VIII Jornadas sobre Programación y Lenguajes

Almendros-Jiménez y Suárez-Cabal Eds.

Gijón, del 8 al 10 de Octubre de 2008

A Framework for Model Transformation in Logic Programming

Jesús M. Almendros-Jiménez[1,2] Luis Iribarne[1,2]

Dpto. de Lenguajes y Computación. Universidad de Almería.

Abstract

In this paper we will present a framework for using logic programming (in particular, Prolog) for specifying model transformations in the context of UML. Our approach describes how the UML metamodel can be represented in Prolog, and how model transformations can be expressed by means of Prolog rules. It uses rules for specifying queries in source models and rules for expressing how to build the target model. Therefore we can distinguish between a *model query language* and a *transformation language*.Our approach will be applied to a well-known example of model transformation in which an UML class diagram for a database can be transformed into an UML diagram representing a relational database.

Keywords: Logic programming, Model transformation, UML

1 Introduction

Model transformation [20,11,14,8] is a key tool of the Model-driven development process. According to the *Model Driven Architecture (MDA)* initiative of the *Object Management Group (OMG)* [16], model transformation provides developers with tools for transforming their models. A simple definition of a model transformation tool is that it is able to mutate one model into another. We can take as an example of model transformation the *code generation* from a *visual model* for representing the architecture of a software system. For instance, most of *UML (Unified Modeling Language)* software development tools are able to generate code from UML class diagrams. In such a model transformation tool, the source model is the class diagram and the target model is code in a certain programming language. However, model transformation is a more general technique of transformation of models. In fact, usually *Model-to-model* (M2M) and *Model-to-Code* (M2C) transformations are considered.

[1] The author's work has been partially supported by the EU (FEDER) and the Spanish MEC under grant TIN2005-09207-C03-02 and TIN2007-61497, respectively.

[2] Email: jalmen@ual.es liribarne@ual.es

In this context, model transformation needs formal techniques for specifying the transformation. In particular, in most of cases transformations can be expressed by means of some kind of *rules*. The rules have to express how any given model can be transformed into another one. Let us now focus our attention in UML, which can be considered the standard of model specification. In the context of UML there are recent proposals of languages whose aim is the specification of transformations of UML models. In order to describe transformations such languages have to move to the *UML metamodel* [16] in which UML itself is defined by means of UML. By expressing transformations of the UML metamodel, any UML model can be transformed into another UML model.

The languages for transforming models range from imperative to declarative ones, but also there are hybrid proposals. Basically, any transformation language has to be able to traverse a model, has to define a certain set of rules, and has to create a new model. In other words, a transformation language should be equipped with a *model query language*, a *rule-based language*, and a *model updating/creation language*. The imperative part is usually responsible of the creation of the target model. The declarative part specifies how to query the source model and how to match the source model into the target model. Finally, some kind of mechanism for rule application (i.e. matching, rule ordering and chaining) is assumed, enabling the generation of new models.

In the context of the *MOF (Meta Object Facility)* metamodeling architecture, the *QVT (Query-View-Transformation)* language [16] has been proposed as standard for model transformation. *QVT* is a hybrid proposal involving a graphical syntax together with an imperative syntax, and the help of *OCL (Object Constraint Language)*. However, there are some other proposals of transformation languages. For instance, the language *ATL (ATLAS transformation language)* [10] provides declarative and imperative constructs. The declarative part of *ATL* is based on rules. Such rules consist of a source pattern matched over source models and of a target pattern that creates target models for each match. *Tefkat* [13,12] is a declarative language whose syntax and execution resemble Prolog (without function symbols), but allowing the user to assert new elements to the target model. There are also *graph transformation languages*. This is the case of *VIATRA2* [5], *GReAT* [1] and AGG [19], among others. Graph transformation languages describe transformations by rewriting graphs. Usually, these languages consist in rules whose match with a graph provides a transformation on the graph, in particular, deleting and adding new elements to the graph. *RubyTL* [18] is an object-oriented language with hybrid nature. It provides declarative and imperative constructs to define transformations. The rules of *RubyTL* can express mappings from source models into target models in which a filter can be added to select certain elements of the source model. The *MT* model transformation language [21] is also a case of hybrid language in which declarative patterns are combined with imperative code.

