PTL: A Model Transformation Language based on Logic Programming

Jesús M. Almendros-Jiménez and Luis Iribarne

University of Almería, SPAIN, {jalmen,luis.iribarne}@ual.es

Jesús López-Fernández and Ángel Mora-Segura

Autonomous University of Madrid, SPAIN, {jesusj.lopez,angel.moras}@uam.es

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 a hybrid language in which *ATL (Atlas Transformation Language)*-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 implementation of PTL is based on the encoding of the ATL-style rules by Prolog rules. Thus, PTL makes use of Prolog as a transformation engine. We have provided a declarative semantics to PTL and proved the semantics equivalent to the encoded program. We have studied an encoding of *OCL (Object Constraint Language)* with Prolog goals in order to map ATL to PTL. Thus a subset of PTL can be considered equivalent to a subset of ATL. The proposed language can be also used for model validation, that is, for checking constraints on models and transformations. We have equipped our language with debugging and tracing capabilities which help developers to detect programming errors in PTL rules. Additionally, we have developed an Eclipse plugin for editing PTL programs, as well as for debugging, tracing and validation. Finally, we have evaluated the language with several transformation examples as well as tested the performance with large models.

Keywords: Logic Programming, Model Transformation, Software Engineering, Mode Driven Engineering, Domain Specific Languages

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, and provides a framework to developers for transforming their models. Therefore, *Model Transformation* [1, 2, 3, 4, 5] is a key technology of MDE. The *Model Driven Architecture (MDA)* proposed by the *Object Management Group (OMG)* [6], distinguishes between *Platform Independent Models (PIMs)* and *Platform Specific Models (PSMs)* as an abstraction mechanism from application domains, programming languages, etc.

A simple definition of a model transformation tool is that it is able to transform one model into another. We can take as an example of model transformation the *code generation* from a visual model for representing the architecture of a software system. For instance, most of the *UML (Unified Modeling Language)* [7] software development tools are able to generate code from UML class diagrams. In such a model transformation tool, the source model is the class diagram, and the target model is the code. However, model transformation is a more general technique for the transformation of models. In fact, usually *Model-to-Model (M2M)* and *Model-to-Code (M2C)* transformations are considered.

MDA proposes (at least) three elements in order to describe a model transformation. The first one are the so-called *meta-meta-models* which are the basis of the model transformation and provide 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-models. 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 also instances of the meta-meta-models. A model transformation maps the source and target models but the transformation is defined with regard to the source and target meta-models.

In this context, there are several proposals (see [8] for a survey) of languages whose aim is the specification of transformations. One of the most relevant is the language *ATL (Atlas Transformation Language)* [9]. ATL is a *domain-specific language* for specifying model-to-model transformations. ATL is a hybrid language, and provides a mixture of declarative and imperative constructors. The declarative part of ATL is based on rules. Such

rules define a source pattern matched to 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.

Model transformation languages should be equipped with the following features.

(a) Firstly, they should be equipped with a well-founded semantics. It is crucial, from an user's point of view, for the understanding of language constructs, enabling a high level and non-procedural interpretation of the behavior of programs. It is also key from the tool development point of view, in order to provide foundations to language interpreters and other tools: compilers, debuggers, etc.

(b) Secondly, they should separate specification from execution, and high-level specifications of transformations should be independent from implementation, hiding low-level details like control flow and operational semantics.

(c) Thirdly, constraint validation should be specified in a more formal language, avoiding procedural mechanisms, and working in a declarative style.

(d) Finally, high-level information should be provided from debugging and tracing. Debugging and tracing should focus on meta-models, model elements and rules, instead of variables values and step by step executions.

Unfortunately, current model transformation languages do not cover all these features, and thus, the study of languages covering all of them should be object of study.

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 a hybrid language in which ATL-style rules are combined with logic rules for defining transformations. The meta-meta-model of PTL is the *EMF (Eclipse Modeling Framework) ECore* meta-meta-model. PTL works with source and target models of ECore. ATL-style rules are used to define mappings from source models to target models while logic rules are used as helpers.

The aim of our work is to provide a framework for model transformation based on logic programming. From a practical point of view, our proposal

can be seen as an application of logic programming to a context in which rule-based systems are required. Our language provides the elements involved in model transformation: meta-models handling and mapping rules. The use of an ATL-style transformation language combined with Prolog rules enables logic programming experts to develop transformations with a logic taste. The hybrid nature of the language (ATL-style and logic rules) makes our language a suitable framework for Prolog programmers. Thus, we believe that our contribution can be interesting to the logic programming community. Due to the wide acceptance of ATL as transformation language, the adoption of a syntax inspired by ATL to write model transformations makes our language easier to use and close to other proposals of transformation languages. The adoption of ATL style syntax facilitates the definition of mappings: basically, source models are mapped to target models of the given source and target meta-models.

From a theoretical point of view, we have studied a declarative semantics for PTL. The declarative semantics interprets meta-models and identifies models that conform to meta-models, while the interpretation of PTL rules provides semantics to the main ATL constructs. From a practical point of view, 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 also studied how to use our language for model validation, in particular, how to validate source and target models, and transformations. Source and target models are validated by considering constraints, and transformations are validated by considering cross constraints on source-target models. Prolog goals are used with this end.

We have equipped our language with debugging and tracing capabilities, which help developers detect programming errors in PTL rules. Debugging permits to detect rules that fail in a certain transformation, and in addition, debugging is able to locate the point of the rule that makes it fail. The failure of a rule comes from (g.1) Boolean conditions that cannot be satisfied, and (g.2) objects of the target model that cannot be created. When a certain rule fails, a certain target model element could be missing, and therefore the transformation could be erroneous. The debugger is able to give (g.1) the name of the rule and the Boolean condition that is false for all the elements of the source model, and (g.2) the name of the rule in which the target model cannot be created, and the failed binding. Tracing permits one 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. Debugging is useful when the target models are *incomplete* while tracing is useful when the target models are *incorrect*.

1.1. Structure of the Paper

The structure of the paper is as follows. Section 2 will motivate the use of Prolog in Model Transformation. Section 3 will summarize the contributions of the approach according to criteria (a), (b), (c) and (d) of Section 1. Section 4 will introduce PTL, a case study, PTL declarative semantics as well as some examples of use. Section 5 will describe the implementation of PTL, prove the soundness and completeness of the implementation, and present the PTL interpreter. Section 6 will show how to map ATL to PTL and how to carry out model validation. Section 7 will present the debugger and tracer, the Eclipse plugin and performance results. Section 8 will review related work. Finally, Section 9 will conclude and present future work.

2. Model Transformation and Prolog

From the early years of model transformation, Prolog has been present as a tool to specify simple transformations [10, 11, 12, 13]. Prolog has been also present in early relevant proposals like VIATRA [14]. Model transformation experts found in logic programming a resource to implement and experiment with transformations and techniques related to transformations. With the development of the technology around model transformation, logic programming is not abandoned [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26], given that experts investigate new aspects of model transformation that existent languages and tools do not cover. Moreover, some authors argue that Prolog is still suitable to learn about model transformations. For instance, in [27] they show how to use Prolog for expressing constraints and model-to-model transformations. They argue that teaching MDE with *MDELite*, a framework which makes use of Prolog as constraint and transformation language, presents many fewer problems with regard to Eclipse MDE tools. They also argue that the understanding of what a meta-model/model is, as well as the specification of transformations with Prolog, is easier from a relational-based representation of models. Writing transformations with Prolog when tuples are used becomes a very easy task. In [28], they argue that declarative approaches concentrate on what relationships exist between the source

and target, compared with imperative approaches which concentrate on how to explicitly transform from the source to the target. Complete and correct transformations can be more probably obtained from declarative transformations which do not have to take into account execution order, source traversal and target creation.

Several works [21, 25, 26] prove that Prolog is useful for the specification of constraints, a key element of model transformation. Firstly, a meta-model is usually a class diagram which basically is a graph with two kinds of nodes: classes and attributes, and two kinds of edges: attribute memberships and associations. The representation of a graph in Prolog is easy, and most of graph operations can be specified with little effort with Prolog predicates. Prolog is untyped and its relational tuple-based structure makes easy the definition of join-based operations, as well as recursive definitions. The success of graph/tuple based languages in constraint specification and validation demonstrates the need for easy to use constraints languages [29, 30].

In [31], they argue that imperative approaches allow the specification of complex transformations more easily than declarative approaches, but induce more overhead code as many issues have to be accomplished explicitly, e.g., specification of control flow. They reason why specifications of transformation are still an error prone tasks as well as tedious: (1) Firstly, languages as ATL, TGGs and QVT work on a high level of abstraction while execution engines work on a low level. As a consequence, debugging, which is a also key element in transformations, is limited to information provided by the engines, which consists of variable values and logging messages. More relevant information like which parts of the transformation are executed is missing. Execution engines act as a black-box for the developer hiding the operational semantics. (2) Secondly, current transformation languages provide a limited view of the transformation: meta-models, models and transformation code as well as trace information are scattered across different artifacts. Normally, current approaches hide the transformation of concrete model elements. Finally, they argue that Prolog-based approaches do not introduce a gap between specification and execution. Although ATL offers facilities for code edition and compilation, when running an ATL program little information is provided to the programmer. When something is wrong, for instance, due to OCL code, information is not provided about where the bug can be found. Debugging with breakpoints, step-by-step transformation execution, and variable values is not enough because the first think one wants to know is which rule is not working, why and where. Debugging support has been

included in some tools: ATL, GReAT, VIATRA, FUJABA, Tefkat but in [28] they identify several debugging questions to be answered, according to three groups: logical bugs (violation of a constraint between the source and target models), well-formedness bugs (violation of a constraint of the target models) and bug smells (relationship between source, target and transformation). They also establish 4 groups for tracing: tracing from a target object to its contributors, tracing from source objects to target objects, source objects that contributed to the creation of target objects; and source objects that did not contribute to the creation of a target object.

Finally, Bernhard Schätz [15] motivates that one of the advantages of using Prolog is its capability to interpret loose characterizations of the resulting model, supporting the exploration of a set of possible solutions. By using Prolog backtracking alternative transformation results can be generated, in order to find an optimized solution, for instance, with respect to a given metric. Also inversion of transformations and thus, the consideration of bidirectional transformations [32, 24], is an open research line in which logic programming can find an application domain. Although these mechanisms are out of the scope of the paper, we find that the use of logic programming for model transformations is hardly a new research line.

3. Contributions of the Approach

PTL tries to cover all the features previously considered (i.e., (a), (b), (c) and (d) of Section 1).

(a) Firstly, a declarative semantics has been provided to the language. Meta-model and model semantics are provided, as well as a formal definition of the PTL constructors in terms of model semantics. Additionally, an equivalence result between the declarative semantics and the operational semantics is proved.

(b) Secondly, specification is separated from execution. Declarative style is used to express both transformation rules and constraint validation. OCL is replaced in PTL by Prolog which gives a more declarative taste to transformation rules and constraint validation. The main purpose of helpers in ATL is to serve as query language against the source models. With this aim, OCL is used in helpers code. In our logic programming based approach, logic (i.e., Prolog style) rules serve as query language. PTL adopts Prolog as language for defining helpers. Prolog

is a fully-fledged programming language equipped with a very large library in many of its implementations (for instance, in SWI-Prolog there are libraries for the development of Web applications, and libraries of constraints solvers, among others). Prolog is useful for several tasks, including search of (optimal) solutions thanks to *backtracking* and the use of the so-called *logic variables* with existential meaning, which can be bound at run-time. Additionally, Prolog is recursive by nature, and is able to handle recursive relations in a very simple and effective way. As an example of use of Prolog for recursive relations, Prolog can be used for *OWL RL reasoning* [33, 34]. In the context of model transformation, Prolog can be used for *materialization* of ontologies, i.e., computing all the logic consequences of a given ontology, which is usually known as ontology reasoning. As far as we know, the handing of recursion is a limitation of OCL: only well-founded recursive queries are supported (e.g. transitive closure of non-DAG graphs) [29]. In the Appendix A we will show an example (Example 3) of materialization. Additionally, OCL is typed and it limits the number of constraints to be expressed. OCL requires in some cases to cast from one type to another. For instance, OCL forces to convert from singleton sets to elements, and back, due to cardinality restrictions. OCL union operators is also limited to elements of the same type. Thus, the adoption of OCL by ATL has as consequence that some transformations and constraints are hard to specify if not impossible.

(c) Thirdly, constraint validation are expressed by Prolog goals making easy the definition of non-procedural based specifications.

(d) Fourthly, the PTL execution is controlled by Prolog, and debugging and tracing of executions handle directly meta-models and model elements, as well as Prolog rules, allowing a high-level specification of debugging as well as tracing. The three methods: debugging, tracing and validation all together offer a repertoire of tools to detect bugs in transformations.

Additionally, having Prolog as execution engine provides an opportunity to develop many of the key elements of model transformation. Prolog has as advantage that the implementation of debuggers, tracers, test case generators, constraint validators, etc., is easy. In our approach we have experimented with test case generators in our performance tests of PTL and ATL. Prolog can be used with little effort to generate random black-box based

test cases. A more elaborated technique for random test cases generation is considered as future work.

In order to evaluate our approach we have studied how to map ATL to PTL. We have studied how to map OCL to Prolog. We have detected a fragment of OCL (an extension of OCL-Lite [35]) for which an encoding by Prolog goals can be defined. It permits to prove that a subset of PTL is equivalent to a subset of ATL. In other words, PTL can be seen as an implementation (*Prolog-based virtual machine*) of a subset of ATL. The subset of ATL covered by PTL is basically the declarative part of ATL plus a fragment of OCL. Thus, PTL provides a declarative semantics for this subset of ATL. Additionally, we can see the Prolog encoding of PTL as a Prolog-based semantics of ATL similar to the proposal of [36] based on Rewriting Logic. Unfortunately, the subset covered by PTL is not full ATL, but an extension is considered as future work. Both (i.e., Prolog-based virtual machine and declarative semantics) can be also seen as a contribution of our work. As a consequence of the mapping of ATL to PTL we are able to evaluate our approach with examples of ATL.

We have tested our implementation with transformations on medium-size models (i.e., thousands of objects), and we have compared execution times with the *ATL/OCL EMF-specific Virtual Machine*. We have also compared execution times for constraints on large modes of standard benchmarking datasets with Prolog.

We have developed an Eclipse plugin in order to integrate editing of PTL programs, execution of transformations as well as debugging of code, tracing of executions, and validation of transformations. The plugin allows to store models and meta-models as well as the source code of transformations and validation rules. The source code of PTL (interpreter, debugger, tracer and validator), together with the examples developed in the paper can be downloaded from[1]. In this site the Eclipse plugin is also available.

Finally, let us remark that this paper is an extended and improved version of our previous work [37], and takes some elements previously studied in [38, 39]. With respect to [37], a more detailed description of the declarative semantics of PTL is provided and proofs of soundness and completeness are included. In addition, a mapping from ATL to PTL, more detailed descriptions of the interpreter, debugger and tracer are given, and the validator is

[1] `http://indalog.ual.es/mdd/ptl2`

metamodel	:= **metamodel** '(' mm_name '[' *definitions* ']' ')'
definitions	:= *definition* | *definition definitions*
definition	:= **class** '(' class_name '[' *attributes* ']' ')' |
	role '(' role_name ',' class_name ',' class_name ',' lower ',' upper [',' container] ')'
attributes	:= attribute_name | attribute_name ',' *attributes*
rule	:= **rule** rule_name **from** *patterns* [**where** *condition*] **to** *objects*
patterns	:= *pattern* | '(' *pattern,...,pattern* ')'
condition	:= *bcondition* | *condition* **and** *condition*
bcondition	:= *expr* == *expr* | *expr* =\= | *expr* |
	expr > *expr* | *expr* >= *expr* |
	expr < *expr* | *expr* <= *expr*
pattern	:= pattern_name **:** mm_name **!** class_name
objects	:= *object,...,object*
object	:= *pattern* '(' *binding,...,binding* ')'
expr	:= value |
	pattern_name |
	pattern_name**@**attribute_name |
	pattern_name**@**role_name |
	pattern_name**@**role_name**@**attribute_name |
	helper_name '(' *expr,...,expr* ')' |
	resolveTemp'(' '(' *expr,...,expr* ')' ',' pattern_name')' |
	sequence'(' '[' *expr,...,expr* ']' ')'
binding	:= attribute_name <- *expr* | role_name <- *expr*

Figure 1: PTL syntax

now included. Finally, the performance evaluation and the description of the Eclipse plugin is also a new contribution of this paper.

4. Prolog Based Transformation Language (PTL)

In this section, we present the main elements of PTL. Firstly, we will define the syntax of PTL programs. We will show a case study to illustrate the language, used in the rest of the paper. Secondly, we will give semantics to PTL constructors. Finally, we will show some examples of use.

A PTL program consists of (a) **meta-model definitions** (source and target meta-models), (b) **mapping rules** and (c) **helpers**. Meta-model definitions define meta-model elements: class and roles together with attributes for classes, and the cardinality of roles. Mapping rules have the form:

rule *rule_name* **from** *patterns* **where** *boolean_condition* **to** *objects*

Helpers are defined by Prolog rules. The syntax of elements (a) and (b) is shown in Figure 1, where *mm_name*, *class_name*, *attribute_name*, *role_name*, *rule_name*, *pattern_name* and *helper_name* are user defined names and *value* can be any Boolean, integer, string, etc. *Lower* and *upper*

are taken from '0','1',..., and '*'. PTL is an untyped language, the inclusion of types will be considered in the future.

