ODM-based UML Model Transformations using Prolog *

Jesús M. Almendros-Jiménez and Luis Iribarne

Dpto. de Lenguajes y Computación University of Almería
04120-Spain
{jalmen,luis.iribarne}@ual.es

Abstract. In this paper we present a framework for the specification of model transformations by means of Prolog rules, using the ODM representation of UML models. In addition, Prolog rules are also used for the validation of source and target models w.r.t. their ODM based metamodels. We have validated our proposal by means of a prototype developed under SWI-Prolog.

1 Introduction

Model Driven Engineering (MDE) is an emerging approach for software development. MDE emphasizes the construction of models from which the implementation should be derived by applying model transformations. Hence, *Model transformation* [35, 20] is a key tool of MDE. According to the *Model Driven Architecture (MDA)* [29] initiative of the *Object Management Group (OMG)* [27], model transformation provides to developers tools for transforming their models.

The MDA approach proposes three elements in order to describe a model transformation: the first one is the so-called *meta-meta-model* which is the basis of the model transformation and provides the language for describing meta-models. The second one consists in the *meta-models* of the models to be transformed. Source and target models must conform to the corresponding meta-model. Such meta-models are modeled according to the meta-meta-model. The third one consists in the source and target *models*. Source and target models are instances of the corresponding meta-models. In addition, source and target meta-models are instances of the meta-meta-model.

Therefore, in order to define a model transformation we should be able to meta-model the source and target models w.r.t. the meta-meta-model, and map source and target meta-models. In this context, model transformation needs formal techniques for specifying the transformation. In particular, in most of the cases transformations can be expressed by means of some kind of *rules*. The rules have to express how any *source model* can be transformed into a *target model* in a given transformation.

On the other hand, the *Ontology Definition Metamodel (ODM)* proposal [30] of the *OMG* aims to define an ontology-based representation of UML models. ODM is

* This work has been supported by the Spanish Ministry MICINN and Ingenieros Alborada IDI under grant TRA2009-0309. This work has been also supported by the EU (FEDER) and the Spanish Ministry MICINN under grants TIN2010-15588, TIN2008-06622-C03-03, and the JUNTA ANDALUCIA (proyecto de excelencia) ref. TIC-6114.

an standard for representing UML models by means of an ontology in which, among others, UML classes are mapped into ontology concepts, UML associations are mapped into ontology roles, and multiplicity restrictions of UML are mapped into cardinality restrictions in roles. ODM is itself an UML meta-model in which UML models can be accommodate. Following the ODM proposal, an UML model can be represented by means of an ontology in which the **TBox** contains the UML meta-model and the **ABox** contains the instance of the UML meta-model which represents the model.

The relationship between logic programming and ontologies is well-known. *OWL* [36], the most prominent ontology language is based on *Description Logic (DL)* [8], a fragment of *first order logic*, and some fragments of *DL* can be encoded into logic programming, for instance, the so-called *Description Logic Programming* approach [15]. Typically, Description Logic is used for representing a **TBox** (*terminological box*) and the **ABox** (*assertional box*). The **TBox** describes concept (and role) *hierarchies* (i.e., relations between concepts and roles) while the **ABox** contains relations between individuals, concepts and roles. Therefore we can see the **TBox** as the meta-data description, and the **ABox** as the description about data. The encoding of (fragments of) DL into logic programming are based on the representation of the **TBox** by means of Prolog rules and the representation of the **ABox** by means of Prolog facts. It means that any instance of an ontology in this context can be represented by means of Prolog facts.

In model transformation a transformation maps the source model into the target model. Therefore a transformation maps the **ABox** of the source model into the **ABox** of the target model. Given that the encoding of a Description logic fragment into logic programming maps the **ABox** of the ontology into Prolog facts, then model transformation consists on the mapping of Prolog facts. Therefore, model transformation can be defined using Prolog rules.

In this paper we present a framework for the specification of model transformations by means of Prolog rules, using the ODM representation of UML models. In addition, Prolog rules are used for the validation of source and target models. We have validated our proposal by means of a prototype developed under SWI-Prolog. The prototype together with the case study can be downloaded from `http://indalog.ual.es/mdd`. Our approach will be applied to a well-known example of model transformation in which an UML class diagram representing a database (as an entity-relationship diagram) is transformed into an UML diagram representing a relational database.

Moreover, OWL is a formalism for knowledge representation in which reasoning (consistence checking and querying) has been largely studied. Recently, a fragment of OWL, named OWL RL [26] has been proposed and a set of rules have been defined for it in order to provide rule based reasoning. Such rules can be implemented in Prolog. In a previous work [2] we have implemented such rule system in the SWI-Prolog interpreter. Now, we have integrated such reasoning with our current proposal as follows. The source and target meta-models of a certain transformation can include constraints that cannot be specified by means of an UML-based meta-model, rather than they can be specified by means of OWL RL. Therefore, we are interested to validate source and target models against their corresponding meta-models by means of OWL RL reasoning.

Therefore, the advantages of this approach are the declarative nature of the transformation specification, the use of a standardized language (Prolog), the ability of executing the transformation and automatically check the well-formedness constraints on the source and target models. With respect to model transformation verification/validation, our approach is limited to validation of source and target models. In addition, our approach is limited to ODM expressivity power and OWL RL reasoning capabilities. We believe that we could extend our work to a more general framework of verification/validation of model transformations in the line of [10], thanks to the logic nature of our proposal.

