PTL: A Prolog-based Model Transformation Language

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

[1] jalmen@ual.es
[2] luis.iribarne@ual.es
Dpto. de Lenguajes y Computación
Universidad de Almería
04120-Almería (Spain)

Abstract: In this paper we present a model transformation language based on logic programming. The language, called PTL (*Prolog-based Transformation Language*), can be considered as an hybrid language in which ATL-style rules are combined with logic rules for defining transformations. ATL-style rules are used to define mappings from source models to target models while logic rules are used as helpers. The proposal has been implemented so that a Prolog program is automatically obtained from a PTL program. We have equipped our language with debugging and tracing capabilities which help developers to detect programming errors in PTL rules.

Keywords: MDD; Logic Programming; Software Engineering

1 Introduction

Model Driven Engineering (MDE) is an emerging approach for software development. MDE emphasizes the construction of models from which the implementation is derived by applying model transformations. Therefore, *Model Transformation* [Tra05] is a key technology of MDE. According to the *Model Driven Architecture (MDA)* initiative of the *Object Management Group (OMG)* [OMG03], model transformation provides a framework to developers for transforming their models.

MDE proposes (at least) three elements in order to describe a model transformation: the first one is the so-called *meta-meta-model* which is 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. Furthermore, source and target meta-models are instances of the meta-meta-model. In order to define a model transformation the source and target models are modeled with respect to the meta-meta-model, and source and target meta-models are mapped.

In this context, model transformation needs formal techniques for specifying the transformation. In most of the cases transformations can be expressed with some kinds of *rules*. The rules have to express how source models can be transformed into another. Several transformation languages and tools have been proposed in the literature (see [CH06] for a survey). The most relevant one is the language *ATL (Atlas Transformation Language)* [JABK08] a *domain-specific language* for specifying model-to-model transformations. ATL is a hybrid language, and pro-

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 197-211, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

vides a mixture of declarative and imperative constructs. The declarative part of ATL is based on rules. Such rules define a source pattern matched over source models and a target pattern that creates target models for each match. ATL transformations are unidirectional, operating on read-only source models and producing write-only target models.

In this paper we present a model transformation language based on logic programming. The language, called PTL (*Prolog-based Transformation Language*), can be considered as an hybrid language in which ATL-style rules are combined with logic rules for defining transformations. ATL-style rules are used to define mappings from source models to target models while logic rules are used as helpers. The proposal has been implemented so that a Prolog program is automatically obtained from a PTL program. Hence, PTL makes use of Prolog as transformation engine. The encoding of PTL programs by Prolog is based on a Prolog library for handling meta-models. We have equipped our language with debugging and tracing capabilities which help developers to detect programming errors in PTL rules. Debugging detects PTL rules that cannot be applied to source models, and tracing shows rules and source elements used to obtain a given target model element.

The aim of our work is to provide a framework for model transformation based on logic programming. From a practical point of view, our language can be exploited by Prolog programmers for defining model transformations. So, our proposal aims to introduce model transformation in the declarative paradigm, and can be intended as an application of logic programming (particularly, Prolog) to a context in which rule-based systems are required. Prolog programmers might be interested in programming model transformations; and our language provides the elements involved in model transformation: meta-models handling and mapping rules. We believe that our contribution can be interesting to the logic programming community.

Mapping rules are defined in ATL style syntax. Due to wide acceptance of ATL as transformation language, the adoption of a syntax inspired in ATL to write model transformations makes our approach easier to use and closer to other proposals of transformation languages. The adoption of ATL style syntax facilitates the definition of mappings: basically, objects are mapped to objects of given models. Nevertheless, we retain logic programming as basis, for instance, by defining helpers with Prolog code instead of OCL code, adopted in ATL. It makes that our framework is a mixture of model transformation and logic languages. One of the main ends of helpers in ATL is to serve as query language against the source models. In a logic programming based approach, logic (i.e., Prolog style) rules serve as query language.

Our approach is equipped with debugging and tracing tools. Using Prolog as transformation engine makes possible to program debugging and tracing tools with little effort. Debugging is useful when the target model is *incomplete* while tracing is useful when the target model is *incorrect*.

In summary, our proposal is based on the following elements:

(a) **Encoding to Prolog rules**. ATL-style mapping rules are translated into Prolog rules. Each rule is encoded by a set of Prolog rules, one rule for each object that is created, and one rule for each role ends set by the rule.

(b) **Prolog library for handling meta-models**. Meta-models can be defined in our approach, and a Prolog library for handling meta-models is automatically generated from meta-model specification.

(c) **MOF meta-metamodel**. MOF in our approach is considered as meta-meta-model, and source and target meta-models have to be defined in terms of the MOF meta-model. With this aim, our language can be also used for defining transformations from (and to) the MOF meta-model to (and from) source (and target) meta-models.

