Model Driven Engineering (MDE) is an emerging approach of software engineering. MDE emphasizes the construction of models from which the implementation should be derived by applying model transformations. The Ontology Definition Meta-model (ODM) has been proposed as a profile for UML models of the Web Ontology Language (OWL). In this context, transformations of UML models can be mapped to ODM/OWL transformations. On the other hand, model validation is a crucial task in model transformation. Meta-modeling permits to give a syntactic structure to source and target models. Nevertheless, semantic requirements have to be imposed to source and target models. A given transformation will be sound when source and target models fulfill the syntactic and semantic requirements. In this paper, we present an approach for model validation in ODM based transformations. Adopting a logic programming based transformational approach we will show how it is possible to transform and validate models. Properties to be validated range from structural and semantic requirements of models (pre and post conditions) to properties of the transformation (invariants). The approach has been applied to a well-known example of model transformation: the Entity-Relationship (ER) to Relational Model (RM) transformation.

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, the model transformation [28][15] is a key tool of MDE. According to the Model Driven Architecture (MDA) [22] initiative of the Object Management Group (OMG) [21], the model transformation provides to developers a framework for transforming their models.

The MDA approach proposes (at least) three levels 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. In order to define a model transformation one should be able to meta-model the source and target models with regard to the meta-meta-model, and map source and target meta-models. Model transformation needs formal techniques for specifying the transformation. In most cases transformations can be expressed by means of some kind of rules.

*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.
On the other hand, the Ontology Definition Metamodel (ODM) proposal [24] of the OMG aims to define an ontology-based representation of UML models. ODM is a standard for representing UML models with OWL 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 accommodated. Following the ODM proposal, an UML model can be represented by an ontology in which the TBox (i.e. the terminological box) contains the UML meta-model while the ABox (i.e. the assertional box) contains the instance of the UML meta-model which represents the model.

Model validation is a key element of MDE. Firstly, (a) source and target models must conform to the corresponding meta-models. Source and target meta-models describe the syntactic structure of source and target models. Nevertheless, some semantic requirements have to be imposed in source and target models. In UML semantic requirements are usually expressed in the Object Constraint Language (OCL) [23]. Secondly, (b) pre-conditions and post-conditions and invariants are imposed in transformations. While source and target models can be well-formed with regard to meta-models, some extra requirements can be required. We can distinguish two specific cases: (b.1) source and target model requirements, and (b.2) transformation requirements. The first case covers requirements of source and target models in isolation. The second case covers requirements on target models with regard to the source models.

In the ODM context, one can argue that the use of OWL as modeling language provides a suitable framework for validation of properties. OCL can be replaced by OWL when specifying requirements imposed to models. OWL reasoning is a widely studied topic of research, and many tools have been developed in this context (for instance, the Protégé tool [17] and the OWL reasoners Hermit, Jena, Fact++, Racer, among others). OWL reasoning ranges from ontology consistence testing to ontology-based inference (i.e. derivation from ontology axioms). The topic can be applied to ODM (an hence to UML) however, validation in model transformation is a wider topic of research. Model validation involves ontology consistence testing and ontology-based inference for cases (a) and (b.1), while case (b.2) involves cross validation of ontologies.

On the other hand, the relationship between logic programming and ontologies is well-known. OWL is based on Description Logic (DL) [4], 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 [11], and OWL RL [20]. Typically, Description Logic is used for representing the TBox and the ABox. The encoding of (fragments of) DL into logic programming is based on the representation of the TBox with Prolog rules and the representation of the ABox with Prolog facts. It means that ontology instances are represented by Prolog facts. In model transformation a transformation maps the source model into the target model, particularly, the ABox of the source model into the ABox of the target model. Model transformation can be seen as a mapping of Prolog facts following the OWL to logic programming encoding, and it can be defined with Prolog rules.

In this paper, we present an approach for model validation in ODM based transformations. Adopting a logic programming based transformational approach we will show how it is possible to transform and validate models. Properties to be validated range from structural and semantic requirements of models (pre and post conditions) to properties of the transformation (invariants). The approach has been applied to the well-known example of model transformation: the Entity-Relationship (ER) to Relational Model (RM) transformation. The proposal is based on the use of logic programming with two ends. Firstly, specification of transformations. Secondly, specification of properties for model validation. We have validated our proposal in a prototype developed under SWI-Prolog. The prototype together with the case study can be downloaded from [http://indalog.ual.es/mdd](http://indalog.ual.es/mdd).