A mapping rule maps a set of objects of the source models into a set of objects of the target models. The rule can be conditioned by a Boolean condition, including equality (i.e. ==) and inequality (i.e. =\=), and the *and* operator. The rule condition starts with *where*. Objects of target models are defined assigning values to attributes and roles, and they can make use in their definition of attribute and role values of the source models, together with *resolveTemp*, helpers and the *sequence* construction. The *resolveTemp* function permits to assign objects created by another rule. 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. The *sequence* construction makes possible to assign more than one role end to a role.

4.1. Declarative Semantics of PTL

PTL has a declarative semantics whose basis is the encoding of PTL by logic programming. The declarative semantics is based on the interpretation of meta-models and PTL rules.

Definition 1 (Metamodel)**.** *Assuming a set $\mathcal{D}$ of domain values partitioned into k domains $d_1, \ldots, d_k$, then a meta-model $\mathcal{MM}$ is a quadruple $\mathcal{MM} = (\mathcal{C}, \mathcal{A}, \mathcal{R}, \mathcal{H})$ where $\mathcal{C}$ is a set of class names $C_1, \ldots, C_n$, $\mathcal{A}$ is a set attribute names, $\mathcal{R}$ is a set of role names $r_1, \ldots, r_m$ and $\mathcal{H}$ is a set of helper names $h_1, \ldots, h_s$; where each class name C_i has an associated set of attribute names $att_1^i, \ldots, att_{l_i}^i \in \mathcal{A}$, where l_i is the number of attributes of the class C_i. In addition, role names r_p have a defined domain (respectively, range), denoted by $domain(r_p)$, (respectively, $range(r_p)$), which is a class name in $\mathcal{C}$. Attributes and helpers have also a domain and range, where $att_j^i \in \mathcal{A}$ has as domain C_i and as range some domain of $\mathcal{D}$; and $h_w : C_{w_1} \ldots C_{w_t} \rightarrow C_{w_{(t+1)}}$ where $w_i \in \{1, \ldots, n\}$. Finally, roles r_p have an associated lower and upper cardinality, $lower(r_p) \in I\!N$ and $upper(r_p) \in I\!N \cup \{\infty\}$.*

Definition 2 (Model)**.** *A model $\mathcal{M}$ is a tuple $\mathcal{M} = (\mathcal{C}^{\mathcal{M}}, \mathcal{A}^{\mathcal{M}}, \mathcal{R}^{\mathcal{M}}, \mathcal{H}^{\mathcal{M}}, \mathcal{O}^{\mathcal{M}})$, which instantiates the elements of a meta-model (i.e., interprets a meta-model), where $\mathcal{O}^{\mathcal{M}}$ is a domain of object names. $\mathcal{C}^{\mathcal{M}}$ is an interpretation $C_1^{\mathcal{M}}, \ldots, C_n^{\mathcal{M}}$ of the class names which are disjoint subsets of $\mathcal{O}^{\mathcal{M}}$. $\mathcal{A}^{\mathcal{M}}$ interprets attributes as a set of (partial) functions $att_j^{<\mathcal{M},i>}$ from $\mathcal{O}^{\mathcal{M}}$ to elements of $\mathcal{D}$, and $\mathcal{R}^{\mathcal{M}}$ interprets roles as a set of (partial) functions $r_1^{\mathcal{M}}, \ldots, r_m^{\mathcal{M}}$ from*

$\mathcal{O}^{\mathcal{M}}$ *to a subset of* $\mathcal{O}^{\mathcal{M}}$*. Finally* $\mathcal{H}^{\mathcal{M}}$ *interprets helpers as (partial) functions* $h_w^{\mathcal{M}} : C_{w_1}^{\mathcal{M}} \ldots C_{w_t}^{\mathcal{M}} \rightarrow C_{w_{(t+1)}}^{\mathcal{M}}$ $w_i \in \{1, \ldots, n\}$.

Definition 3 (Conformance)**.** *A model* $\mathcal{M}$ *conforms to* $\mathcal{MM}$ *whenever* $r_p^{\mathcal{M}}(o_i) \in C_j^{\mathcal{M}}$ *and* $lower(r_p) \leq \#r_p^{\mathcal{M}}(o_i) \leq upper(r_p)$*, for each* $r_p \in \mathcal{R}$ *and* $o_i \in C_i^{\mathcal{M}}$*, where* $domain(r_p) = C_i$ *and* $range(r_p) = C_j$*;* $att_j^{<\mathcal{M},i>}(o_i) \in d_l$ *for each* $o_i \in C_i^{\mathcal{M}}$ *where* $domain(att_j^i) = C_i$ *and* $range(att_j^i) = d_l$.

Definition 4 (Union of Models)**.** *Given two models* $\mathcal{M}_1$ *and* $\mathcal{M}_2$ *of a meta-model* $\mathcal{MM}$*, we can build the union of models as follows* $C^{\mathcal{M}_1 \cup \mathcal{M}_2} = C^{\mathcal{M}_1} \cup C^{\mathcal{M}_2}$ *for each* $C \in \mathcal{C}$ *of* $\mathcal{MM}$*,* $\mathcal{A}^{\mathcal{M}_1 \cup \mathcal{M}_2}$*,* $\mathcal{R}^{\mathcal{M}_1 \cup \mathcal{M}_2}$ *and* $\mathcal{H}^{\mathcal{M}_1 \cup \mathcal{M}_2}$ *are the union of the graphs of each function*[2]*, and* $\mathcal{O}^{\mathcal{M}_1 \cup \mathcal{M}_2} = \mathcal{O}^{\mathcal{M}_1} \cup \mathcal{O}^{\mathcal{M}_2}$.

Let us remark that the union of two models that conform to a meta-model does not necessarily conform to the meta-model due to cardinality restrictions. Model unions will be used in the definition of the declarative semantics of PTL.

Now, we can provide declarative semantics to PTL as follows. Given a *source model* $\mathcal{M}$, a PTL program $\mathcal{P}$ with rules $r_1, \ldots, r_n$*, defines the target model* $\mathcal{M}' = \cup_{1 \leq i \leq n} [\![r_i]\!]^{\mathcal{M}}$, where $[\![r]\!]^{\mathcal{M}}$ denotes the model obtained by r from $\mathcal{M}$, which is defined in Figure 2. The main cases are the following. Case (Rule) defines the semantics of a PTL rule. The semantics is defined as the union of the target objects obtained from bindings to source objects that satisfy the Boolean condition. Cases (Patt1) and (Patt2) bind source patterns to object names of the source model. Cases (Mod1) and (Mod2) create the target object from the target pattern, and attribute and role bindings are obtained in cases (Mod3) and (Mod4). Cases from (Expr1) to (Expr7) and from (Bool1) to (Bool3) cover with the value of an expression.

Notation:
We have adopted the following notation.

(1) The expression $[\![expr]\!]^{\mathcal{M}}_{\overline{(pn,v)}}$ where $\overline{(pn, v)} \equiv (pn_1, v_1), \ldots, (pn_n, v_n)$ denotes the **value** of *expr* in the model $\mathcal{M}$, with respect to the assignments $pn_1 \rightarrow v_1, \ldots, pn_n \rightarrow v_n$ of pattern names to object names of the model $\mathcal{M}$

[2]Assuming that at arguments with inconsistent results, the result is undefined.

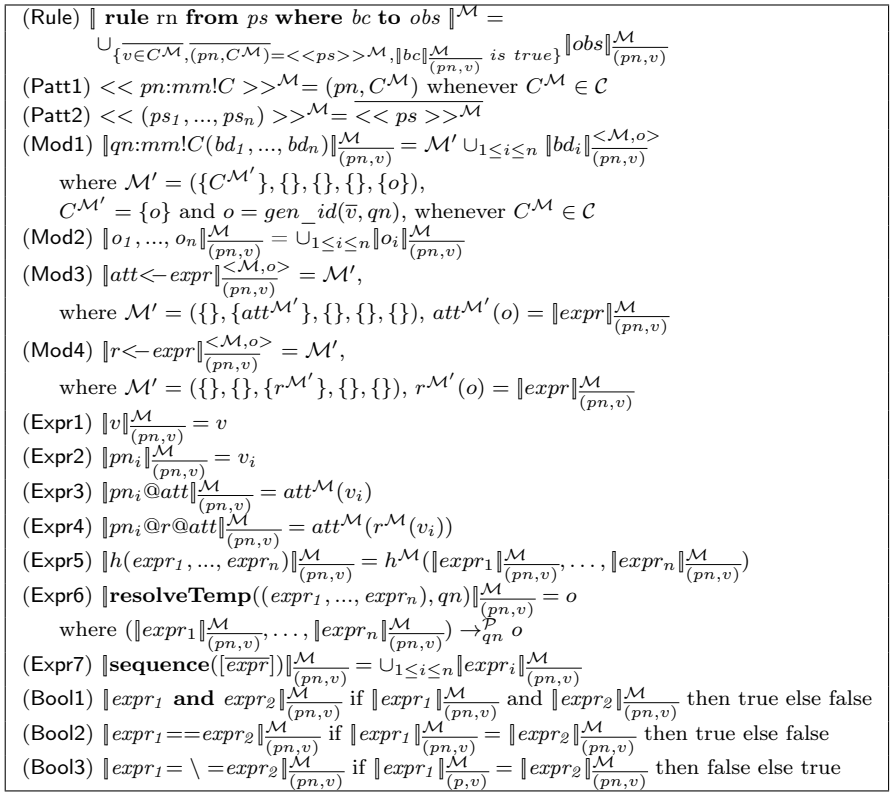


Figure 2: Declarative Semantics of PTL

(2) The expression $[\![expr]\!]^{<\mathcal{M},o>}_{(pn,v)}$ denotes the **value** of $expr$ in the object o replacing $pn_1 \rightarrow v_1, \ldots, pn_n \rightarrow v_n$ in $\mathcal{M}$

(3) The expression $[\![bd]\!]^{\mathcal{M}}_{(pn,v)}$ where bd is a binding represents the **model** obtained replacing $pn_1 \rightarrow v_1, \ldots, pn_n \rightarrow v_n$ of pattern names to object names of the model $\mathcal{M}$

(4) The expression $[\![o]\!]^{\mathcal{M}}_{(pn,v)}$ where o is an object represents the **model** obtained with respect to the assignments $pn_1 \rightarrow v_1, \ldots, pn_n \rightarrow v_n$ of pattern names to object names of the model $\mathcal{M}$

(5) The expression $\overline{v \in C^{\mathcal{M}}}$ denotes the sequence $v_1 \in C_1^{\mathcal{M}}, \ldots, v_n \in C_n^{\mathcal{M}}$

(6) The expression $\overline{<< ps >>}^{\mathcal{M}}$ denotes the sequence $<< ps_1 >>^{\mathcal{M}}, \ldots,$ $<< ps_n >>^{\mathcal{M}}$ and represent a sequence of assignments of pattern

names to class interpretations $\overline{(pn, C^{\mathcal{M}})} = (pn_1, C_1^{\mathcal{M}}) \ldots, (pn_n, C_n^{\mathcal{M}})$

(7) The expression $(v_1, \ldots, v_n) \rightarrow_p^{\mathcal{P}} o$, used in the *resolveTemp* definition (case (Expr6)), denotes that a sequence of object names $(v_1, \ldots, v_n)$, $v_i \in \mathcal{O}^{\mathcal{M}}$, $1 \leq i \leq n$, is transformed into the **object** o of pattern name p with regard to $\mathcal{P}$, that is, there exists (*rule rn from ps where bc to obs*) $\in \mathcal{P}$, $ps \equiv ps_1, \ldots, ps_n$ such that $<< ps >>^{\mathcal{M}} = \overline{(pn, C^{\mathcal{M}})}$, $\overline{v \in C^{\mathcal{M}}}$, $[\![bc]\!]^{\mathcal{M}}_{\overline{(pn,v)}}$ is true, and $gen\_id(\bar{v}, p) = o$

(8) Finally, *gen_id* is a bijective function that takes n object names of $\mathcal{M}$ and a pattern name p and returns an object name of $\mathcal{M}'$. We assume that rules cannot map the same objects to the same object name (as usual in ATL).

4.2. Case Study

In this section we would like to summarize the main elements of the model transformation setting. With this aim, we describe a case study, used in the rest of the paper.

4.2.1. Source and Target Models

The model A of Figure 3 represents the modeling of a database. We will call this kind of modeling *"entity-relationship"* modeling of a database in contrast to the model B of Figure 4 which will be called *"relational"* modeling of a database.

The model A of Figure 3 can be summarized as follows. *Data* (i.e. entities) are represented by classes (i.e., *Student* and *Course*), including attributes. *Stores* are defined for each data (i.e., *DB_Students* and *DB_Courses*); stores are composed of data, therefore specifying a composition relationship between store and data. Stores are unique for each data. *Relations* are represented by associations and relation names are association names. Besides, *roles* are defined (i.e., *the_students*, *the_courses*, *is_registered* and *register*). Data *attributes* are class attributes. Each data has a unique key attribute. It can be considered as a *constraint* of the source model of the proposed transformation. Relations can be adorned with *qualifiers* and *navigability*. Qualifiers are used to specify the key attributes of each data becoming foreign keys of the corresponding association. Therefore qualifiers have to be selected from the key attributes of the corresponding data. Such a requirement can be considered as a *constraint* of the source model of the proposed

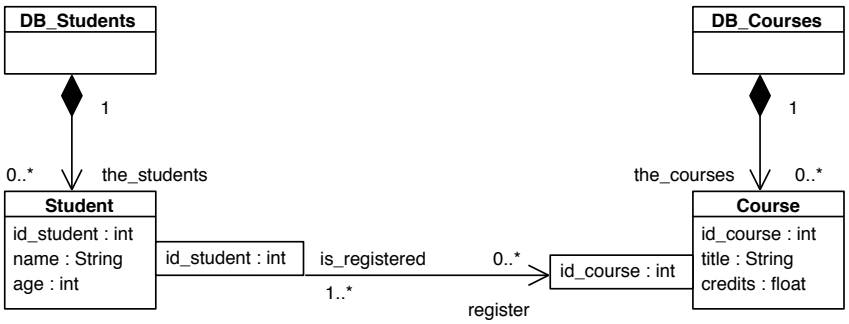


Figure 3: Entity-relationship modeling of the Case Study

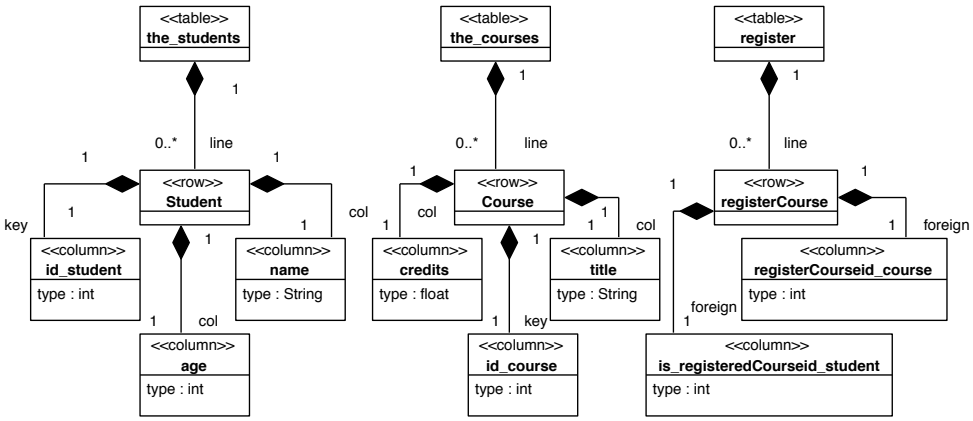


Figure 4: Relational modeling of the Case Study

transformation. The navigability describes which of them can be accessed and which mapping should be efficiently implemented.

Figure 4 shows the relational modeling of the same database. Such modeling is also defined by a class diagram. In model B, *Tables* are composed of *rows*, and rows are composed of *columns*. 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*.