(d) **XMI support**. In order to execute a transformation, source models have to be stored in XMI format. Source models are loaded from XMI files, and target models are obtained in XMI format. Our approach allows to define the location of source models and destination of target models. So, our approach can be integrated with XMI-compliant tools.

(e) **Prolog helpers**. Helpers are defined with Prolog rules. Such Prolog rules make use of the Prolog library for handling meta-models.

(f) **Debugging capabilities**. Debugging permits to detect rules that cannot be applied in a certain transformation, and in addition, debugging is able to locate the point of the program in which a certain programming error is found. Such programming error comes from (f.1) Boolean conditions that cannot be satisfied in mapping rules, and (f.2) objects of the target model that cannot be created. When a certain rule cannot be applied, a certain target model element will be missing, and therefore the transformation can be considered erroneous. The debugger is able to give (f.1) the name of rule and the Boolean condition that is false for all the elements of the source model, and (f.2) the name of the rule in which the target model cannot be created, and the failed binding.

(g) **Tracing capabilities**. Tracing permits to visualize how a certain target model element is obtained. Tracing shows all the rules and source model elements that contribute to a target model element. Tracing is useful when a target model element is obtained but is wrong.

Our approach will be applied to a well-known example of model transformation in which a class diagram describing an entity-relationship modeling of a database is transformed into a class diagram representing a relational database. The Prolog-based compiler and interpreter of our PTL language have been developed under *SWI-Prolog*. The code of the case study together with the source code of PTL can be downloaded from http://indalog.ual.es/mdd.

The structure of the paper is as follows. Section 2 will introduce the model transformation setting. Section 3 will present the PTL language. Section 4 will describe the encoding of PTL into Prolog. Section 5 will show how to debug and trace a given PTL transformation. Section 6 will review related work. Finally, Section 7 will conclude and present future work.

2 Model Transformation

In our framework, meta-meta-models are models similar to the model of Figure 1. The meta-meta-model represents the elements of the (UML) class diagram. For instance, a *class* has a *name* and can include attributes, represented by the role *ownedAttribute*, which belongs to the class *Property*. Properties can also be *roles* of *associations*. The members and owned ends of an association are represented by the roles *memberEnd* and *ownedEnd*. *NavigableOwnedEnd* sets navigable role ends. Each property is described with a *name*, a *type*, whether it is *composite* and