The structure of the paper is as follow. Section 2 will introduce the model transformation framework and will describe a case study of transformation. Section 3 will present the Prolog-based approach. Section 4 will show model validation by means of OWL RL. Section 5 will discuss related work. Finally, Section 6 will conclude and present future work.

2 Model Transformation

The elements to be considered in a given ontology based transformation using Prolog as transformation language can be summarized as follows:

- Firstly, we have to consider the *meta-model of the source model* which defines the elements occurring in source model. Any given instance of the defined meta-model is transformed by applying the transformation rules.
- Secondly, we have to consider the *meta-model of the target model* which defines the elements occurring in target model. Any instance of the source meta-model is transformed into an instance of the meta-model of the target model.
- Finally, we have to define *Prolog rules* for transforming any instance of the source meta-model into an instance of the target meta-model. The instance of the target meta-model defines the target model of the transformation.

The contribution of Prolog in this approach is the use of a well-known rule language for expressing transformations. The question now is, how to express transformations in Prolog? Our proposal is as follows.

- The ODM proposal provides a representation of UML models by means of an ontology. The **TBox** represents the meta-model and the **ABox** represents properly the model. We can represent an **ABox** in Prolog by means of facts.
- In particular, the source model is defined by means of an instance of the source meta-model and therefore a source model can be mapped into a set of Prolog facts.
- Now, a transformation between a source model and a target model can be seen as a transformation of the set of Prolog facts of the source model into a set of Prolog facts representing the target model.
- Our proposal is to use Prolog for defining rules for transforming Prolog facts representing UML models.

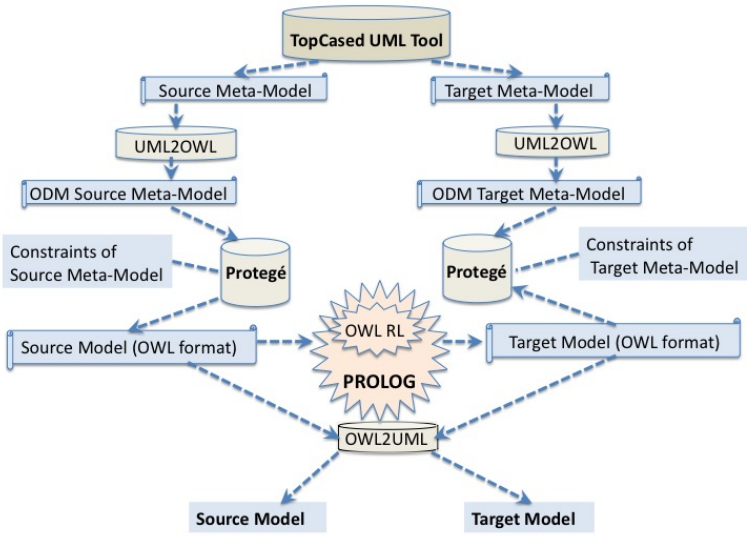

Fig. 1. Integration with UML/OWL tools

Our approach has been implemented and tested with some examples. We have integrated our approach with other UML and OWL tools (see Figure 1). We have used the *TopCased* UML tool [34] for designing the source and target meta-models. In addition, we have used a UML2OWL transformer (available from [17]) in order to have the ODM-based representation of source and target meta-models. We have also used the *Protegé* tool [21] for defining the instance of the source meta-model, and for exporting the source model (i.e. meta-model+instance) into an OWL document. *Protegé* is also used for defining constraints of the source meta-model. Such constraints are expressed by means of OWL RL. Then, the SWI-Prolog interpreter is used for validating the source model, and for transforming the instance of the source model into the instance of the target model. Once the target model is computed, *Protegé* can be used for defining constraints on the target meta-model and SWI-Prolog is used for validating the target model. After, the *Protegé* tool is also used for exporting the target model together with the target meta-model to a OWL document. Finally, an OWL2UML transformer has been used for obtaining the target model from the ODM-based representation.

2.1 Case Study

As an example of model transformation we will consider two UML profiles for database design and we will describe how to transform the first profile into the second one.

The model of Figure 2 represents the modeling of a database by means of UML. We will call to this kind of modeling, the *"entity-relationship"* modeling of a database in contrast to the model of Figure 3 which will be called *"relational"* modeling of a