In this paper we will present a framework for using logic programming (in particular, Prolog) for specifying model transformations in the context of UML. Our approach describes how the UML metamodel can be represented in Prolog, and how model transformations can be expressed by means of Prolog rules. It uses rules for specifying queries in source models and rules for expressing how to build the

target model. Therefore we can distinguish between a *model query language* and a *transformation language*. Our approach will be applied to a well-known example of model transformation in which an UML class diagram for a database can be transformed into an UML diagram representing a relational database.

The motivation of our approach is the use of a well-known programming language for expressing model transformations. A Prolog programmer will be able to write his/her own transformations. He/she only needs to know how Prolog stores the UML metamodel and he/she can run transformations with the support of a Prolog tool. With respect to how to integrate UML with a Prolog tool, UML models can be exported in most of UML tools in the so-called *XMI (XML Metadata Interchange)* [16] a dialect of *XML*. Therefore a Prolog tool should be able to import XMI, more concretely XML (most of Prolog tools have a XML library), and it should be able to export XML (and therefore XMI). When importing a XMI document, a Prolog program can generate a Prolog representation of the UML model to be transformed. The Prolog rules representing the model transformation will generate the Prolog representation of a new UML model, which is exported to a XMI document to be loaded from an UML tool. Therefore the *loading* and *creation/update* of models can be considered as separate tasks, in which a separate Prolog program is responsible for loading of XMI documents and the generation of facts, and another Prolog program is responsible of the execution of the Prolog rules and generation of new facts for creating/updating a XMI document.

Our approach contributes to the framework of model transformation with declarative languages. Declarative languages have been already used in this context in some works. One of the most relevant is [9], which describes the attempts to use several technologies for model transformation including logic programming. In particular, they use as examples the *Mercury* and *F-Logic* logic languages. The language *Tefkat* [13,12] is a declarative language whose syntax resembles a logic language with some differences (for instance, it uses *forall* construct for traversing models). The work of [22] is also a contribution to the research line by using *inductive logic programming* for deriving rules for model transformation.

Some declarative languages have been also used in the context of UML. The language *Maude* [6] has been used in several works for UML model and metamodel representation. This is the case of [4] in which structural and behavioural diagrams are integrated by means of *Maude*, and in [17] in which UML models and metamodels are formalized.

The structure of the paper is as follows. Section 2 presents the UML metamodel. Section 3 introduces the model transformation of UML models. Section 4 describes the approach of model transformation with logic programming. Finally, Section 5 concludes and presents future work.

2 UML Metamodel

The question now is how to describe model transformations. Fortunately, UML models, that is, instances of UML diagrams, can be mapped into instances of the so-called UML metamodel. The UML metamodel is a (UML) representation of

