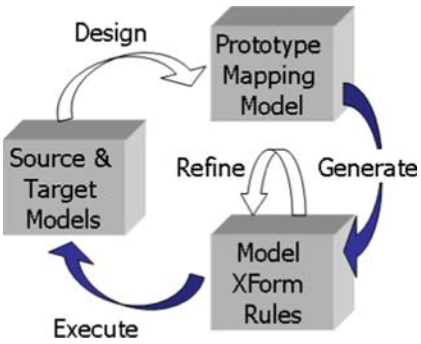

Fig. 2 Model transformation by example: process overview

4 Model transformation by example (MTBE)

4.1 Overview of model transformation by example

Model transformations by example (MTBE) [34] is defined as a highly iterative and interactive process as illustrated in Fig. 2.

Step 1: Set-up of prototype mapping models. The transformation designer assembles an initial set of interrelated source and target model pairs, which are called *prototype mapping models* in the rest of the paper. These prototype mapping models typically capture critical situations of the transformation problem by showing how the source and target model elements should be interrelated by appropriate mapping constructs.

Step 2: Automated derivation of rules. Based on the prototype mapping models, the MTBE framework should synthesize a set of model transformation rules, which correctly transform as many prototypical source models into their target equivalents as possible.

Step 3: Manual refinement of rules. The transformation designer can refine the rules manually at any time by adding attribute conditions or providing generalizations of existing rules.

Step 4: Automated execution of rules. The transformation designer validates the correctness of the synthesized rules by executing them on additional source–target model pairs as test cases, which will serve as additional prototype mapping models. Then the development process is started all over again.

The main vision of the "model transformations by example" approach is that the transformation designer mainly uses the concepts of the source and target languages as the "transformation language", which is very intuitive. He or she does not need to learn a new formalism for capturing model transformations.

While the current paper focuses on automating the "model transformation by example" approach, we still regard MTBE as a highly iterative and interactive process. Our experience also shows that it is very rare that the *final* set of transformation rules is derived right from the *initial* set of prototype models. Furthermore, transformation designer can overrule the automatically generated rules at any time, especially, when certain critical abstractions or generalizations are not detected automatically.

Concerning correctness issues, one would expect as a minimum requirement that the derived model transformation rules should correctly transform all prototypical source models into their target equivalent. However, this is not always practical, since overspecification or incorrect specification in prototype mapping models may decrease the chance of deriving a meaningful set of model transformation rules. Since MTBE takes prototype mapping models as specifications (unintended) omissions in them might easily result in incorrect rules. Therefore, MTBE approaches should ideally tolerate a certain amount of "noise" when processing prototype mapping models.

4.2 Steps of automation

From a technical point of view, the process of model transformation by example can be split into the following phases to support the semi-automatic generation of transformation rules:

1. *Set up an initial prototype mapping model.* In the first step, an initial prototype mapping model is set up manually from scratch or by using existing source and target models.
2. *Context analysis.* Then we identify (positive and negative) constraints in the source and target models for the presence of each mapping node. For instance, only top-level classes are related to database tables or a table related to a class always contains a primary key column. For this purpose, we first examine the contexts of all mapped source and target nodes.
3. *Connectivity analysis.* For each edge in the target metamodel, we identify contextual conditions (in the source and mapping models) for the existence of that target edge.
4. *Derive transformation rules for target nodes.* Then we derive transformation rules for all (types of) mapping nodes that derive only target nodes using the information derived during context analysis. Informally, the context of source nodes will identify the precondition of the derived model transformation rules, while the context of target nodes will define the postcondition of model transformation rules. As a result, we create target nodes from source nodes interconnected by a mapping structure (of some type).

5. *Derive transformation rules for target edges*. Finally, we derive transformation rules for each target edge based upon the information gained during connectivity analysis of source and target elements.
6. *Iterative refinement.* The derived rules can be refined at any time by extending the prototype mapping model or manually generalizing the automatically generated rules.

In the current paper, we discuss how the MTBE approach can be automated by using inductive logic programming [22] as an underlying framework. Our goal is to show that it is possible to construct relatively small prototype mapping models for practical problems from which the complete set of model transformation rules can be derived semi-automatically. Furthermore, we also identify critical transformation problems where our approach failed to derive a complete solution.

5 Inputs of model transformation by example

Model transformation by example takes a set of prototypical interconnected source and target model pairs as inputs. We now discuss how such interconnections can be described by mapping metamodels (Sect. 5.1), and how prototype mapping models are defined based upon this metamodel (Sect. 5.2). We also collect our assumptions for the structure of prototype mapping models that we rely upon for automation (Sect. 5.3).

5.1 Mapping metamodel

In many model transformation approaches, source and target metamodels (of Fig. 1) are complemented with another metamodel, which specifies the interconnections allowed between elements of the source and target languages. This metamodel has various names. In triple graph grammars [29] it is called a correspondence metamodel, while it is also called a weaving (meta)model in [4,8]. In previous works of the authors, the term reference metamodel was used [35], which unfortunately coincides with reference models introduced in [16]. In order to avoid unintended clashes in terminology, in the current paper, we refer to this metamodel as *mapping metamodel* (see Fig. 3). Instances of this metamodel are called *(prototype) mapping models*.

In our case, three types of mapping nodes are defined, namely, *Cls2Tab*, *Asc2Tab*, *Attr2Col*, which pair classes to tables, associations to tables and attributes to columns, respectively. In addition, various typed edges interconnect these mapping nodes to elements in the source and target metamodels.

– In case of *Cls2Tab*, edge *class* points to a *Class* in the source language, edge *tab* links to the corresponding *Table*, edge *pkey* marks the primary key *Column* in the database model, while *kind* denotes the *Column* storing the type information for the instances.
– In case of *Asc2Tab*, an edge *assoc* points to an *Association*, and a *tab* edge denotes the *Table* derived as main corresponding target element. In addition, a pair of edges (prefixed with *src* and *trg*, respectively) are used to mark the *Columns* storing the identifiers of classes and foreign key *FKey* restrictions (as denoted by edges *srccol*, *srckey*, *trgcol* and *trgkey*.
– Finally, in case of *Attr2Col*, an edge *attr* links a source *Attribute* with a corresponding target *Column*.

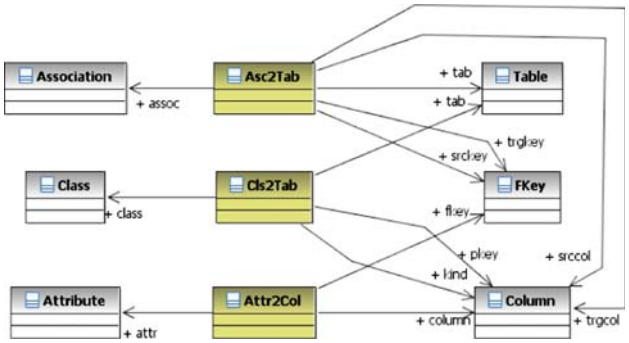

Fig. 3 Mapping metamodel

We require that all edges in the mapping metamodel (i.e. those associations that lead out from classes of the mapping metamodel) have an at most one multiplicity, which is not depicted explicitly in Fig. 3. As a consequence, each mapping node in a model uniquely identifies all the interconnected elements in the source and target models.

Furthermore, for the sake of convenience, we assume that these (outgoing) edges of the mapping metamodel can be (totally) ordered for each type of mapping nodes and edges pointing to source elements precede edges leading to target elements. For instance, we can assume that the edges of the *Cls2Tab* node are ordered in the following way: *class*, *tab*, *pkey* and *kind*.