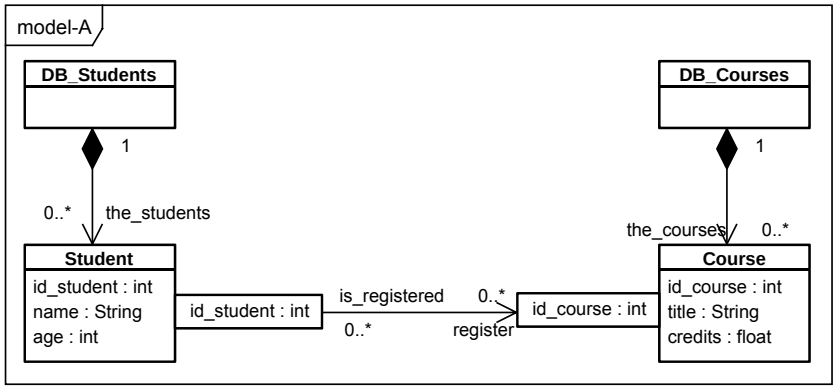

Fig. 2. Entity-relationship modeling of the Case Study

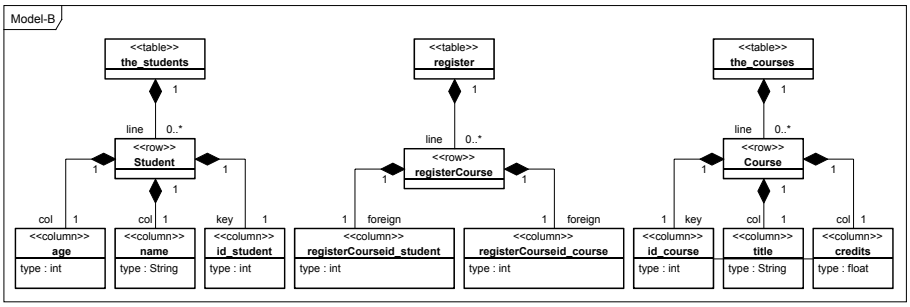

Fig. 3. Relational modeling of the Case Study

database. The proposed UML profile for entity-relationship modeling aims to adapt the UML class diagram to the traditional entity-relationship model, with some extensions for object-oriented modeling. The proposed UML profile consists of the following elements:

- Entities are represented by means of UML classes (i.e., *Student* and *Course*).
- A container is defined for each entity (i.e., *DB_Students* and *DB_Courses*). The container is responsible of the storing of the objects of the entity. Containers are not usual in the entity-relationship model, however in an object-oriented approach the objects belonging to an entity have to be stored in a container. Our UML profile assumes that containers are unique for each entity.
- Relationships are represented by means of UML associations. Relation names are association names. In addition, the proposed UML profile allows to define role names to each end of the associations (i.e., *is_registered* and *register*).
- Entity attributes are class attributes. Each entity *has key attributes* (one or more).

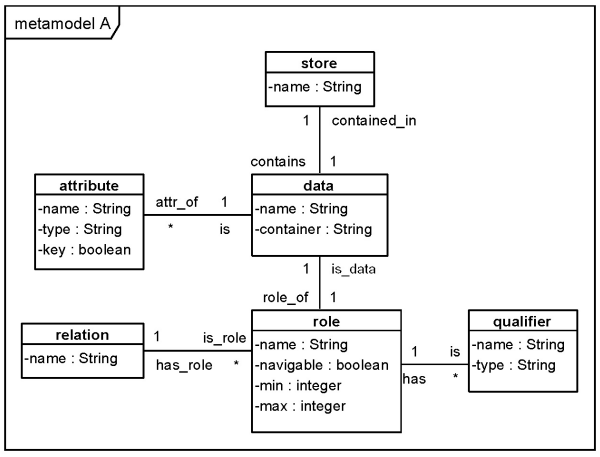

Fig. 4. Meta-model of the Source Models

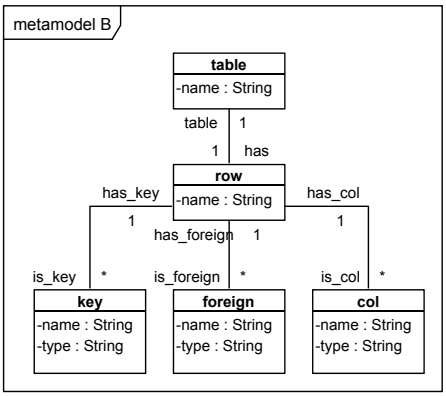

Fig. 5. Meta-model of the Target Models

– Associations can be adorned in our UML profile for entity-relationship modeling with qualifiers. The role of qualifiers in our UML profile is to specify the key attributes of each entity used as foreign keys of the corresponding association. A constraint in our UML profile is that *qualifiers have to be selected from the key attributes* in the corresponding entity.
– Navigability can be specified in our profile. In a relational model based implementation of the profile, one table should be implemented for each container and one table for each navigable association.

Figure 3 shows the relational modeling of the same database. Such modeling also defines an UML class diagram based profile for database design. It introduces the fol-