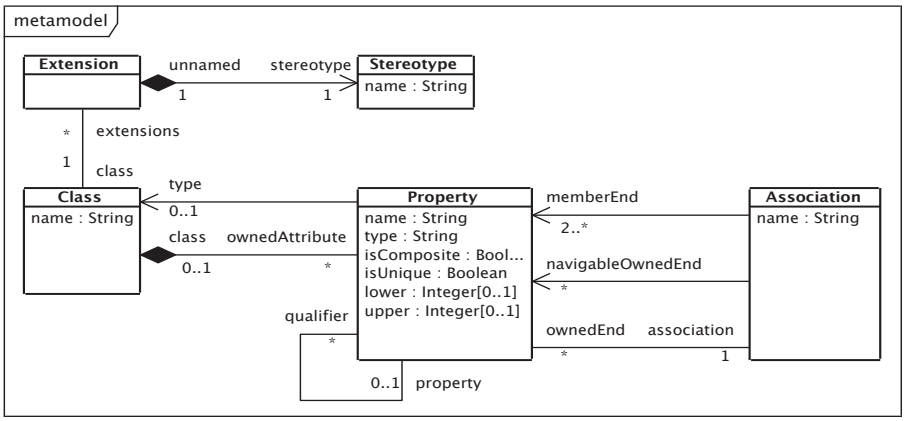

Figure 1: MOF Metamodel of the Class Diagram

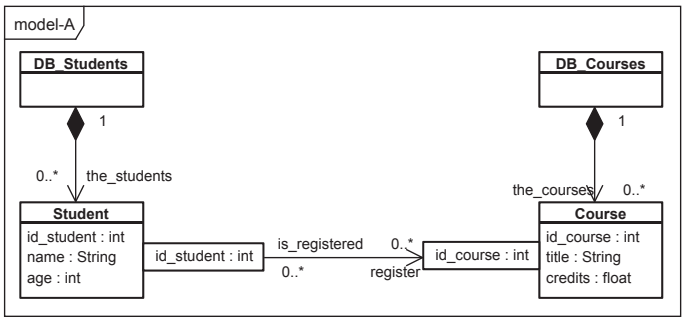

Figure 2: Entity-relationship modeling of the Case Study

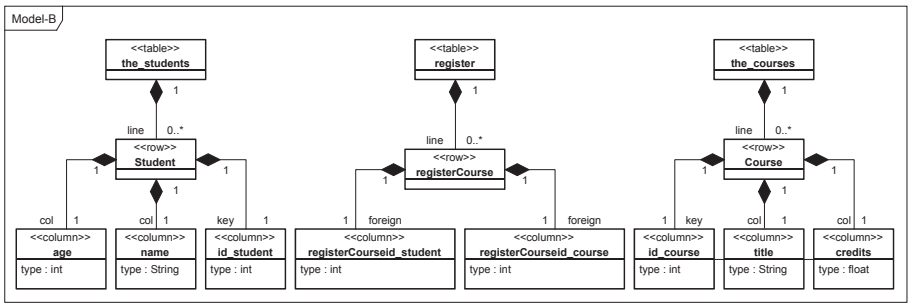

Figure 3: Relational modeling of the Case Study

unique or not, and the *lower* and *upper* bounds of the number of elements of the properties. The association with role *qualifier* links roles to qualifiers of the association. A class can be linked to *extension*'s which are *Stereotype*'s.

In order to define a transformation we have to define the source and the target meta-models. Such meta-models are defined in terms of the MOF meta-meta-model. Source and target models are instances of the corresponding meta-models. Moreover, source and target meta-models are instances of the MOF meta-meta-model.

Now, we would like to show an example of transformation with respect to the meta-meta-model of Figure 1. The model of Figure 2 represents the modeling of a database. We will call this kind of modeling as *"entity-relationship"* modeling of a database in contrast to the model of Figure 3 which will be called *"relational"* modeling of a database.

The entity-relationship 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*); *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 relational modeling of the same database. Such modeling also defines a class diagram for database design. Tables are composed of rows, and rows are composed of columns. It introduces the following new stereotypes: $<<$ *table* $>>$, $<<$ *row* $>>$ and $<<$ *column* $>>$. Furthermore, *line* is the role of the rows in the table, *key* is the role of the key attributes in rows, *foreign* is the role of the foreign keys in rows, and *col* is the role of non keys and non foreign keys in rows. Finally, each column has an attribute called *type*.

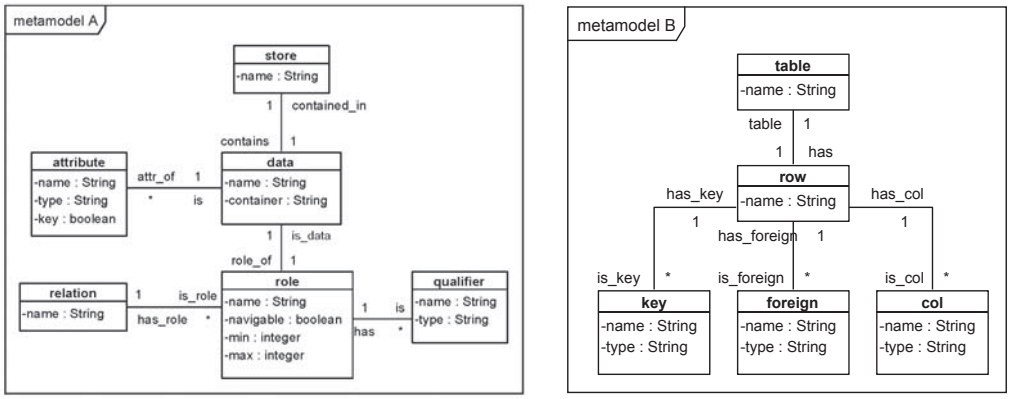

Figure 4: Meta-model of the Source/Target Models

Figure 4 represents the meta-models of both types of modeling. 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). 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

```
metamodel(
er,
[
         class(data, [name,container]),
         class(store, [name]),
         class(attribute, [name,type,key]),
         class(relation, [name]),
         class(role, [name,navigable,min,max]),
         class(qualifier, [name,type]),
         role(contains,store,data,"1","1"),
         role(contained_in,data,store,"1","1"),
         role(attr_of,data,attribute,"0","*"),
         role(is,attribute,data,"1","1"),
         role(has_role,role,relation,"1","1"),
         role(is_role,relation,role,"1","*"),
         role(has,qualifier,role,"1","1"),
         role(is,role,qualifier,"0","*"),
         role(is_data,role,data,"1","1"),
         role(role_of,data,role,"1","1")
]
).
```

Figure 5: Source Metamodel

```
role_id(er, A) :-role(er, A, [name(_), navigable(_), min(_), max(_)]).
data_container(er, A, B) :-data(er, A, [name(_), container(B)]).
attribute_type(er, A, B) :-attribute(er, A, [name(_), type(B), key(_)]).
qualifier_has(er, A, B) :-associationEnds(er, has, A, B).
relation_is_role(er, A, B) :-associationEnds(er, is_role, A, B).
```

Figure 6: Prolog library

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 role *foreign* in the associations of Figure 3.