These conditions enable to represent (model-level) mapping structures as tuples

$$ref(r_i, src_1, \ldots, src_n, trg_1, \ldots, trg_m)$$

where ref corresponds to the type of the mapping node, r_i is the unique identifier of the node, $src_1, \ldots, src_n$ are nodes of the source model (defined by the order of mapping edges), while $trg_1, \ldots, trg_m$ are nodes of the target model linked by appropriate mapping edges.

Assumptions on mapping metamodel. It is worth pointing out that we assume that transformation designers provide the mapping metamodel as input to model transformation

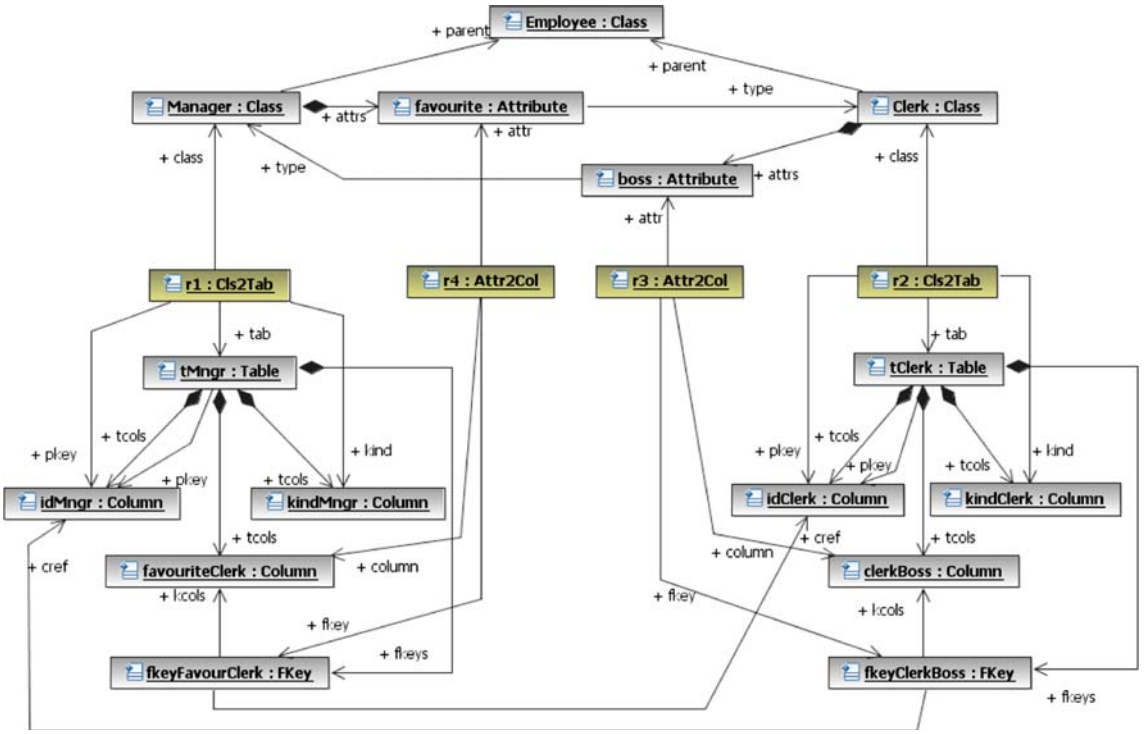

Fig. 4 A prototype mapping model

by example. This metamodel already contains certain hints from the transformation designer on the actual transformation rules. For instance, we learn from the metamodel that classes are expected to be transformed to tables and columns.

However, the mapping metamodel does not specify (i) under what condition a class is transformed into a table, (ii) if the same class can be mapped to different tables, (iii) if two classes are mapped into the same tables, or (iv) if classes are ever transformed to columns at all. These contextual conditions will be defined below implicitly by prototype mapping models. The automation discussed in this paper will exclusively focus on automating the derivation of model transformation rules to precisely capture contextual conditions with respect to its inputs, namely, the mapping metamodel and prototype mapping models.

Finally, we regard the derivation of mapping metamodel as a separate automation problem, which is subject to future work. First results for (semi-)automating this step were reported in [11].

5.2 Prototype mapping models

Prototype mapping models can be obtained by interrelating any existing (real) source and target models. However, prototype mapping models are preferably small, thus they are rather created by hand to incorporate critical situations of the transformation problem. These prototype mapping models can also serve as test cases later on. A sample prototype mapping model is depicted in Fig. 4 for demonstration purposes.

This prototype mapping model contains three UML classes (*Employee*, *Manager* and *Clerk* where the first is the *parent* of the other two), an attribute *boss* of type *Manager* belonging to class *Clerk* and an attribute *favourite* of type *Clerk* belonging to the class *Manager*.

The prototype mapping model captures that two relational database tables are present in the target model (*tMngr* and *tClerk*). Both tables have three columns:

- The columns of *tMngr*: *idMngr* for the unique identifier, *kindMngr* for storing the kind of instance, and *favouriteClerk* which is the target equivalent of the *favourite* attribute together with a foreign key *fkeyFavourClerk*
- The columns of *tClerk*: *idClerk*, *kindClerk* and *clerkBoss*, where the latter is the target equivalent of the *boss* attribute together with a corresponding foreign key *fkeyClerkBoss*.

It is worth pointing out that not all source nodes are necessarily linked to a mapping node. In such a case, the corresponding source element does not have an equivalent in the target model (see the prototype mapping model of Fig. 5 for an example). However, each target element is required to be linked to exactly one mapping node, otherwise, their existence is not causally dependent on the source model, which will be one of our assumptions.

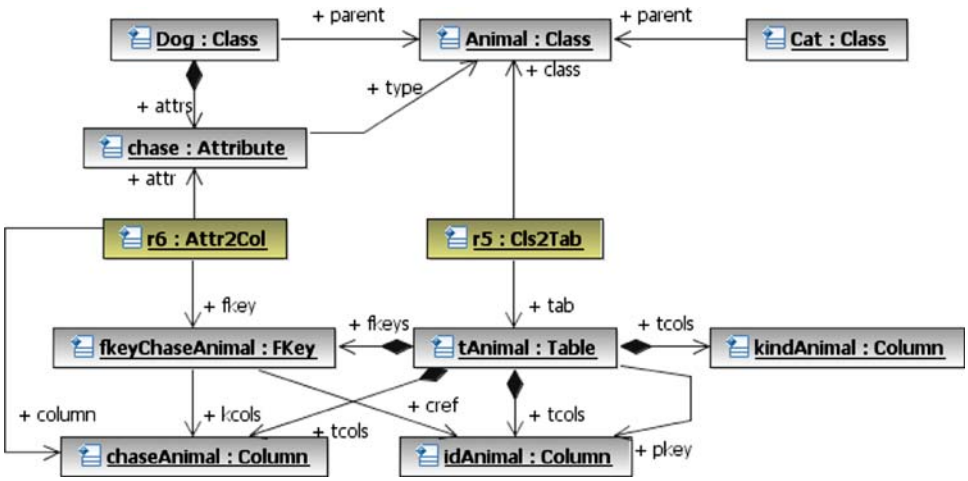

Fig. 5 Prototype mapping model with unmapped source elements

5.3 Assumptions

For this derivation process we make the following assumptions for the structure of the prototype mapping models:

Assumption 1: Each mapping node is connected to all the source nodes on which it is causally dependent. This requirement guarantees that prototype mapping models describe exactly the intent of the transformation designers, thus they can be used as a specification for the automation step.