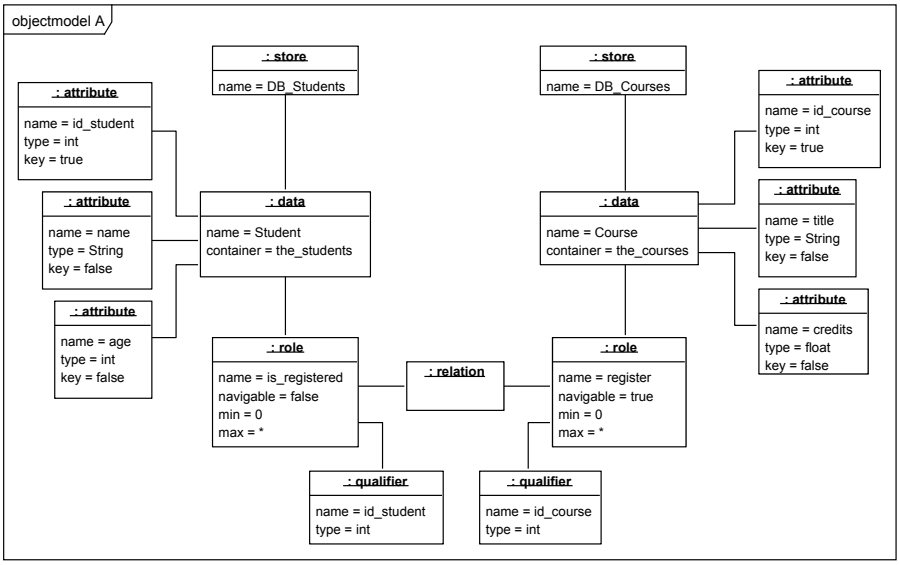

Fig. 6. Object Model of Source Model

lowing new stereotypes: $<<$ *table* $>>$, $<<$ *row* $>>$ and $<<$ *column* $>>$ for specifying tables, rows of tables and columns of tables. In addition, *line* is used as role name of each row in the table, *key* is used as role for key attributes in rows, *foreign* is used as role for foreign keys in rows, and *col* is used for non keys and non foreign keys in rows. Finally, each column has an attribute called *type*. A constraint of the target meta-model is that *foreign keys have to be keys* of other tables.

Both profiles can be meta-modeled by means of UML. Figures 4 and 5 represent the meta-modeling of the profiles. In the first case, *DB_Students* and *DB_Courses* are instances of the class *store*, *Student* and *Course* are instances of the class *data*, the attributes of *Student* class and *Course* class are instances of the class *attribute*, and the relationships between entities are represented in the meta-model by means of the classes *relation*, *role* and *qualifier*. In the second case, tables and rows of the target model are instances of the corresponding classes, and the same can be said from *key*, *col* and *foreign* classes. 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 meta-model of the Figure 4. It can be seen in Figure 6. The UML model of Figure 3 can be also represented as an UML object diagram (see Figure 7).

Now, the problem of model transformation is how to transform an UML class diagram of the type A (as Figure 2) into an UML class diagram of type B (as Figure 3). The transformation is as follows. The transformation generates two tables called *the_students* and *the_courses* including each one three columns grouped into rows. The table *the_students* includes for each student the attributes of *Student* of Figure 2. The same can be said for the table *the_courses*. Given that the association between *Student*

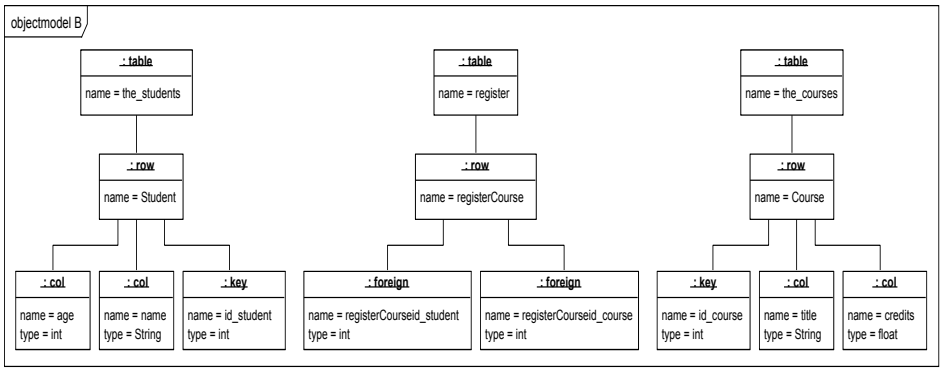

Fig. 7. Object Model of Target Model

and *Course* is navigable from *Student* to *Course*, a table of pairs is generated to represent the assignments of students to courses, using the role name of the association end, that is, *register* concatenated with *Course*, for naming that table. The columns *id_student* and *id_course* taken from qualifiers, play the role of foreign keys which are represented by means of the role *foreign* in the associations of Figure 3.

The transformation can be considered as a transformation between object diagrams of source and target meta-models. Therefore, a transformation should be able to define a set of rules from which an instance of the target meta-model is obtained from the instance of the source meta-model. Our proposal is to use logic programming (in particular, Prolog) for defining such transformation.

3 Prolog for Model Transformation

In this section, we will show how Prolog can be used for defining transformation rules in our approach. In order to adapt Prolog to such context, we have to consider the following elements:

- The Prolog interpreter has to import and export OWL files. This is the case of SWI-Prolog which includes a library for importing and exporting RDF(S)/OWL triples. The SWI-Prolog library stores RDF triples in a database, and they can be retrieved by means of a predicate called `rdf`. The RDF library includes predicates: `rdf_reset_db/0` which resets the database, `rdf_load(+File,+Options)` for importing triples, `rdf_save(+File)` for exporting triples, and finally, `rdf_assert(+Subject,+Property,+Object)` for inserting a new triple in the current database.
- A Prolog predicate `transform(+SourceModelFile,+TargetModelFile)` is defined for transforming a source model (stored in a OWL file) into a target model (stored also in a OWL file). The Prolog code of such predicate is as follows:

```
transform(_,_):-rdf_reset_db,fail.
transform(_,_):-retractall(new(_,_,_)),fail.
transform(FileIn,_):-rdf_load(FileIn,[]),fail.
transform(_,_):-newrdf(A,B,C),assert(new(A,B,C)),fail.
transform(FileIn,_):-rdf_reset_db,fail.
transform(_,_):-new(A,B,C),rdf_global_term(B,D),rdf_assert(A,D,C),fail.
transform(_,FileOut):-rdf_save(FileOut),rdf_reset_db.
```

– The source model is stored in Prolog and it can be retrieved by means of the `rdf` predicate. The transformation rules define new triples representing the target model. A new predicate called `newrdf` is defined in the transformation rules.