3 PTL

PTL is a transformation language based on logic programming. A PTL program consists of **meta-model definitions** (source and target meta-models), **mapping rules** and **helpers**.

3.1 Metamodel definitions

Basically, meta-model definitions define meta-model elements: class and roles together with attributes for classes, and for each role the multiplicity and the role ends. From the meta-model definitions the PTL compiler automatically generates a Prolog library for handling the meta-models. Basically, the library contains predicates for each class and role of each meta-model in order to access to class attributes and role ends.

For instance, in Figure 5 we can see the definition of the source meta-model of Figure 4, and the Prolog library of Figure 6 contains (some of) the predicates generated to handle the

rule	:= **rule** rule_name **from** *pointers* [**where** *condition*] **to** *objects*
pointers	:= *pointer* | '(' *pointer*,...,*pointer* ')'
condition	:= *bcondition* | *condition* **and** *condition*
bcondition	:= *access* **==** *access* | *access* **=\=** *access*
pointer	:= pointer_name **:** metamodel **!** class
objects	:= *object*,...,*object*
object	:= *pointer* '(' *binding*,...,*binding* ')'
access	:= value | pointer_name | pointer_name@attribute | pointer_name@role@attribute | helper '(' *access*,...,*access* ')' | **resolveTemp**'(' '(' *access*,...,*access* ')', pointer_name')' | **sequence**'('['*access*,...,*access*']')'
binding	:= attribute <– *access* | role <– *access*

Figure 7: Mapping rules of PTL

er metamodel of Figure 5. We have three kinds of predicates: (a) those for *accessing class attributes*, for instance, *data_container* and *attribute_type*, (b) those for *accessing role ends*, for instance, *qualifier_has* and *relation_is_role*, and (c) a special kind of predicates that retrieve the *identifier* of a certain object (for instance, *role_id*).

The first and third kind of predicates call predicates representing class objects, which are called as the class name, and they have as arguments: the name of the meta-model, the object identifier and a Prolog list with the attributes: each attribute is represented by a Prolog term of the form: *attribute_name(value)*. The second kind of predicates calls to a predicate called *associationEnds*, representing role end objects. The *associationEnds* element has as arguments: the name of the meta-model, the name of the role, and the identifiers of the role ends.

3.2 Mapping rules

The syntax of the mapping rules is described in Figure 7, where *rule_name*, *pointer_name* and *helper* are rule, object and helper names, respectively, and *value* can be any basic datatype.

Basically, a mapping rule maps a set of objects of the source model into a set of objects of the target model. The rule can be conditioned by means of a Boolean condition, including equality (i.e. ==) and dis-equality (i.e. =\=), and the *and* operator. The rule condition is marked with *where*. Objects of target models are defined assigning values to attributes and roles, and they can make use in their definition of attribute values of the source models, *resolveTemp*, helpers and the *sequence* construction. The *resolveTemp* function permits to assign objects created from some other rules. With this aim, *resolveTemp* has as parameters the objects to which the rule is applied, and the name of the object created by the rule.

In Figure 8, we can see also examples of mapping rules in our approach. The rule *table2_er2rl* defines the tables and rows obtained from navigable roles (in the case study, *register* and *registerCourse*). The name of the table is the name of the role, and the name of the row is built from the concatenation of the name of the role, and the name of the role end. Moreover, we have to set the role ends from (to) tables to (from) rows (i.e., *has* and *table*) together with the *is_foreign* role end. The *is_foreign* role end is a sequence of two elements (*registerCourseid_student* and *registerCourseid_course*). For this reason the *sequence* constructor is chosen. Besides, *resolveTemp* retrieves *registerCourseid_course*, and a helper called *inverse_qualifier* is used for the retrieval of *registerCourseid_student*. The rule *foreign2_er2rl* computes the foreign class *regis-*

```
rule table2_er2rl from
p:er!role where (p@navigable==true and p&max=="*") to
     (t:rl!table(
        name <- p@name,
        has <- r
                ),
       r:rl!row(
         name <- concat(p@name,p@is_data@name),
         table <-t,
         is_foreign <- sequence([resolveTemp((p@is,p),f1),inverse_qualifier(p)])
               )
      ).
rule foreign2_er2rl from
(p:er!qualifier,q:er!role) where (p@has == q and q@navigable==false) to
        (f2:rl!foreign(
             name <-concat(concat(q@name,q@is_data@name),p@name),
             type <- p@type,
             has_foreign <- inverse_row(p)
                      )
         ).
```

Figure 8: Examples of PTL Mapping Rules

```
inverse_row(A, E) :-
        associationEnds(er, has, A, B),
        associationEnds(er, has_role, B, C),
        associationEnds(er, is_role, C, D),
        role_navigable(er, D, true),
        resolveTemp(D, r, E).
```

Figure 9: Helpers

terCourseid_student, which is computed from roles and qualifiers which are not navigable. The rule sets the role end *has_foreign* with a helper called *inverse_row*, which computes the row *registerCourse*.