Assumption 2: Each mapping node is connected to all the target nodes which cannot exist without each other (i.e. belong to the same component). For instance, in our mapping, a table generated from a class always has a primary key column, thus both are linked to the same mapping node. Note that this is a major syntactic difference compared to [34], however, it fully corresponds to the idea of weaving models [4] where complex mapping structures are used.

Assumption 3: Each target node is linked to exactly one mapping node. For instance, a table and its primary key column will be linked to the same mapping node. This requirement prescribes that the transformation is deterministic in a sense that the creation of each target node is uniquely identified by a mapping node and a corresponding mapping edge, and no merging of target nodes is required.

Assumption 4: The existence of a node in the target model depends only on the existence of a certain mapping node, i.e. source and target models are dependent on each other only indirectly via mapping structures.

Assumption 5: The existence of an edge in the target model depends only on contextual conditions of the source model and the existence of certain structure (but does not directly depend on the target model itself).

While these assumptions seem to cover a large set of practical model transformations, we will also discuss certain transformation problems in Sect. 7 where some of these conditions do not hold, and we ran into problems when automating MTBE.

6 Automating model transformation by example

We now discuss how inductive logic programming can be used to automate the model transformation by example approach. First we sketch how prototype mapping models can be constructed and mapped into corresponding Prolog clauses (Sect. 6.1). Then, we discuss one by one how to automate each phase of MTBE (Sects. 6.2–6.5).

6.1 From prototype mapping models to Prolog clauses

Source and target models are mapped into corresponding Prolog clauses following a straightforward representation where each node type in the source or target metamodel is mapped into a unary predicate, and each edge type in the metamodel is transformed into a binary predicate. In order to avoid name clashes, the name of the source node type is generated as a prefix for binary predicates for edges.

- On the one hand, (source or target) model nodes correspond to a ground predicate (over the unique identifier which represents the object), which predicate defines the type of that node.
- On the other hand, (source or target) edges have no unique identifiers in our representation, the corresponding binary predicate defines the source and target nodes, respectively.

Note that this is not a conceptual limitation of our approach: this mapping to Prolog clauses could be easily

refined to incorporate identifiers of edges. Unsurprisingly, there is a penalty in the performance of the underlying ILP engine to incorporate this change.

For instance, in the UML part of the prototype mapping model of Fig. 4, a class *Clerk* with an attribute *boss* of type class *Manager* can be represented by the following Prolog clauses (provided that *clerk*, *manager* and *boss* are unique identifiers this time).

```
% Clauses for source model
class(clerk).
class(manager).
attribute(boss).
attribute(favourite).
class_attrs(manager,favourite).
class_attrs(clerk,boss).
attribute_type(boss,manager).
attribute_type(favourite,clerk).
```

The representation of its corresponding target model (see Fig. 4) is the following.

```
% Clauses for target model
table(tMngr).
column(idMngr).
column(kindMngr).
column(favouriteClerk).
table_tcols(tMngr,idMngr).
table_tcols(tMngr,kindMngr).
table_tcols(tMngr,favouriteClerk).
table_pkey(tMngr,idMngr).
fkey(fkeyFavourClerk).
table_fkey(tMngr,fkeyFavourClerk).
fkey_cref(fkeyFavourClerk,idClerk).
fkey_kcols(fkeyFavourClerk,
           favouriteClerk).

table(tClerk).
column(idClerk).
column(kindClerk).
column(clerkBoss).
table_tcols(tClerk,idClerk).
table_tcols(tClerk,kindClerk).
table_tcols(tClerk,clerkBoss).
table_pkey(tClerk,idClerk).
fkey(fkeyClerkBoss).
table_fkey(tClerk,fkeyClerkBoss).
fkey_cref(fkeyClerkBoss,idMngr).
fkey_kcols(fkeyClerkBoss,clerkBoss).
```

Representation of mapping models. Mapping (weaving) models are represented in a slightly different way in order to improve the performance of the ILP engine. In Sect. 5.1, we made an assumption on the structure of mapping models, namely, that each mapping node and its outgoing edges can be represented by a tuple

$$ref(r_i, src_1, \ldots, src_n, trg_1, \ldots, trg_m).$$

As a consequence a straightforward representation of mapping models is the following:

```
% Clauses for mapping model
cls2tab(r1,manager,tMngr,idMngr,kindMngr).
cls2tab(r2,clerk,tClerk,idClerk,kindClerk).
attr2col(r3,boss,clerkBoss,fkeyClerkBoss).
attr2col(r4,favourite,favouriteClerk,
            fkeyFavourClerk).
```

Furthermore, each mapping node is uniquely identified by (ordered) source nodes $src_1, \ldots, src_n$ as well as (ordered) target nodes $trg_1, \ldots, trg_m$. Therefore, the identifier r_i can be omitted from the tuple, moreover, the tuple can be projected to the source and target elements without changing its truth value, i.e.

$$\begin{aligned} ref(r_i, src_1, \ldots, src_n, trg_1, \ldots, trg_m) &\iff \\ ref(src_1, \ldots, src_n) &\iff \\ ref(trg_1, \ldots, trg_m). & \end{aligned}$$

As a consequence, we can interchangeably use the following clauses in the different phases of MTBE:

```
% Alternative clauses for mapping models
cls2tab(r1,manager,tMngr,idMngr,kindMngr).
cls2tab(manager).
cls2tab(tMngr,idMngr,kindMngr).
```

Inheritance and helper edges. While the simplified metamodels of Fig. 1 do not contain generalization, we emphasize that the inheritance hierarchy of metamodel nodes and edges (i.e. the generalization of classes and the refinement of association ends as in MOF 2.0) can be mapped into Prolog clauses of the form

superclass(X) :- subclass(X)

when *superclass* is a generalization of *subclass* in some metamodel. Such clauses will be part of our background knowledge.

Various model transformation approaches introduce the concept of helper (derived) edges in order to reduce the complexity of individual transformation rules. When the actual model transformation rules are derived, we assume that such helper edges of the source language are already part of our background knowledge. As a result, helper edges enable the ILP engine to derive more general hypothesis as an output.

Background knowledge on helper edges can be obtained in two different ways.

– We allow domain experts to directly add helper information to the knowledge base.

– However, we can also use ILP to derive such helper information automatically in a preprocessing phase carried out on the source and target languages separately.

In our running example, the background knowledge may contain the following definitions defined by some domain expert (currently on the Prolog level).

```
ancestor(X,Y) :- class_parent(X,Y).
ancestor(X,Y) :- class_parent(Z,Y),
                 ancestor(X,Z).
```

This definition of the *ancestor* relation can also be taught in a preprocessing phase by the ILP engine with an appropriate set of positive and negative examples (i.e. using the same technique as described below). In such a case, the domain expert only gives a hint that the *ancestor* relation may be important (by adding it to the metamodel as a derived element), and supplies an appropriate set of examples. In the sequel, we assume the presence of such helper relations in the background knowledge, which were derived by analyzing the source and target models prior to the model transformation itself.

Moreover, by using abductive learning [1,21] techniques, Aleph can also synthesize the *ancestor* relation on demand, i.e. when learning other predicates. In this case, we only need to assume that our background knowledge contains sound but not necessarily complete information on helper edges like *ancestor*, i.e. the domain expert needs to give some positive examples for the *ancestor* edge.

Furthermore, there is intensive research on predicate learning [7] in the field of inductive logic programming when absolutely no hint is required from domain experts on helper edges, i.e. not even the existence of *ancestor* edge is required to be given as a hint. However, Aleph unfortunately does not yet support this feature.

6.2 Context analysis

In this phase, we identify constraints in the source and target models for the presence of each mapping node.