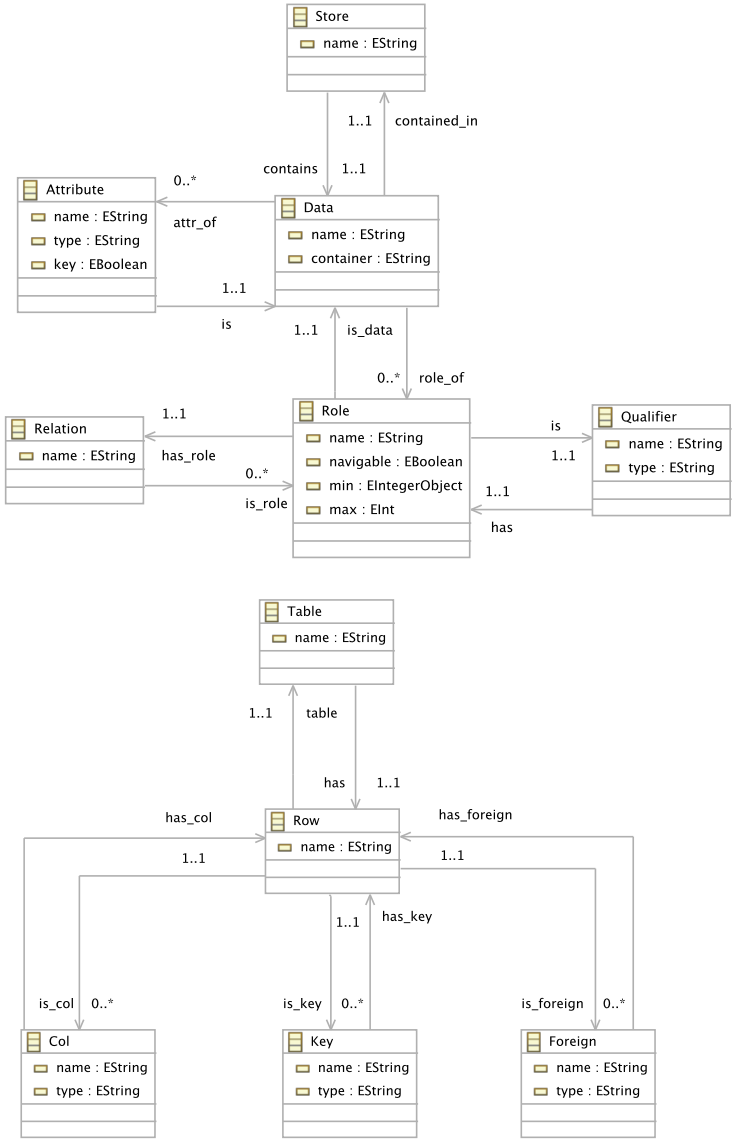


Figure 5: EMF-based Meta-model of the Source/Target Models

4.2.2. Source and Target Meta-models

Figure 5 represents the EMF meta-models A and B of both types of modeling. In the meta-model A, *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 class *Student* and class *Course* are instances of the class *attribute*. Also *is_registered* and *register* are instances of the class *role*, and *id_student*, *id_course* of the class *qualifier*. In the meta-model B, 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.

4.2.3. Transformation

Now, the problem of model transformation is how to transform a model of type A (like Figure 3) into a model of type B (like Figure 4). The transformation is as follows.

The transformation generates two tables: *the_students* and *the_courses*. Each table includes three columns that are grouped into rows. The table *the_students* includes for each student the attributes of *Student* of Figure 3. 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, called *register*, is generated to represent the assignments of students to courses, using *registerCourse* as name of the row. The columns *is_registeredCourseid_student* and *registerCourseid_course* taken from qualifiers, play the role of foreign keys.

4.2.4. Validation

Now, we show how source and target models are validated by considering constraints, and transformations are validated by considering cross constraints on source-target models. We can see these constraints in Figure 6. Constraints are specified by PTL programmers and should be checked for each transformation. Using the Eclipse plugin, constraints can be written in several files and checked for the transformation. It is worth observing that constraints about source and target models in isolation are insufficient for proving the soundness of the transformation. For instance, a target model can have foreign keys, but a cross constraint is needed: foreign key names are concatenations of role, role end and key names. We will show in Section 6.1 that such constraints can be expressed by OCL, and how to translate OCL expressions into Prolog goals.

4.3. PTL Example

Now, we describe how the transformation can be implemented in our PTL language. PTL consists of **meta-model definitions** (see Figure 7), **mapping rules** (see Figure 8) and **helpers** (see Figure 9).

	Constraints on Source Models
(v1)	All attributes of a data have distinct names
(v2)	Each data has a unique key attribute
(v3)	All data have distinct names
(v4)	All data have distinct containers
(v5)	All qualifiers are key attributes
(v6)	All role names of a data are distinct
	Constraints on Target Models
(v7)	All table names are distinct
(v8)	All row names are distinct
(v9)	All rows have either all keys and columns or all foreign keys
	Cross Constraints on Source-Target Models
(v10)	Key and column names and types are names and types of data attributes
(v11)	Table names are either container names or role names
(v12)	Row names are either data names or concatenations of role and role ends
(v13)	Foreign key names are concatenations of role, role end and key names

Figure 6: Model validation: constraints

```
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,"0","*"),
   role(has,qualifier,role,"1","1"),
   role(is,role,qualifier,"1","1"),
   role(is_data,role,data,"1","1"),
   role(role_of,data,role,"0","*")
]
).
```

Figure 7: Example of meta-model definition in PTL

Figure 7 shows the definition of the meta-model A of Figure 5. Classes and roles are defined together with attribute names and cardinality restric-

```
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),
        resolveTemp((inverse1_qualifier(p),inverse2_qualifier(p)),f2)]))
     ).
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<-resolveTemp(inverse_row(p),r)
            )
    ).
```

Figure 8: Examples of rules in PTL

```
inverse_row(IdQ,IdRole2):-qualifier_has(er,IdQ,IdRole),
            role_has_role(er,IdRole,IdAss),
            relation_is_role(er,IdAss,IdRole2),
            role_navigable(er,IdRole2,true).
```

Figure 9: Example of helper in PTL

tions. The rule *table2_er2rl* of Figure 8 defines how navigable roles are transformed into tables and rows (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 role *is_foreign*. The role *is_foreign* contains a sequence of two elements (*is_registeredCourseid_student* and *registerCourseid_course*). For this reason the *sequence* constructor is chosen. Besides, *resolveTemp* retrieves with two helpers called *inverse1_qualifier*, *inverse2_qualifier* the elements *is_registeredCourseid_student* and *registerCourseid_course*. The rule *foreign2_er2rl* of Figure 8 computes the *foreign* class *is_registeredCourseid_student*, which is computed from roles and qualifiers which are not navigable. The rule also sets the role *has_foreign* with a helper called *inverse_row*, which computes the row *registerCourse*. Figure 9 shows the definition of the helper *inverse_row*. Helpers make use of the

Prolog meta-model library (see Section 3).

It is worth observing that helpers are defined with the following convention: helpers can be defined in PTL 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. The full PTL program of the case study is shown in the Appendix B and can be downloaded from [3]. In the Appendix A we have included a more complete batch of examples developed with PTL.

5. Encoding with Prolog rules

Now, we show how PTL mapping rules are encoded by Prolog rules.

5.1. Prolog library for meta-models

```
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 10: Prolog library

For each meta-model definition a Prolog library is automatically generated. For instance, the Figure 10 contains (some of) the predicates generated to handle the *er* meta-model of Figure 7. We have three kinds of predicates: (a) those for *accessing class attributes* (for instance, *data_container* and *attribute_type*) (b) those for *accessing roles* (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 predicates (a) and (c) 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 predicates of type (b) call to a predicate called *associationEnds*, representing role end objects. The *associationEnds* predicate has as arguments: the name of the meta-model,

[3] `http://indalog.ual.es/mdd/ptl2`

$$
\begin{array}{l}
(r1)\ object(mt_i, ct_i, Var, \overline{Var}, [att_1(Var'_1), \ldots, att_n(Var'_n)], pnt_i) :- \\
\qquad rn(\overline{Var}), \\
\qquad enc(expr_1, \overline{(pns, Var)}, Var'_1), \ldots, enc(expr_n, \overline{(pns, Var)}, Var'_n), \\
\qquad gen\_id(\overline{Var}, pnt_i, Var). \\
\\
(r2)\ associationObjects(mt_i, r_j, Var, Var'_j) :- rn(\overline{Var}), \\
\qquad enc(expr'_j, \overline{(pns, Var)}, Var'_j), \\
\qquad object(mt_i, ct_i, Var, \overline{Var}, \_, pnt_i). \\
\\
(r3)\ rn(\overline{Var}) :- cs_1\_id(ms_1, Var_1), \ldots, cs_n\_id(ms_n, Var_n), \\
\qquad enc(bc, \overline{(pns, Var)}).
\end{array}
$$

Figure 11: Encoding of PTL

the name of the role, and the identifiers of the role ends. Predicates *object* and *associationEnds* will be stored as Prolog facts representing the elements of a certain (source and target) model.

5.2. Rules encoding

Given a PTL mapping rule of the form:

rule *rn* **from** *ps* **where** *bc* **to** *obs*

where $ps \equiv \overline{pns : ms!cs}$ and $obs \equiv \overline{pnt : mt!ct(\overline{bd})}$ then the encoding (for each $pnt_i : mt_i!ct_i(\overline{bd}_i)$) is defined in Figure 11 where in (r1) att_p<-$expr_p$ are the bindings to attributes in $\overline{bd}_i$ and in (r2) r_j<-$expr'_j$ are the bindings to roles in $\overline{bd}_i$.

The predicate *object* encodes the creation of objects of the target model. The predicate *associationObjects* encodes the creation of links between objects of the target model. There are rules *object* and *associationObjects* 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 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 *gen_ id* is used. The Prolog library for meta-models and the representation of models by Prolog facts makes possible to encode PTL rules.

(e1) $expr \equiv v$ then $Var = v$

(e2) $expr \equiv pns_j$ then $Var = Var_j$

(e3) $expr \equiv pns_j@att_j$ then $cs_j\_att_j(ms_j, Var_j, Var)$

(e4) $expr \equiv pns_j@r_j$ then $associationEnds(ms_j, r_j, Var_j, Var)$

(e5) $expr \equiv pns_j@r_j@att_j$ then
$associationEnds(ms_j, r_j, Var_j, Var_c), C\_att_j(ms_j, Var_c, Var)$
whenever $range(r_j) = C$

(e6) $expr \equiv h(\overline{expr})$ then
$\overline{enc(expr, \overline{(pns, Var)}, V)}, h(V_1, \ldots, V_n, Var)$

(e7) $expr \equiv resolveTemp(\overline{expr}, qn)$ then
$\overline{enc(expr, \overline{(pns, Var)}, V)}, object(\_, \_, Var, \overline{V}, \_, qn)$

(e8) $expr \equiv sequence([\overline{expr}])$ then
$\overline{enc(expr, \overline{(pns, Var)}, Var)}$

Figure 12: Encoding of expressions

(bc1) $expr \equiv expr_1$ *and* $expr_2$ then
$enc(expr_1, \overline{(pns, Var)}), enc(expr_2, \overline{(pns, Var)})$

(bc2) $expr \equiv expr_1 == expr_2$ then
$enc(expr_1, \overline{(pns, Var)}, Var), enc(expr_2, \overline{(pns, Var)}, Var)$

(bc3) $expr \equiv expr_1 =\backslash= expr_2$ then
$enc(expr_1, \overline{(pns, Var)}, Var_1), enc(expr_2, \overline{(pns, Var)}, Var_2), Var_1 \backslash= Var_2$

Figure 13: Encoding of Boolean expressions

In the previous definition, $enc(expr, \overline{(pns, Var)}, Var)$ is a conjunction of Prolog atoms, encoding an expression $expr$ in a variable Var, with respect to an assignment of pattern names to variables $pns_1 \rightarrow Var_1, \ldots, pns_n \rightarrow Var_n$. It is defined in Figure 12. Finally, $enc(bc, \overline{(pns, Var)})$ is the encoding of a Boolean condition with respect to an assignment of pattern names to variables, $pns_1 \rightarrow Var_1, \ldots, pns_n \rightarrow Var_n$. It is defined in Figure 13.

For instance, we can see in Figure 14 the encoding of PTL rule *foreign2_ er2rl* of Figure 8. The *object* and *associationObjects* predicates 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*. Such predicate retrieves the objects of the source model and encodes the Boolean

```
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),
                gen_id((A, B), f2, L).

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).

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

Figure 14: Encoded Rule

condition of the rule.

The encoding has taken into account the declarative semantics defined for PTL. The following theorem establishes the soundness and completeness of the encoding.

Given a PTL program $\mathcal{P} = (\mathcal{MM}^{\mathcal{P}}, \mathcal{R}^{\mathcal{P}}, \mathcal{H}^{\mathcal{P}})$ where $\mathcal{MM}^{\mathcal{P}}$ are the meta-model definitions of $\mathcal{P}$, $\mathcal{R}^{\mathcal{P}}$ the ATL style rules of $\mathcal{P}$ and $\mathcal{H}^{\mathcal{P}}$ the helpers of $\mathcal{P}$, we denote by $enc(\mathcal{P})$ the encoding of $\mathcal{MM}^{\mathcal{P}}$ and $\mathcal{R}^{\mathcal{P}}$ together with the rules of $\mathcal{H}^{\mathcal{P}}$. Given a model $\mathcal{M}$, we denote by $enc(\mathcal{M})$ the encoding of $\mathcal{M}$ with Prolog facts.

Theorem 1 (Soundness and Completeness)**.** *Given a program $\mathcal{P}$, a model $\mathcal{M}$ that interprets the source meta-models of $\mathcal{P}$, and $\mathcal{M}' = \cup_{1 \leq i \leq n} [\![r_i]\!]^{\mathcal{M}}$, where $\mathcal{R}^{\mathcal{P}} \equiv r_1, \ldots, r_n$, then:*

- *$object(mt, ct, o, \overline{v}, [att_1(a_1), \ldots, att_m\ (a_m)], q)$ is a logic consequence of $enc(\mathcal{P}) \cup enc(\mathcal{M})$ iff there exist $\overline{v} \in \mathcal{O}^{\mathcal{M}}$, mt target meta-model of $\mathcal{P}$, $o \in ct^{\mathcal{M}'}$ where $att_i^{\mathcal{M}'}(o) = a_i$ and $gen\_id(\overline{v}, q) = o$*

- *$associationObjects(mt, r_j, o, o_j)$ is a logic consequence of $enc(\mathcal{P}) \cup enc(\mathcal{M})$ iff there exists mt target meta-model of $\mathcal{P}$ and $o, o_j \in \mathcal{O}^{\mathcal{M}'}$ where $r_j^{\mathcal{M}'}(o) = o_j$*

Proof. *We have to prove that a Prolog atom ϕ is a logic consequence of the program $enc(\mathcal{P}) \cup enc(\mathcal{M})$ in the following cases:*

(a) case $\phi \equiv rn(\overline{v})$ iff exists a rule (rule rn from ps where bc to obs) $\in \mathcal{P}$, $ps \equiv \overline{pns : ms!cs}$ such that $\overline{v \in C^{\mathcal{M}}}, \overline{(pns, C^{\mathcal{M}})} = << ps >>^{\mathcal{M}}$ and $[\![bc]\!]^{\mathcal{M}}_{\overline{(pns,v)}}$ is true.

(b) case $\phi \equiv object(mt, ct, o, \overline{v}, [att_1(a_1), \ldots, att_m\ (a_m)], pnt_i)$ iff there exists $\overline{v} \in \mathcal{O}^{\mathcal{M}}$, mt target meta-model of $\mathcal{P}$, and a rule (rule rn from ps where bc to obs) $\in \mathcal{P}$, $ps \equiv \overline{pns : ms!cs}$, $obs \equiv \overline{pnt : mt!ct(\overline{bd})}$, where att_p<$-expr_p$, $1 \leq p \leq m$ are the bindings to attributes in $\overline{bd}_i$ such that $o \in \mathcal{O}^{\mathcal{M}'}$, $o \in ct^{\mathcal{M}'}$, $att_p^{\mathcal{M}'}(o) = a_p$, $1 \leq p \leq m$, $[\![expr_p]\!]^{\mathcal{M}'}_{\overline{(pns,v)}} = a_p$, $[\![bc]\!]^{\mathcal{M}}_{\overline{(pns,v)}}$ is true and $gen\_id(\overline{v}, pnt_i) = o$

(c) case $\phi \equiv associationObjects(mt, r_j, o, o_j)$ iff there exists $\overline{v} \in \mathcal{O}^{\mathcal{M}}$, mt target meta-model of $\mathcal{P}$ and a rule (rule rn from ps where bc to obs) $\in \mathcal{P}$, $ps \equiv \overline{pns : ms!cs}$, $obs \equiv \overline{pnt : mt!ct(\overline{bd})}$, where r_j<$-expr'_j$, $1 \leq j \leq k$ are the bindings to roles in $\overline{bd}_i$ such that $o \in \mathcal{O}^{\mathcal{M}'}$, $o \in ct^{\mathcal{M}'}$, $r_j^{\mathcal{M}'}(o) = o_j$, $[\![expr'_j]\!]^{\mathcal{M}'}_{\overline{(pns,v)}} = o_j$, $[\![bc]\!]^{\mathcal{M}}_{\overline{(pns,v)}}$ is true and $(v_1, \ldots, v_n) \rightarrow^{\mathcal{P}}_{pnt_i} o$

(d) case $\phi \equiv enc(expr, \overline{(pns, v)}, v)$ iff $[\![expr]\!]^{\mathcal{M}}_{\overline{(pns,v)}} = v$.

(e) case $\phi \equiv enc(bc, \overline{(pns, v)})$ iff $[\![bc]\!]^{\mathcal{M}}_{\overline{(pns,v)}}$ is true.