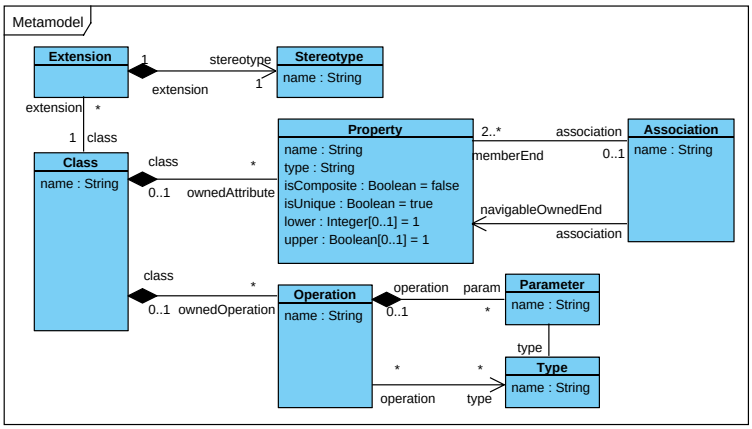

Fig. 1. UML metamodel for the UML class diagram

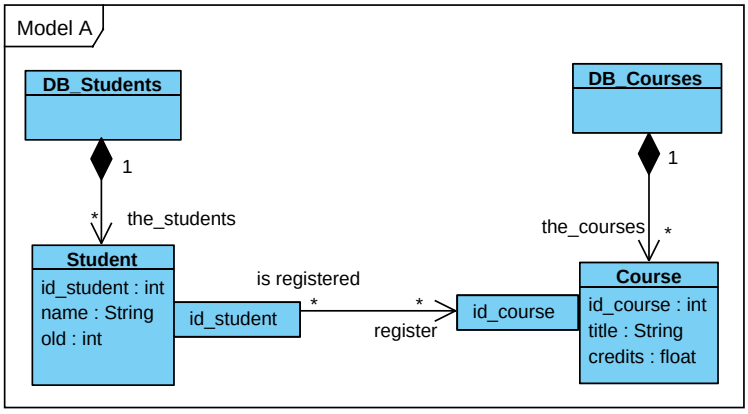

Fig. 2. Example of UML class diagram

the UML elements. As an example, the UML metamodel in Figure 1 [3] represents the elements of the UML class diagram of Figure 2. In such UML metamodel, the elements to be included in an UML class diagram are specified by means of an UML class diagram itself [4]. For instance, a *class* can include a *name* and *attributes*, represented by the *role ownedAttribute*, which belongs to the class *Property*. A class can also include *operations*, represented by the *role ownedOperation*, which can have typed *parameters* and a *type* (the returned type from the operation). Finally, a class can be linked to *extensions* which are *stereotypes*. Stereotypes can represent particular cases of classes like interfaces, tables, etc. Some of the *properties* of a *class* can represent the membership of the *class* to an association (represented by the role *memberEnd*). The role *navigableownedEnd* specifies which of the classes involved in an association are navigable. Each *property* is described by means of the *name*, the *type*, whether it is *composite* or not (whose value is true for *compositions*), whether it is unique or not (for *keys*), and the *lower* and *upper* bound of the number of elements of the associations ('0..*', '1..*','1..3', etc). Therefore, basically, the UML metamodel defines the required (the lower bound of the multiplicity is 1) and optional elements (the lower bound of the multiplicity is 0) of an UML class diagram.

[3] This is a simplified version of the UML metamodel of [15].
[4] In fact, all the UML elements are described in the UML metamodel by means of an UML class diagram.

For instance, the UML class diagram of the Figure 2 conforms to the UML metamodel in Figure 1, given that the elements included in such UML class diagram are classes (*DB_Students*, *DB_Courses*, *Student* and *Course*) including attributes (for instance, *Student* includes *id_student*, *name* and *old*) and there are three associations. The associations have the roles *the_students*, *the_courses*, *register* and *is_registered* which are *properties* of the corresponding classes in the UML metamodel. In the association between *Student* and *Course* there are two external keys (i.e. *id_student* and *id_course*) taken from *DB_Students* and *DB_Courses* containers. The association between *DB_students* and *Student* is a navigable composed association. The same can be said for the association between *DB_courses* and *Course*. Stereotypes have not been used in this UML class diagram, and the same can be said for operations.