Context analysis for the source model. In case of context analysis of the source model, our background knowledge B will consist of all the facts derived from the source model (by taking the source projection of the prototype mapping model), and the positive and negative facts (E^+ and E^-) will consist of tuples on the mapping node in question. This way, a separate ILP problem will be derived for each mapping node.

For instance, we will identify contextual conditions in the source model when a mapping node of type *attr2col* should appear. For this purpose, we construct B from Fig. 4 to obtain the facts listed above. For positive facts, we have *attr2col(boss)*, and *attr2col(favourite)* and for negative facts, we can state *attr2col(manager)*, *attr2col(employee)* and *attr2col(clerk)*. In Aleph, all these specifications need to be listed in separate files, but for presentation purposes, we will list them together.

```
% Background knowledge
class(clerk).
class_attrs(clerk,boss).
attribute(boss).
class(manager).
class_attrs(manager,favourite).
attribute_type(boss,manager).
attribute_type(favourite,clerk).
% Positive facts
attr2col(boss).
attr2col(favourite).
% Negative facts
attr2col(clerk).
attr2col(manager).
attr2col(employee).
```

When deriving negative facts from a prototype mapping model, we currently build on closed world assumption as a frame condition, i.e. negative facts are derived for all identifiers part of the source model and not listed in positive facts. However, this is not the only possibility, and we can explicitly ask the user to provide (an incomplete) list of negative examples. While this latter approach seems to be more cumbersome, in many case the ILP engine can better tolerate noise, i.e. when transformation designers unintendedly include (or exclude) certain elements from the prototype mapping model.

Aleph will automatically derive the following rule as hypothesis by using core inductive logic programming algorithms:

```
attr2col(X) :- attribute(X).
```

Note that this result fulfills our expectations provided that only well-formed models are considered when language-specific constraints are checked separately.

However, we might want to specify that attributes are required to be attached to classes, and that each attribute is required to have a type in order to be transformed into a corresponding column. Since ILP derives the most general solution, these constraints are not incorporated in the solution by default. Therefore, we enrich our background knowledge and negative facts with fictitious model elements as follows.

```
% Background knowledge
% ... as before +
attribute(notype).
```

```
attribute(noattrs).
class_attrs(notype,manager).
class_attrs(noattribute,clerk).
attribute_type(noattrs,manager).
attrs(noattribute,manager).
% Negative facts
% ... as before
% Negative facts
% ... as before +
attr2col(noattrs).
attr2col(notype).
attr2col(noattribute).
```

As a result, the ILP engine will derive the following rule:

```
attr2col(A) :- attribute(A),
               class_attrs(B,A),
               attribute_type(A,C).
```

This rule will properly handle incomplete model as well. The price we paid for that is that both the background knowledge B and the negative facts E^- need to be extended with carefully selected cases, which can be cumbersome. Fortunately, if a sufficient number of real source and target model pairs are available, they might already cover cases to handle such incomplete models. Anyhow, it is a subject of future research to incorporate language constraints into automatically generated transformation rules.

In the general case, the ILP engine might derive multiple rules as a hypothesis, which means a disjunction of the different bodies. This is normal for the context analysis of the source model, since the same type of mapping nodes can be used with different source contexts.

Context analysis for the target model. As for the context analysis of the target model, the ILP engine needs to identify trivial hypotheses due to Assumption 3 (discussed in Sect. 5.3), which prescribes causal dependency of target nodes on the mapping structure. Therefore, we reverse the direction of our investigations, and use predicates corresponding to mapping structures as background knowledge, predicates corresponding to the nodes of the target model as positive and negative examples. Below we present an example for identifying the target context for *FKey*s.

```
% Background knowledge
attr2col(favouriteClerk,fkeyFavourClerk).
attr2col(clerkBoss,fkeyClerkBoss).
cls2tab(tMngr,idMngr,kindMngr).
cls2tab(tClerk,idClerk,kindClerk).
% Positive facts
fkey(fkeyClerkBoss).
% Negative facts
fkey(clerkBoss).
fkey(tMngr).
fkey(idMngr).
fkey(kindMngr).
fkey(tClerk).
fkey(idClerk).
fkey(kindClerk).
```

As a result, the ILP engine will derive the following hypothesis for the target model, which states that a mapping node of type *attr2col* should always be connected to an *FKey* in the target model.

```
fkey(A) :- attr2col(B,A).
```

In contrast to the context analysis of the source model, it is an error now if the ILP engine derives multiple rules for the target model as Assumption 3 would be violated.

In addition to this context analysis, one could also carry out context analysis with reverse roles, which can help constructing reverse transformations, but this step is out of scope for the current paper.

Aleph-specific settings. There are a couple of important Aleph-specific parameters which need to be set appropriately to drive the search engine for the proper hypothesis. We identified the following most important ones.

- *Number of variables* (*i*). We need to limit the number of fresh variables used in constructing a hypothesis. For instance, *B* and *C* were such fresh variables in the example of *attr2col* above.
- *Clause length* (*clauselength*). An upper bound needs to be set on the number of clauses used when constructing a hypothesis. For instance, the hypothesized *attr2col* rule contains three clauses (and the header).
- *Determination*. In case of determination, we need to set which clauses may have an impact on the hypothesized clause. Without the loss of generality, we assume that all source clauses are allowed to have an impact on the existence all mapping nodes. For instance, *determination(attr2col, type)* prescribes that predicate *type* may induce *attr2col*.
- *Modes*. Mode settings (i) prescribe the determinism or non-determinism of a predicate, (ii) define which variables of a clause are input and which are output variables, and (iii) assign (pseudo-)types to each predicate. For instance,

  :- mode(, attrs(+class,-attribute))*

  denotes that predicate *attrs* may succeed several times (*) if its first parameter is an input variable (+) and its second parameter is an output variable (-).

Note that our pessimistic approach for setting determination and modes causes only minor performance penalty in case of small knowledge bases, and a prototype mapping model is, typically, relatively small. We have experienced performance problems only in our complex case study of [14] when a more careful adjustment of these parameters was required.

6.3 Learning negative constraints

In a typical model transformation, the existence of a certain mapping structure may depend on the non-existence of certain structures in the source model. ILP systems frequently contain support to identify such negative constraints by means of *constraint learning*. The technique of constraint learning aims at identifying negative constraints of the form *false :- b1, b2, ... , bn.*, i.e. the conjunction of the bodies should never happen.

Learning of negative constraints will be demonstrated on the intuitive mapping rule that only top-level classes should be transformed into database tables. For this purpose, we use an additional prototype mapping model, which is illustrated in Fig. 5.

In case of constraint learning, we only need to construct the background knowledge without positive and negative facts from the source model and the mapping structures as follows:

```
% Background knowledge
class(animal).
class(dog).
class_parent(dog,animal).
class_attrs(dog,chase).
attribute(chase).
class(cat).
attribute_type(chase,animal).
cls2tab(animal).
```

With appropriate Aleph settings, the following constraints will be induced automatically (which are specific to the mapping node *cls2tab*):

```
false :- cls2tab(A),
         class_parent(A,A).
false :- cls2tab(A),
         class_parent(B,A).
```

The first constraint is, in fact, a language restriction of UML (i.e. no class is a parent of itself), while the second derived constraint is a negative constraint of the transformation itself.