Our approach adopts the OWL to logic programming mapping as basis. Firstly, transformations can
be expressed with Prolog rules. Secondly, model validation can be encoded with Prolog. Using Prolog atoms and some elements of Prolog meta programming we are able to validate source and target models as well as transformations. The advantages of the approach are the following. Firstly, the declarative nature of the specification, secondly, the use of a standardized language (Prolog), and the ability of executing transformations and automatically validate source and target models. Besides, the use of Prolog as validation language enriches the capabilities of ODM and OWL constructors for expressing model requirements. While ODM is an OWL profile for UML models the expressivity power of ODM is limited, and model validation needs to express more complex requirements.

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. 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: (a) We have to consider the meta-model of the source model which defines the elements occurring in source model. Instances of the source meta-model are transformed by applying the transformation rules. (b) We have to consider the meta-model of the target model which defines the elements occurring in target model. (c) We have to define Prolog rules for transforming instances of the source meta-model into instances of the target meta-model. (d) We have to define Prolog rules for validating the transformation. The validation consists in source model validation, target

Figure 1: Integration with UML/OWL tools
model validation and transformation validation.

The question now is, how to express transformations and validations in Prolog? Our proposal is as follows. (a) The ODM proposal provides a representation of UML models with an ontology. The TBox represents the meta-model and the ABox properly represents the model. We can represent the ABox with Prolog facts. Fortunately, SWI-Prolog used in our prototype is equipped with a library for loading and writing OWL files into Prolog facts. (b) 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. Prolog rules can be used for transforming Prolog facts. (c) Model validation consists in checking properties on source and target models in isolation as well as checking cross properties on both models. Model validation with Prolog consists in checking properties about the set of Prolog facts representing source and target models.

Our approach has been implemented and tested with some examples. We have integrated our approach by using several UML and OWL tools (see Figure 1). We have used the TopCased UML tool [27] for designing the source and target meta-tools. In addition, we have used a UML2OWL transformer (available from [14]) in order to have the ODM-based representation of source and target meta-models. We have also used the Protégé tool [17] for defining the instance of the source meta-model, and for exporting the source model (i.e. meta-model+instance) into an OWL document. 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, SWI-Prolog is used for validating the target model, and validating the transformation. Later, the Protégé tool is also used for exporting the target model together with the target meta-model to an OWL document. Finally, an OWL2UML transformer has been used for obtaining the target model from the ODM-based representation.

2.1 Case Study

In this section we will describe the case study used in the paper. It is a well-known example of model transformation. Basically, the entity-relationship (ER) model is transformed into the relational model (RM). The model of Figure 2 represents the modeling of a database with an ER style diagram, while the model of Figure 3 is a RM style modeling of the same database. The ER modeling of Figure 2 can be summarized as follows. Entities are represented by classes (i.e., Student and Course), including attributes; Containers are defined for each entity (i.e., DB_Students and DB_Courses); Containers are composed of entities, therefore specifying a composition relationship between the container and the
Relationships are represented by associations. Relation names are association names. Besides, association ends are defined (i.e., the_students, the_courses, is_registered and register). Relationships can be adorned with qualifiers and navigability. The role of qualifiers is to specify the key attributes of each entity being foreign keys of the corresponding association. Figure 3 shows the RM modeling of the same database. It introduces the following new UML stereotypes: <<table>>, <<row>> and <<column>> for specifying tables, rows of tables and columns of tables, respectively. Tables are composed of rows, and rows are composed of columns. Furthermore, line is the role of the rows in the table and key, foreign and col is the role of the key, foreign, and non key and non foreign attributes in rows, respectively. Finally, each column has two attributes name and type. Figure 4 represents the meta-models of ER and RM models. In the first case, DB_Students and DB_Courses are instances of the class store, while Student and Course are instances of the class data, and the attributes of Student class and Course class are instances of the class attribute. In the second case, tables and rows of the target model are instances of the corresponding classes, and the same can be said about key, col and foreign classes. Now, the problem of model transformation is how to transform a class diagram of the type A (like Figure 2) into a class diagram of type B (like Figure 3).

