Automating Model Transformation by Example Using Inductive Logic Programming*

Dániel Varró
varro@mit.bme.hu

Zoltán Balogh
zoli.balogh@gmail.com

Budapest University of Technology and Economics
Department of Measurement and Information Systems
H-1117, Magyar tudosok krt. 2., Budapest, Hungary

ABSTRACT

Model transformation by example [18] 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 [14] which aims at the inductive construction of first-order clausal theories from examples and background knowledge.

Categories and Subject Descriptors

D.1.6 [**Programming Techniques**]: Logic Programming—*Inductive Logic Programming*; D.3.2 [**Language Classifications**]: Specialized Application Languages—*Model transformation languages*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*Mathematical Induction*; I.2.2 [**Artificial Intelligence**]: Automatic Programming; I.6.5 [**Simulation and Modeling**]: Model Development

Keywords

model transformation, by-example synthesis, inductive logic programming

1. INTRODUCTION

The efficient design of automated model transformations between modeling languages have 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.

However, a common deficiency of all these 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 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.

Model transformation by example (MTBE) is an novel approach (introduced in [18]) 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 fully or semi-automatically.

However, the heuristic approach presented in [18] lacks theoretical foundations and highly relied on human interaction in order to derive appropriate transformation rules.

The current paper proposes to automate the model transformation by example approach using inductive logic programming [14] (ILP). 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 [18] using relatively small examples.

2. MOTIVATING EXAMPLE

For the motivating example of the current paper, we revisit the object-relational mapping problem discussed in [18] where UML class diagrams are mapped into relational database tables by using one of the standard solutions.

The source and target languages (UML and relational databases, respectively) are captured by their corresponding 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 for classes and associations, respectively.

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*).

*This work was partially supported by the SENSORIA European project (IST-3-016004). The first author was also supported by the J. Bolyai Scholarship.

SAC'07 March 11-15, 2007, Seoul, Korea

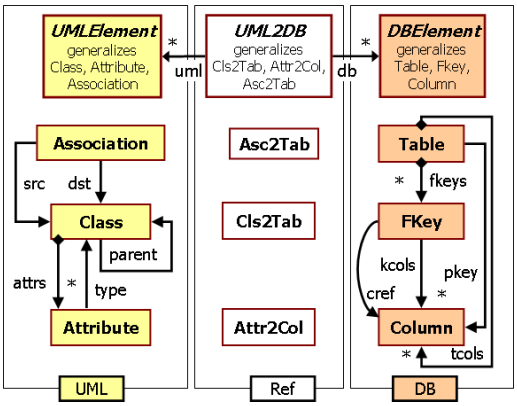

Figure 1: Metamodels of the example

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*). A foreign key refers to certain columns (*cref*) of another table, and it is related to the columns of local referring table by *kcols* edges.

These metamodels are extended with a *reference metamodel* to interconnect the elements of the source and the target language. As the main conceptual difference to the metamodel used in [18], a new abstract model element is defined for each domain. Furthermore, in the current paper, we require that a reference element is linked to all nodes in the source and target domains that should be present in the corresponding model when this reference node exists by ordered edges (*uml* and *db*).

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 automatically derive transformation rules corresponding to these informal guidelines by only relying on a prototypical set of interrelated source and target models as the specification of the transformation.

3. AN OVERVIEW OF APPROACHES

3.1 Inductive Logic Programming

Inductive logic programming [14] (ILP) aims at the inductive construction of 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 [14] 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 [2] that we used for our experiments are closely related to Prolog, and the following restrictions are quite typical:

- B is restricted to Prolog clauses, which are of the form $Head :- Body_1, Body_2, \ldots Body_n$, i.e. the conjunction of body clauses implies the head.
- E^+ and E^- are restricted to ground facts.

Further language and search restrictions can 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 Sec. 4.2.

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 can be given as follows:

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

Finally, some negative facts 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 by default.

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

By default settings (which is used in the current paper), Aleph will set up a hypothesis only for clauses used in the positive and negative examples. However, with additional tool-specific settings, Aleph will be able to derive the following more intuitive hypothesis as well:

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

3.2 Model transformation by example (MTBE)

The highly iterative and interactive process of MTBE was defined in [18] as follows:

Step 1: Manual set-up of prototype mapping models. The transformation designer assembles prototype mapping models which capture critical situations of the transformation problem by showing how the source and target model elements should be interrelated by appropriate reference (mapping) constructs.

Step 2: Automated derivation of rules. Based upon the prototype mapping models, the MTBE framework should synthesize the set of model transformation rules.

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.

In the current paper, we discuss how Step 2 can be automated by using an inductive logic programming 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 automatically. The automated derivation of rules will be carried out in the following phases.