A practical problem of the Aleph system we needed to face is that all synthesized constraints are listed instead of listing only the most general ones. Therefore, we need to prune the enumerated list of constraints according to clause entailment in order to keep only the most general constraints. For instance, the following constraint is derived by Aleph, but should be filtered out as presenting redundant knowledge with respect to the previous results:

```
false :- cls2tab(A),
         class_parent(A,A),
         class_parent(B,A).
```

We carry out this pruning by submitting the derived constraints to the Prover9 first-order theorem prover [26]. For this purpose, we present the identified constraints as assumptions to the theorem prover, and the prover tries to construct a formal proof that the more complex constraint implies any of the more simple ones. As for the example above, our assumptions are the following:

$$\forall A \neg(\text{cls2tab}(A) \wedge \text{class\_parent}(A, A)) \quad (1)$$

$$\forall A \forall B \neg(\text{cls2tab}(A) \wedge \text{class\_parent}(B, A)) \quad (2)$$

$$\forall A \forall B \neg(\text{cls2tab}(A) \wedge \text{class\_parent}(A, A) \wedge \text{class\_parent}(B, A)). \quad (3)$$

As a theorem we aim to prove that

$$\forall A \forall B(\text{cls2tab}(A) \wedge \text{class\_parent}(A, A) \wedge \text{class\_parent}(B, A)) \rightarrow (\text{cls2tab}(A) \wedge \text{class\_parent}(A, A) \vee \text{cls2tab}(A) \wedge \text{class\_parent}(B, A)).$$

Prior to submitting similar problems to the theorem prover, the constraints are ordered according to their length. All (non-filtered) constraints up to length $n-1$ can be used when proving the entailment of a constraint of length n. Fortunately, such theorems are proved immediately by Prover9, therefore, the effort related to the use of sophisticated theorem provers is negligible.

6.4 Connectivity analysis

In case of connectivity analysis, we derive different kind of ILP problems for each edge in the target metamodel. The background knowledge B now contains all elements from the source model and all mapping structures as well. This time, the tuple of the mapping structure contains both source and target mappings as follows:

$$ref(src_1, \ldots, src_n, trg_1, \ldots, trg_m).$$

Positive and negative facts (E^+ and E^-) are derived this time from an edge in the target metamodel by deriving a separate ILP problem from each edge type.

As a demonstration, we carry out the connectivity analysis for target edge *cref*, which is performed (from an ILP perspective) in a similar way as context analysis.

```
% Background knowledge
% Source model
class(employee).
class(clerk).
class_parent(clerk,employee).
class_attrs(clerk,boss).
attribute(boss).
class(manager).
class_parent(manager,employee).
class_attrs(manager,favourite).
attribute_type(boss,manager).
attribute_type(favourite,clerk).
% Mapping model
attr2col(favourite,favouriteClerk,
            fkeyFavourClerk).
attr2col(boss,clerkBoss,fkeyClerkBoss).
cls2tab(manager,tMngr,idMngr,kindMngr).
cls2tab(clerk,tClerk,idClerk,kindClerk).
% Helper edges
ancestor(X,Y) :- class_parent(X,Y).
ancestor(X,Y) :- class_parent(Z,Y),
                 ancestor(X,Z).

% Positive facts
cref(fkeyClerkBoss,idMngr).
cref(fkeyFavourClerk,idClerk).
% Negative facts = all type consistent
% pairs which are not in positive facts
cref(fkeyClerkBoss,kindMngr).
cref(fkeyClerkBoss,idClerk).
cref(fkeyClerkBoss,kindClerk).
cref(fkeyClerkBoss,clerkBoss).
cref(fkeyClerkBoss,favouriteClerk).
cref(fkeyFavourClerk,kindMngr).
cref(fkeyFavourClerk,idMngr).
cref(fkeyFavourClerk,kindClerk).
cref(fkeyFavourClerk,clerkBoss).
cref(fkeyFavourClerk,favouriteClerk).
```

Further explanation is needed to understand how negative facts are derived this time. Here, we enumerate all *FKey* and *Column* pairs in the model which are not connected by a *cref* edge. In this step, we impose closed world assumption on our model, thus the set of (type-conforming) object identifiers are exactly those that can be found in the model.

The Aleph ILP system will derive the following hypothesis for this prototype mapping model.

```
cref(A,B) :- cls2tab(C, D, B, E),
             attr2col(F, G, A),
             attribute_type(F, C).
```

This rule states that a target node A (of type *FKey*) is connected to a target node B (of type *Column*) with an edge of type *cref*, if class C which belongs to the table D containing B as its primary key is the type of attribute F, which is the source equivalent of the foreign key A.

During a validation phase, domain experts might reveal that this is only a partial solution since *type* edges may lead into a subclass (descendant) of class E, and not necessarily into E itself. Therefore, if we incrementally refine our knowledge base by adding the prototype mapping model of Fig. 5, a new Prolog inference rule is derived in addition to the previous one, which fully corresponds to our expectations:

```
cref(A,B) :- attr2col(C,A,D),
             attribute_type(C,E),
             ancestor(E,F),
             cls2tab(F,G,B,H).
```

This rule extends the previous one by stating that maybe an ancestor F of the class E (which is the type of attribute C) has the corresponding table which stores the mapped primary key column B.

With appropriate additional training for associations, Aleph will derive six rules which "generates" a *cref* edge (using *src* and *dst* instead of *type*, and *asc2tab* instead of *attr2col*).

6.5 Generation of model transformation rules

Now we discuss how to derive model transformation rules based upon the inference rules derived by the ILP engine during context analysis and connectivity rules. We use graph transformation rules [9] as the underlying transformation language, which provides a pattern and rule based manipulation technique for graph models frequently used in various model transformation tools. Each rule application transforms a graph by replacing a part of it with another graph. We derive a different set of rules for generating target nodes and edges.

Graph transformation rules. Formally, a *graph transformation rule* contains a left-hand side graph LHS, a right-hand side graph RHS, and a negative application condition graph NAC. The LHS and the NAC graphs are together called the precondition PRE of the rule.

The *application* of a GT rule to a *host model* M replaces a matching of the LHS in M by an image of the RHS. This is performed by (i) finding a matching of LHS in M (by pattern matching), (ii) checking the negative application conditions NAC (which prohibit the presence of certain nodes and edges) (iii) removing a part of the model M that can be mapped to LHS but not to RHS yielding the context model, and (iv) gluing the context model with an image of the RHS by adding new objects and links (that can be mapped to the RHS but not to the LHS) obtaining the *derived model* M′.

In the paper, we use a (slightly modified) graphical representation initially introduced in [12] where the union of these graphs is presented. Elements to be deleted are marked by the *del* keyword, elements to be created are labeled by *new*, while elements in the *NAC* graph are denoted by *neg*.

Rules for generating target nodes. The first kind of GT rules that we derive are required for generating all the target nodes.

For this purpose, we combine the source and the target context of a certain mapping node as discussed in Algorithm 1.

- The LHS of the rule is constructed for each source context of a mapping node (Lines 2–5).
- The RHS of the rule contains a copy of the LHS and the entire target context of the same mapping node interconnected by an appropriate mapping structure (Lines 6–12).
- A separate NAC is derived for each negative constraint containing a mapping node of a certain type (Lines 13–15).
- Finally, the mapping structure generated by the RHS is added to prevent applying the rule twice on the same source object.

As a demonstration, we list the graph transformation rule derived from mapping node *cls2tab* in Fig. 6. The rule expresses that for each class *C* without a child superclass *CP*, a table *T* is generated with two columns *Id* and *Kind* (which are, in fact, attached to table *T* later on by transformation rules generating edges).

Rules for generating target edges. A second set of graph transformation rules aims at interconnecting previously generated target nodes with appropriate edges. For this purpose, we need to combine the Prolog rules derived during connectivity analysis with target contexts. During connectivity analysis, we identify what conditions are required in the SOURCE language in order to generate a target edge of a certain type.