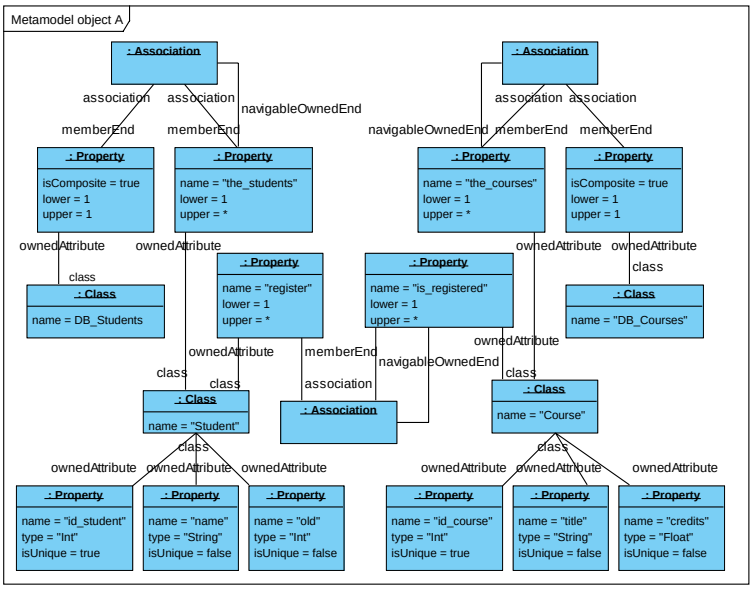

Fig. 3. Object Diagram of the Example

Now, we will show how the UML class diagram of the Figure 2, can be represented by means of an UML object diagram which is an instance of the UML metamodel of the Figure 1. It can be seen in Figure 3. From top to bottom, we can see how by means of objects, the associations with roles *the_students* and *the_courses*, and the association with roles *is_registered* and *register* are represented as links to the corresponding classes. In the bottom of the Figure 3, each class *Student* and *Course* is linked to properties representing the associated attributes. Finally, the role *navigableOwnedEnd* specifies that *the_courses*, *the_students* and *register* are navigable.

3 Model Transformation

As an example of model transformation we will consider how an UML class diagram is transformed into a new UML class diagram. The source model (Model of type A) represents a description in UML of a database (Figure 2). The target model (Model of type B) is an UML representation of a relational (tables, rows and columns) database (Figure 4).

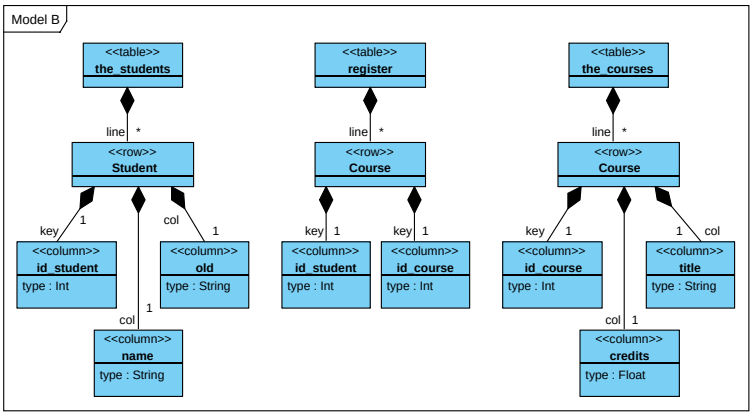

Fig. 4. Relational modeling of the Example

In the relational modeling of the example there are three tables. There are two tables called *the_students* and *the_courses* including each one three columns grouped into rows. The table of *the_students* includes for each student the associated attributes in Figure 2. The same can be said for the table *the_courses*. Moreover, there is an additional table called *register* with two columns which represents the pairs *id_student*, *id_course* for each element of the association of Figure 2. Given that the association between *Student* and *Course* is navigable from *Student* to *Course*, it is supposed that a table of pairs is generated from such association to represent the assignments of students to courses, using the role name of the association end, that is, *register*, for naming the cited table. Some of the columns play the role of (external) keys of tables which is represented by means of the role *key* in the associations of Figure 4. This is the case of *id_student* and *id_course*.

According to the UML metamodel of Figure 1, the UML model of Figure 4 can be represented as in Figure 5. *Tables* and *columns* can be represented by means of extensions of classes with the stereotype <<table>> and <<column>>. For this reason, the classes representing the tables *the_students*, *the_courses* and *register* are linked to extensions which are stereotyped as *"table"*. The same can be said for the classes representing the rows: *Student* and *Course*, and the columns: *id_course*, *id_student*, *old*, *name*, etc. The other elements, that is, *line*, *key* and *col* roles, and attributes *"type"* are represented as properties and associations.

4 Logic Programming for Model Transformation

The steps to be followed for using logic programming for model transformation are two. The first step consists in the use of Prolog facts for representing the UML metamodel and, in particular, how to instantiate the Prolog facts to represent an UML metamodel object diagram associated to an UML model. The second step consists in the use of Prolog rules for representing a model transformation.