(f) case $\phi \equiv object(\_, \_, o, \overline{v}, \_, q)$ iff $(v_1, \ldots, v_n) \rightarrow^{\mathcal{P}}_{q} o$

Where the cases (b) and (c) correspond with the two cases considered in the theorem statement. We prove this result by induction on the length **n** *of the derivation of ϕ from $enc(\mathcal{P}) \cup enc(\mathcal{M})$.*

Case n=0*:*
*They are atoms obtained from cases (e1) to (e4) in Figure 12 and therefore case (d) of the theorem. From the representation of $\mathcal{M}$ with Prolog facts, and by rules from (*Expr1*) to (*Expr4*) of Figure 2 we can conclude the result.*

Case n>0*:*

— *In the case (a): $rn(\overline{v})$ is obtained from rule (r3) of Figure 11; now, $cs_i\_id(ms_i, v_i)$, $1 \leq i \leq n$ holds iff $\overline{v \in C^{\mathcal{M}}}, \overline{(pns, C^{\mathcal{M}})} = << ps >>^{\mathcal{M}}$ from cases (*Patt1*) and (*Patt2*) of Figure 2; by induction hypothesis $enc(bc, \overline{(pns, v)})$ is a logic consequence of $enc(\mathcal{P}) \cup enc(\mathcal{M})$ iff $[\![bc]\!]^{\mathcal{M}}_{\overline{(pns,v)}}$ is true, and thus we obtain the result.*

— *In the case (b): $object(mt, ct, o, \overline{v}, [att_1(a_1), \ldots, att_m\ (a_m)], q)$ is obtained from rule (r1) of Figure 11; now, by induction hypothesis $rn(\overline{v})$ is a logic consequence of $enc(\mathcal{P}) \cup enc(\mathcal{M})$ iff $[\![bc]\!]^{\mathcal{M}}_{\overline{(pns,v)}}$ is true; in addition, by induction hypothesis $enc(expr_p, \overline{(pns, v)}, a_p)$ is a logic consequence of $enc(\mathcal{P}) \cup enc(\mathcal{M})$ iff $[\![expr_p]\!]^{\mathcal{M}}_{\overline{(pns,v)}} = a_p$; in addition $gen\_id(\overline{v}, pnt_i) = o$. Thus applying cases (*Mod1*), (*Mod2*) and (*Mod3*) of Figure 2, we obtain the result.*

— *In the case (c) we can reason analogously to the previous case, applying (r2) of Figure 11, with cases (*Mod1*), (*Mod2*) and (*Mod4*) of Figure 2.*

— *In the case (d): $enc(expr, \overline{(pns, v)}, v)$ and cases from (e5) to (e8) of Figure 12; we can also reason by induction hypothesis with cases from (*Expr5*) to (*Expr8*) of Figure 2.*

— *In the case (e): $enc(bc, \overline{(pns, v)})$; we can also reason by induction hypothesis, from cases from (bc1) to (bc3) of Figure 13 and cases from (*Bool1*) to (*Bool3*) of Figure 2.*

— *In the case (f): $object(\_, \_, o, \overline{v}, \_, q)$ is obtained from rule (r1) of Figure 11. We can conclude the result by induction hypothesis and the definition $(v_1, \ldots, v_n) \rightarrow^{\mathcal{P}}_q o$.*

5.3. PTL interpreter

Now we show how a PTL program is executed. The PTL program has to include meta-model definitions (i.e., source and target models), and the set of PTL (i.e., mapping and helper) rules. We have to specify in the PTL program the location of source and target models with the directives *input* and *output*.

A predicate `ptl` is called with the file name in which the PTL code is included, for example: `?- ptl('er2rl.ptl')`. The predicate `ptl` is defined in Figure 15.

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

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 15: PTL interpreter

The `ptl` predicate automatically generates the Prolog library of the metamodels 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 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 15.

6. Mapping of ATL to PTL

As part of the evaluation of PTL we have considered how to map a fragment of ATL to PTL. In this way, we can align PTL with a standardized language, and compare PTL with ATL in expressivity and performance. With respect to expressivity a subset of PTL is equivalent to a core fragment of ATL. With regard to performance, we will show in Section 7.4 a comparison between ATL and PTL, as well as a comparison between OCL and Prolog as constraint validation languages. PTL has the same syntax as ATL for declarative mapping rules. The difference from PTL and ATL is the use of OCL as mechanism to query the source model. PTL uses Prolog instead

OCL-LiteExpr ::=	PathAttributes | PathObjects SelectExpr |
	oclIsTypeOf(mm!C) |
	not OCL-LiteExpr |
	OCL-LiteExpr and OCL-LiteExpr |
	OCL-LiteExpr or OCL-LiteExpr |
	OCL-LiteExpr implies OCL-LiteExpr
	PathAttributes = PathAttributes
	PathAttributes <> PathAttributes
PathAttributes ::=	$mm!C$.att | $mm!C$.r.att | v.att | v.r.att
PathObjects ::=	$mm!C$.r | $mm!C$.allInstances() | v.r
SelectExpr ::=	BooleanOp |
	→ select(v | OCL-LiteExpr) SelectExpr
	→ collect(v | PathAttributes)
BooleanOp ::=	→ exists(v | OCL-LiteExpr) |
	→ forall(v | OCL-LiteExpr) |
	→ size()>0 | → size()=0 |
	→ isEmpty() | → notEmpty()

Figure 16: OCL expressions

OCL. Nevertheless, we can still consider a mapping from OCL to Prolog. This mapping can be defined for a (expressive enough) fragment of OCL. Therefore, a subset of PTL can be mapped to a subset of ATL (the declarative part of ATL) using the fragment of OCL. Nevertheless, PTL is equipped with Prolog, which is a fully-fledged programming language and, thus, PTL is more expressive than the ATL fragment. A mapping for a more rich fragment is considered as future work. Additionally, full ATL includes additional elements: ECore inheritance, imperative rules, lazy rules, called rules (among them the entry point), rule inheritance, as well as the ATL refining mode. In order to incorporate such mechanisms to PTL we should modify not only the implementation but the semantics. It is also considered as future work.

The fragment of OCL we consider is defined in Figure 16. This is a modified version of OCL-Lite [35]. We have extended OCL-Lite to include *allInstances* and *collect* as well as =, <> operators. We have removed *oclAsType* and other typing operators because PTL is untyped. Let us remark that some other OCL expressions can be translated to this OCL

(ocl1)	$enc(\text{mm!C}.att) = C\_att(mm, D, E)$
(ocl2)	$enc(\text{mm!C}.r.att) = C\_id(mm, D), C\_r(mm, r, D, E), R\_att(mm, E, F)$ if $range(r) = R$
(ocl3)	$enc(\text{mm!C}.r\ s) = C\_id(mm, D), C\_r(mm, r, D, E), enc(s, mm!R, (E, F))$ if $range(r) = R$
(ocl4)	$enc(\text{mm!C}.allInstances()\ s) = enc(s, mm!C, (D, E))$

Figure 17: Encoding of OCL expressions (I)

fragment. This is the case, for instance, of *isUnique*, *one* and *reject*.

The encoding of this fragment of OCL is defined by the function *enc(OCL-LiteExpr)* which returns a Prolog goal. The encoding is shown in Figures 17 and 18, where Figure 17 encodes non contextualized expressions (i.e., starting from $mm!C$) and the Figure 18, encodes contextualized expressions.

PathObjects are encoded by the library of meta-models (see Figure 17). For instance, `er!data.allInstances()` is encoded by `data_id(er,A)`, where `A` represents the identifier of the object. *PathAttributes* are encoded also by the library of meta-models (see Figure 17). For instance, `er!data.name` is encoded by `data_name(er,A,B)`, where `A` represents the identifier of the object, and `B` the name.

Expressions can be contextualized, that is, a given OCL variable can range from a given model and metamodel. For encoding contextualized expressions (see Figure 18), we use two Prolog variables that are used as input and output of the encoded expression. For instance, `v.name` in the context `er!data` is encoded by `data_name(er,A,B)`, where `A` represents the identifier of the object, and `B` the name.

OCL *not*/*and*/*or*/*implies*/$=$/$<>$ are encoded by Prolog \+/,/;/->/=/\== (see Figure 18). OCL *exists*, *select* and *collect* are encoded by Prolog goals, while OCL *forAll* is encoded by the negation of the goals. For instance, the following OCL *select* subexpression:

```
er!data.allInstances() -> select(d | d.attr_of -> exists(a | a.key))
```

is encoded by:

```
:-data_id(er,A),data_attr_of(er,A,D),attribute_id(er,D),
          attribute_key(er,D,B),B=true
```

Analogously, the following OCL *collect* subexpression:

(ocl5)	$enc(\text{v}, mm!C, A) = C\_id(mm, A)$
(ocl6)	$enc(\text{v}.att, mm!C, (A, B)) = C\_att(mm, A, B), B = true$
	if $range(att) = Boolean$
(ocl7)	$enc(\text{v}.att, mm!C, (A, B)) = C\_att(mm, A, B)$
	otherwise
(ocl8)	$enc(\text{v}.r.att, mm!C, (A, B)) = C\_r(mm, A, E), D\_att(mm, E, B)$
	if $range(r) = R$
(ocl9)	$enc(\text{v}.r\ s, mm!C, (A, B)) = C\_r(mm, r, A, D), enc(s, mm!R, (D, B))$
	if $range(r) = R$
(ocl10)	$enc(oclIsTypeOf(mm!C), mm'!C', (A, B)) = C\_id(mm, A)$
(ocl11)	$enc(not\ o, mm!C, (A, B)) = \backslash + enc(o, mm!C, (A, B))$
(ocl12)	$enc(o1\ and\ o2, mm!C, (A, B)) = enc(o1, mm!C, (A, D)), enc(o2, mm!C, (D, B))$
(ocl13)	$enc(o1\ or\ o2, mm!C, (A, B)) = enc(o1, mm!C, (A, B)); enc(o2, mm!C, (A, B))$
(ocl14)	$enc(o1\ implies\ o2, mm!C, (A, B)) =$
	$enc(o1, mm!C, (A, D))$–> $enc(o2, mm!C, (D, B)); false$
(ocl15)	$enc(p1 = p2, mm!C, (A, B)) =$
	$enc(p1, mm!C, (A, D)), enc(p2, mm!C, (A, E)), D == E$
(ocl16)	$enc(p1 <> p2, mm!C, (A, B)) =$
	$enc(p1, mm!C, (A, D)), enc(p2, mm!C, (A, E)), D\backslash == E$
(ocl17)	$enc(\rightarrow exists(v\|c), mm!C, (A, B)) = enc(v, mm!C, A), enc(c, mm!C, (A, B))$
(ocl18)	$enc(\rightarrow forAll(v\|c), mm!C, (A, B)) =$
	$\backslash + (enc(v, mm!C, A), \backslash + enc(c, mm!C, (A, B)))$
(ocl19)	$enc(\rightarrow select(v\|o)s, mm!C, (A, B)) =$
	$enc(v, mm!C, A), enc(o, mm!C, (A, D)), enc(s, mm!C, (D, B))$
(ocl20)	$enc(\rightarrow collect(v\|p), mm!C, (A, B)) =$
	$enc(v, mm!C, A), enc(p, mm!C, (A, B))$
(ocl21)	$enc(\rightarrow size() > 0, mm!C, (A, B))$=$enc(v, mm!C, A)$
(ocl22)	$enc(\rightarrow size() = 0, mm!C, (A, B))$=$\backslash + enc(v, mm!C, A)$
(ocl23)	$enc(\rightarrow notEmpty(), mm!C, (A, B))$=$enc(v, mm!C, A)$
(ocl24)	$enc(\rightarrow isEmpty(), mm!C, (A, B))$=$\backslash + enc(v, mm!C, A)$

Figure 18: Encoding of OCL expressions (II)

```
er!data.allInstances() -> collect(d | d.attr_of)
```

$enc(er!data.allInstances() \rightarrow forAll(d \mid d.attr\_of \rightarrow exists(a \mid a.key))) = G$

G=$enc(\rightarrow forAll(d|d.attr\_of \rightarrow exists(a|a.key))), er!data, (A, B))$

$enc(\rightarrow forAll(d|d.attr\_of \rightarrow exists(a|a.key)), er!data, (A, B)) = \setminus + (G1, \setminus + G2)$

G1=$enc(d, er!data, A)$

G2=$enc(d.attr\_of \rightarrow exists(a|a.key), er!data, (A, B))$

$enc(d, er!data, (A, D))$=$data\_id(er, A)$

$enc(d.attr\_of \rightarrow exists(a|a.key), er!data, (A, B))$=$data\_attr\_of(er, A, D), G3$

$G3 = enc(\rightarrow exists(a|a.key), er!data, (D, B))$

$range(attr\_of) = attribute$

$enc(\rightarrow exists(a|a.key), er!attribute, (D, B)) = G4, G5$

G4=$enc(a, er!attribute, D)$

G5=$enc(a.key, er!attribute, (D, B))$

$enc(a, er!attribute, D) = attribute\_id(er, D)$

$enc(a.key, er!attribute, (D, B)) = attribute\_key(er, D, B)), B = true$

Figure 19: Example of OCL expression encoding

.

is encoded by:

```
:-data_id(er,A),data_attr_of(er,A,D)
```

Finally, OCL *size>0* (and the equivalent *notEmpty()*) are encoded the same as OCL *exists*, while OCL *size=0* (and the equivalent *isEmpty()*) is encoded by the negation, thus the same as OCL *forAll*.
Let us see an example of encoding of using the rules of Figures 17 and 18. Let us suppose the OCL expression:

```
er!data.allInstances() -> forAll(d | d.attr_of -> exists(a | a.key))
```

The encoding is shown in Figure 19, resulting in:

```
 \+ (data_id(er,A), \+ (data_attr_of(er,A,D),attribute_id(er,D),
     attribute_key(er,D,B),B=true))
```

6.1. Model Validation with Prolog and OCL

Model validation is carried out with Prolog. Next validation rules of the constraints expressed in Figure 6 are shown.

```
vrule(1):- \+(data_attr_of(er,D,A1),data_attr_of(er,D,A2),A1\==A2,
            attribute_name(er,A1,N1),attribute_name(er,A2,N2),N1==N2).
vrule(2):- \+(data_id(er,D),\+ (data_attr_of(er,D,A),
```

```
            attribute_key(er,A,true))).
vrule(2):- \+(data_id(er,D),(data_attr_of(er,D,A),attribute_key(er,A,true),
            data_attr_of(er,D,B),attribute_key(er,B,true),A\==B)).
vrule(3):- \+(data_id(er,D1),data_id(er,D2),D1\==D2,
            data_name(er,D1,N1),data_name(er,D2,N2),N1==N2).
vrule(4):- \+(data_id(er,D1),data_id(er,D2),D1\==D2,
            data_container(er,D1,N1),data_container(er,D2,N2),N1==N2).
vrule(5):-\+(qualifier_id(er,Q),\+ (attribute_id(er,A),
            qualifier_name(er,Q,N1),attribute_name(er,A,N2),N1==N2)).
vrule(6):- \+(data_role_of(er,D,R1),data_role_of(er,D,R2),R1\==R2,
            role_name(er,R1,N1),role_name(er,R2,N2),N1==N2).
vrule(7):-\+(table_id(rl,T1),table_id(rl,T2),T1\==T2,table_name(rl,T1,N1),
            table_name(rl,T2,N2),N1==N2).
vrule(8):-\+(row_id(rl,R1),row_id(rl,R2),R1\==R2,row_name(rl,R1,N1),
            row_name(rl,R2,N2),N1==N2).
vrule(9):- \+(row_id(rl,R),((row_is_foreign(rl,R,F),row_is_key(rl,R,K));
            (row_is_foreign(rl,R,F),row_is_col(rl,R,C)))).
vrule(10):- \+(key_id(rl,K),\+(attribute_id(er,A),key_name(rl,K,N1),
            attribute_name(er,A,N2),N1==N2)).
vrule(10):- \+(key_id(rl,K),\+(attribute_id(er,A),key_type(rl,K,N1),
            attribute_type(er,A,N2),N1==N2)).
vrule(11):-\+(table_id(rl,T), \+(data_id(er,D),data_container(er,D,N1),
            table_name(rl,T,N2),N1==N2),\+ (role_id(er,R),
            role_name(er,R,N1),table_name(rl,T,N2),N1==N2)).
vrule(12):- \+(row_id(rl,R), \+(data_id(er,D),data_name(er,D,N1),
            row_name(rl,R,N2),N1==N2),\+ (role_id(er,RL),
            data_id(er,D),role_name(er,RL,N1),data_name(er,D,N3),
            row_name(rl,R,N2),concat(N1,N3,N2))).
vrule(13):- \+(foreign_id(rl,F), \+(data_id(er,D),data_name(er,D,N1),
            role_id(er,R),role_name(er,R,N2),key_id(rl,K),
            key_name(rl,K,N3),foreign_name(rl,F,N4),
            concat(N2,N1,AUX),concat(AUX,N3,N4))).
```

The equivalent OCL expressions are as follows:

```
(1) er!data.allInstances()->forAll(d| d.attr_of->isUnique(a|a.name))
(2) er!data.allInstances()->forAll(d| d.attr_of->exists(a|a.key))
(2) er!data.allInstances()->forAll(d| d.attr_of->isUnique(a|a.key))
(3) er!data.allInstances()->isUnique(d|d.name)
(4) er!data.allInstances()->isUnique(d|d.container)
(5) er!qualifier.allInstances()->forAll(q| er!attribute.allInstances()
       ->exists(a|a.name=q.name))
(6) er!data.allInstances()->forAll(d|d.role_of->isUnique(r|r.name))
(7) rl!table.allInstances()->isUnique(t|t.name)
(8) rl!row.allInstances()->isUnique(r|r.name)
(9) rl!row.allInstances()->forAll(r | r.is_foreign->isEmpty()
       or (r.is_key->isEmpty() and r.is_col->isEmpty()))
(10) rl!key.allInstances()->forAll(k | er!attribute.allInstances()
       -> exists(a | a.name=k.name))
(10) rl!key.allInstances()->forAll(k | er!attribute.allInstances()
       -> exists(a | a.type=k.type))
(11) rl!table.allInstances()->forAll(t | (er!data.allInstances()
       -> exists(d | d.container=t.name))
       or (er!role.allInstances()->exists(r | r.name=t.name)))
(12) rl!row.allInstances()->forAll(r | (er!data.allInstances()
       -> exists(d | d.name=r.name)) or (er!data.allInstances()
       -> exists(d | er!role.allInstances()
```

```
        ->exists(rl | r.name=rl.name+d.name))) )
(13) rl!foreign.allInstances()->forAll(f | er!role.allInstances()
        -> exists(r | er!data.allInstances()
        -> exists(d | rl!key.allInstances()
        -> exists(k | f.name=r.name+d.name+k.name))) )
```

For validating the constraints on models, we have implemented a predicate called `validate` to obtain results of validation. For instance, let us suppose that there are two data called *Course*, instead of one, then the validator give us the number of the rules that have been violated.

```
?- validate('er2rl.ptl','constraints_ptl.pl').
Validation..............
Validation failure on rule: 3
Validation failure on rule: 8
true.
```

7. Debugging and Tracing

Now, we would like to show how to debug programs and trace executions in our language.