3.1 Transformation of the Case Study

In order to show how Prolog rules are defined in our approach for model transformation, we will consider the case study of Section 2.1 in which the model A (see Figure 2) is transformed into the model B (see Figure 3). Now, the model A can be represented by means of an ontology in which the **TBox** contains the meta-model of Figure 4, together with a **ABox** with the instance of Figure 6. The transformation has to map the **ABox** of the model A to the **ABox** of the model B (see Figure 6). For instance, the OWL document containing the **ABox** of the model A defines, among others, the following elements:

```
<store rdf:about="#01_DB_Students_store">
<store.name rdf:datatype="&xsd;string">DB_Students
</store.name>
<store.contains rdf:resource="#02_Student_data"/>
</store>

<data rdf:about="#02_Student_data">
<data.name rdf:datatype="&xsd;string">Student
</data.name>
<data.container rdf:datatype="&xsd;string">the_students
</data.container>
<data.contained_in rdf:resource="#01_DB_Students_store"/>
<data.attr_of rdf:resource="#03_id_student_attribute"/>
<data.attr_of rdf:resource="#04_name_attribute"/>
<data.attr_of rdf:resource="#05_age_attribute"/>
<data.role_of rdf:resource="#06_is_registered_role"/>
</data>
```

which represent the instances *DB_Student* and *Student* of the classes *store* and *data*, respectively. Now, the OWL document of the source model can be imported from Prolog and it can be retrieved by means of the Prolog predicate `rdf`. For instance:

```
Subject='http://metamodelA.ecore#02_Student_data'
Property='http://metamodelA.ecore#data.name'
Object=literal(type('http://www.w3.org/2001/
XMLSchema#string', 'Student'))
```

is obtained by means of the Prolog goal `:-rdf(Subject,Property,Object).` Now, the predicate `rdf` can be used in the transformation rules to define the triples of the target model. The transformation rules define a new predicate `newrdf`. For instance, the following rules define the identifiers of the elements of the class *table* of the model B:

```
newrdf(IdTable,rdf:type,'http://metamodelB.ecore#table'):-
    rdf(IdData,rdf:type,'http://metamodelA.ecore#data'),
    atom_concat(IdData,'table',IdTable).
newrdf(IdTable,rdf:type,'http://metamodelB.ecore#table'):-
    rdf(_,'http://metamodelA.ecore#data.role_of',IdRole),
    rdf(IdRole,'http://metamodelA.ecore#role.navigable',Navigable),
    Navigable=literal(type(_,true)),
    atom_concat(IdRole,'table',IdTable).
```

The first rule defines triples `(IdTable,rdf:type,'http://metamodel-B.ecore#table')` obtained from triples `(IdData,rdf:type,'http://me-tamodelA.ecore#data')`, where `IdTable` is the identifier of the table, which is generated concatenating the word `'table'` to `IdData`. The second rule defines the identifiers of tables obtained from navigable roles, which are generated concatenating the word `'table'` to the identifier of the role. In such a way that the following Prolog goal obtains the tables of the target model:

```
?- newrdf(IdTable,rdf:type,'http://metamodelB.ecore#table').
IdTable = 'http://metamodelB.ecore#02_Student_datatable' ;
IdTable = 'http://metamodelB.ecore#09_Course_datatable' ;
IdTable = 'http://metamodelB.ecore#13_register_roletable' ;
```

which represent the identifiers of the tables *Student*, *Course* and *register* of the Figure 7. Now, the identifiers of the rows of Figure 7 can be defined as follows:

```
newrdf(IdRow,rdf:type,'http://metamodelB.ecore#row'):-
    rdf(IdData,rdf:type,'http://metamodelA.ecore#data'),
    atom_concat(IdData,'row',IdRow).
newrdf(IdRow,rdf:type,'http://metamodelB.ecore#row'):-
    rdf(IdData,'http://metamodelA.ecore#data.role_of',IdRole),
    rdf(IdRole,'http://metamodelA.ecore#role.navigable',Navigable),
    Navigable=literal(type(_,true)),
    rdf_split_url(_,Pointer,IdData),
    atom_concat(IdRole,Pointer,Id),
    atom_concat(Id,'row',IdRow).
```