- The LHS of such GT rule is constituted from the inference rules derived by connectivity analysis. These inference rules only identify source edges and interconnecting mapping structures, thus the types of the related nodes need to be inferred. (Lines 2–5)
- The RHS is simply a copy of the LHS extended with the corresponding target edge. (Lines 6–7)
- The same edge is added as a NAC is derived to prevent multiple application of the rule on the same match. (Line 8)

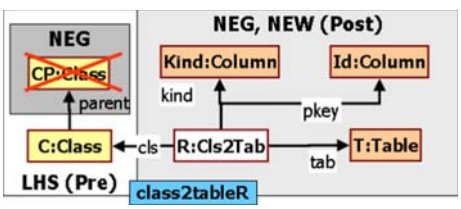

Fig. 6 GT rule to derive for target nodes

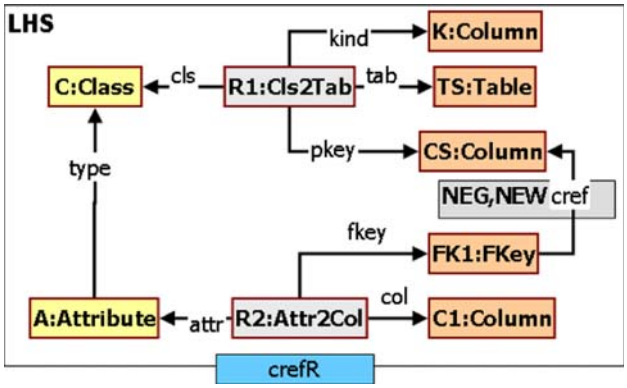

Fig. 7 A GT rule to derive target *cref* edges

As a demonstration, we present one GT rule (out of the six) derived for generating *cref* edges in Fig. 7. In the current paper, we systematically separated graph transformation rules into node creation rules and edge creation rules for separation of transformation concerns. It is subject to future research to merge these automatically generated rules in order to obtain a more compact set of transformation rules.

Helper edges can be treated in two ways. A first solution is when each helper edge is added explicitly to the metamodel, thus they are ordinary edges. These edges are then derived by ordinary graph transformation rules. Alternatively, a helper edge can be represented as a recursive pattern [3] which is supported by model transformation frameworks like VIATRA or TEFKAT.

Practical assessment of MTBE for the object-relational mapping. As an initial case study for our approach, we implemented the object-relational mapping with MTBE. The source model used in the prototype mapping models contained altogether ten classes, three attributes and four associations. Its target equivalent contained 8 tables with 2 or 3 columns for each, and 11 primary key constraints.

Using this relatively small prototype mapping model, we were able to automatically generate 20 graph transformation rules (3 for deriving target nodes and 17 for deriving target edges) by using MTBE, which turned out to be the complete transformation.

Obviously, this prototype mapping model was not created directly from scratch, but it was a result of three iterations. However, all rules have been derived automatically, which exceeded our expectations expressed in [34].

7 Known limitations

In order to assess the conceptual and practical limitations of our approach, we have chosen a complex model transformation problem as another case study, where a large abstraction gap needs to be bridged between the source and target languages. The transformation was originally defined in

Algorithm 1 An algorithm for generating graph transformation rules to derive target nodes

RN: a node from the mapping metamodel
p_{RN}: a node predicate derived from the mapping metamodel
$SrcCtx$: list of inference rules derived as the source context for RN
$TrgCtx$: list of inference rules derived as the target context for RN
$NegConstr$: negative constraints derived for RN

fun *generate_GT_rule_for_nodes*(RN, $SrcCtx$, $TrgCtx$, $NegConstr$)=
1: **for all** inference rule p^{src}_{RN} *:- a1, a2, an* in $SrcCtx$ **do**
2: create an empty GT rule $R = (LHS, RHS, NAC)$
// Creating the LHS
3: add a node for each variable appearing in the body *a1, a2, an*
4: infer types for the nodes based on the metamodels
5: add an edge to the LHS for each predicate of the body *a1,a2,an*
// Creating the RHS
6: copy LHS to RHS
7: add a mapping node $r1$ for p_{RN}
8: add edges to interconnect $r1$ with all source nodes identified by p^{src}_{RN}
9: **for all** inference rule p^{trg}_{RN} *:- b1* in $TrgCtx$ **do**
10: add a target node n_{b1} to RHS for the body *b1*
11: add an edge to interconnect $r1$ with the new target node n_{b1}
12: **end for**
// Creating NACs based on constraints
13: **for all** negative constraint *false :- n1, nk* in $NegConstr$ **do**
14: add the graph corresponding to the body of the constraint as a new NAC
15: **end for**
// NAC to prevent applying a rule twice on a matching
16: add a mapping node $r2$ node to a new NAC
17: add edges to all the source nodes *a1, a2, an* from $r2$
18: **end for**

Algorithm 2 An algorithm for generating graph transformation rules to derive target edges

RE: an edge from the target metamodel
$p_{RE}(A, B)$: an edge predicate derived from the target metamodel
$Connect$: list of inference rules derived connectivity analysis for RE

fun *generate_GT_rule_for_edges*(RE, $SrcCtx$, $TrgCtx$, $NegConstr$)=
1: **for all** inference rule $p^{RE}(A, B)$ *:- a1, a2, an* in $Connect$ **do**
2: create an empty GT rule $R = (LHS, RHS, NAC)$
// Creating the LHS
3: add a node for each variable appearing in the body *a1, a2, an*
4: infer types for the nodes based on the metamodels
5: add an edge to the LHS for each predicate of the body *a1,a2,an*
// Creating the RHS
6: copy LHS to RHS
7: add a target edge $e1$ of type RE leading from node A to node B
// Creating NAC
8: add the same edge as a NAC as well
9: **end for**

[14], and it aims to derive a Petri net representation of UML statecharts for model analysis purposes.

On the one hand, most of the transformation rules have been generated automatically even for this complex case study. However, two conceptual limitations of the current MTBE approach using ILP techniques have also been revealed. These limitations do not violate our assumptions (see Sect. 5.3), but since they arise from practical problems, they demonstrate that these assumptions might be too restrictive for certain model transformation problems.

In order to avoid the presentation of the complex transformation itself, we have taken these problems out of their context and present an analogy in the context of the object-relational mapping.

7.1 Non-deterministic transformations

Let us assume that our object-relation transformation requires that each UML class can only be processed if all its parents

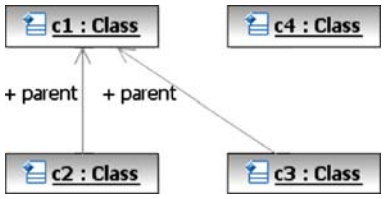

Fig. 8 A sample class hierarchy for ordering classes

have already been processed. Thus, we need to define a total order between the classes as an output, which can be denoted by a chain of *next* edges, for instance. Thus, the goal is to derive transformation rules for generating this total order using some sample chains as prototype mapping models.

Figure 8 depicts one class hierarchy where class *c1* is the parent of class *c2* and *c3*, while class *c4* is not in a *parent* relation with any of the previous classes. One possible ordering of these classes is *c1,c2,c3,c4*, but *c1,c4,c3,c2* is also a possible ordering.

In this case, the transformation has a non-deterministic result, i.e. any total order respecting the partial order imposed by the *parent* relation is a valid result. However, the main problem is that this transformation has two consecutive phases: GT rules generating the first *next* edge are different from GT rules generating subsequent *next* edges. However, these phases were not revealed by the ILP engine.

On the other hand, when the transformation rules are successfully derived, their confluence can be investigated by well-known techniques of graph transformation (such as critical pair analysis [15]).