3.3 Helpers

Helpers can de defined with logic rules. For instance, in Figure 9 we can see the definition of the *inverse_row* helper. Helpers can make use of the meta-model Prolog library and the *resolveTemp* construction. It is worth observing that helpers are defined with the following convention: we can define helpers with several arguments, but the last one has to be the result of the helper. In other words, predicates associated to helpers work as functions. The previous convention requires that helpers in PTL mapping rules have n arguments while code of helpers has $n+1$ arguments.

4 Encoding with Prolog rules

Now, we would like to show how PTL mapping rules are encoded with Prolog rules. The schema of the encoding is as follows. Given a PTL mapping rule of the form:

rule *rule_name* **from** *pointers* **where** *bcondition* **to** *objects*

```
(1) object(rl, foreign, L, (A, B), [name(K), type(C)], f2) :-
        foreign2_er2rl((A, B)),
        qualifier_type(er, A, C),
        qualifier_name(er, A, I),
        role_is_data(er, B, E),
        data_name(er, E, G),
        role_name(er, B, F),
        concat(F, G, H),
        concat(H, I, K),
        generate_ids((A, B), f2, L).

(2) associationObjects(rl, has_foreign, C, B) :-
        foreign2_er2rl((A, D)),
        qualifier_id(er, A),
        inverse_row(A, B),
        object(rl, foreign, C, (A, D), _, f2).

(3) foreign2_er2rl((A, B)) :-
        qualifier_id(er, A),
        role_id(er, B),
        qualifier_has(er, A, B),
        role_navigable(er, B, false).
```

Figure 10: Encoded Rule

where $objects \equiv object_1, \ldots, object_n$, $object_i \equiv pointer_i(binding_1, \ldots, binding_k)$ and $pointer_i \equiv q_i : mm_i!class_i$ then the encoding is as follows:

(1) $object(mm_i, class_i, Var, \overline{Vars}, enc[atts], q_i) : -rule\_name(\overline{Vars}'), enc[bind\_att]$.
(2) $associationObjects(mm_i, role_j, Var_1, Var_2) : -rule\_name(\overline{Vars}), enc[role_j <- access]$.
(3) $rule\_name(\overline{Vars}) : -enc[bcondition]$.

where $bind\_att$ is the subset of $binding_1, \ldots, binding_k$ of bindings of the form $attribute <- access$; expressions $role_j <- access$ are each one of the remaining bindings ($j \in \{1, ..., k\}$); and finally, the element $enc[atts]$ is the encoding of attributes, and the elements $enc[bind\_att]$, $enc[role_j <- access]$ and $enc[bcondition]$ are the encoding of such expressions using the Prolog library for meta-models and Prolog helpers.

The predicate *object* encodes the creation of objects of the target model. The *associationObjects* predicate encodes the creation of links between objects of the target model. There are *object* and *associationObjects* rules for each object and each link created by one rule. Hence, one PTL mapping rule is encoded by a set of Prolog rules, one rule for each object that is created, and one rule for each role ends set by the rule. The *object* predicate generates a unique identifier for each object of the target model. With this aim, a Prolog predicate called *generate_ids* is used.

For instance, we can see in Figure 10 how this Prolog library permits to encode the PTL rule *foreign2_er2rl* of Figure 8. The main predicates of the encoding are the predicates *object* (see (1) of Figure 10) and *associationObjects* (see (2) of Figure 10) which define new class objects and role end objects in the target model. They make use of a predicate called the same as the rule, in this case, *foreign2_er2rl* (see (3) of Figure 10). Such predicate retrieves the objects of the source model encoding the Boolean condition of the rule.

Finally, *resolveTemp* construction of PTL can be easily defined with our proposal of encoding:

```
resolveTemp(B, C, A) :- object(_, _, A, B, _, C).
```

```
ptl(Program):-[Program],
                generate_metamodels,
                generate_rules,
                load_models,
                clean_transformation.
```

Figure 11: PTL predicate

```
load_model(A) :-object(A, B, C, D, E, F),
        assert(objectM(A, B, C, D, E, F)),
        fail.
load_model(A) :-associationObjects(A, B, C, D),
        assert(associationObjectsM(A, B, C, D)),
        fail.
load_model(_).
```

Figure 12: Load_model predicate

4.1 PTL interpreter

Now, we would like to show how a PTL program is executed from Prolog. The predicate *ptl* of Figure 11 is called with the file name in which the PTL code is included. For instance:

```
?- ptl('er2rl.ptl').
```

The *ptl* predicate automatically generates the Prolog library of the meta-models defined in the PTL program (predicate *generate_metamodels*), it encodes PTL rules with Prolog rules (predicate *generate_rules*), and it generates the target models from the source models (predicate *load_models*). Since the execution is carried out in main memory, it cleans memory at the end (predicate *clean_transformation*).