7.1. Debugging

Debugging is able to find rules that failed, and provides the location in which the error is found. PTL mapping rules fail due to Boolean conditions that are not satisfied and target objects that cannot be created. Debugging also handles helpers failure. 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 syntactically correct). Now, we find that the target model is wrong. In such a case we can query the debugger, obtaining:

```
?- debug('er2rl.ptl').
DEBUGGER........................
Debugger: Rule Condition of: table2_er2rl cannot be satisfied.
Found error in: role_name
true.
```

The debugger shows the name of the PTL rule (i.e. *table2_ er2rl*) that fail and the error found (i.e. *role_ name*). But the debugger can also detect that target objects cannot be created, for instance, let us suppose that we write *p@navigable==false*, instead of *p@navigable==true*. Then the debugger answers:

```
debug_term(Head):-clause(Head,(C,Condition)),
               C=..[Rule,A],
               (((findall(A,C,L),L=[])->
               (clause(C,Cond),
               debug_condition(Rule,Cond,A),!);
                   (findall(A,(C,Condition),L),L=[])->
               (C,debug_object(Rule,Condition,A),!);
               (C,Condition))).

debug_condition(Rule,Cond,A):-
               write('Debugger: Rule Condition of: '),
               write(Rule),write(' cannot be satisfied.'),nl,
               debug_sequence(Cond,A).

debug_object(Rule,Cond,A):-
               write('Debugger: Objects of: '),
               write(Rule),write(' cannot be created.'),nl,
               debug_sequence(Cond,A).

debug_sequence((O,RO),A):-!,
               ((findall(A,O,L),L=[])->
               write('Found error in: '),
               O=..[Name|_],write(Name),nl;
               (O,debug_sequence(RO,A))).
```

Figure 20: Debugger

```
?- debug('er2rl.ptl').
DEBUGGER.........................
Debugger: Objects of: table2_er2rl cannot be created.
Found error in: resolveTemp
true.
```

In this case the objects of the 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. The PTL interpreter has been modified in order to cover debugging. The debugger checks PTL rules that fail and prints the location in which Boolean conditions and object creations fail. The main predicates of the debugger are shown in Figure 20.

7.2. Tracing

Nevertheless, we 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, we 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 we omit the Boolean condition *q@navigable==false*. We review the target model and find a wrong element: *registerCourseid_ course*. Now, we can trace the execution from the identifier of the wrong element, obtaining the applied rules and the identifiers of the source model elements as follows:

```
?- trace('er2rl.ptl', id74id62f1).

TRACER.............................
Tracing the element: id74id62f1
Traced Execution....
Rule: foreign1_er2rl

Element: er:role
Id: id62
Role: er:navigable
Value: true

Element: er:qualifier
Role: er:type
Value: int
Id: id74
Role: er:name
Value: id_course

Element: er:data
Id: id18
Role: er:name
Value: Course

Element: er:role
Id: id62
Role: er:name
Value: register
```

The trace shows the applied rule (i.e. *foreign1_ er2rl*), together with the 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. The main predicates of the tracer are shown in Figure 21.

7.3. Eclipse Plugin

We have developed an Eclipse plugin to integrate editing of PTL programs, execution of transformations, debugging, tracing, and validations of models. The plugin calls to SWI-Prolog in order to achieve these tasks. In each task the user selects the files involved, that is, the source code of PTL programs. The results of the tasks can be visualized from the Eclipse environment by inspecting log files generated from SWI-Prolog. The Eclipse

```
trace_elem(O):-clause(O,Condition),call(Condition),trace_atoms(Condition),!.

trace_atoms((O,RO)):-!,(recorded(pattern,O)->write('Rule: '),O=..[Name|_],
        write(Name),nl,trace_elem(O);
        (recorded(head,O)->print_trace(O),nl,trace_elem(O);
        (O=..[Name|[_,A|_]],class_def(Name,_,_)->
        print_trace(O,A),trace_elem(O);
        (recorded(access,O)->trace_elem(O);
        (recorded(metamodel,O)->true;
        (O=objectM(A,B,C,D,E,F)->
        trace_elem(object(A,B,C,D,E,F));true))))))),
        trace_atoms(RO).
```

Figure 21: Tracer

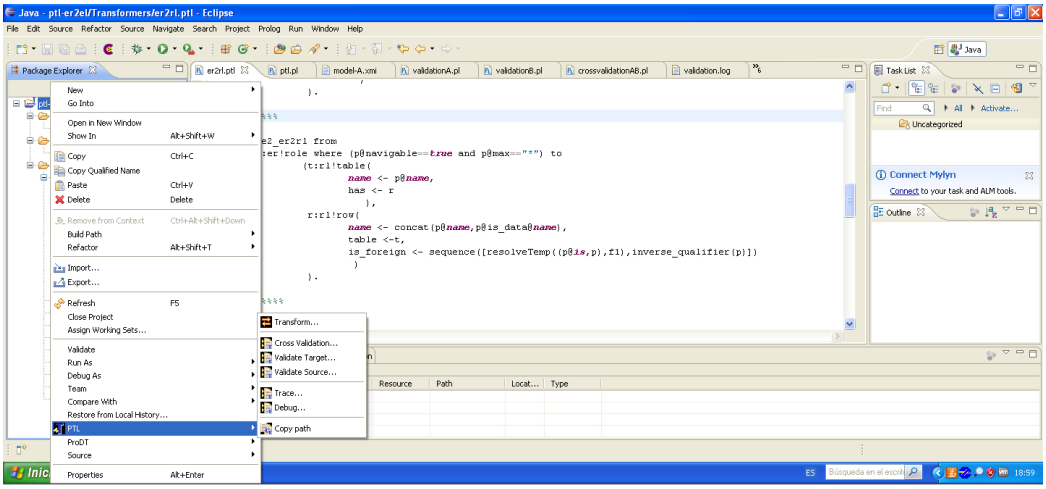


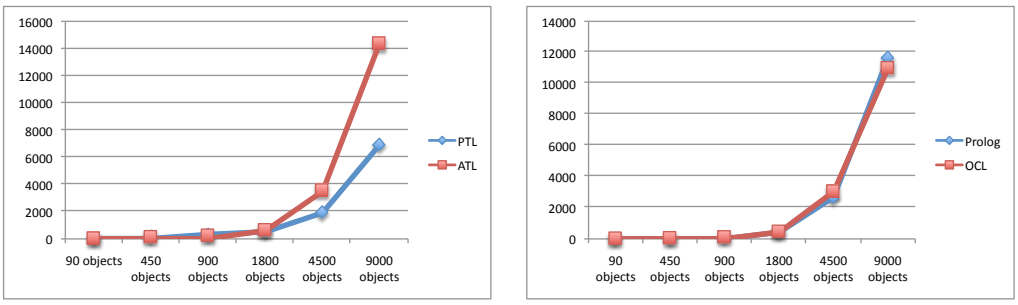
Figure 22: Eclipse Plugin of PTL

environment can be used by PTL programmers to write PTL and Prolog code. The Eclipse plugin is equipped with a menu in which the PTL programmer can select from transformation, validation, debugging and tracing (see Figure 22). The plugin can be downloaded from[4].

7.4. Performance

We have tested the performance of PTL for (1) the transformation and the constraints of the case study with medium-size models (from 90 to 9000

[4] `http://indalog.ual.es/mdd/ptl2`



(a) (b)

Figure 23: Comparison of Execution Times of (a) PTL/ATL (case study) and (b) Prolog/OCL (constraint (v5)). Expressed in ms.

objects), compared with the ATL/OCL EMF-specific Virtual Machine. We have implemented a random test case generator in Prolog with this end. Additionally, we have tested performance of PTL for (2) constraint validation using the standard benchmarking large model datasets (from 6k (i.e., 6 thousand) to 1423k (i.e., more that one million) objects) available from[5]. We have used a Mac OS X machine with a 1.7 GHz processor Intel Core i7, 8GB 1600 MHz RAM.

With respect to (1) Figure 23 shows the results of execution times. The times range from 27 ms to 6,905 ms in PTL, and from 12 ms to 14,354 ms in ATL in the case of transformations. In the case of constraints, the times range from 2 ms to 11,646 ms in Prolog, and 7ms to 10,971 ms in the case of OCL. Execution times include loading and writing of the model in the case of transformations. We can see that we have similar execution times for constraints, while in the case of transformations we have an improvement.

With respect to (2) Figure 24 shows the results of execution times. We have tested the execution times of four constraints of the meta-model of Figure 25. It describes a railway system in which a train *route* can be defined by a set of sensors between two neighboring signals. The state of a *signal* can be stop (when it is red), go (when it is green) or failure. *Sensors* are associated with track elements, which can be a track segment or a switch. The status of a switch can be left, right, straight or failure. A route can have

[5] `http://incquery.net/performance`

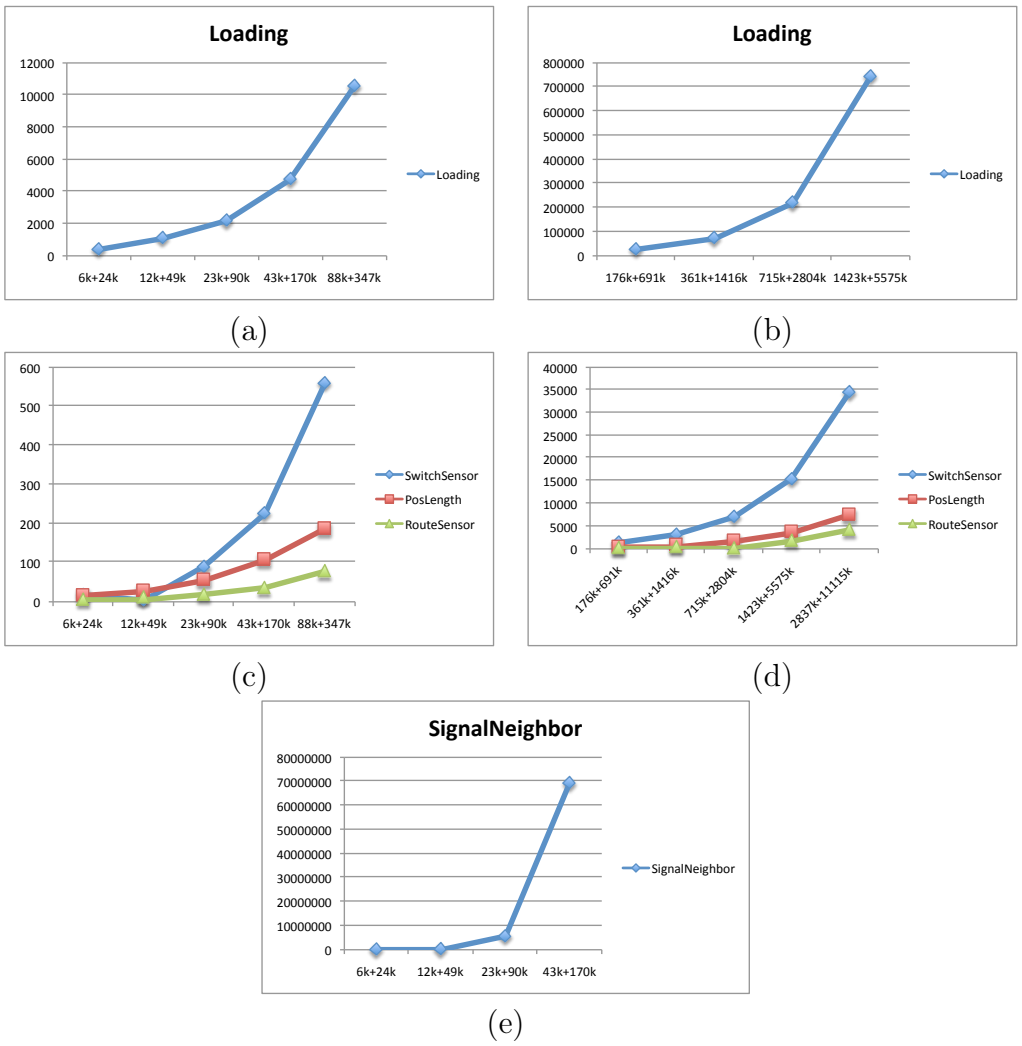


Figure 24: Execution Times of Railway Models. (a) and (b) Loading times of models. (c) and (d) Constraint checking times for Q1 to Q3. (e) Constraint checking times for Q4. Expressed in ms.

associated switch positions, which describe the required state of a switch belonging to the route. Different route definitions can specify different states for a specific switch. The constraints to be checked are the following:
(Q1)-SwitchSensor: Every switch must have at least one sensor connected to it.

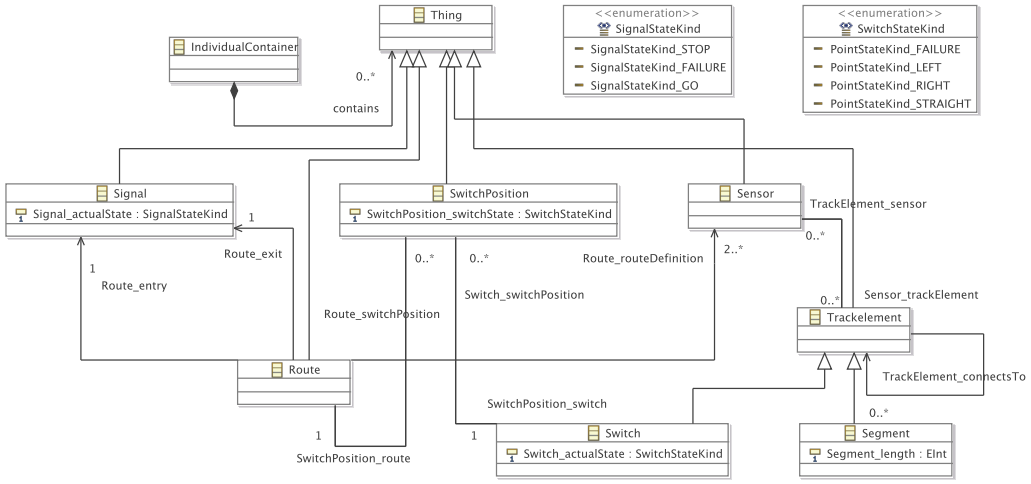


Figure 25: Metamodel of Railway

(Q2)-PosLength: A segment must have positive length.
(Q3)-RouteSensor: All sensors that are associated with a switch that belongs to a route must also be associated directly with the same route.
(Q4)-SignalNeighbor: A route is incorrect, if it has a signal, and a sensor connected to another sensor by two track element, but there is no other route that connects the same signal and the other sensor..

The code in Prolog of the previous queries is shown in the Appendix C.

In Figure 24 we show in (a) and (b) the time required to load models (from 6k nodes and 24k edges to 88k nodes and 347k edges in (a) and from 176k nodes and 691k edges to 2837k nodes and 11115k edges in (b)), which range from 395 ms to 47,536,510 ms (approximately 13 hours). In (c) and (d) we show the time required to check the constraints Q1 to Q3 for the same size of models which range from 5ms (Q3 and the smallest size) to 34,514 ms (Q1 and the largest model). Q4 is the constraint that taking longest (see (d) of Figure 24). In fact, we have only obtained execution times for sizes from 6k to 43k nodes. From this size, the constraint checking takes more than two days. In this case the execution times range from 45,400 ms to 69,278,447 (approximately 19 hours).

To improve performance in both tests we have introduced some optimiza-

tions. The first optimization is the use of the RDF library of SWI-Prolog [40] for storing models as triples. It greatly increases the performance. The second optimization is particular to (1) and (2).