---

Note: In traditional ER modeling containers are omitted, however in a UML-based modeling are incorporated.
The transformation is as follows. The transformation generates two tables called *the_students* and *the_courses* each including three columns that are 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* 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 the cited table. The columns *id_student* and *id_course* taken from qualifiers, play the role of foreign keys which are represented by 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 (see Figures 5 and 6). A transformation should be able to define a set of rules from which instances of the target meta-model are obtained from instances of the source meta-model.

Figure 5: Object Model of Source Model

Figure 6: Object Model of Target Model
Table 1: Model validation: requirements

<table>
<thead>
<tr>
<th>Source Model</th>
<th>Target Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) All attributes of a data have distinct names (SR) (WF)</td>
<td>(17) All col names of a row are distinct (SR) (WF)</td>
</tr>
<tr>
<td>(3) Each data has a key attribute (SR) (TR)</td>
<td>(19) All key names of a row are distinct (SR) (WF)</td>
</tr>
<tr>
<td>(5) Each data is contained in exactly one store (SC) (WF)</td>
<td>(21) Each table is associated to exactly one row (SC) (WF)</td>
</tr>
<tr>
<td>(7) All data have distinct containers (SR) (TR)</td>
<td>(23) Each key is associated to exactly one row (SC) (TR)</td>
</tr>
<tr>
<td>(9) All qualifier names of a role are distinct (SR) (TR)</td>
<td>(25) Each foreign is associated to exactly one row (SC) (TR)</td>
</tr>
<tr>
<td>(11) Each relation has two roles (SC) (WF)</td>
<td>(27) All row names are distinct (SR) (WF)</td>
</tr>
<tr>
<td>(13) Each role is associated to exactly one relation (SC) (TR)</td>
<td>(29) All rows have either all keys and cols or all foreigns (SR) (TR)</td>
</tr>
<tr>
<td>(15) All role names of a data are distinct (SR) (TR)</td>
<td>(18) All foreign names of a row are distinct (SR) (WF)</td>
</tr>
<tr>
<td></td>
<td>(20) All foreigns of a row are keys of another row (SR) (WF)</td>
</tr>
<tr>
<td></td>
<td>(22) Each row is associated to exactly one table (SC) (WF)</td>
</tr>
<tr>
<td></td>
<td>(24) Each col is associated to exactly one row (SC) (TR)</td>
</tr>
<tr>
<td></td>
<td>(26) All table names are distinct (SR) (WF)</td>
</tr>
<tr>
<td></td>
<td>(28) All rows have exactly one key (SC) (TR)</td>
</tr>
<tr>
<td>Cross Requirements</td>
<td></td>
</tr>
<tr>
<td>(30) Key and col names and types are names and types of attributes</td>
<td>(31) Table names are either container names or role names</td>
</tr>
<tr>
<td>(32) Row names are data names or concatenations of role names and data</td>
<td>(33) Foreign names are concatenations of roles, data and key</td>
</tr>
</tbody>
</table>

2.2 Model Validation

While source and target meta-models impose structural constraints (SC) on source and target models, we can specify several semantic requirements (SR) on source and target models. In addition, we can describe cross requirements on models. In Table 1 we can see a set of requirements classified as (SC) and (SR). Some requirements express conditions of well-formed models (WF), while some of them are required by the transformation (TR).

For instance, (2) requires that each data has a unique key attribute. This is a semantic requirement. Key attributes are attributes having key set to true, and the existence of a unique key attribute cannot be expressed in the UML diagram. Moreover, this requirement is a pre-condition of the transformation because key attributes are used as foreign keys in the target model. Case (5) is a structural constraint of well-formed models: each data is associated to exactly one store. It is not needed in the transformation and can be expressed in the UML diagram with a cardinality constraint. Cases (6), (7), (9), (12) and (15) are related to naming of elements of source models, and therefore are semantic requirements. They (except (12)) are required by the transformation: data and container names are used for naming tables and rows in the target model, while role and qualifier names (concatenated with data names) are also used for naming rows and foreign keys. (12) is required by the well-formedness of the source model. In the target model tables, rows, cols, keys and foreigns are not shared (cases (21)-(25)). Case (20) is a semantic requirement that describes the relationship between foreign keys and keys in a well-formed target model. Case (29) is required by the transformation which assigns either keys and cols or foreigns to rows. Finally, cases (30)-(33) describe the relationship between names of the target model and names of the source model.