The PTL program has to include meta-model definitions (i.e., source and target models), and the set of PTL (mapping and helper) rules. Moreover, we have to specify in the PTL program the location of source and target models with the directives *input* and *output*. These directives also admit to specify the XMI files in which models are stored.

Now, we would like to explain how PTL rules are executed. The previous *load_models* predicate calls to an auxiliary *load_model* predicate, which at the same time, calls to *object* and *associationObjects* predicates, which encode elements created by the rules, and each element is asserted into Prolog (main) memory, in the form of a Prolog fact, called *objectM* and *associationObjectsM*, respectively. The code of *load_model* predicate is shown in Figure 12.

4.2 Transformation execution

In order to execute a particular transformation, we have to specify how source and target meta-models are defined in terms of the MOF meta-meta-model.

With this aim, we have to define a transformation from the MOF meta-meta-model to the source meta-models, and from the target meta-model to the MOF meta-meta-model. It is required in order to load an XMI file containing the source models and to write the target-models into an XMI file[1]. In other words, in order to execute a transformation we have to consider (at least)

[1] A Prolog program has been implemented in order to serialize the MOF meta-model to XMI.

```
transform(Files):-clean_ptl,
                  transform_files(Files).
transform_files([]):-!.
transform_files([F|RF]):-ptl(F),
                  transform_files(RF).
```

Figure 13: Transform predicate

three PTL programs. Hence a transformation is executed as follows:

```
?- transform(['mm2er.ptl','er2rl.ptl','rl2mm.ptl']).
```

The code of the *transform* predicate can be seen in Figure 13.

5 Debugging and Tracing in PTL

In this section, we would like to show how to debug and trace executions in our proposal. Debugging and tracing permit to find programming errors.

5.1 Debugging

Debugging is able to find rules that cannot be applied, and provides the location in which the error is found. PTL mapping rules are not applied due to Boolean conditions that are not satisfied and objects that cannot be created. A Boolean condition is not satisfied whenever a certain equality or inequality is false.

Let us suppose that the PTL rule *table2_er2rl* of Figure 8 includes *p@name*=="*" instead of *p@max*=="*". This is a typical programming error and it cannot be detected by the compiler (i.e, the PTL program is well-typed and syntactically correct). Now, the engineer finds that the target model is wrong. In such a case (s)he can query the debugger, obtaining:

```
?- debugging(['mm2er.ptl','er2rl.ptl','rl2mm.ptl']).
Debugger: Rule Condition of: table2_er2rl cannot be satisfied.
Found error in: role_name
```

The debugger shows the name of the PTL rule (i.e. *table2_er2rl*) that cannot be applied and the found error (i.e. *role_name*).

But the debugger can also detect that target objects cannot be created, for instance, let us suppose that the engineer writes *p@navigable==false*, instead of *p@navigable==true* then the debugger answers as follows:

```
?-  debugging(['mm2er.ptl','er2rl.ptl','rl2mm.ptl']).
Debugger: Objects of: table2_er2rl cannot be created.
Found error in: resolveTemp
```

In this case the objects of rule *table2_er2rl* cannot be created, and the programming error comes from *resolveTemp*, because the call does not succeed.

In summary, debugging is useful when some elements of the target model are not created. In other words, when a certain element of the target model is missing. The PTL interpreter has been modified in order to cover with debugging. The debugger checks PTL rules that cannot be applied and prints the location in which Boolean conditions and object creations fail.

5.2 Tracing

However, the engineer can find a programming error due to the opposite case: a certain target model element is created but it is wrong. In such a case, the engineer can trace from the wrong target element to find the reason (i.e., applied rules and source model elements) of the creation of such element. A target element can be created from a missing Boolean condition.

Let us suppose that in rule *foreign2_er2rl* of Figure 8 the engineer omits the Boolean condition *q@navigable==false*. The engineer reviews the target model and finds a wrong element: *is_registeredStudentid_student*. Now, (s)he can trace the execution from the XMI identifier of the wrong element, obtaining the following answer:

```
?- tracing(['mm2er.ptl','er2rl.ptl',
    'rl2mm.ptl'],'275284307q275325284r2f2c').
Tracing the element: 275284307q275325284r2f2c

Rule: foreign1_rl2mm
Element: foreign
Metamodel: rl

Rule: foreign2_er2rl
Element: qualifier
Metamodel: er

Rule: qualifier_mm2er
Element: association
Metamodel: mm
xmi:id is wfP6O_iGCkzsbgVq

Element: property
Metamodel: mm
xmi:id is wfP6O_iGCkzsbgVr

Element: property
Metamodel: mm
xmi:id is CrZGO_iGCkzsbhYY
```

The trace shows the applied rules (i.e. *foreign1_rl2mm*, *foreign2_er2rl* and *qualifier_mm2er*) together with the XMI ids of source elements.