In the case of (1), some optimizations have been carried out in the PTL interpreter which affects the ordering in which the encoding rules are executed. In particular, the execution is divided in three steps: (a) *rule condition evaluation*, (b) *object creation* and (c) *association creation*. Given that rule conditions are evaluated for each object created by a rule, (a) rule conditions are computed before and stored in a cache. After, (b) each object is created for each rule condition. Finally, (c) associations are created. In particular, *resolveTemp* is evaluated at the end, when all the rules have been applied. In the case of (2), Prolog atom reordering is crucial to get a better performance, and the use of the RDF library of SWI-Prolog to handle large datasets. In particular, atom reordering has improved the performance of queries (Q1) to (Q4) by considering type checking after join (similar to the proposed in [41, 42]. For instance, the rule:

```
(Q3) routeSensor(Sen,Sw,Sp,R):-
      contains_Route_switchPosition(concept,R,Sp),
      contains_SwitchPosition_switch(concept,Sp,Sw),
      contains_TrackElement_sensor(concept,Sw,Sen),
      'contains_xsi:type'(concept,R,'Concept:Route'),
      'contains_xsi:type'(concept,Sw,'Concept:Switch'),
      'contains_xsi:type'(concept,Sp,'Concept:SwitchPosition'),
      'contains_xsi:type'(concept,Sen,'Concept:Sensor'),
      \+ contains_Route_routeDefinition(concept,R,Sen).
```

has better performance than:

```
(Q3) routeSensor(Sen,Sw,Sp,R):-
      'contains_xsi:type'(concept,R,'Concept:Route'),
      'contains_xsi:type'(concept,Sw,'Concept:Switch'),
      'contains_xsi:type'(concept,Sp,'Concept:SwitchPosition'),
      'contains_xsi:type'(concept,Sen,'Concept:Sensor'),
      contains_Route_switchPosition(concept,R,Sp),
      contains_SwitchPosition_switch(concept,Sp,Sw),
      contains_TrackElement_sensor(concept,Sw,Sen),
      \+ contains_Route_routeDefinition(concept,R,Sen).
```

The first one executes join before type checking (i.e., `'contains_xsi:type'`), while the second one checks types before. The improvement is due to the number of objects to be retrieved which in the first case is lower. Let us remark that the execution times of Figure 24 have been obtained without type checking, which is not required in this example because all the meta-model roles are distinct.

In summary, we have found that PTL works fine for medium-size models,

and for large models performance is penalized for constraints with multiple joins. In[6] there is a complete comparison of execution times of other systems, for the same kind of constraints, revealing that some of them work better with large models and multiple joins.

8. Related Work

Model transformation encompasses a variety of technical spaces, including model-ware, grammar-ware, and XML-ware, a variety of transformation representations including graphs, trees, and DAGs, and a variety of transformation paradigms including rule-based graph transformation, term rewriting, and implementations in general-purpose programming languages. We concentrate here in a comparison with strongly related approaches. Some other related approaches are summarized in Table 1.

8.1. Model Transformation Languages

D. Varró in his Ph.D thesis [14] describes the foundations of VIATRA and proposes to transform models into a Prolog term representation which is dynamically stored as facts, modified at run-time. Graph pattern matching is achieved by unification. VIATRA control structures are implemented as Prolog predicates. Automated program generation by model transformation is used to generate a Prolog program that implements a transformation. From a UML description of the transformation, a *GraTra* description is obtained from which a Prolog term based representation is generated. Finally, a Prolog code tree is handled to obtain a final Prolog program. The similarities with D. Varró's work are obvious. Models are transformed into Prolog terms, and transformations into Prolog code. However, the context is different. It works with graph-based transformations while our proposal works with ATL-style rules and OCL constraints encoded by Prolog rules. VIATRA satisfies the requirements (a), (b) and (c) of Section 1.

Bernhard Schätz [15] describes also how to define model transformations by using Prolog and considering models as terms. He represents EMF ECore-based meta-models as Prolog terms, and instantiates them to construct models. Construction predicates to deconstruct and reconstruct a term are provided. Only two classes of construction predicates are required: the union

[6] `http://incquery.net/performance`

Table 1: Related work summary

Language	*Contribution*	*Cite*
	Transformation Languages	
QVT	Standard for model transformation	[43, 44]
ATL	DSL for M2M transformations	[45, 9]
RubyTL	OO language, declarative and imperative	[46]
MT	Declarative and imperative	[47]
GReAT	Graph transformation language	[48]
AGG	Graph transformation language	[49]
TGG	Triple Graph Grammars based transformation language	[50]
	Logic based Transformation Languages	
VIATRA	Graph transformation language	[14, 51]
PETE	EMF transformations	[15, 52, 53]
Tefkat	Declarative language based on logic programming	[13, 12, 54]
JTransformer	Model requirements and code generation	[20]
Maude	Declarative language based on Rewritting Logic	[55, 56, 57]
Formula 2.0	Declarative language based on open-world logic programs and behavioral types	[17]
JTL	Bi-directional declarative transformation language based on Answer Set Programming (ASP)	[24]
	Logic Programming in MDE	
MDELite	Tool for Model Transformation	[27]
Mercury and F-Logic	*Mercury* and *F-Logic* logic languages for Model Transformation	[11]
MoMat	Prolog for Querying and Verification of Models	[18]
Prolog	Logic programming based model querying	[58]
Abductive logic programming	Reverse Model Engineering	[22]
Inductive logic programming	(Semi-) automatic way to derive transformation rules based on Inductive Logic Programming.	[16]
	Model Validation	
EMF-IncQuery	Incremental Model Validation	[29, 30, 59]
OCL	OCL to other Formalisms	[60, 61, 25, 62, 63, 64, 26, 65, 66]
Videas	Answer-set programming and Model Validation	[19]
Prolog	Comparison OCL/Prolog	[21]
Prolog	Consistence Checking of Sequence Diagrams	[23]
ATL/OCL	Validation and Verification	[67, 68]
	Debugging and Tracing	
ATL	ATL Debugging	[69]
-	Model Debugging	[28]
TNs	Model Debugging	[31]

and the composition operators. He focuses on graph transformations using a relational calculus to manipulate nodes (elements) and edges (relations). He shows how a single and homogeneous mechanism is enough to define transformations. Debugging is possible in his approach by tracing execution

and analyzing rule application. With regard to performance, he claims that transformation of EMF ECore models into their Prolog term representation leads also to some loss of efficiency. Nevertheless, he shows that experiments with real-world size models are feasible. He also has studied how to prove properties from models by using a theorem prover based on high-order logics [52]. He has developed the PETE transformation framework [53] provided as an Eclipse plugin. The similarities with Bernhard Schätz's work fall on the Prolog term based representation of models. However, he directly works with Prolog transformations, and Prolog back-tracking is a key tool of his proposal. In our case, PTL can work non deterministically and can compute several solutions from a given rule (see examples of Appendix A, Example 3), however the mechanism to compute target models is different. Bernhard Schätz's work satisfies the requirements (b), (c) and (d) of Section 1.

The *Tefkat* language [12, 13] 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, [54] proposes meta-model transformations in which evolutionary aspects are formalized using the Tefkat language. Tefkat has a trace model, which links the target, source and transformation. A transformation is represented as a model, having a meta-model. The trace model references the transformation as well as the source and target models. Our tracing technique is not based on a meta-model rather it is based on the backward execution of the PTL program. The representation of the program and the execution as models, as well as to provide meta-models for them, is considered as future work. Again, evolutionary aspects are out of the scope of this paper, but we believe that these can be included in our proposal. Tefkat satisfies the requirements (b) and (d) of Section 1.

In [20], 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 Eclipse. JTransformer satisfies the requirements (b) and (d) of Section 1.

FORMULA 2.0 [17] is a formal specification language based on open-world logic programs and behavioral types. With FORMULA 2.0, models and their instances can be defined and verified. Transformations can be easily defined by declarative Prolog-like transformation rules. FORMULA 2.0 sat-

isfies the requirements (b) and (c) of Section 1. JTL (Janus Transformation Language) [24] is a bi-directional declarative transformation language based on Answer Set Programming (ASP) using the DLV solver to compute source and target models from a set of atoms representing models, and QVT style transformation rules. JTL satisfies the requirements (b) and (c) of Section 1.

The rewriting logic has found an application in model engineering. The *Maude* language [70] has been chosen in several works about model transformation. For instance, in [55] models and meta-models have been formalized in *Maude*, and the same authors have developed an Eclipse plug-in called *Maudelling* that enables the transformation of models and meta-models to the corresponding *Maude* specifications. *Mova* [56] and *Moment* [57] are also *Maude*-based modeling tools for model verification. In the case of *Maude*-based tools, they also have to represent meta-models and models with the constructions of the language: sorts, classes, etc. The introduction of models in *Maude* enables to take profit from many of the tools (model checking, verification, etc.,) available on *Maude*. In [36], they have described the encoding of ATL with Maude. Basically, ATL is encoded by means of rewriting logic, the logic foundation of Maude. They make use of Maude for simulating transformations and reachability analysis.

8.2. Constraint Validation

OCL mappings to equivalent formalisms have been studied in several works. For instance, from OCL to SQL [60], OCL to first-order logic [61], OCL to a Prolog-based CSP formalism [25], OCL to equational logic [62, 63], OCL to Higher Order Logic [64] and OCL to graph patterns [65, 66]. OCL to Prolog has been studied in [26], where inconsistency rules are translated to Prolog queries and model construction operations to Prolog facts. The Prolog engine has been integrated into the modeling environments Eclipse EMF and Rational Software Architect. In [19] Answer-set programming facts are used to represent models that are validated against meta-model constraints. In [21] the authors have compared OCL and Prolog for querying UML models. They have found that if execution time of queries is linear then Prolog is faster. In [23] 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 a Prolog reasoning engine is used to automatically find inconsistencies. There are some works [68, 67] dealing with the so-called *tracts*, and *contracts* for specifying with OCL constraints in

transformations. The spirit of tracts is the same as our constraints on source and target models, and cross constraint validation on source-target models. EMF-IncQuery [29, 30] is a validation language based on graph patterns. EMF-IncQuery has a syntax similar to Prolog goals to express constraints. Incremental validation of constraints has been studied for EMF-IncQuery [59, 71], which is considered as future work.

8.3. Debugging

In [31] they propose Transformation Nets (TNs), a DSL on top of Colored Petri Nets (CPNs) for developing, executing and debugging model transformations. The run-time model is specified by a meta-model, and it can be exploited for model-based debugging by using OCL queries to find the origin of a bug. OCL is used to define conditional breakpoints. Approaches like VIATRA which produce debug reports that trace an execution, do not provide interactive debugging facilities. Although the ATL debugger [69] allows the step-wise execution of ATL byte-code, only low-level debugging information is provided, e.g., variable values.

9. Conclusions and Future Work

In this paper we have presented a model transformation language based on logic programming. We have also described a declarative semantics and the implementation of the language which consists in the encoding of mapping rules by Prolog rules. Furthermore, we have shown how to debug programs and trace executions, as well as validate models and transformations. Finally, we have provided a mapping from ATL to PTL in order to evaluate PTL with examples and test performance.

As future work we would like to extend our language with the following elements:

— A *type system* to be used in compilation: types will be included in meta-model definitions and used to detect the main programming errors. Debugging and tracing is still useful in the presence of types. Related to compilation also rule overlapping detection and cardinality checking will be also considered.

— *Meta-meta-models* other than ECore: Other meta-meta-models can be considered in our language whenever a suitable XML-based serialization is provided and Prolog predicates to load and write models are

implemented. For instance, we are interested in transformations of *BPMN (Business Process Modeling Notation)* models, for which an XML serialization, called *XPDL* is available.

— *Bi-directional and optimized transformations*: Our language is one-directional. A natural extension of our approach is to consider bi-directional transformations, where source models are computed from target models. In this case, Prolog backtracking can be used to generate several source models from a change of the target model. Also, the generation of several target models from a source model, and selection criteria of optimized target models (with respect to a given metric) is considered as future work.

— *Debugging*: other techniques of debugging can be used in PTL programs: breakpoints, step-by-step transformation execution, running transformation to the next breakpoint; visualization of patterns values, etc.

— *Tracing*: 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, that is, which elements of the target model are obtained from a given source element and the rules that are used. In addition, the definition of a meta-model of the trace is also considered as future work.

— *Validation*: validation can be improved by providing more detailed information about the violated rules: class and attributes involved, elements that violates the requirements, etc. Incremental validation is also considered as future work.

– *Random Test Case Generation*: we are also interested in the definition of a technique for generating random test cases using black and white-box methods. Test cases can be used for validating PTL as well as ATL transformations. Automatic mapping of ATL to PTL will be also useful.

– *Full ATL*: Incorporate other mechanisms of ATL to PTL: ECore inheritance, lazy rules, called rules (among them the entry point) and rule inheritance. A mapping of a richer fragment of OCL to Prolog will be also considered.

— *Code generation*: we are also interested in the use of our language for code generation.

Acknowledgment

The authors would like to thank the anonymous reviewers for their helpful and constructive comments that greatly contributed to improving the final version of the paper. This work was funded by the EU ERDF and the Spanish Ministry of Economy and Competitiveness (MINECO) under Projects TIN2013-41576-R and TIN2013-44742-C4-4-R, and the Andalusian Regional Government (Spain) under Project P10-TIC-6114.

[1] K. Czarnecki, S. Helsen, Classification of Model Transformation Approaches, in: OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003.

[2] F. Jouault, I. Kurtev, On the interoperability of model-to-model transformation languages, Sci. Comput. Program. 68 (3) (2007) 114–137.

[3] T. Mens, P. V. Gorp, A Taxonomy of Model Transformation, Electr. Notes Theor. Comput. Sci. 152 (2006) 125–142.

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

[5] M. Amrani, J. Dingel, L. Lambers, L. Lúcio, R. Salay, G. Selim, E. Syriani, M. Wimmer, Towards a model transformation intent catalog, in: Proceedings of the First Workshop on the Analysis of Model Transformations, ACM, 2012, pp. 3–8.

[6] OMG, Model driven architecture, Tech. rep. (2014).
URL `http://www.omg.org/cgi-bin/doc?ormsc/14-06-01`

[7] OMG, Unified Modeling Language Specification, version 2.4.1, Tech. rep., Object Management Group (2011).
URL `http://www.omg.org/spec/UML/`

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

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

[10] J.-M. Favre, Towards a basic theory to model model driven engineering, in: 3rd Workshop in Software Model Engineering, WiSME, 2004.

[11] A. Gerber, M. Lawley, K. Raymond, J. Steel, A. Wood, Transformation: The Missing Link of MDA, in: Procs of ICGT'02, LNCS 2505, Springer, 2002, pp. 90–105.

[12] M. Lawley, J. Steel, Practical Declarative Model Transformation with Tefkat, in: MoDELS Satellite Events, LNCS 3844, Springer, 2006, pp. 139–150.

[13] 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, ACM, New York, NY, USA, 2007, pp. 971–977.

[14] D. Varró, Automated model transformations for the analysis of IT systems, Ph.D. thesis (2004).

[15] B. Schätz, Formalization and Rule-Based Transformation of EMF Ecore-Based Models, in: D. Gašević, R. Lämmel, E. Wyk (Eds.), Software Language Engineering, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 227–244.

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

[17] E. K. Jackson, W. Schulte, Formula 2.0: A language for formal specifications, in: Unifying Theories of Programming and Formal Engineering Methods, Springer, 2013, pp. 156–206.

[18] H. Storrle, A Prolog-based Approach to Representing and Querying UML Models, in: Intl. Ws. Visual Languages and Logic (VLL'07), Vol. 274, 2007, pp. 71–84.
URL `http://ceur-ws.org/Vol-274/`

[19] J. Oetsch, J. Puehrer, M. Seidl, H. Tompits, P. Zwickl, VIDEAS: Supporting Answer-Set Program Development using Model-Driven Engineering Techniques, in: Proceedings of the MELO 2011Workshop: Model-Driven Engineering, Logic and Optimization: friends or foes?, 2011.

[20] M. Goldberg, G. Wiener, A Declarative Approach for Software Modeling, in: Practical Aspects of Declarative Languages, Springer LNCS 7149, 2012, pp. 18–32.

[21] 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, IEEE, 2008, pp. 81–88.

[22] T. Hettel, M. Lawley, K. Raymond, Towards Model Round-Trip Engineering: An Abductive Approach, in: Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations, ICMT'09, Springer-Verlag, 2009, pp. 100–115.

[23] Z. Khai, A. Nadeem, G.-s. Lee, A Prolog Based Approach to Consistency Checking of UML Class and Sequence Diagrams, in: T.-h. Kim, H. Adeli, H.-k. Kim, H.-j. Kang, K. J. Kim, A. Kiumi, B.-H. Kang (Eds.), Software Engineering, Business Continuity, and Education, Vol. 257 of Communications in Computer and Information Science, Springer Berlin Heidelberg, 2011, pp. 85–96.

[24] A. Cicchetti, D. Di Ruscio, R. Eramo, A. Pierantonio, JTL: a bidirectional and change propagating transformation language, in: Software Language Engineering, Springer, 2011, pp. 183–202.

[25] J. Cabot, R. Clarisó, D. Riera, Verifying UML/OCL operation contracts, in: Integrated Formal Methods, Springer, 2009, pp. 40–55.

[26] X. Blanc, I. Mounier, A. Mougenot, T. Mens, Detecting model inconsistency through operation-based model construction, in: Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on, IEEE, 2008, pp. 511–520.