7.2 Counting in transformations

The second problem is related to counting during model transformations. Let us assume that each table in the target database model should have an attribute counting the number of classes it represents. For instance, when a top-level class is transformed into a table, this attribute of the table should count the number of descendants of that class.

When transforming the class hierarchy presented in Fig. 8, two tables corresponding to classes *c1* and *c4* are generated. The counter for the corresponding table of *c1* should be set to 3, while the counter related to *c4* is set to 1.

Here the problem is that such a counting is only possible if (i) we use higher-order logic where one can refer to the number of matches for a certain predicate, or (ii) processing the matches sequentially as long as an unprocessed match is found.

7.3 Practical limitations

One complexity aspect of the model transformation [14] is that one source node is related to many target nodes due to the abstraction gap between high-level UML models and low-level Petri nets.

As a consequence, both the number of variables (*i*) and the clause length (*clauselength*) of the hypothesis (which are the most important parameters of the ILP engine concerning performance) had to be kept at a relatively large numbers (above 10). Thus, we experienced cases when the generation of inference rules was not instantaneous (but still completed within 5–10 s).

However, more critical performance would be experienced when the source model have the same complexity, i.e. when the derivation of a target edge depends on a large source context with a large number of edges. While our experience shows that this is not so common in a typical model transformation problem, these issues should be kept in mind as potential practical limitations of using an ILP engine for the model transformation by example approach.

It is worth pointing out that Aleph requires at least two positive examples in order to carry out generalization, otherwise only the facts themselves will be retrieved instead of inference rules.

8 Prototype tool support

We have implemented a prototypical tool chain (illustrated in Fig. 9) to automate MTBE by integrating an off-the-shelf model transformation tool with an ILP engine using Eclipse as the underlying tool framework.

– Source and target metamodels and models as well as prototype mapping models are constructed and stored as ordinary models in the VIATRA model space.

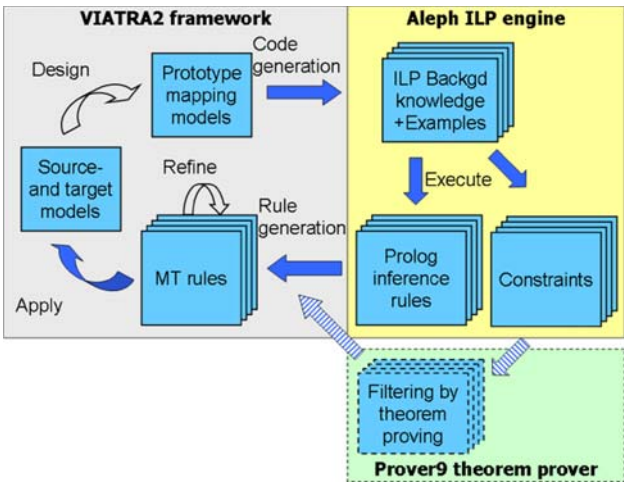

Fig. 9 A prototype tool chain for automating MTBE

– Then a first transformation takes prototype mapping models and generates a set of ILP problems in the way discussed in Sect. 6.
– These models are fed into the Aleph ILP engine to induce inference rules (for context analysis or connectivity analysis) or learn negative constraints. Obviously, this step is hidden from the user, as Aleph runs in the background.
– Ongoing work aims at integrating the Prover9 theorem prover in order to filter redundant constraints as described in Sect. 6.3.
– Based upon the discovered inference rules, transformation rules are synthesized in the graph transformation based language of the VIATRA2 framework [3].
– These transformation rules are then executed as ordinary transformations within VIATRA2 to complete the lifecycle of our model transformation by example approach.

This initial tool chain was already a great help for us in carrying out our experiments. Since a different ILP problem is submitted to Aleph for each type of mapping node or target edge, their manual derivation was already infeasible in practice.

However, additional future work should be carried out to improve the usability of the tool chain. Probably the most critical issue is that prototype mapping models need to be defined using the abstract syntax of the language, which is frequently too complex notation for domain experts. Ideally, mappings could be defined using the concrete syntax of source and target languages.

9 Related work

While the current paper is based on [34,35], Strommer et al. independently presented a very similar approach for model transformation by example in [36]. As the main conceptual difference between the two approaches is that [36] presents an object-based approach which finally derives ATL [17] rules for model transformation, while [34] is graph-based and derives graph transformation rules.

In a recent paper of Strommer [31], their MTBE approach is applied to a model transformation problem in the business process modeling domain and several new MTBE operators used on the concrete syntax were identified. Disregarding the string manipulation (which is obviously out of scope for the current paper), all the rest can be incorporated in our approach. In this sense, our limitations in Sect. 7 only arise in significantly more complex model transformation problems.

Naturally, the model transformation by example approach show correspondence to various "by-example" approaches. Query-by-example [38] aims at proposing a language for querying relational data constructed from sample tables filled with example rows and constraints. A related topic in the field of databases is semantic query optimization [20,30], which aims at learning a semantically equivalent query which yields a more efficient execution plan that satisfy the integrity constraints and dependencies.

The by-example approach has also been proposed in the XML world to derive XML schema transformers [10,19,24,37], which generate XSLT code to carry out transformations between XML documents. Advanced XSLT tools are also capable of generating XSLT scripts from schema-level (like MapForce from Altova [2]) or document (instance-)level mappings (such as the pioneering XSLerator from IBM Alphaworks, or the more recent StylisStudio [32]).

Programming by example [6,27], where the programmer (often the end-user) demonstrates actions on example data, and the computer records and possibly generalizes these actions, has also proven quite successful.

While the current paper heavily uses advanced tools in inductive logic programming [22], other fields of logic programming has also been popular in various model transformation approaches like answer set programming for approximating change propagation in [5] or F-Logic as a transformation language in [13].

The derivation of executable graph transformation rules from declarative triple graph grammar (TGG) rules is investigated in [18]. While TGG rules are quite close to the source and target modeling languages themselves, they are still created manually by the transformation designer.

10 Conclusions

In the current paper, we proposed to use inductive logic programming tools to automate the model transformation by example approach where model transformation rules are derived from an initial prototypical set of interrelated source and target models. We believe that the use of inductive logic programming is a significant novelty in the field of model transformations.

Let us briefly summarize our experience in using ILP and Aleph. Our experiments carried out on the object relational mapping and a complex model analysis transformation [14] demonstrated that ILP (and Aleph) is a very promising way for implementing MTBE due to the following reasons.

– Partly to our own surprise, we were able to derive all the transformation rules of the object-relational mapping automatically with Aleph, which exceeded our expectations in [34].
– For rule training, we used relatively small prototype mapping models.

– We used default Aleph settings almost everywhere (only the number of new variables and clauses were set manually). Determination and mode settings were derived systematically when using Aleph for MTBE.

Of course, we experienced certain limitations as well, which were presented in Sect. 7. Therefore, our main intentions for future work is to resolve these limitations.

– Non-deterministic transformations were problematic, especially, when an edge of a certain type needs to be generated in a certain order.
– We failed to implement counting in transformation rules, when a target attribute contains the number of matches of a source pattern. This way, handling of attributes values is subject to future work. We expect that data mining techniques could be usable to resolve this issue.

Despite these limitations we believe that the model transformation by example approach has a strong potential, and inductive logic programming turns out to be a powerful tool when implementing it.