4.1 Representing the UML metamodel in Logic Programming

We will represent the UML metamodel by means of Prolog facts. For instance, the UML metamodel of Figure 1 can be represented as follows:

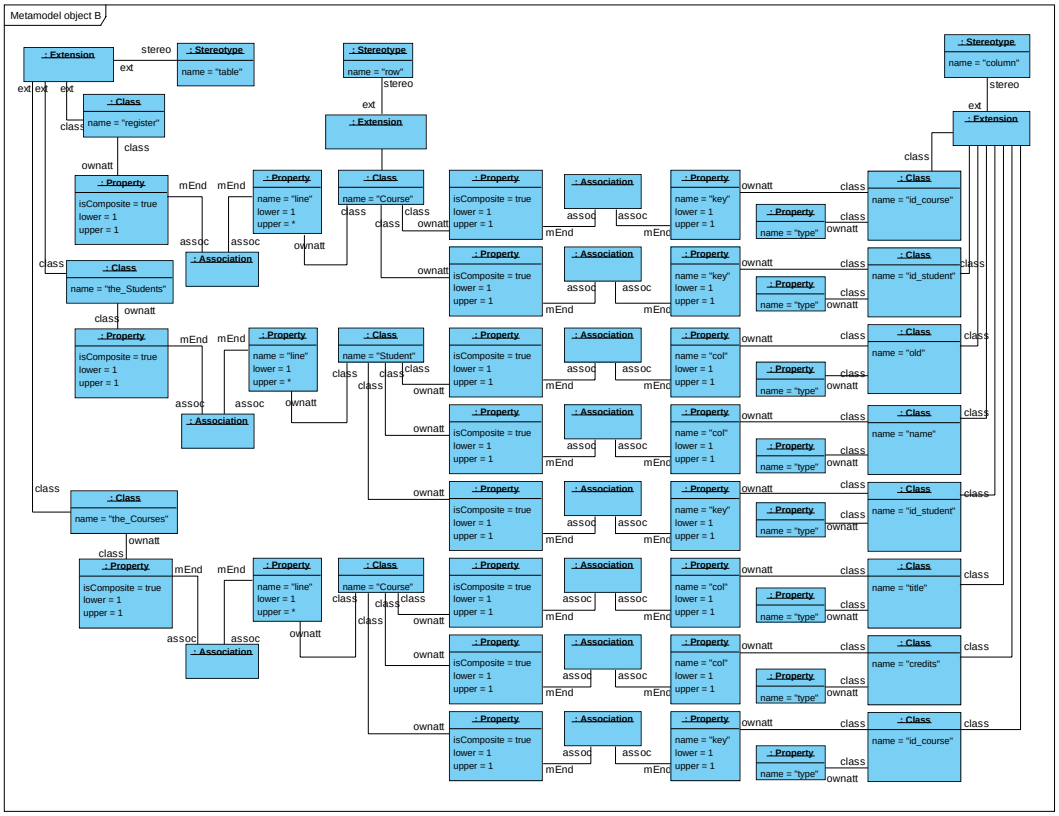

Fig. 5. Object Diagram of the Relational Modeling of the Example

```
class(Name,Id_class).
extension(Id_extension).
stereotype(Name,Id_stereotype).
property(Name,Type,IsComposite,IsUnique,Lower,Upper,Id_property).
association(Name,Id_association).
associationEnds(ownedAttribute,Id_class,Id_property).
associationEnds(extension,Id_class,Id_extension).
associationEnds(stereotype,Id_extension,Id_stereotype).
associationEnds(memberEnd,Id_association,Id_property).
associationEnds(navigableOwnedEnd,Id_association,Id_property).
...
```

where each element of the metamodel is represented by means of a fact. In particular, each class is represented by means of a fact of the form *class(Name,Id_class)* where *Name* represents the attribute "name" of each class, and *Id_class* represents an identifier for each object of type class. This identifier has to be added to the Prolog representation in order to be able to distinguish each object of the type class. We will assume that the identifier in the case of classes is the name itself. Extensions, stereotypes, associations and properties are also represented by means of facts including the attributes as parameters of the predicates, adding also an identifier for each object. The associations in the metamodel: *ownedAttribute,class,extension*, etc, are also represented by means of facts called *associationEnds* in which the first parameter indicates the role of the association, and they have two additional parameters representing (the identifiers of) each pair of objects belonging to the association.