[27] D. Batory, E. Latimer, M. Azanza, Teaching model driven engineering from a relational database perspective, in: Model-Driven Engineering Languages and Systems, Springer, 2013, pp. 121–137.

[28] M. Hibberd, M. Lawley, K. Raymond, Forensic debugging of model transformations, in: Model Driven Engineering Languages and Systems, Springer, 2007, pp. 589–604.

[29] G. Bergmann, Z. Ujhelyi, I. Ráth, D. Varró, A Graph Query Language for EMF Models, in: J. Cabot, E. Visser (Eds.), Theory and Practice of Model Transformations, Vol. 6707 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 167–182.

[30] Z. Ujhelyi, G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, D. Varró, Emf-incquery: An integrated development environment for live model queries, Science of Computer Programming 98, Part 1 (0) (2015) 80 – 99, fifth issue of Experimental Software and Toolkits (EST): A special issue on Academics Modeling with Eclipse (ACME2012).

[31] J. Schoenboeck, G. Kappel, A. Kusel, W. Retschitzegger, W. Schwinger, M. Wimmer, Catch me if you can–debugging support for model transformations, in: Models in Software Engineering, Springer, 2010, pp. 5–20.

[32] P. Stevens, Bidirectional model transformations in QVT: semantic issues and open questions, Software & Systems Modeling 9 (1) (2010) 7–20.

[33] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, C. Lutz, OWL 2 Web Ontology: Reasoning in OWL 2 RL and RDF Graphs using Rules, Tech. rep., `http://www.w3.org/TR/owl2-profiles/#Reasoning_in_OWL_2_RL_and_RDF_Graphs_using_Rules` (2009).

[34] J. M. Almendros-Jiménez, OWL RL in Logic Programming: Querying, Reasoning and Inconsistency Explanations, in: A. Bikakis, A. Giurca (Eds.), RuleML, Vol. 7438 of Lecture Notes in Computer Science, Springer, 2012, pp. 248–255.

[35] A. Queralt, A. Artale, D. Calvanese, E. Teniente, OCL-Lite: Finite reasoning on UML/OCL conceptual schemas, Data & Knowledge Engineering 73 (2012) 1–22.

[36] J. Troya, A. Vallecillo, Towards a rewriting logic semantics for ATL, Theory and Practice of Model Transformations (2010) 230–244.

[37] J. M. Almendros-Jiménez, L. Iribarne, A Model Transformation Language Based on Logic Programming, in: P. van Emde Boas et al. (Eds.): 39th International Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2013, LNCS 7741, Springer-Verlag, 2013, pp. 382–394.

[38] J. M. Almendros-Jiménez, L. Iribarne, Model Validation in Ontology Based Transformations, in: J. Silva, F. Tiezzi (Eds.), WWV, Vol. 98 of EPTCS, 2012, pp. 17–30.

[39] J. M. Almendros-Jiménez, L. Iribarne, Transformation and Validation with SWRL and OWL of ODM-Based Models, in: A. Abelló, L. Bellatreche, B. Benatallah (Eds.), Model and Data Engineering - 2nd International Conference, MEDI, Vol. 7602 of LNCS, Springer, 2012, pp. 103–115.

[40] J. Wielemaker, G. Schreiber, B. Wielinga, Prolog-based infrastructure for RDF: scalability and performance, in: The Semantic Web-ISWC 2003, Springer, 2003, pp. 644–658.

[41] G. V. Batz, M. Kroll, R. Geiß, A first experimental evaluation of search plan driven graph pattern matching, in: Applications of Graph Transformations with Industrial Relevance, Springer, 2008, pp. 471–486.

[42] G. Varró, K. Friedl, D. Varró, Adaptive graph pattern matching for model transformations using model-sensitive search plans, Electronic Notes in Theoretical Computer Science 152 (2006) 191–205.

[43] OMG, MOF 2.0 Query/Views/Transformations V1.1, Tech. rep. (2011). URL `http://www.omg.org/spec/QVT/1.1/`

[44] 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, Springer, 2008, pp. 377–393.

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

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

[47] L. Tratt, The MT model transformation language, in: SAC '06: Proceedings of the 2006 ACM Symposium on Applied Computing, ACM, New York, NY, USA, 2006, pp. 1296–1303.

[48] 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 (6-10 Oct. 2003) 364–368.

[49] 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, Vol. 3062, LNCS, 2004, pp. 446–453.

[50] A. Königs, Model transformation with triple graph grammars, in: Model Transformations in Practice Satellite Workshop of MODELS, 2005, p. 166.

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

[52] B. Schätz, Verification of model transformations, Proceedings of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009). Electronic Communications of the EASST 18.

[53] B. Schatz, Prolog EMF Transformation Eclipse-Plugin, Tech. rep. (2009).
URL `http://www4.informatik.tu-muenchen.de/~schaetz/PETE/PETEFrame.html`

[54] D. I. Hearnden, Deltaware: Incremental Change Propagation for Automating Software Evolution in Model-Driven Architecture, Ph.D. thesis, Centre or Institute School of Information Tech & Elec Engineering,

Univ. of Queensland (2007).
URL http://espace.library.uq.edu.au/view/UQ:152739

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

[56] M. Clavel, M. Egea, V. T. D. Silva, MOVA: A Tool for Modeling, Measuring and Validating UML Class Diagrams.

[57] A. Boronat, R. Heckel, J. Meseguer, Rewriting Logic Semantics and Verification of Model Transformations, in: FASE '09: Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 18–33.

[58] P. Dohrmann, S. Herold, Designing and Applying a Framework for Logic-Based Model Querying, in: 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), IEEE, 2010, pp. 164–171.

[59] G. Bergmann, Á. Horváth, I. Ráth, D. Varró, A. Balogh, Z. Balogh, A. Ökrös, Incremental evaluation of model queries over EMF models, in: Model Driven Engineering Languages and Systems, Springer, 2010, pp. 76–90.

[60] B. Demuth, The Dresden OCL toolkit and its role in Information Systems development, in: Proc. of the 13th International Conference on Information Systems Development (ISD'2004), 2004.

[61] M. Clavel, M. Egea, M. A. García de Dios, Checking unsatisfiability for OCL constraints, Electronic Communications of the EASST 24.

[62] A. Boronat, MOMENT: a formal framework for MOdel manageMENT, PhD in Computer Science, Universitat Politenica de Valencia (UPV), Spain.

[63] M. Clavel, M. Egea, ITP/OCL: A rewriting-based validation tool for UML+ OCL static class diagrams, in: Algebraic Methodology and Software Technology, Springer, 2006, pp. 368–373.

[64] A. D. Brucker, B. Wolff, HOL-OCL: a formal proof environment for UML/OCL, in: Fundamental Approaches to Software Engineering, Springer, 2008, pp. 97–100.

[65] G. Bergmann, Translating OCL to graph patterns, in: Model-Driven Engineering Languages and Systems, Springer, 2014, pp. 670–686.

[66] T. Arendt, A. Habel, H. Radke, G. Taentzer, From Core OCL Invariants to Nested Graph Constraints, in: H. Giese, B. Konig (Eds.), Graph Transformation, Vol. 8571 of Lecture Notes in Computer Science, Springer International Publishing, 2014, pp. 97–112.

[67] J. Cabot, R. Claris, E. Guerra, J. de Lara, Verification and Validation of Declarative Model-to-Model Transformations, Systems and Software 2 (83) (2010) 283–302.

[68] A. Vallecillo, M. Gogolla, Typing model transformations using tracts, in: Theory and Practice of Model Transformations, Springer, 2012, pp. 56–71.

[69] F. Jouault, I. Kurtev, Transforming models with ATL, in: Satellite Events at the MoDELS 2005 Conference, Springer, 2006, pp. 128–138.

[70] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, The Maude 2.0 System, in: R. Nieuwenhuis (Ed.), Rewriting Techniques and Applications (RTA 2003), no. 2706 in LNCS, Springer-Verlag, 2003, pp. 76–87.

[71] A. Egyed, Automatically detecting and tracking inconsistencies in software design models, Software Engineering, IEEE Transactions on 37 (2) (2011) 188–204.

Appendix A

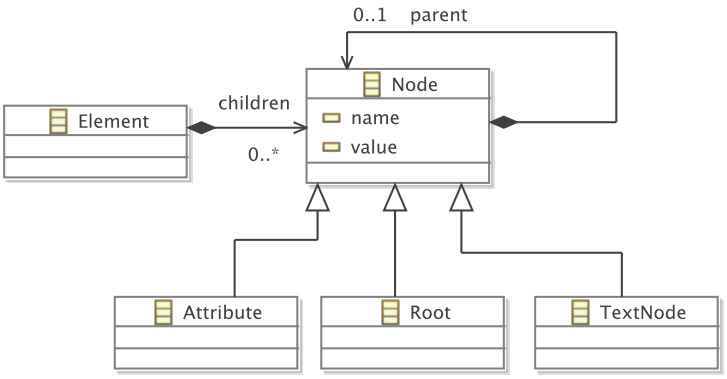
In this Appendix we would like to show some examples of use of PTL. Firstly, we will show two examples (and three transformations) taken from the ATL zoo[7]. Next, we will illustrate the advantages of PTL as hybrid language, by describing an example of transformation in which Prolog rules are used for the materialization of relations.

Example 1. *The first example taken from the ATL zoo shows how to transform a set of books. This example consists of two transformations. The first transformation takes as source model an XML document in which nodes (i.e.,* children*) are books and chapters, and transforms the document into a target model of* books *and* chapters*. The second transformation takes as source model the target of the first transformation and summarizes the content, by computing the number of pages, and the authors of the book appending the name of the author of each chapter, obtaining a target model of* publications*. In Figure 26 we can see the meta-models of the transformation. An example of source and target models is as follows:*

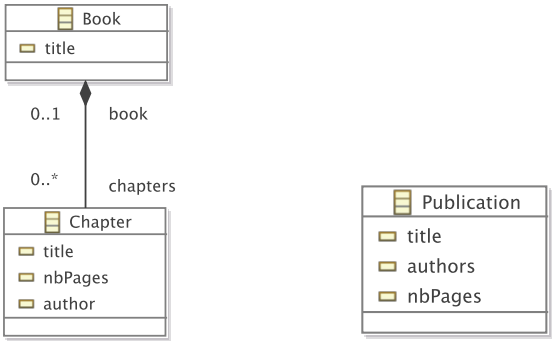
```
<root:Root>
  <children xsi:type="Element" name="book">
    <children xsi:type="Attribute" name="title" value="livre"/>
    <children xsi:type="Element" name="chapter">
      <children xsi:type="Attribute" name="title" value="chapter 1"/>
      <children xsi:type="Attribute" name="nbPages" value="13"/>
      <children xsi:type="Attribute" name="author" value="toto"/>
    </children>
    <children xsi:type="Element" name="chapter">
      <children xsi:type="Attribute" name="title" value="chapter 2"/>
      <children xsi:type="Attribute" name="nbPages" value="17"/>
      <children xsi:type="Attribute" name="author" value="toto"/>
    </children>
    <children xsi:type="Element" name="chapter">
      <children xsi:type="Attribute" name="title" value="chapter 3"/>
      <children xsi:type="Attribute" name="nbPages" value="20"/>
      <children xsi:type="Attribute" name="author" value="titi"/>
    </children>
  </children>
</root:Root>
```

```
<xmi:XMI>
  <Book title="livre">
    <chapters title="chapter 1" nbPages="13" author="toto"/>
  </Book>
  <Book title="livre">
```

[7] http://www.eclipse.org/atl/atlTransformations/.



(a) XML metamodel



(b) Book metamodel (c) Publication metamodel

Figure 26: Metamodels of Example 1

```
    <chapters title="chapter 2" nbPages="17" author="toto"/>
  </Book>
  <Book title="livre">
    <chapters title="chapter 3" nbPages="20" author="titi"/>
  </Book>
</xmi:XMI>
```

```
<xmi:XMI>
 <Publication title="livre" authors="toto and toto and titi" nbPages="50"/>
</xmi:XMI>
```

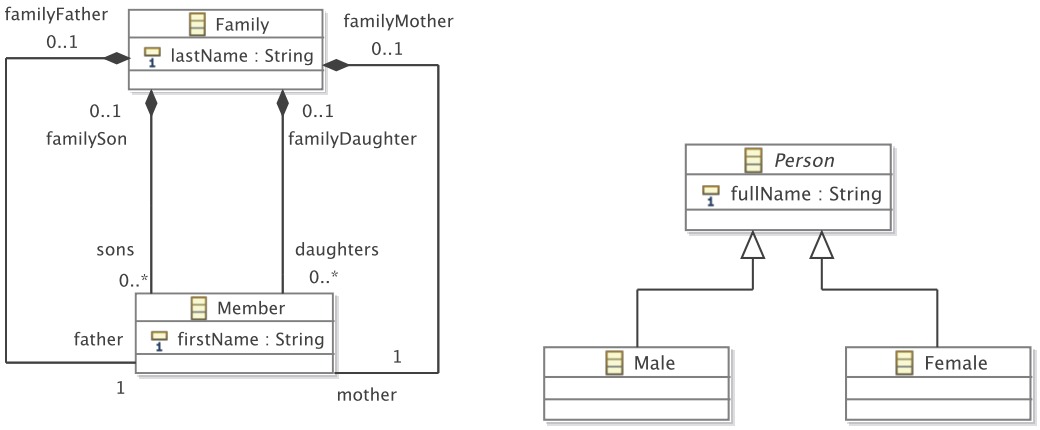
Now, the code of PTL for the first transformations is as follows:

```
metamodel(xmlf,[
  class(children,['xsi$type',name,value]),
```

```
  role(children,children,children,"0","*",container)]).

metamodel(book,[
  class('Book',[title]),
  class('Chapter',[title,nbPages,author]),
  role(chapters,'Book','Chapter',"0","*",container)]).

helper(getTitle).
getTitle(E,Y):- children_name(xmlf,E,'title'),children_value(xmlf,E,Y).

helper(getPages).
getPages(E,Y):- children_name(xmlf,E,'nbPages'),children_value(xmlf,E,Y).

helper(getAuthor).
getAuthor(E,Y):- children_name(xmlf,E,'author'),children_value(xmlf,E,Y).

rule book
  from (e:xmlf!children,l:xmlf!children,f:xmlf!children,
        g:xmlf!children,h:xmlf!children,k:xmlf!children)
    where
      (e@name == "book" and (e@children == l and
      (e@children==f and (f@name=="chapter" and
      (l@name=="title" and (f@children==g and
      (g@name=="title" and (f@children==h and
      (h@name=="nbPages" and (f@children==k and
       k@name=="author"))))))))))
  to
    (book:book!'Book'(title <- l@value,
          chapters <- resolveTemp((f,g,h,k),chapter))).

rule chapter
    from (e:xmlf!children,f:xmlf!children,
        g:xmlf!children,h:xmlf!children)
    where
      (e@name=="chapter" and (e@children == f
        and (e@children == g and e@children == h)))
    to
    (chapter:book!'Chapter'(title<-getTitle(f),
            nbPages <- getPages(g),
            author <- getAuthor(h))).
```

We have defined two rules (i.e., `book` *and* `chapter`*) and three helpers (i.e.,* `getTitle`*,* `getPages` *and* `getAuthor`*). Helpers make use of the library for the meta-models. Rule* `book` *makes use of* `resolveTemp` *to link books to chapters of the second rule. The code of PTL for the second transformation is as follows:*

```
metamodel(book,[
  class('Book',[title]),
  class('Chapter',[title,nbPages,author]),
  role(chapters,'Book','Chapter',"0","*",container)]).

metamodel(publication,[class('Publication',[title,authors,nbPages])]).

helper(getAuthors).
```



(a) Families metamodel (b) Persons metamodel

Figure 27: Metamodels of Example 2

```
getAuthors(B,Authors):-bagof(Author,(C^'Book_chapters'(book,B,C),
                        'Chapter_author'(book,C,Author)),L),
                        concat_atom(L,' and ',Authors).

helper(getSumPages).
getSumPages(B,Total):- bagof(Pages,(C^SPages^'Book_chapters'(book,B,C),
                        'Chapter_nbPages'(book,C,SPages),
                        atom_number(SPages,Pages)),LPages),
                        sumlist(LPages,Total).

rule book2Publication from
      (b:book!'Book') where (getSumPages(b)>2)
        to
        (out:publication!'Publication'(title <- b@title,
                          authors <- getAuthors(b),
                          nbPages <- getSumPages(b))).
```

Here, we have used just one rule and two helpers (i.e., `getAuthors` *and* `getSumPages`*). The helpers make use of the library for meta-models and also use the Prolog built-in* `bagof` *for collecting authors and pages of books. They also use the Prolog built-in predicates* `concat_atom` *and* `sumlist`*.*