Phase 1: Context analysis. Here we identify (positive and negative) constraints in the source and target models for the presence of each reference 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.

Phase 2: Connectivity analysis. For each edge in the target metamodel, we identify contextual conditions (in the source and reference models) for the existence of that target edge.

Phase 3: Transformation rules for target nodes. Finally, we derive transformation rules for all reference nodes that derive only target nodes using the information derived during context analysis. Furthermore, we also derive transformation rules for each target edge based upon connectivity analysis.

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

Assumption 1: Each reference node is connected to all the source and target nodes on which it is causally dependent (which is a major syntactic difference compared to [18]).

Assumption 2: Each target node is linked to exactly one reference node. For instance, a table and its primary key column will be linked to the same reference node.

Assumption 3: The existence of a node in the target model depends only on the existence of a certain reference node, i.e. source and target models are indirectly dependent on each other along reference structures. structure (but does not directly depend on the target model itself).

Assumption 4: 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).

4. 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 mapped into corresponding Prolog clauses, then we discuss one by one how to automate each phase of MTBE. For demonstrating these concepts, we use the prototype mapping model of Fig. 2.

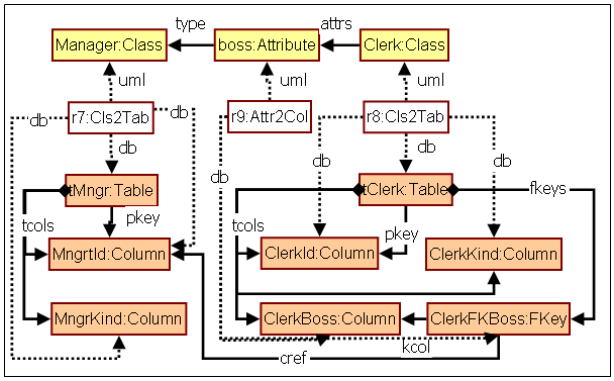

Figure 2: A prototype mapping model

4.1 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.

A node in an instance 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, edges have no unique identifiers in our representation, the corresponding binary predicate defines the source and target nodes, respectively.

For instance, in the UML part of the prototype mapping model of Fig. 2, 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).

```
class(clerk).
attrs(clerk,boss).
attribute(boss).
type(boss,manager).
class(manager).
```

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) is mapped into Prolog clauses of the form *classifier(X) :- class(X)*.

Moreover, we may derive additional helper Prolog clauses for the transitive closure of some edges in the source and target models, which may be used during induction to derive a more general hypothesis. For the sake of simplicity, we assume the existence of these helper constructs, however, they can be derived automatically in a preprocessing phase using the same ILP technique. For instance, the ancestor relation can be defined as the transitive closure of the parent relation as follows.

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

4.2 Context analysis

In the first phase, we identify constraints in the source and target models for the presence of each reference node. Since this step is carried out in an identical way for both the source and target models, we only describe how to derive constraints in the source model.

For this purpose, our background knowledge B will consist of all the facts derived from the source model, and the positive and negative facts (E^+ and E^-) will consist of facts on the reference node in question. Of course, a separate ILP problem should be derived for each reference node.

For instance, we will identify conditions in the source model when a reference node of type *attr2col* should appear. For this purpose, we construct B from Fig. 2 to obtain the facts listed above. For positive facts, we have *attr2col(boss)*, and for negative facts, we can state *attr2col(manager)* 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).
attrs(clerk,boss).
attribute(boss).
type(boss,manager).
class(manager).
% Positive facts
attr2col(boss).
% Negative facts
attr2col(clerk).
attr2col(manager).
```

By using ILP, Aleph will automatically derive the following rule as hypothesis, which already fully fulfills our expectations if only complete source models are considered (i.e. language specific constraints are checked separately).

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

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.

```
% Background knowledge
% ... as before +
attribute(notype).
attribute(noattrs).
attrs(notype,manager).
attrs(noattribute,clerk).
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), attrs(B,A), 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.

Aleph-specific settings. Of course, there are some 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 above.
- *Clause length* (*clauselength*). An upper bound need to be set on the number of clauses used when constructing a hypothesis. For instance, the last hypothesized rule contains three clauses.
- *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 reference 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 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.

