A minimal OCL-based Profile for Model Transformation

Roxana Giandini (1) (2)   Gabriela Pérez (1)   Claudia Pons (1) (3)
(1) LIFIA, Facultad de Informática,Universidad Nacional de La Plata, Buenos Aires, Argentina
(2) Dpto. Sistemas de Información, Facultad Regional La Plata,
Universidad Tecnológica Nacional.
(3) CONICET (Consejo Nacional de Investigaciones Científicas y Técnicas)
[giandini, gperez, cpons]@lifia.info.unlp.edu.ar

Abstract

The MDD (Model Driven Development) initiative covers a broad spectrum of research areas such as modeling languages, definition of transformation languages among models, and construction of support tools. Currently, some of these aspects are being established and applied, while others are still in the process of definition. Consequently, it is necessary to make every effort to convert MDD and its concepts and related techniques into a coherent proposal, based on open standards, and supported by mature tools and techniques.

Transformations among models require specific languages for their definition. These languages must have a formal base, for instance, a metamodel that supports them and allows for an automated treatment. This paper presents a declarative language for model transformations inspired on OMG (Object Management Group) standards. Our proposal is expected to be a minimal extension of the already existing OMG specifications, and it basically uses OCL (Object Constraint Language) language to specify transformation relations.

1. Introduction

In the MDD paradigm [1], models are thought to be the primary conductors in all software development aspects. A PIM (Platform Independent Model) is transformed into one or more PSMs (Platform Specific Model); hence, a specific PSM is generated for each specific technological platform. Model transformation is the MDD engine; models are no longer mere contemplative entities and become productive entities.

The MDD initiative covers a broad spectrum of research areas: modeling languages, definition of languages for model transformation, construction of support tools for the different tasks involved, application of concepts to development methods and specific domains, etc. Currently, some of these aspects are well-based, and are being applied with some success; however, other aspects are still undergoing their definition process. In this context, it is necessary to make every effort to convert MDD and its concepts and related techniques into a coherent proposal, based on open standards, and supported by mature tools and techniques. Model transformations require specific languages for their definition; these languages should have a formal base, for example, a metamodel that supports them and allows for an automated treatment.

In this paper, we present a pure declarative language to express transformations among models whose initial metamodel is inspired in the QVT (Query\View\Transformation) language [4], which is the OMG [2] specification for transformations, still undergoing a definition process. The language we propose aims at being minimal for expressing queries and transformations among models.

The organization of this paper is as follows: in Section 2 the concept of model transformation is introduced by an example specified in QVT. Section 3 presents and discusses some disadvantages about QVT notation, thus informally realizing our proposed statement redefining the example. Section 4 presents a step-by-step construction for a transformation profile, based on the definition of stereotypes and the use of OCL language giving thus a formal base to our proposal. In Section 5, this profile is applied to the example. Finally, conclusions and future working lines are presented.

2. A Model Transformation example

For a better understanding of the model transformation notion, we present a simple example extracted from QVT manual. This example shows the textual specification of a transformation called UML2Rdbms that defines a relation Class2Table that in turn transforms UML classes (which fulfill the constraint of being persistent, as indicated in the when clause of this relation) into Relational Model tables. For each class a corresponding table under the same name must exist. Also the transformation presents another relation: the Attr2Col, which specifies that the class attributes are in agreements with the columns under the same name in the corresponding table (the relation only transforms not multivalued attributes whose type is a basic data type). This relation is invoked in the where clause of Class2Table and it means that each time Class2Table is fulfilled for a class and a table, Attr2Col for its attributes and columns, respectively, are also fulfilled:

Transformation UML2Rdbms (Uml: UML2.0, Rel: RDBMS) {

TopLevel **Relation** Class2Table {
checkonly domain Uml c: Class {name = n }
checkonly domain Rel t: Table {name = n }
when { isPersistent = true }
where { Attr2Col (c, t) } }

Relation Attr2Col {
checkonly domain Uml c: Class { attribute = a: Attribute {name = an, type = p: DataType {name = dt}} }
checkonly domain Rel t: Table {column: col:Column {name = an, type.name = dt } }
when { not a.isMultivalued() }
where { col.type = a.type } }
}

A graphic instantiation of this transformation could be, for instance, the one shown in Figure 1 between two concrete models: Uml (from UML) and Rel (from Rdbms). In this case, for the class Person of the Uml model, there is a Table in the Rel relational model under the same name. Attributes undergo the same process: for each attribute of the class Person there is a column in the table Person under the same name and with the same type. That is to say, for the attribute `name´, there is a column `name´, and both are strings.