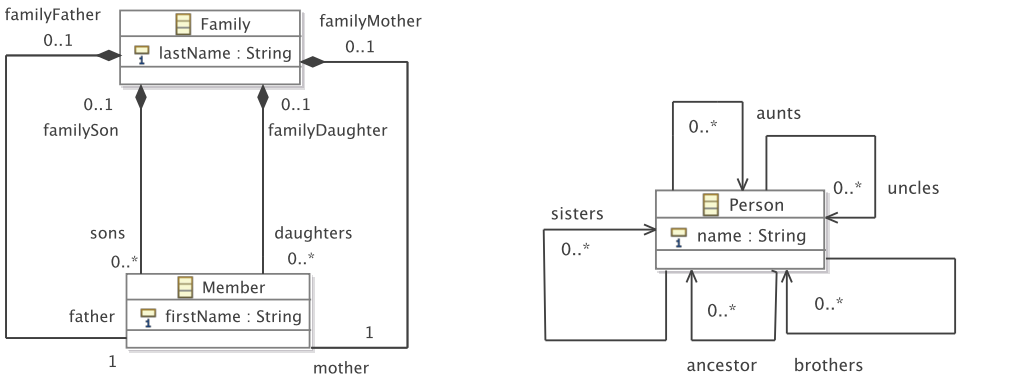
Example 2. *The second example taken from the ATL zoo is the well-known transformation Families to Persons. The metamodels are shown in Figure 27. The transformation maps a Family representation to a Male-Female representation, according to the relations* `father`*,* `mother`*,* `sons` *and* `daughters`*.*

The code of PTL for this transformation is as follows:

```
metamodel(families,[
   class('Family',[lastName]),
   class('Member',[firstName]),
   role(father,'Family','Member',"0","1",container),
   role(mother,'Family','Member',"0","1",container),
   role(sons,'Family','Member',"0","*",container),
   role(daughters,'Family','Member',"0","*",container)]).

metamodel(persons,[
   class('Male',[fullName]),
   class('Female',[fullName])]).

helper(isMale).

isMale(X,true):- 'Family_father'(families,_,X);'Family_sons'(families,_,X).

helper(isFemale).

isFemale(X,true):-
       'Family_mother'(families,_,X);'Family_daughters'(families,_,X).

helper(familyName).

familyName(X,Z):-
    ('Family_father'(families,Y,X),'Family_lastName'(families,Y,Z));
    ('Family_sons'(families,Y,X),'Family_lastName'(families,Y,Z));
    ('Family_mother'(families,Y,X),'Family_lastName'(families,Y,Z));
    ('Family_daughters'(families,Y,X),'Family_lastName'(families,Y,Z)).

rule member2Male
  from
    s : families!'Member' where (isMale(s) == true)
  to
    t : persons!'Male'(
      fullName <- concat(concat(s@firstName," "),familyName(s))).

rule member2Female
  from
    s : families!'Member' where (isFemale(s) == true)
  to
    t : persons!'Female'(
      fullName <- concat(concat(s@firstName," "),familyName(s))).
```

Here Prolog is used by the helpers `isFemale`, `isMale` *and* `familyName`, *easy to specify due to the relational based nature of Prolog. Rules (i.e.,* `member2Male` *and* `member2Female`*), use helpers in Boolean conditions and object creations. They also use the Prolog built-in* `concat`*. With the previous PTL program, we can transform the following model:*

```
<xmi:XMI>
<Family lastName="March">
    <father firstName="Jim"/>
    <mother firstName="Cindy"/>
```



(a) Families metamodel (b) Materialized Relations metamodel

Figure 28: Metamodels of Example 3

```
    <sons firstName="Brandon"/>
    <daughters firstName="Brenda"/>
</Family>
</xmi:XMI>
```

into the following one:

```
<xmi:XMI>
  <Male fullName="Jim March"/>
  <Male fullName="Brandon March"/>
  <Female fullName="Cindy March"/>
  <Female fullName="Brenda March"/>
</xmi:XMI>
```

Example 3. *The third example shows the advantages of PTL as hybrid language. The example materializes the relationships of the instance of* Families. *More concretely, the relations of a* Person *ancestors, sisters, brothers, aunts and uncles are materialized. In Figure 28, two metamodels of the transformation are shown. The first one, the source metamodel, is the same as the source metamodel of Example 2. The second one, the target metamodel, represents the materialized relations of each Person. The source model is as follows:*

```
<xmi:XMI>
<Family lastName="March">
    <father firstName="Jim"/>
```

```
    <mother firstName="Cindy"/>
    <sons firstName="Brandon"/>
    <daughters firstName="Brenda"/>
  </Family>
  <Family lastName="Sailor">
    <father firstName="Peter"/>
    <mother firstName="Jackie"/>
    <sons firstName="David"/>
    <sons firstName="Dylan"/>
    <daughters firstName="Kelly"/>
  </Family>
  <Family lastName="Sailor">
    <father firstName="David"/>
    <mother firstName="Brenda"/>
    <sons firstName="Martin"/>
    <sons firstName="Robert"/>
    <daughters firstName="Mary"/>
  </Family>
</xmi:XMI>
```

and the target model is as follows:

```
<xmi:XMI>
  <Person name="Peter"/>
  <Person name="Jackie"/>
  <Person
      name="David"
      ancestor="//@Person.1 //@Person.0 "
      brothers="//@Person.3 "
      sisters="//@Person.4 "/>
  <Person
      name="Dylan"
      ancestor="//@Person.1 //@Person.0 "
      brothers="//@Person.2 "
      sisters="//@Person.4 "/>
  <Person
      name="Kelly"
      ancestor="//@Person.1 //@Person.0 "
      brothers="//@Person.2 //@Person.3 "/>
  <Person name="Jim"/>
  <Person name="Cindy"/>
  <Person
      name="Brandon"
      ancestor="//@Person.5 //@Person.6 "
      sisters="//@Person.8 "/>
  <Person
      name="Brenda"
      ancestor="//@Person.5 //@Person.6 "
      brothers="//@Person.7 "/>
  <Person
      name="Martin"
      ancestor="//@Person.1 //@Person.2 //@Person.5 //@Person.6 //@Person.8
          //@Person.0 "
      aunt="//@Person.4 "
      brothers="//@Person.10 "
      sisters="//@Person.4 //@Person.11 "
      uncle="//@Person.3 //@Person.7 "/>
```

```
  <Person
      name="Robert"
      ancestor="//@Person.1 //@Person.2 //@Person.5 //@Person.6 //@Person.8
          //@Person.0 "
      aunt="//@Person.4 "
      brothers="//@Person.9 "
      sisters="//@Person.4 //@Person.11 "
      uncle="//@Person.3 //@Person.7 "/>
  <Person
      name="Mary"
      ancestor="//@Person.1 //@Person.2 //@Person.5 //@Person.6 //@Person.8
          //@Person.0 "
      aunt="//@Person.4 "
      brothers="//@Person.9 //@Person.10 "
      sisters="//@Person.4 "
      uncle="//@Person.3 //@Person.7 "/>
</xmi:XMI>
```

The code of PTL for this transformation is as follows:

```
metamodel(families,[class('Family',[name]),
    role(father,'Family','Family',"0","1"),
    role(mother,'Family','Family',"0","1"),
    role(sons,'Family','Family',"0","*"),
    role(daughters,'Family','Family',"0","*")]).

metamodel(familiesFull,[class('Person',[name]),
    role(ancestor,'Person','Person',"0","*"),
    role(sisters,'Person','Person',"0","*"),
    role(brothers,'Person','Person',"0","*"),
    role(uncles,'Person','Person',"0","*"),
    role(aunts,'Person','Person',"0","*")]).

helper(ancestor).

ancestors(X,Y):-'Family_father'(families,X,Y).
ancestors(X,Y):-'Family_mother'(families,X,Y).
ancestors(X,Z):-'Family_father'(families,X,Y),ancestors(Y,Z).
ancestors(X,Z):-'Family_mother'(families,X,Y),ancestors(Y,Z).

helper(sisters).

sisters(X,Y):-'Family_father'(families,X,Z),
            'Family_daughters'(families,Z,Y),Y\==X.
sisters(X,Y):-'Family_mother'(families,X,Z),
            'Family_daughters'(families,Z,Y),Y\==X.

helper(brothers).

brothers(X,Y):-'Family_father'(families,X,Z),
            'Family_sons'(families,Z,Y),Y\==X.
brothers(X,Y):-'Family_mother'(families,X,Z),
            'Family_sons'(families,Z,Y),Y\==X.

helper(uncle).

uncles(X,Y):- 'Family_father'(families,X,Z),brothers(Z,Y).
```

```
uncles(X,Y):- 'Family_mother'(families,X,Z),brothers(Z,Y).

helper(aunt).

aunts(X,Y):- 'Family_father'(families,X,Z),sisters(Z,Y).
aunts(X,Y):- 'Family_mother'(families,X,Z),sisters(Z,Y).

rule completion from f:families!'Family' to
      (ff:familiesFull!'Person'(name <- f@name,
          ancestors <- resolveTemp(ancestors( f ),ff),
          sisters <- resolveTemp(sisters( f ),ff),
          brothers <- resolveTemp(brothers( f ),ff),
          uncles <- resolveTemp(uncles(f),ff),
          aunts <- resolveTemp(aunts(f),ff))).
```

We can see in this example that Prolog is used for computing the relation `ancestors` *which is the transitive closure of both* `Family_mother` *and* `Family_father` *relations. In addition,* `uncles` *and* `aunts` *are defined in terms of* `brothers` *and* `sisters` *relations, respectively. Thus Prolog makes possible to handle recursive relations in a very simple way. The program has just one rule (i.e.,* `completion`*), which is a recursive rule thanks to the use of* `resolveTemp`*.*

We can also specify validation rules for the examples. They are shown below:

```
Example 1 (Source Model) All chapters have an author:

:- \+ (children_id(xmlf,A),children_name(xmlf,A,'book'),
        children_children(xmlf,A,B),children_name(xmlf,B,'chapter'),
    \+ (children_children(xmlf,B,C),children_name(xmlf,C,'author'))).

Example 2 (Source Model) Name of father and mother are distinct:

:- \+ ('Family_father'(families,A,B),
      'Family_mother'(families,A,C),
      'Member_firstName'(families,B,E),
      'Member_firstName'(families,C,F),
      E==F).

Example 3 (Target Model) Uncles are brothers of ancestors.

:- \+ ('Person_uncles'(familiesFull,A,B),
        \+ ('Person_brothers'(familiesFull,C,B),
          'Person_ancestors'(familiesFull,A,C))).
```

Appendix B

In this Appendix we show the full set of rules of the ER2RL example.

```
input('metamodelA','root',er,'object-model-A.xmi').
output('metamodelB','root',rl,'object-model-B.xmi').

helper(inverse1_qualifier).

inverse1_qualifier(IdP,IdQ):-
      role_has_role(er,IdP,IdAss),
      relation_is_role(er,IdAss,IdRole),
      role_navigable(er,IdRole,false),
      role_is(er,IdRole,IdQ).

helper(inverse2_qualifier).

inverse2_qualifier(IdP,IdRole):-
      role_has_role(er,IdP,IdAss),
      relation_is_role(er,IdAss,IdRole),
      role_navigable(er,IdRole,false).

helper(inverse2_row).

inverse_row(IdQ,IdRole2):-qualifier_has(er,IdQ,IdRole),
      role_has_role(er,IdRole,IdAss),
      relation_is_role(er,IdAss,IdRole2),
      role_navigable(er,IdRole2,true).

rule table1_er2rl from
  p:er!data to
    (t:rl!table(
      name <- p@container,
      has <- r),
     r:rl!row(
      name <- p@name,
      table <- t,
      is_key <- resolveTemp(p@attr_of,k),
      is_col <- resolveTemp(p@attr_of,c))).

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),
          resolveTemp((inverse1_qualifier(p),inverse2_qualifier(p)),f2)]))).

rule key_er2rl from
  p:er!attribute where (p@key==true)
    to
    (k:rl!key(
        name <- p@name,
        type <- p@type,
        has_key <- resolveTemp(p@is,r))).
rule col_er2rl from
   p:er!attribute where (p@key==false) to
```

```
   (c:rl!col(
      name <- p@name,
      type <- p@type,
      has_col <- resolveTemp(p@is,r))).

rule foreign1_er2rl from
  (p:er!qualifier,q:er!role) where (p@has == q   and q@navigable==true) to
    (f1:rl!foreign(
      name <- concat(concat(q@name,q@is_data@name),p@name),
      type <- p@type,
      has_foreign <- resolveTemp(q,r))).

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 <- resolveTemp(inverse_row(p),r))).
```

```
module er2rl;
create OUT : rl from IN : er;

helper def: inverse1_qualifier(r:e!role) : er!qualifier =
  r.has_role.is_role->select(r | not r.navigable)->collect(e|e.is);

helper def: inverse2_qualifier(r:er!role) : er!role =
  r.has_role.is_role->select(r | not r.navigable);

helper def: inverse2_row(q:er!qualifier) : er!role =
  q.has.has_role.is_role->select(r | r.navigable)->first();

rule table1_er2rl {
  from
    p:er!data
  to
    t:rl!table(
      name <- p.container,
      has <- r),
     r:rl!row(
      name <- p.name,
      table <- t,
      is_key <- thisModule.resolveTemp(p.attr_of,'k'),
      is_col <- thisModule.resolveTemp(p.attr_of,'c'))
}

rule table2_er2rl {
  from
    p:er!role (p.navigable and p.max=5)
  to
    t:rl!table(
      name <- p.name,
      has <- r),
    r:rl!row(
      name <- p.name + p.is_data.name,
      table <- t,
      is_foreign <-
            Sequence{
```

```
            thisModule.resolveTemp(
                Tuple{p1:er!qualifier=p.is, p2:er!role=p},'f1'),
            thisModule.resolveTemp(
                Tuple{
                p1:er!qualifier=thisModule.inverse1_qualifier(p),
                p2:er!role=thisModule.inverse2_qualifier(p)},'f2')})
}

rule key_er2rl {
  from
    p:er!attribute (p.key)
  to
    k:rl!key(
        name <- p.name,
        type <- p.type,
        has_key <- thisModule.resolveTemp(p.is,'r'))
}

rule col_er2rl {
  from
    p:er!attribute (not p.key)
  to
    c:rl!col(
      name <- p.name,
      type <- p.type,
      has_col <- thisModule.resolveTemp(p.is,'r'))
}

rule foreign1_er2rl {
  from
    p:er!qualifier, q:er!role (p.has=q and q.navigable)
  to
    f1:rl!foreign(
         name <- (q.name + q.is_data.name) + p.name,
         type <- p.type,
         has_foreign <- thisModule.resolveTemp(q,'r'))
}

rule foreign2_er2rl {
  from
    p:er!qualifier, q:er!role (p.has=q and not q.navigable)
  to
    f2:rl!foreign(
      name <- (q.name + q.is_data.name) + p.name,
      type <- p.type,
      has_foreign <- thisModule.resolveTemp(thisModule.inverse2_row(p),'r'))
}
```

Appendix C

In this Appendix we show the queries (Q1) to (Q4) in Prolog.

```
(Q1) switchSensor(Individual):-
      'contains_xsi:type'(concept,Individual,'Concept:Switch'),
      \+ contains_TrackElement_sensor(concept,Individual,_).
```

```
(Q2) posLength(Source,Target):-
      contains_Segment_length(concept,Source,Target),
      'contains_xsi:type'(concept,Source,'Concept:Segment'),
      atom_number(Target,D),D=<0.

(Q3) routeSensor(Sen,Sw,Sp,R):-
      contains_Route_switchPosition(concept,R,Sp),
      contains_SwitchPosition_switch(concept,Sp,Sw),
      contains_TrackElement_sensor(concept,Sw,Sen),
      'contains_xsi:type'(concept,R,'Concept:Route'),
      'contains_xsi:type'(concept,Sw,'Concept:Switch'),
      'contains_xsi:type'(concept,Sp,'Concept:SwitchPosition'),
      'contains_xsi:type'(concept,Sen,'Concept:Sensor'),
      \+ contains_Route_routeDefinition(concept,R,Sen).

(Q4) signalNeighbor(R1):-exitSignalSensor(SigA,R1,Sen1A),
      connectingSensors(Sen1A,Sen2A),
      rDefinition(R3A,Sen2A),
      R3A \== R1,
      \+ entrySignalSensor(SigA,_R2A,Sen2A).

exitSignalSensor(Sig,R1,Sen1):-exitSignal(R1,Sig),
        rDefinition(R1,Sen1).

entrySignalSensor(Sig,R2,Sen2):-entrySignal(R2,Sig),
      rDefinition(R2,Sen2).

entrySignal(R,Sig):-contains_Route_entry(concept,R,Sig),
        'contains_xsi:type'(concept,R,'Concept:Route'),
      'contains_xsi:type'(concept,Sig,'Concept:Signal'),

exitSignal(R,Sig):-contains_Route_exit(concept,R,Sig),
      'contains_xsi:type'(concept,R,'Concept:Route'),
      'contains_xsi:type'(concept,Sig,'Concept:Signal').

rDefinition(R,Sen):-contains_Route_routeDefinition(concept,R,Sen),
        'contains_xsi:type'(concept,R,'Concept:Route'),
      'contains_xsi:type'(concept,Sen,'Concept:Sensor').

connectingSensors(Sen1,Sen2):-sensorTrackelement(Sen1,Te1),
        sensorTrackelement(Sen2,Te2),
        trackelementConnected(Te1,Te2).

trackelementConnected(Te1,Te2):-
      contains_TrackElement_connectsTo(concept,Te1,Te2).

sensorTrackelement(Sen,Te):-
      contains_Sensor_trackElement(concept,Sen,Te),
      'contains_xsi:type'(concept,Sen,'Concept:Sensor').
```