It is worth observing that the requirements about source and target models in isolation are not enough for the soundness of the transformation. For instance, source and target models can both have keys, but a cross requirement is needed: the keys of the target models are the keys of the source model.
3 Prolog for Model Transformation and Validation

In this section, we will show how Prolog can be used for defining transformation and validation rules in our approach. With this aim, we have to consider the following elements.

3.1 Prolog based Transformation

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 with the predicate `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:

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

The transformation rules define new triples representing the target model. Hence, a new predicate called `newrdf` is defined by the transformation rules. For instance, the following rules define the individuals of the class `table` of the model B from the model A of the case study:

```prolog
newrdf(IdTable,rdf:type,'http://metamodelB.ecore#table'):-
    rdf(IdData,rdf:type,'http://metamodelA.ecore#data'),
    generate_id([IdData,'table1'],IdTable).

newrdf(IdTable,rdf:type,'http://metamodelB.ecore#table'):-
    rdf(IdRole,'http://metamodelA.ecore#role.navigable',E),
    E=literal(type(_,true)),
    generate_id([IdRole,'table2'],IdTable).
```

The first rule defines triples `(IdTable,rdf:type,'http://metamodelB.ecore#table')` obtained from triples `(IdData,rdf:type,'http://metamodelA.ecore#data')`, where `IdTable` is the identifier of the table, which is generated with the call `generate_id` from the data identifier `IdData` and ‘table1’. The second rule defines the individuals of class `table` obtained from navigable roles, which are generated from the role identifier `IdRole` and ‘table2’. In such a way that the following Prolog goal obtains the tables of the target model:

```prolog
?- newrdf(IdTable,rdf:type,'http://metamodelB.ecore#table').
IdTable = 'http://metamodelA.ecore#02_Student_datatable1' ;
IdTable = 'http://metamodelA.ecore#09_Course_datatable1' ;
IdTable = 'http://metamodelA.ecore#13_register_roletable2' ;
false.
```
which represent the individuals of classes Student, Course and register of the Figure 6. Now, the individuals of the class row of Figure 6 can be defined as follows:

```
newrdf(IdRow,rdf:type,'http://metamodelB.ecore#row'): -
    rdf(IdData,rdf:type,'http://metamodelA.ecore#data'),
    generate_id([IdData,'row1'],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',E),
    E=literal(type(_,true)),
    generate_id([IdRole,IdData,'row2'],IdRow).
```

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

```
newrdf(IdForeign,rdf:type,'http://metamodelB.ecore#foreign'): -
    rdf(IdRole,'http://metamodelA.ecore#role.navigable',E),
    E=literal(type(_,true)),
    rdf(IdRole,'http://metamodelA.ecore#role.is',IdQualifier),
    rdf(IdData,'http://metamodelA.ecore#data.role_of',IdRole),
    generate_id([IdRole,IdData,IdQualifier,'foreign1'],IdForeign).
```

```
newrdf(IdForeign,rdf:type,'http://metamodelB.ecore#foreign'): -
    rdf(IdRole,'http://metamodelA.ecore#role.navigable',E),
    E=literal(type(_,true)),
    rdf(IdRole,'http://metamodelA.ecore#role.has_role',IdRelation),
    rdf(IdRelation,'http://metamodelA.ecore#relation.is_role',IdRole2),
    rdf(IdRole2,'http://metamodelA.ecore#role.is',IdQualifier),
    IdRole2\=IdRole,
    rdf(IdData,'http://metamodelA.ecore#data.role_of',IdRole),
    generate_id([IdRole2,IdData,IdQualifier,'foreign2'],N).
```