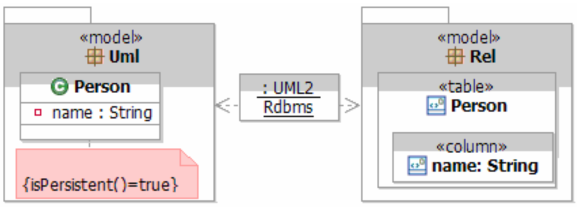

Figure 1. Transformation UML2Rdbms graphic instantiation.

3. Our proposal – Informal Introduction

In the previous example, certain issues that can be generally found in QVT examples could be observed. We will give some details of these issues over this particular example; they could be considered general disadvantages that hinder the QVT use and understanding.

Some of these issues are:

- Transformation relations are expressed through template expressions in each domain. Then, a pattern matching among these expressions will be performed to concrete the transformation. Due to the fact that in these expressions, variables can be recursively defined, this nesting and the consequent pattern matching could be broadcast producing complicated specifications.
- The relation Attr2Col suggests its application over all attributes of the Class parameter, but this is not explicitly indicated. Similarly, the same situation happens for the parameter Table. To denote that without ambiguity, we could use OCL language.
- Also, the same relation transforms specifically attribute to column, but its parameters are Class and Table. It would be clearer if each relation has as parameters the elements that actually transforms.

Consequently, we propose a new way to specify the same transformation considering the mentioned disadvantages, making more readable the notation, following a more friendly syntax and maximizing the OCL language use.

3.1 Redefining the example

In our proposal, each domain is specified with just the participating variables, while the conditions, that must be fulfilled among them, are specified in the where clause of the relation. It means, we are using neither template expressions nor pattern matching. Furthermore, in the relation Class2Table we explicitly specify in OCL the iteration over class attributes and we call the relation with the adequate parameters: Attribute and Column. Then, variables used in domain and codomain have respectively the proper type: Attribute and Column. Preconditions that attributes must fulfill, appear in the when clause, while post conditions are included in the where clause.

The example redefined in our notation is as follows:

Transformation UML2Rdbms (Uml: UML2.0, Rel: RDBMS) {

TopLevel **Relation** Class2Table {
checkonly domain Uml c: Class
checkonly domain Rel t: Table
when { c.isPersistent = true }

```
    where { c.name = t.name and c.allAttribute-> forAll
    (a |     ( t.column -> exists (co | Attr2Col (a, co))))   }
    }
    Relation  Attr2Col {
    checkonly domain Uml a: Attribute
    checkonly domain Rel co: Column
    when        {not        a.isMultivalued()        and
a.type.oclIsKindOf (DataType)}
    where { co.type = a.type and co.name = a.name } }
}
```

In the next section we discuss an approach to define a formal base to the proposed notation. In order to realize this, a UML Infrastructure extension is constructed. This profile is based in stereotypes definition and maximizes the use of the OCL to avoid the creation of unneeded new elements. Stereotypes and constraints definition is the standard extension mechanism of UML. This mechanism promotes to extend existing metaclasses instead of defining new ones. In turn, it is commendable to minimize the amount of new stereotypes maintaining the simplicity and reducing the user's training time, encouraging the use of this profile.

4. A minimal OCL-based Infrastructure Extension for Model Transformation

Previously to present the analysis of how and where the transformation language is defined, let us review some concepts on techniques for metamodeling and definition languages.

4.1 Model Transformation in the 4-layer Metamodeling Architecture

Metamodeling is a mechanism that allows for the formal construction of modeling languages such as UML and RDBMS. The 4-layer Metamodeling Architecture is the OMG proposal aiming at standardizing modeling-related concepts, from the most abstract to the most concrete ones.

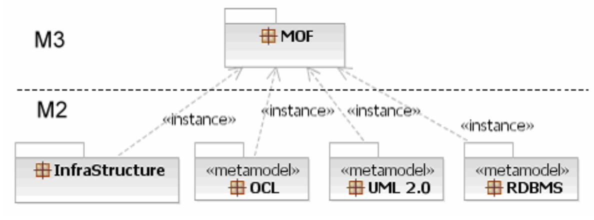

Figure 2. Layers M3 and M2 of the 4-layer Architecture

The levels defined in this architecture are commonly called: M3, M2, M1, and M0. Level M3 is the most abstract one, where MOF [3] is located; level M2 defines languages for specifying models, for instance UML; level M1 defines instances of a metamodel; Level M0 models the real system.