The first rule defines the identifiers of rows obtained from instances of data (i.e., the identifiers of *the_courses* and *the_students*), and the second rule defines the identifiers of rows obtained from navigable roles (i.e., the identifier of *registerCourse*). Now, *key*, *col* and *foreign* elements have to defined. For instance, the identifiers of instances of the *foreign* class are defined as follows:

```
newrdf(Id,rdf:type,'http://metamodelB.ecore#foreign'):-
    rdf(_,'http://metamodelA.ecore#data.role_of',IdRole),
    rdf(IdRole,'http://metamodelA.ecore#role.navigable',Navigable),
    Navigable=literal(type(_,true)),
    rdf(IdRole,'http://metamodelA.ecore#role.is',IdQualifier),
    rdf(IdData,'http://metamodelA.ecore#data.role_of',IdRole),
    rdf_split_url(_,Pointer1,IdData),
    rdf_split_url(_,Pointer2,IdQualifier),
    atom_concat(IdRole,Pointer1,Id1),
    atom_concat(Id1,Pointer2,Id2),
    atom_concat(Id2,'foreign',Id).
```

In this case, instances of the *foreign* class are obtained from navigable roles, using the identifier of the qualifier and the identifier of the role to generate the identifier. Now, the association roles of the Figure 6 have to be defined. For instance, the role *has* from the class *table* of Figure 4 is defined as follows:

```
newrdf(Id,'http://metamodelB.ecore#table.has',IdRow):-
    rdf(IdData,rdf:type,'http://metamodelA.ecore#data'),
    atom_concat(IdData,'table',Id),
    atom_concat(IdData,'row',IdRow).
newrdf(Id,'http://metamodelA.ecore#table.has',IdRole):-
    rdf(IdData,'http://metamodelA.ecore#data.role_of',IdRole),
    rdf(IdRole,'http://metamodelA.ecore#role.navigable',Navigable),
    Navigable=literal(type(_,true)),
    rdf_split_url(_,Pointer,IdData),
    atom_concat(IdRole,'table',Id),
    atom_concat(IdRole,Pointer,Id),
    atom_concat(Id,'row',IdRow).
```

The first rule defines the rows of tables obtained from instances of *data*, and the second rule defines the rows of tables obtained from navigable roles. Now, the role *is_col* from *row* of Figure 4 is defined as follows:

```
newrdf(IdRow,'http://metamodelB.ecore#row.is_col',IdCol):-
    rdf(IdData,rdf:type,'http://metamodelA.ecore#data'),
    atom_concat(IdData,'row',IdRow),
    rdf(IdData,'http://metamodelA.ecore#data.attr_of',IdAtt),
    rdf(IdAtt,'http://metamodelA.ecore#attribute.key',Key),
    Key=literal(type(_,false)),
    atom_concat(IdAtt,'col',IdCol).
```

in which columns are obtained from non key attributes. Finally, attributes of classes of Figure 4 have to be defined. For instance, *name* of class *table* is defined as follows:

```
newrdf(IdTable,'http://metamodelB.ecore#table.name',ContName):-
    rdf(IdData,'http://metamodelA.ecore#data.container',ContName),
    atom_concat(IdData,'table',IdTable).
```

where the table names are obtained from container names (i.e., *the_students* and *the_courses*). In addition, column names (i.e., *age*, *name*, *title* and *credits*) are obtained from non key attribute names:

```
newrdf(IdCol,'http://metamodelB.ecore#col.name',AttName):-
    rdf(IdData,rdf:type,'http://metamodelA.ecore#data'),
    rdf(IdData,'http://metamodelA.ecore#data.attr_of',IdAttribute),
    rdf(IdAttribute,'http://metamodelA.ecore#attribute.name',AttName),
    rdf(IdAttribute,'http://metamodelA.ecore#attribute.key',Key),
    Key=literal(type(_,false)),
    atom_concat(IdAttribute,'col',IdCol).
```

4 Model Validation

Finally, we would like to show how to validate source and target models by means of OWL RL. There are some constraints on source and target models that could not be specified by means of the UML-based meta-model. Such constraints can be specified by means of the OWL RL in order to validate the source and target models w.r.t. their corresponding meta-model. For instance, in our case study we have the following constraints:

(1) *Data* entities of the source meta-model have at least a key.
(2) *Qualifier* entities of source models have to be selected from keys of *Data* entities.
(3) *Foreign* keys of the target models have to be *keys* of other tables.

Such constraints can be checked by means of OWL RL by adding the following formulas in Description Logic:

(1) $KeyAtt(id\_student)$, $KeyAtt(id\_course)$
(2) $Qualifier \sqsubseteq KeyAtt$
(3) $Foreign \sqsubseteq Key$

where *KeyAtt* is a new concept of the source meta-model defined as the elements having the attribute *key* equal to *true*: $KeyAtt \equiv \exists\ key.\ \{true\}$ and $KeyAtt \sqcap \exists key.\{false\} = \bot$.

Now, we can use the OWL RL reasoner for validating source and target models. For instance, let us suppose that in the source model the *Data* entity *Student* has *id_student* attribute with *key* value set to *false*, and declared of type *KeyAtt*. Then, the OWL RL reasoner shows as output the following message:

```
Warning: Disjoint Classes
Element:
http://metamodelA.ecore#03_id_student_attribute
Class:
http://metamodelA.ecore#keyatt
Class Restriction:
Property:
http://metamodelA.ecore#attribute.key
Has Value:
literal(type(http://www.w3.org/2001/XMLSchema#boolean, false))
```

The meaning of such message is that an element (i.e., *id_student*) has been found in the intersection of *KeyAtt* and $\exists key.\{false\}$ classes. Let us now suppose that a certain *Qualifier*, for instance, *id_course* is not selected from *KeyAtt*. Then the OWL RL reasoner shows the following message:

```
Warning: Same and Different Individuals
http://metamodelA.ecore#10_id_course_attribute
http://metamodelA.ecore#10_id_course_attribute
```

The previous message adverts that *id_course* is equal and different from itself, given that the reasoner tries to make it equal to *id_course_qualifier*.