This UML metamodel representation can be instantiated by means of an

UML metamodel object diagram. For instance, the UML metamodel object diagram of Figure 3 can be represented by means of Prolog facts as follows:

```
class('DB_Students','DB_Students').    class('Student','Student').
class('DB_Courses','DB_Courses').      class('Course','Course').
```

The above facts represent each class of the model of Figure 2 by means of a fact. Properties and classes are identified by means of their names. Associations are identified by means of a composed name built from the identifiers of the associated classes. For instance, the association between *DB_Students* and *Student* is identified by means of *DB_Students-Student*. Therefore, we have three (navigable) associations:

```
association('','DB_Students-Student').  association('','DB_Courses-Course').
association('','Student-Course').
```

Now, attributes are represented by means of properties in the metamodel, therefore in Prolog they can be represented as follows:

```
property('id_student','String',false,true,'1','1','id_student').
property('name','String',false,false,'1','1','name').
property('old','Integer',false,false,'1','1','old').
```

where each attribute is identified by means of its name itself. Now, each attribute has to be associated to each class as follows:

```
associationEnds(ownedAttribute,'Student','id_student').
associationEnds(ownedAttribute,'Student','name').
associationEnds(ownedAttribute,'Student','old').
```

representing the association *ownedAttribute* between classes and properties. In addition, the roles have to be represented in Prolog as follows:

```
property('',null,true,false,'1','1','Student-DB_Student').
property('the_students',null,false,false,'0','*','DB_Student-Student').
```

Each role is identified by means of the pairs of identifiers of the association ends. Now, each member of each association has to be annotated by means of a Prolog fact:

```
associationEnds(memberEnd,'DB_Students-Student','DB_Students-Student').
associationEnds(memberEnd,'DB_Students-Student','Student-DB_Student').
```

Finally, the navigability has to be also represented in Prolog as follows:

```
associationEnds(navigableownedEnd,'DB_Students-Student','DB_Students-Student').
```

4.2 Rules for Model Transformation

Now, we will show how to define Prolog rules for transforming the UML model of Figure 2 into the UML model of Figure 4. In particular, our transformation technique takes the facts representing the model of Figure 2, obtaining a set of facts representing the model of Figure 4. The transformation rules work on the representation of the UML metamodel by means of facts. In general, and as already mentioned, the rules can be classified into *model query rules* and *transformation rules*. The first kind queries the Prolog representation of the source model and defines new elements of the new diagram. The second kind translates the new elements to the same kind of representation of the source model, in order to obtain the target model.

4.2.1 Model Query Language

The model query rules define the new elements of the target model. For instance, in the running example, they can be defined as follows:

```
table(Name,Id_table):-property(Name,_,_,_,_,'*',Id_table),
                      associationEnds(navigableownedEnd,_,Id_table).
row(Name,Id_table,Id_row):-class(Name,Id_row),property(Name,_,_,_,_,'*',Id_table),
                      associationEnds(navigableownedEnd,Id_table,Id_row).
column(Id_column,Id_row,Id_column):-row(_,Id_row),
                      associationEnds(ownedAttribute,Id_row,Id_column).
```

The first rule specifies the names and identifiers of the tables. Tables are navigable properties of the source model whose multiplicity is unbounded (in the example, *the_students*, *the_courses* and *register*). The second rule specifies the name and identifier of each row, together with the identifier of the table to which it belongs. In the running example, the rows are *Student* and *Course*, the first belonging to *the_students*, and the second belonging to *register* and *the courses*. Finally, the third rule specifies the name and identifier of each column, together with the identifier of the row to which it belongs. In the running example, for instance, *id_student*, *name* and *old* belongs to the *Student* row. Now, the attributes of the model of Figure 4 can be specified by means of the following rules:

```
attribute('type',Type,Id_class,Id_attribute):-property(_,Type,_,_,_,_,Id_attribute),
            column(_,_,Id_class),associationEnds(ownedAttribute,Id_class,Id_attribute).
```

The attributes of the model of Figure 4 have the form "type:Type" for each "Type" of an attribute of a column. Now, the associations of the model of Figure 4 can be specified as follows:

```
associationlink('line',Id_table,Id_row,'1','1','0','*'):-table(_,Id_table),
                                  row(_,Id_table,Id_row).
associationlink('key',Id_row,Id_column,'1','1','1',1):-column(_,Id_row,Id_column),
                                  property(_,_,_,true,'1','1',Id_column).
associationlink('col',Id_row,Id_column,'1','1','1',1):-column(_,Id_row,Id_column),
                                  property(_,_,_,false,'1','1',Id_column).
```

The association roles are *line*, *key* and *col*. The first association represents the link between tables and rows. The second association represents the link between the row and the column in the case of keys (i.e. *isUnique* is equal to true). Finally, the third association represents the link between the row and the column in the case of non keys.