4.3 Learning negative constraints

In a typical model transformation, the existence of a certain reference 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*. Constraint learning aims at identifying negative constraints of the form *false :- b1, b2.*, 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 only need to construct the background knowledge without positive and negative facts as follows:

```
% Background knowledge
class(customer).
class(product).
class(vipcustomer).
parent(vipcustomer,customer).
attrs(vipcustomer,favourite).
type(favourite,product).
attrs(product,owner).
type(owner,customer).
cls2tab(customer).
cls2tab(product).
```

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

```
false :- cls2tab(A), parent(A,A).
false :- cls2tab(A), 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. For this purpose, the enumerated list of constraints are pruned 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:

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

4.4 Connectivity analysis

In case of connectivity analysis, we derive another kind of ILP problems for each edge in the target model. The background knowledge B now contains all elements from the source model and all reference structures as well. However, positive and negative facts are derived this time from an edge in the target metamodel (by deriving separate ILP problems from each edge).

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(clerk).
attrs(clerk,boss).
attribute(boss).
type(boss,manager).
class(manager).
% cls2tab(class, table, pkey, kind)
cls2tab(clerk, tClerk, clerkId, clerkKind).
cls2tab(manager, tMngr, mngrId, mngrKind).
% attr2col(attribute, column, fkey)
attr2col(boss, clerkBoss, clerkFKBoss).
% Positive facts
cref(clerkFKBoss,mngrId).
% Negative facts = all type consistent pairs
% which are not in positive facts
cref(clerkFKBoss,mngrKind).
cref(clerkFKBoss,clerkId).
cref(clerkFKBoss,clerkKind).
cref(clerkFKBoss,clerkBoss).
```

Further explanation is needed how negative facts are derived this time. Here, we enumerate all pairs of *FKey* and *Column* pairs in the model which are not connected by a *cref* edge.