5 Related Work

Several transformation languages and tools have been proposed in the literature (see [13] for a survey). In the context of *MOF (Meta Object Facility)* meta-modeling architecture, the *QVT (Query-View-Transformation)* language [28, 23] has been proposed as standard for model transformation. QVT is actually a family of languages: the relations and core languages which are declarative, and the operational mapping language which is an imperative language. The language *ATL (ATLAS transformation language)* [19, 18] is a domain-specific language for specifying model-to-model transformations. ATL is inspired by QVT. ATL is a hybrid language, and provides a mix of declarative and imperative constructs.

Our approach follows a different direction, aiming to use a declarative language (i.e., Prolog) for expressing transformations by using the ontology based representation of models. In addition, transformation and validation are integrated. Our proposal contributes also to the framework of model transformation with declarative languages.

Declarative languages have already been used in this context in some works. A first approach is [14], 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 approach [7] introduces *inductive logic programming* in model transformation. The motivation of the work is that designers need to understand how to map source models to target models. With this aim, they are able to derive transformation rules from an initial and critical set of elements of the source and target models. The rules are generated in a (semi-) automatic way. We believe that (semi-) automatic derivation of transformation rules could be an extension of our work, but it is out of the scope of the current work.

The language *Tefkat* [25, 24] is a declarative language whose syntax resembles a logic language with some differences (for instance, it uses *forall* construct for traversing models). In this framework, [16] proposes metamodel transformations in which evolutionary aspects are formalised using the Tefkat language. In our case, we have adopted the syntax of Prolog for transforming models. The advantage of using Prolog is that a Prolog programmer can use our tool without training. In addition, the programmer can use the existing Prolog libraries in the code. In particular, the OWL RL library can be used for validation of models. With respect to incrementality, we have still not covered this aspect in our work.

The language *Maude* [11] has been also used in several works about model transformation. For instance, in [31] UML models and metamodels have been formalized in *Maude*, and the same authors have developed an Eclipse plug-in called *Maudelling* that enables the transformation of models and metamodels to the corresponding *Maude* specifications. *Mova* [12] and *Moment* [9] are also *Maude*-based modeling tools for verification of UML models. In the case of *Maude*-based tools, they also have to represent (UML) metamodels and models by means of the constructions of the language: sorts, classes, etc. The introduction of models in *Maude* enables to use many of the tools (model checking, verification, etc) available on *Maude*. Comparing *Maude* proposals with our approach, we believe that *Maude* is equipped with powerful tools for model checking and verification and it makes model transformation an interesting application of the *Maude* language. Our proposal aims in the same sense to find a framework for model transformation based on Prolog. Prolog can provide easy specification and rapid prototyping, along with the wide acceptance of Prolog as programming language.

Prolog has been already used in the context of MDD in the MoMaT framework [32]. In particular, they use Prolog for representing and verifying models in the same sense as our proposal. Our approach can be considered as an extension of such work, allowing meta-modeling and transformation/validation of models. However, our work take as basis the ODM representation of models, and uses OWL RL reasoning for validation. In summary, our approach to model transformation with Prolog can be seen as the basis for the building a ODM/Prolog and logic-based tool for transforming and analysing models.

There are also so-called *graph transformation languages*, which can be considered of declarative nature. This is the case of *VIATRA2* [6], *GReAT* [1] and AGG [33], among others. Graph transformation languages describe transformations by graph rewriting. Usually, these languages consist of rules whose match with a graph provides a transformation on the graph, in particular, deleting and adding new elements to the graph. *VIATRA2* has some features which make the proposal closer to our approach. Although the specification of model transformations is supported by graph transformations, Prolog is used as transformation engine. Thus XMI models and rules are translated into a Prolog graph notation. In our proposal Prolog is used as transformation engine but also as transformation specification language. Graph transformations provides a very intuitive and visual mechanism for expressing transformations. Our approach is not focused on visual representation of transformations, however, it is based on meta-modeling of source and target models which makes easy the definition of transformations.

Finally, *Datalog* has been used in [5] to transform data schemas (from OO schemas to SQL and from SQL to XML, for example). Our model transformation approach has been applied a similar problem to the presented in [5]. In the same line of research (i.e, database transformations), the *WOL* language [22] is a declarative language for specifying database transformations and constraints. WOL is based on Horn clause logic expressions using a small number of simple predicates and primitive constructors. Our running example can be seen as an example of tranformation in WOL, however, our approach can be applied to other kinds of UML transformations.

6 Conclusions and Future Work

In this paper we have presented a framework for the specification and validation of model transformations by means of Prolog rules, using the representation of UML models by means of ODM. Our approach has been applied to a well-known example of model transformation in which an UML class diagram representing a database (as an entity-relationship diagram) is transformed into ab UML diagram representing a relational database. We have validated our proposal by means of a prototype developed under SWI-Prolog.