In the case of the tracer, the PTL interpreter is also modified. The rules are applied backward from the target model element and each applied rule and source model element is printed.

6 Related Work

Logic programming based languages have already been explored in the context of model engineering in some works.

A first approach is [GLR+02], 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 [BV09] 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 [LR07, LS06] 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, [Hea07] proposes metamodel transformations in which evolutionary aspects are formalised using the Tefkat language.

VIATRA2 [BV07] is a well-known language which has some features which make the proposal close to our approach. Although the specification of model transformation is supported by graph transformations, Prolog is adopted as transformation engine. Thus XMI models and rules are translated into a Prolog graph notation.

Prolog has been also used in the *Model Manipulation Tool (MoMaT)* [Sto07] for representing and verifying models. In [GW12], 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 [DH10], in which a logic-based facts represent meta-models. In [Sch09] 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.

In [CFLL08] they authors have compared OCL and Prolog for querying UML models. They have found that Prolog is faster when execution time of queries is linear. *Abductive logic programming* is used in [HLR09] for reversible model transformations, in which changes of the source model are computed from a given change of the target model. In [KNL11] 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, ATL debugging has been explored in [Ecl11], that asserts debugging instructions in ATL transformations. Each time an attribute is assigned to a value, a log file is updated with the rule name, the attribute and the value. In [Jou05], they have studied traceability, that is, links between source and target models, which are instances of a traceability meta-model.

Most of the quoted works make use of Prolog for representing meta-models and model elements. The representation varies from one to another, but in essence Prolog facts are used for representing model instances while rules are used for representing transformations and constraints on models. In our case Prolog facts are also used for representing model instances however they are not directly handled by the programmer given that they are automatically generated from XMI files. Prolog rules are used as helpers in PTL but the main component of PTL are mapping rules which are rules inspired in ATL. Mapping rules are automatically translated into Prolog rules by the compiler.

In our case, debugging is intended to provide a log of the rules that cannot be applied in a certain transformation. The debugging process shows also a log with the name of the rules that cannot be applied together with the Boolean conditions and object creations that fail. The case of tracing enables to navigate from a certain target model element to the source model, that is, applied rules and source elements that contribute to the target element. In our experience using the debugging and tracing tools they are very useful to correct usual programming errors in our

language.

7 Conclusions and Future Work

In this paper we have presented a model transformation language based on logic programming. We have also described the implementation the language which consists in the encoding of mapping rules by Prolog rules. Furthermore, we have shown how to debug and trace the execution of rules. As future work, we could also extend debugging and tracing capabilities. For instance, we have considered tracing from the target model to the source model, however, tracing from source model to target model would also be possible and interesting. Finally, we would like to improve the performance of our system. The execution times we have obtained for small examples are satisfactory. However, we would like to optimize Prolog code and Prolog representation, for instance, storing Prolog facts in secondary memory for large models.

Acknowledgements: 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.

Bibliography

[BV07] A. Balogh, D. Varró. The Model Transformation Language of the VIATRA2 Framework. *Science of Programming* 68(3):187–207, October 2007.

[BV09] Z. Balogh, D. Varró. Model Transformation by Example Using Inductive Logic Programming. *Software and Systems Modeling* 8(3):347–364, 2009.

[CFLL08] J. Chimia-Opoka, M. Felderer, C. Lenz, C. Lange. Querying UML models using OCL and Prolog: A performance study. In *Software Testing Verification and Validation Workshop, 2008. ICSTW'08. IEEE International Conference on*. Pp. 81–88. 2008.

[CH06] K. Czarnecki, S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal* 45(3):621–645, 2006.

[DH10] P. Dohrmann, S. Herold. Designing and Applying a Framework for Logic-Based Model Querying. In *Software Engineering and Advanced Applications (SEAA), 2010 36th EUROMICRO Conference on*. Pp. 164–171. 2010.

[Ecl11] Eclipse. ATL to BindingDebugger. Technical report, http://www.eclipse.org/m2m/atl/atlTransformations/#ATL2BindingDebugger, 2011.

[GLR+02] A. Gerber, M. Lawley, K. Raymond, J. Steel, A. Wood. Transformation: The Missing Link of MDA. In *Procs of ICGT'02*. Pp. 90–105. LNCS 2505, Springer, London, UK, 2002.