In this context, a specific transformation is located in layer M2 so as to be able to relate generic instances of concrete metamodels (located in M2), such as those of UML and RDBMS, while instances of these metamodels (for example a UML Class and an RDBMS Table) are actually transformed. That is to say, models that are concretely involved in the transformation (layer M1) are parameters for the transformation language.

It is reasonable to think that the metamodel for transformations and its instantiation cannot coexist in the same layer since they represent different levels of abstraction. Then definition of languages for model transformation may be realized in the layer M3 of the 4-layer Modeling Architecture.

However, MOF represents a close Meta-metamodel upon which metamodels (MOF instances) are instantiated and it is located in layer M3. Therefore, the transformation metamodel should be located in layer M2, together with the remaining metamodels (UML, OCL, etc.) as shown in Figure 2. Let us analyze other OMG specifications, closely related to MOF: one of these specifications is the Infrastructure [7] for modeling languages. The Infrastructure is the most simplified specification defining the basic constructors and the common concepts for modeling language. It could be said that Infrastructure is independent from the UML itself. The UML metamodel is complemented by the Superstructure [6] specification, that defines the constructors at UML 2.0 user's level.

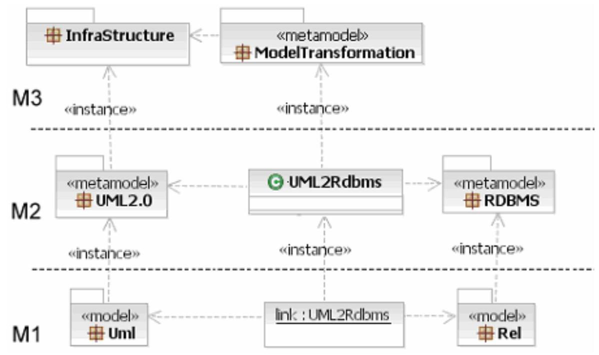

Figure 3. Model Transformation in 4-layer Architecture

The Infrastructure case is interesting for its recursive definition regarding MOF: on the one hand, it can be defined as a MOF instance (in layer M2 as shown in Figure 2); on the other hand, MOF itself is based on, or uses elements of the Infrastructure Core package for its definition, thus allowing for the identification of the Infrastructure as a meta-metamodel. If the new transformation metamodel is defined as an extension of

elements of the Core package, it is questionable to think that it lies in level M3 together with the Infrastructure (see Figure 3). Then, a transformation metamodel instance in layer M2 relates metamodel elements, while in layer M1 will be a correspondence link among models - instances of such metamodels, which are the ones that are concretely transformed.
In the QVT specification document, the metamodel is defined as a "MOF and OCL extension" [5]. Our point of view, following what has been earlier analyzed and the OMG definitions, is that it is not technically correct to extend MOF since it represents a close Meta-metamodel upon which metamodels are instantiated. Hence, we propose to minimally extend Infrastructure 2.0 and use OCL 2.0 to implement the transformation metamodel.
The QVT definition consists of three packages: two declarative packages (Relations and Core) and one imperative package. A large number of the languages proposed for model transformation [8, 9, 10, 11 and 12] are based on the QVT language. Most of these language proposals are hybrid, being both declarative and imperative. To define a transformation relation among models, it seems to be enough to specify it declaratively and then choose in which imperative language it could be executed. Just then, executable statements in agreement with the declarations should be written. Thus, we chose to propose a pure declarative language for transformations, whose initial metamodel is inspired in the QVT Relations package. permitting us to express relations and queries of model transformation.

4.2 Construction of the minimal profile

Following what has been stated in the above section, our proposal is to extend Infrastructure, by creating a specific profile (instance of the Profile metaclass from the Infrastructure) capable of expressing particular features of model transformation.
The extension offers a formal base that reflects our textual notation proposed in section 3. It is defined in five stages, as suggested in other works defining profiles:

First stage: definition of the proposed metamodel

Based on the QVT Relations package metamodel, the metamodel proposed (see Figure 4) uses the OCL existing metaclasses and minimizes the metamodel proposed in our previous work [18]. Specifically speaking, the concrete syntax (BNF grammar) of that package was adapted with the purpose of simplify its use and understanding, while keeping the necessary basic concepts; eliminating several abstract classes (such as Rule, RelationDomain, Pattern, PatternDomain, etc.) and other concepts that appear in the transformation domain (such us Query and Helper) were taken into account.
In our metamodel, a transformation consists of typed