References

1. Ade, H., Denecker, M.: AILP: Abductive inductive logic programming. In: Proceedings of the 14th International Joint Conference on Artificial Intelligence, pp. 1201–1209. Morgan Kaufmann, Montréal (1995)
2. Altova:. MapForce 2006. http://www.altova.com/features_xml2xml_mapforce.html
3. Balogh, A., Varró, D.: Advanced model transformation language constructs in the VIATRA2 framework. In: ACM Symposium on Applied Computing—Model Transformation Track (SAC 2006), pp. 1280–1287. ACM Press, Dijon (2006)
4. Bézivin, J.: On the unification power of models. Softw. Syst. Model. **4**(2), 171–188 (2005)
5. Cicchetti, A., di Ruscio, D., Eramo, R.: Towards propagation of changes by model approximations. In: Proceedings of the 10th International Enterprise Distributed Object Computing Conference Workshops (EDOC 2006), p. 24. IEEE Computer Society. Workshop on Models of Enterprise Computing (2006)
6. Cypher, A. (ed.): Watch What I Do: Programming by Demonstration. MIT Press, Cambridge (1993)
7. De Raedt, L., Lavrač, N.: Multiple predicate learning in two inductive logic programming settings. J. Pure Appl. Logic **4**(2), 227–254 (1996)
8. Didonet Del Fabro, M., Bézivin, J., Jouault, F., Valduriez, P.: Applying generic model management to data mapping. In: Journées Bases de Données Avancés (BDA), pp. 343–355 (2005)
9. Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.): Handbook on Graph Grammars and Computing by Graph Transformation, vol. 2: Applications, Languages and Tools. World Scientific, Singapore (1999)
10. Erwig, M.: Toward the automatic derivation of XML transformations. In: First International Workshop on XML Schema and Data Management (XSDM'03), LNCS, vol. 2814, pp. 342–354. Springer, Heidelberg (2003)
11. Fabro, M.D.D., Valduriez, P.: Semi-automatic model integration using matching transformations and weaving models. In: SAC 2007: Proceedings of the 2007 ACM Symposium on Applied computing, pp. 963–970. ACM, New York (2007)
12. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story diagrams: a new graph transformation language based on UML and Java. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) Proceedings of Theory and Application to Graph Transformations (TAGT'98), LNCS, vol. 1764. Springer, Heidelberg (2000)
13. Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A.: Transformation: The missing link of MDA. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) Proceedings of ICGT 2002: First International Conference on Graph Transformation, LNCS, vol. 2505, pp. 90–105. Springer, Barcelona (2002)
14. Huszerl, G., Majzik, I., Pataricza, A., Kosmidis, K., Dal Cin, M.: Quantitative analysis of UML Statechart models of dependable systems. Comput. J. **45**(3), 260–277 (2002)
15. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of typed attributed graph transformation systems. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) Proceedings of ICGT 2002: First International Conference on Graph Transformation, LNCS, vol. 2505, pp. 161–176. Springer, Barcelona (2002)
16. Jouault, F., Bézivin, J.: KM3: A DSL for metamodel specification. In: Proceedings of Formal Methods for Open Object-Based Distributed Systems (FMOODS 2006), LNCS, vol. 4037, pp. 171–185. Springer, Heidelberg (2006)
17. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Model Transformations in Practice Workshop at MODELS 2005, LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2005)
18. Königs, A., Schürr, A.: MDI—a rule-based multi-document and tool integration approach. J. Softw. Syst. Model., Special Section on Model-based Tool Integration (2006, in press)
19. Lechner, S., Schrefl, M.: Defining web schema transformers by example. In: Marik, V., Retschitzegger, W., Stepankova, O. (eds.) DEXA, LNCS, vol. 2736, pp. 46–56. Springer, Heidelberg (2003)
20. Lowden, B.G.T., Robinson, J.: Constructing inter-relational rules for semantic query optimisation. In: Hameurlain, A., Cicchetti, R., Traunmüller, R. (eds.) Proceedings of Database and Expert Systems Applications, 13th International Conference, DEXA 2002, Aix-en-Provence, France, 2–6 September, LNCS, vol. 2453, pp. 587–596. Springer, Heidelberg (2002)
21. Moyle, S.: Using theory completion to learn a navigation control program. In: Matwin, S., Sammut, C. (eds.) Proceedings of Twelfth International Conference on ILP (ILP 2002), vol. LNAI, pp. 182–197. Springer, Heidelberg (2003)
22. Muggleton, S., de Raedt, L.: Inductive logic programming: Theory and methods. J. Logic Program. **19-20**, 629–679 (1994)
23. Object Management Group. QVT: Request for Proposal for Queries, Views and Transformations. http://www.omg.org
24. Ono, K., Koyanagi, T., Abe, M., Hori, M.: XSLT stylesheet generation by example with WYSIWYG editing. In: Proceedings of the 2002 Symposium on Applications and the Internet (SAINT 2002), pp. 150–161. IEEE Computer Society, Washington, DC, USA (2002)
25. Progol. http://www.doc.ic.ac.uk/~shm/progol/
26. Prover9: Automated Theorem Prover. http://www.cs.unm.edu/~mccune/prover9/
27. Repenning, A., Perrone, C.: Programming by example: programming by analogous examples. Commun. ACM **43**(3), 90–97 (2000)
28. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformations: Foundations. World Scientific, Singapore (1997)
29. Schürr, A.: Specification of graph translators with triple graph grammars. In: Tinhofer, B. (ed.) Proceedings of WG94: International Workshop on Graph-Theoretic Concepts in Computer Science, LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1994)

30. Shekhar, S., Hamidzadeh, B., Kohli, A., Coyle, M.: Learning transformation rules for semantic query optimization: A data-driven approach. IEEE Trans. Knowl. Data Eng. **5**(6), 950–964 (1993)
31. Strommer, M., Murzek, M., Wimmer, M.: Applying model transformation by-example on business process modeling languages. In: Proceedings of Third International Workshop on Foundations and Practices of UML (ER 2007), LNCS, vol. 4802, pp. 116–125. Springer, Heidelberg (2007)
32. StylisStudio. http://www.stylusstudio.com
33. The Aleph Manual. http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/
34. Varró, D.: Model transformation by example. In: Proceedings of Model Driven Engineering Languages and Systems (MODELS 2006), LNCS, vol. 4199, pp. 410–424. Springer, Genova (2006)
35. Varró, D., Balogh, Z.: Automating model transformation by example using inductive logic programming. In: ACM Symposium on Applied Computing—Model Transformation Track (SAC 2007). ACM Press, New York (2007)
36. Wimmer, M., Strommer, M., Kargl, H., Kramler, G.: Towards model transformation generation by-example. In: Proceedings of HICSS-40 Hawaii International Conference on System Sciences, p. 285. IEEE Computer Society, Hawaii, USA (2007)
37. Yan, L.L., Miller, R.J., Haas, L.M., Fagin, R.: Data-driven understanding and refinement of schema mappings. In: Proceedings of ACM SIGMOD Conference on Management of Data, pp. 485–496 (2001)
38. Zloof, M.M.: Query-by-example: the invocation and definition of tables and forms. In: Kerr, D.S. (ed.) VLDB, pp. 1–24. ACM, New York (1975)

Author's Biography

Zoltán Balogh graduated in 2008 at the Budapest University of Technology and Economics. He has been working on the topic of model transformation by example. Since graduation, he works as a software engineer at OptXware Research and Development Ltd.

Dániel Varró is an assistant professor at the Budapest University of Technology and Economics. His research interests are related to model driven software and systems engineering with special focus on model transformations. He regularly serves in the programme committee of various international conferences in the field. He is the founder of the VIATRA model transformation framework, and the principal investigator at his university of the SENSORIA and DIANA European Projects. Previously, he was a visiting researcher at SRI International, at the University of Paderborn and TU Berlin. He is also a co-founder of the OptXware Research and Development Ltd.