[GW12] M. Goldberg, G. Wiener. A Declarative Approach for Software Modeling. In *Practical Aspects of Declarative Languages: 14th International Symposium, PADL 2011, Philadelphia, Pa, January 23-24, 2012. Proceedings*. Volume 7149, pp. 18–32. 2012.

[Hea07] D. 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, Univ. of Queensland, 2007.

[HLR09] T. Hettel, M. Lawley, K. Raymond. Towards model round-trip engineering: an abductive approach. *Theory and Practice of Model Transformations*, pp. 100–115, 2009.

[JABK08] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming* 72(1-2):31–39, 2008.

[Jou05] F. Jouault. Loosely coupled traceability for ATL. In *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany*. Volume 91, pp. 29–37. 2005.

[KNL11] Z. Khai, A. Nadeem, G. Lee. A Prolog Based Approach to Consistency Checking of UML Class and Sequence Diagrams. *Software Engineering, Business Continuity, and Education*, pp. 85–96, 2011.

[LR07] M. Lawley, K. Raymond. Implementing a practical declarative logic-based model transformation engine. In *SAC'07: Proceedings of the 2007 ACM Symposium on Applied Computing*. Pp. 971–977. ACM, New York, NY, USA, 2007. doi:http://doi.acm.org/10.1145/1244002.1244216

[LS06] M. Lawley, J. Steel. Practical Declarative Model Transformation with Tefkat. In *MoDELS Satellite Events*. Pp. 139–150. LNCS 3844, Springer, 2006.

[OMG03] OMG. MDA Spec. Technical report, http://www.omg.org/mda/specs.htm, 2003.

[Sch09] B. Schätz. Formalization and rule-based transformation of EMF Ecore-based models. *Software Language Engineering*, pp. 227–244, 2009.

[Sto07] H. Storrle. A Prolog-based Approach to Representing and Querying UML Models. In *Intl. Ws. Visual Languages and Logic (VLL'07)*. Volume 274, pp. 71–84. 2007.

[Tra05] L. Tratt. Model transformations and tool integration. *Software and System Modeling* 4(2):112–122, 2005.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Intérprete PsiXML para gramáticas de mini-Lenguajes XML en aplicaciones Web

Enrique Chavarriaga[1], Fernando Díez[1] y Alfonso Díez[2]

[1]Escuela Politécnica Superior. Universidad Autónoma de Madrid
Avda. Tomás y Valiente 11. 28049 Madrid.
jesusenrique.chavarriaga@uam.es, fernando.diez@uam.es
[2]BET Value SLR
Fuentes 10. 28013 Madrid.
adiez@betvalue.com

Resumen: El uso de lenguajes y protocolos XML es una de las herramientas de trabajo más explotadas para la creación de aplicaciones Web. Lenguajes basados en XML como ASP.NET o Java Server Face, en combinación con lenguajes robustos de programación como C# y Java, respectivamente, se emplean con frecuencia para generar páginas dinámicas en la arquitectura servidor de un sistema. Por su parte, considerado como metalenguaje, XML permite definir gramáticas como XSLT, MathML, SVG, y/o SMIL que enriquecen el modelo de presentación de la página web. Actualmente la web 2.0 nos ofrece tecnologías, servicios y herramientas que permiten construir páginas verdaderamente funcionales, agradables y usables en la arquitectura cliente. En este trabajo combinamos la definición de una gramática de lenguaje específico XML con los beneficios de un intérprete de lenguaje de programación. Introducimos el concepto de mini-lenguaje XML como aquellas gramáticas basadas en un conjunto de *tags* asociados a un modelo de diagramas de clases, evaluables sobre un intérprete especializado XML. Este intérprete puede ser usado en diferentes facetas de una aplicación Web como componente reutilizable en el sistema. Incluiremos como caso de estudio el diseño de una página Web para el manejo de diferentes fuentes Web, en particular fuentes RSS.

Palabras Clave: Intérprete de Programación, Lenguajes XML, Gramáticas de Lenguajes XML, Fuentes RSS.

1 Introducción

En las últimas décadas el desarrollo de Internet ha constituido, de algún modo, una revolución tecnológica de facto a nivel mundial. La tecnología basada en el uso de la red como plataforma de difusión del conocimiento a todos los niveles ha posibilitado un crecimiento inconmensurable de multitud de nuevas oportunidades de negocio. En este marco dinámico de cambio global, el diseño y la creación de aplicaciones Web, basadas en la construcción y presentación de páginas Web (W3C: Web Design an Applications), supone en la actualidad un reto tecnológico. Son numerosas y muy diversas las tecnologías necesarias para la creación de valor en este mercado. Las principales tecnologías están basadas en HTML para el manejo de

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 213-227, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.