Our approach has to be extended in the future as follows: (a) Firstly, we would like to test our prototype with other kinds of UML diagrams and transformations, and also with bigger examples; (b) Secondly, we are also interested in the use of our approach for model driven development of user interfaces in the line of our previous works [3, 4]; (c) Finally, we believe that our work will lead to the development of a logic based tool for transformation under the ECLIPSE framework.

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. A Prolog Library for OWL RL. In *Proceedings of the Logic in Databases, LID'2011, EDBT/VLDB*. ACM, 2011.

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*, 19:695–720, 2008.
4. J. M. Almendros-Jiménez and L. Iribarne. UML Modeling of User and Database Interaction. *The Computer Journal*, 52(3):348–367, 2009.
5. P. Atzeni, P. Cappellari, R. Torlone, P.A. Bernstein, and G. Gianforme. Model-independent schema translation. *The VLDB Journal*, 17(6):1347–1370, 2008.
6. A. Balogh and D. Varró. The Model Transformation Language of the VIATRA2 Framework. *Science of Programming*, 68(3):187–207, October 2007.
7. Zoltán Balogh and Daniel Varró. Model transformation by example using inductive logic programming. *Software and Systems Modeling*, 8(3):347–364, 2009.
8. Alex Borgida. On the relative expressiveness of Description Logics and Predicate Logics. *Artificial Intelligence*, 82(1-2):353–367, 1996.
9. Artur Boronat, Reiko Heckel, and José Meseguer. Rewriting logic semantics and verification of model transformations. In *FASE '09: Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, pages 18–33, Berlin, Heidelberg, 2009. Springer-Verlag.
10. J. Cabot, R. Claris, E. Guerra, and J. de Lara. Verification and Validation of Declarative Model-to-Model Transformations. *Systems and Software*, 2(83):283–302, 2010.
11. 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 LNCS, pages 76–87. Springer-Verlag, June 2003.
12. Manuel Clavel, Marina Egea, and Viviane Torres Da Silva. Mova: A tool for modeling, measuring and validating uml class diagrams, `http://maude.sip.ucm.es/mova/`. 2008.
13. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.
14. 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.
15. Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description Logic Programs: Combining Logic Programs with Description Logic. In *Proc. of the International Conference on World Wide Web*, pages 48–57, USA, 2003. ACM.
16. David I. Hearnden. *Deltaware: Incremental Change Propagation for Automating Software Evolution in Model-Driven Architecture*. PhD thesis, Centre or Institute School of Information Tech & Elec Engineering,The University of Queensland, 2007.
17. Guillaume Hillairet. ATL Use Case - ODM Implementation (Bridging UML and OWL). Technical report, `http://www.eclipse.org/m2m/atl/atlTransformations/`, 2007.
18. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.
19. 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.
20. Frédéric Jouault and Ivan Kurtev. On the interoperability of model-to-model transformation languages. *Sci. Comput. Program.*, 68(3):114–137, 2007.
21. H. Knublauch, R.W. Fergerson, N.F. Noy, and M.A. Musen. The Protégé OWL plugin: An open development environment for semantic web applications. *The Semantic Web–ISWC 2004*, pages 229–243, 2004.
22. A. Kosky, S. Davidson, and P. Buneman. Semantics of database transformations. *Semantics of Databases. Springer LNCS*, 1358:55–91, 1998.

23. I. Kurtev. State of the Art of QVT: A Model Transformation Language Standard. In *3rd Int. Symposium on Applications of Graph Transformation with Industrial Relevance*, pages 377–393. Springer, 2008.
24. M. Lawley and J. Steel. Practical Declarative Model Transformation with Tefkat. In *MoDELS Satellite Events*, pages 139–150. LNCS 3844, Springer, 2006.
25. 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.
26. Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, and Carsten Lutz. OWL 2 Web Ontology: Reasoning in OWL 2 RL and RDF Graphs using Rules. Technical report, `http://www.w3.org/TR/owl2-profiles/#Reasoning_in_OWL_2_RL_and_RDF_Graphs_using_Rules`, 2009.
27. OMG. MDA Specifications. Technical report, `http://www.omg.org/mda/specs.htm`, 2003.
28. OMG. MOF 2.0 Query/Views/Transformations RFP. Technical report, `http://www.omg.org/docs/ad/05-01-06.pdf`, 2008.
29. OMG. Model driven architecture. Technical report, `http://www.omg.org/cgi-bin/doc?omg/03-06-01`, 2008.
30. OMG. Ontology Definition Metamodel (ODM) . Technical report, `http://http://www.omg.org/spec/ODM/1.0/`, 2009.
31. 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.
32. H. Storrle. A PROLOG-based Approach to Representing and Querying UML Models. *Intl. Ws. Visual Languages and Logic (VLL07)*, 274:71–84.
33. G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *Applications of graph transformations with industrial relevance: second international workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27-October 1, 2003; revised selected and invited papers*, volume 3062, pages 446–453. LNCS, 2004.
34. TopCased. Topcased: The open-source tool kit for critical systems. Technical report, `http://www.topcased.org/`, 2010.
35. Laurence Tratt. Model transformations and tool integration. *Software and System Modeling*, 4(2):112–122, 2005.
36. W3C. OWL Ontology Web Language. Technical report, `http://www.w3.org`, 2004.