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 5 have to be defined. For instance, the role has from the class table of Figure 4 is defined as follows:

```
newrdf(IdTable,'http://metamodelB.ecore#table.has',IdRow): -
    rdf(IdData,rdf:type,'http://metamodelA.ecore#data'),
    generate_id([IdData,'table1'],IdTable),
    generate_id([IdData,'row1'],IdRow).
```

```
newrdf(IdTable,'http://metamodelB.ecore#table.has',IdRow): -
    rdf(IdData,'http://metamodelA.ecore#data.role_of',IdRole),
    rdf(IdRole,'http://metamodelA.ecore#role.navigable',E),
    E=literal(type(_,true)),
    generate_id([IdRole,'table2'],IdTable),
    generate_id([IdRole,IdData,'row2'],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.

Finally, attributes of classes of the target metamodel of Figure 4 have to be defined. For instance, name of class table is defined as follows:
where the table names are obtained from container names (i.e., the_students and the_courses).

3.2 Prolog based Validation

Model validation is achieved with Prolog. Table 2 includes some of the Prolog rules of the constraints expressed in Table 1. The full set of rules can be downloaded from http://indalog.ual.es/mdd.

For validating the requirements on models, we can call each rule and in the case of success it indicates that the requirement is violated. In other words, the condition of the rule expresses the negation of the requirement. Prolog meta programming predicates are used. For instance, case (2) uses the setof predicate to collect the set of keys of a given data. When the collected list is empty in some case, some data violates the requirement.

4 Related Work

Validation and verification of model transformations is an emerging topic of research. We have found some similarities of our work with the work proposed in [5]. The authors work in the context of the ATLAS Transformation language (ATL) and OCL, but handle the same kind of properties of our approach (unique names for relations and attributes together with existence of keys). A more general framework for transformation validation and verification is proposed in [6] including verification and validation of properties about transformation rules. Our approach focused on properties about meta-models, assuming that when some requirement is violated either models or rules are incorrect. Prolog has been also used in the Model Manipulation Tool (MoMaT) [26] for representing and verifying models. In [16] they propose consistency checking of class and sequence diagrams based on Prolog. Consistency checking rules as well as UML models are represented in Prolog, and Prolog reasoning engine is used to automatically find inconsistencies.

On the other hand, logic programming based languages have already been explored in the context of model engineering in some works. A first approach is [9], which describes the attempts to adopt several technologies for model transformation including logic programming. Particulary, they focused on Mercury and F-Logic logic languages. The approach [3] has introduced 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. The Tefkat language [19, 18] is a declarative language whose syntax resembles a logic language with some differences (for instance, it incorporates a forall construct for traversing models). In this framework, [12] proposes metamodel transformations in which evolutionary aspects are formalised using the Tefkat language. In [10], they present a declarative approach for modeling requirements (designs and patterns) which are encoded as Prolog predicates. A search routine based on Prolog returns program fragments of the model implementation. Traceability and code generation are based on logic
programming. They use *JTransformer*, which is a logic-based query and transformation engine for Java code, based on the Eclipse IDE. Logic programming based model querying is studied in [8], in which logic-based facts represent meta-models. In [25] they study a transformation mechanism for the EMF Ecore platform using Prolog as rule-based mechanism. Prolog terms are used and predicates are used for deconstructing and reconstructing a term of a model. *Abductive logic programming* is used in [13] for reversible model transformations, in which changes of the source model are computed from a given change of the target model. Finally, [7] has compared OCL and Prolog for querying UML models. They have found that Prolog is faster when execution time of queries is linear.
5 Conclusions and Future Work

In this paper we have presented a framework for the specification and validation of model transformations with 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 an ER diagram is transformed into a UML diagram representing a relational database. We have validated our proposal with a prototype developed under SWI-Prolog.

Our approach has to be extended in the future as follows: (a) Firstly, we would like to improve our prototype. Particularly, validation is now achieved by Prolog rules in which success and fail is returned with the analysis. We would like to show more complete analysis results, showing the model elements that violate the requirements, justifications, diagnosis, reparation, etc. (b) Secondly, we would like to test our prototype with other kinds of UML diagrams and transformations, and also with bigger examples; (c) Thirdly, we are also interested in the use of our approach for model driven development of user interfaces in the line of our previous works [1][2]; (d) Finally, we believe that our work will lead to the development of a logic based tool for transformation and validation of models.

References