4.2.2 Transformation Language

The second kind of rules, the transformation rules, translate the new elements, defined by means of the model query language into the same kind of representation as the source model.

The new classes (facts called *class2*) can be obtained from tables, rows and columns. In addition, a new stereotype is added to each class which is an extension to the class according to the UML metamodel. Extensions are identified by means of the name of the table (i.e. the name of the class) [5]. Stereotypes are identified by means of the name of the stereotype. New associations between the classes and the extensions, and between the extensions and the stereotypes are added:

```
class2(Name,Id_table):-table(Name,Id_table).
stereotype2('table','table').
associationEnds2(extension,Id_table,
       Id_table):-table(_,Id_table).
```

```
extension2(Id_table):-table(_,Id_table).


associationEnds2(stereotype,Id_table,
       'table'):-table(_,Id_table).
```

Now, new properties are added, including attribute and association roles as follows:

[5] In the running example, each class has at least one extension.

```
property2(Name,Type,false,false,'1','1',Id_attribute):-attribute(Name,Type,_,Id_attribute).
property2(Name,null,false,false,Lower2,Upper2,Id_association):-
        associationlink(Name,_,Id_link1,Id_link2,_,_,Lower2,Upper2),
        atom_concat(Id_link2,'-',Aux),atom_concat(Aux,Id_link1,Id_association).
```

Rules for linking attributes to classes, and rules for specifying associations can be defined as follows:

```
associationEnds2(ownedAttribute,Id_class,Id_attribute):-attribute(_,_,Id_class,Id_attribute).
association2(Id_association):- associationlink(_,_,Id_link1,Id_link2,_,_,_,_),
        atom_concat(Id_link1,'-',Aux),atom_concat(Aux,Id_link2,Id_association).
```

Associations are identified by concatenating the name of the associated classes. Finally, membership to associations and navigability (in the running example all associations are navigable) can be specified as follows:

```
associationEnds2(memberEnd,Id_association,Id_link1):-associationlink(_,_,Id_link1,Id_link2,_,
        _,_,_),atom_concat(Id_link1,'-',Aux),atom_concat(Aux,Id_link2,Id_association).
associationEnds2(navigableOwnedEnd,Id_association,Id_link):-
        associationEnds2(memberEnd,Id_association,Id_link).
```

4.2.3 Loading and Creation/Update of Models

In order to use the previous rules for obtaining new facts representing the object diagram of the Figure 5, a Prolog interpreter can be used. However, the interpreter has to be modified as follows. Prolog can be used for computing new facts from a set of rules by implementing a *bottom-up interpreter* (see for instance [7]).

Alternatively, each predicate of the metamodel (*class*,*stereotype*, *property*, etc) can be called as goal (with variables) and each *computed answer* represents a fact [6]. In the running example, we would obtain the following set of facts, among others, for the case of the first table:

```
class2('the_students','the_students').  class2('Student','Student').
extension2('the_students').  extension2('Student').
stereotype2('table','table').  stereotype2('row','row').
associationEnds2(stereotype,'the_students','table').
associationEnds2(extension,'the_students','the_students').
associations2('the_students-Student').
associationEnds2(memberEnd,'the_students-Student','the_students').
associationEnds2(navigableOwnedEnd,'the_students-Student','Student').
property2('line',null,false,false,'0','*','Student-the_students').
property2('type','int',false,false,'1','1','id_student').
```

5 Conclusions and Future Work

We have studied how to use logic programming for model transformation in the context of UML. We have described how to apply our proposed technique to transform an UML class diagram representing a database into an UML representation of a relational database. We will study several extensions of our work in the future. Firstly, we are interested in the implementation of the proposed technique: loading of XMI documents into facts, execution of rules for generating new facts, exporting of new facts into XMI documents. In addition, we will study how to use logic programming for model transformation of other kinds of UML diagrams. In particular, we are interested in the use of our approach for generating user interfaces from use case and state diagrams. We have provided the basis for such model transformation

[6] The predicate **assert** available in most Prolog implementations can be used for automating the process.

in previous works [3,2]. Finally, we believe that the integration of UML models and logic programming can lead to the development of a logic based tool for verification and validation of UML models and transformations.