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

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

Note 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 refine our prototype mapping model by incrementally adding the example of Sec. 4.3, 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), type(C,E),
  ancestor(E,F), cls2tab(F,G,B,H).
```

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*).

4.5 Generation of model transformation rules

While the main emphasis of the current paper was on the use of ILP, we also sketch how model transformation rules are derived in the form of graph transformation rules [7] in accordance with source and target contexts and connectivity rules. Graph transformation provides a pattern and rule based manipulation of graph models, which is frequently used in various model transformation tools. Each rule application transforms a graph by replacing a part of it by another graph.

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 objects and links) (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 [9] 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 reference node as follows:

- The LHS of the rule is constructed from (i) each source context of a reference node with additional NAC derived from (ii) each negative constraint related to a reference node, (iii) and the difference of RHS \ LHS to prevent applying the rule twice on the same source object.
- The RHS contains RHS and a target context of the same reference node *without target edges*.

As a demonstration, we list the graph transformation rule derived from reference node *cls2tab* in Fig. 3. The rule expresses that for each class *C* without a child superclass *CP*, a table *T* is generated with two columns *Id* and *Kind*.

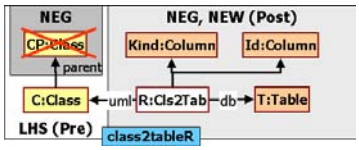

Figure 3: GT rule to derive for target nodes

Rules for generating target edges. Graph transformation rules aim 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. Context analysis, in turn, identify additional restrictions in the target model which need to be fulfilled (such as type restrictions of target nodes).

- The LHS of such GT rule is constituted of the merged context and connectivity patterns (Prolog rules) along reference structures of corresponding types. Multiple rules are derived if these patterns can be merged in different ways. The NACs are derived similarly as before to preserve negative constraints and to prevent multiple application on the same match.
- The RHS is simply a copy of the LHS extended with the corresponding target edge.

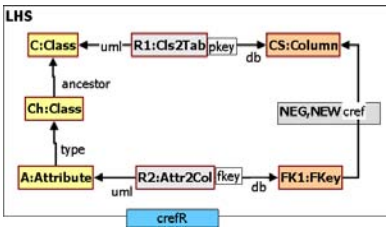

Figure 4: GT rule to derive target edges

As a demonstration, we present one GT rule derived for generating *cref* edges in Fig. 4. In order to distinguish between the elements linked to the same reference object, we will use appropriate qualifiers in a UML style: qualifiers *pkey* and *fkey* of *db* edges identify the appropriate target nodes (i.e. the primary key column and not the table itself).

5. RELATED WORK

While the current paper is based on [18], Strommer et al. independently present a very similar approach for model transformation by example in [19]. As the main conceptual difference between the two approaches is that [19] presents an object-based approach which finally derives ATL [4] rules for model transformation, while [18] is graph-based and derives graph transformation rules.

Naturally, the model transformation by example approach has direct roots in various "by-example" approaches. Query-by-example [21] 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 [13,17], 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 [8, 12, 15, 20], 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 [3]) or document (instance-)level mappings (such as the pioneering XSLerator from IBM Alphaworks, or the more recent StylisStudio [1]).

Programming by example [6, 16], 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 [14], 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 [10].

The derivation of executable graph transformation rules from declarative triple graph grammar (TGG) rules is investigated in [11]. While TGG rules are quite close to the source and target modeling languages themselves, they are still created manually by the transformation designer.

6. 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 different version of the object relational mapping 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 automatically with Aleph, which exceeded our expectations in [18].
- For rule training, we used relatively small prototype mapping models with about 100 graph objects.
- We used default Aleph settings almost everywhere (only the number of new variables and clauses were set manually). Determination and mode settings were written systematically when using Aleph for MTBE.

Of course, we experienced certain disadvantages as well, especially, with Aleph (and thus not really with the ILP approach itself). The resolution of these definitely requires the deep knowledge of Aleph.

- All (and not only most general) negative constraints are enumerated by constraint analysis, which often resulted in a verbose set of negative constraints.
- In some cases, the incremental extension of the prototype mapping model failed, i.e. the hypothesis was sensitive to the order of facts defined in the background knowledge.

Future work will primarily focus on implementing the transformations presented in the current paper. Hopefully, this paper demonstrated that the transformation from prototype mapping models to ILP problems is fairly straightforward. On the other hand, it is a more complex transformation to assemble graph transformation rules from the hypotheses. Furthermore, we aim at fine-tuning Aleph for ILP problems derived from the prototype mappings.

7. REFERENCES

[1] *StylisStudio*. `http://www.stylusstudio.com`.

[2] *The Aleph Manual*. `http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/`.

[3] Altova:. *MapForce 2006*. `http://www.altova.com/features_xml2xml_mapforce.html`.

[4] ATLAS Group. *The ATLAS Transformation Language*. `http://www.eclipse.org/gmt`.

[5] A. Cicchetti, D. di Ruscio, and R. Eramo. Towards propagation of changes by model approximations. In *International Workshop on Models of Enterprise Computing*. 2006. To appear.

[6] A. Cypher (ed.). *Watch What I Do: Programming by Demonstration*. The MIT Press, 1993.

[7] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (eds.). *Handbook on Graph Grammars and Computing by Graph Transformation*, vol. 2: Applications, Languages and Tools. World Scientific, 1999.

[8] M. Erwig. Toward the automatic derivation of XML transformations. In *1st Int. Workshop on XML Schema and Data Management (XSDM'03)*, vol. 2814 of *LNCS*, pp. 342–354. Springer, 2003.

[9] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph transformation language based on UML and Java. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (eds.), *Proc. Theory and Application to Graph Transformations (TAGT'98)*, vol. 1764 of *LNCS*. Springer, 2000.

[10] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The missing link of MDA. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg (eds.), *Proc. ICGT 2002: Firs International Conference on Graph Transformation*, vol. 2505 of *LNCS*, pp. 90–105. Springer-Verlag, Barcelona, Spain, 2002.

[11] A. Königs and A. Schürr. MDI - a rule-based multi-document and tool integration approach. *Journal of Software and Systems Modelling*, 2006. DOI: 10.1007/s10270-006-0016.

[12] S. Lechner and M. Schrefl. Defining web schema transformers by example. In V. Marik, W. Retschitzegger, and O. Stepankova (eds.), *DEXA*, vol. 2736 of *LNCS*, pp. 46–56. Springer, 2003.

[13] B. G. T. Lowden and J. Robinson. Constructing inter-relational rules for semantic query optimisation. In A. Hameurlain, R. Cicchetti, and R. Traunmüller (eds.), *Proceedings of Database and Expert Systems Applications, 13th International Conference, DEXA 2002, Aix-en-Provence, France, September 2-6,*, vol. 2453 of *LNCS*, pp. 587–596. Springer, 2002.

[14] S. Muggleton and L. de Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, vol. 19-20:pp. 629–679, 1994.

[15] K. Ono, T. Koyanagi, M. Abe, and M. Hori. 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.

[16] A. Repenning and C. Perrone. Programming by example: programming by analogous examples. *Comm. of the ACM*, vol. 43(3):pp. 90–97, 2000.

[17] S. Shekhar, B. Hamidzadeh, A. Kohli, and M. Coyle. Learning transformation rules for semantic query optimization: A data-driven approach. *IEEE Trans. Knowl. Data Eng.*, vol. 5(6):pp. 950–964, 1993.

[18] D. Varró. Model transformation by example. In *Proc. MODELS 2006*, pp. 410–424, Vol. 4199 of LNCS. Springer, 2006.

[19] M. Wimmer, M. Strommer, H. Kargl, and G. Kramler. Towards model transformation generation by-example. In *Proc. of HICSS-40 Hawaii International Conference on System Sciences*. Hawaii, USA., 2007. To appear.

[20] L. L. Yan, R. J. Miller, L. M. Haas, and R. Fagin. Data-driven understanding and refinement of schema mappings. In *Proc. ACM SIGMOD Conference on Management of Data*. 2001.

[21] M. M. Zloof. Query-by-example: the invocation and definition of tables and forms. In D. S. Kerr (ed.), *VLDB*, pp. 1–24. ACM, 1975.