References

[1] A. Agrawal. Graph rewriting and transformation (GReAT): a solution for the model integrated computing (MIC) bottleneck. *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 364–368, 6-10 Oct. 2003.

[2] J. M. Almendros-Jiménez and L. Iribarne. Designing GUI Components for UML Use Cases. In *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)*, pages 210–217. IEEE Computer Society Press, 2005.

[3] J. M. Almendros-Jiménez and L. Iribarne. An Extension of UML for the Modeling of WIMP User Interfaces. *Journal of Visual Languages and Computing, Elsevier, in press*, 2008.

[4] N. Aoumeur and G. Saake. Integrating and Rapid-Prototyping UML Structural and Behavioural Diagrams Using Rewriting Logic. In *Procs. of CAiSE*, pages 296–310. LNCS 2348, Springer, 2002.

[5] A. Balogh and D. Varró. The Model Transformation Language of the VIATRA2 Framework. *Science of Programming*, 68(3):187–207, October 2007.

[6] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The Maude 2.0 System. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 76–87. Springer-Verlag, June 2003.

[7] Michael Codish. Efficient goal directed bottom-up evaluation of logic programs. *J. Log. Program.*, 38(3):355–370, 1999.

[8] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *OOPSLA03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[9] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The Missing Link of MDA. In *ICGT '02: Proceedings of the First International Conference on Graph Transformation*, pages 90–105, London, UK, 2002. LNCS 2505, Springer.

[10] F. Jouault and I. Kurtev. On the architectural alignment of ATL and QVT. In *SAC '06: Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1188–1195, New York, NY, USA, 2006. ACM.

[11] Frédéric Jouault and Ivan Kurtev. On the interoperability of model-to-model transformation languages. *Sci. Comput. Program.*, 68(3):114–137, 2007.

[12] M. Lawley and J. Steel. Practical Declarative Model Transformation with Tefkat. In *MoDELS Satellite Events*, pages 139–150. LNCS 3844, Springer, 2006.

[13] Michael Lawley and Kerry Raymond. Implementing a practical declarative logic-based model transformation engine. In *SAC '07: Proceedings of the 2007 ACM Symposium on Applied Computing*, pages 971–977, New York, NY, USA, 2007. ACM.

[14] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electr. Notes Theor. Comput. Sci.*, 152:125–142, 2006.

[15] OMG. Unified Modeling Language Specification, version 2.0. Technical report, Object Management Group, 2005.

[16] OMG. Object Management Group. Technical report, www.omg.org, 2008.

[17] J. R. Romero, J. E. Rivera, F. Durán, and A. Vallecillo. Formal and tool support for model driven engineering with Maude. *Journal of Object Technology*, 6(9):187–207, October 2007.

[18] J. Sánchez-Cuadrado, J. García-Molina, and M. Menárguez-Tortosa. RubyTL: A Practical, Extensible Transformation Language. In *Procs of Model Driven Architecture - Foundations and Applications*, pages 158–172. LNCS 4066, Springer, 2006.

[19] Gabriele Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *AGTIVE*, pages 446–453, 2003.

[20] Laurence Tratt. Model transformations and tool integration. *Software and System Modeling*, 4(2):112–122, 2005.

[21] Laurence Tratt. The MT model transformation language. In *SAC '06: Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1296–1303, New York, NY, USA, 2006. ACM.

[22] D. Varró and Z. Balogh. Automating model transformation by example using inductive logic programming. In *SAC '07: Proceedings of the 2007 ACM Symposium on Applied Computing*, pages 978–984, New York, NY, USA, 2007. ACM.