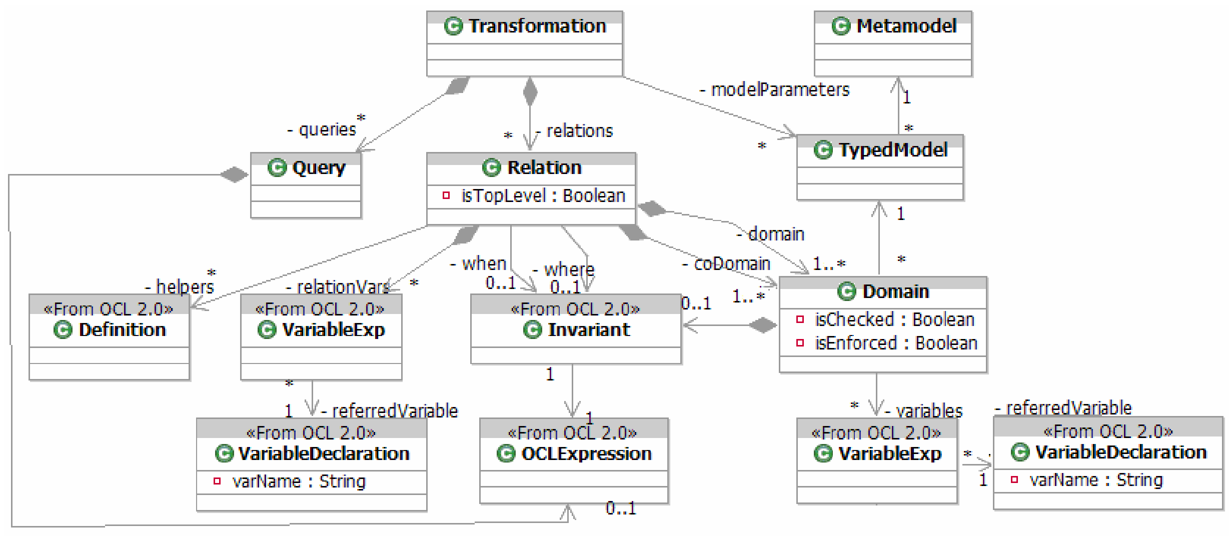

Figure 4. Metamodel proposed for Model Transformation

Then, we implemented it as an Infrastructure extension using the existing OCL metaclasses. This metamodel is expected to be minimal in the set of metamodels models, relations and queries. Each typed model has a metamodel as type - any MOF instance – and it participates in the transformation as source and/or target. A relation has a Boolean attribute isTopLevel, representing its level. A top level relation is executed automatically; otherwise the relation must be explicitly called by another one to be executed. Also relations may include variable declarations, when and where clauses - preconditions that should be fulfilled so as to achieve the transformation and postconditions that have to be fulfilled when finishing the transformation execution, respectively - and helpers. A helper, an element not taken into account in the QVT metamodel, defines additional operations in order to perform navigation upon the models participating in the transformation. Helpers could optionally have input parameters and use recursion. They consist of variable declarations and expressions that specify the additional operations definition. The application of a helper does not produce side effects.

Moreover, relations are defined from domains to codomains; each domain corresponds to a model and can only be checked or else, forced. When a domain is checked, its constraints will be evaluated regarding the current model-associated values. When a domain is forced, it is also checked, but in this case, the associated model can be updated (that is to say, object creation and/or erasure may occur) to realize the transformation.

In the QVT metamodel, TemplateExp metaclass represents complex expressions that consist in variable declarations and optionally other constraints. They are specified into the relation domains. The purpose of a TemplateExp is to arbitrarily define nested objects and/or collections templates for pattern matching and instantiation. In the QVT document the TemplateExp hierarchy is specify by new metaclasses added to the OCL metamodel.

To be able to understand the Template Expression issue, let us use the example in section 2. In the relation Attr2Col, the TemplateExp of the UML domain, is c : Class { attribute = a: Attribute, { name = an, type = p: DataType {name = dt}} } and the TemplateExp of the Rel codomain is t: Table { column: col:Column {name = an, type.name = dt }.

We propose to separate the variable definitions that could be nested, from the expressions which are used to match patterns in candidate models. For this reason, we define variables through the OCL Variable-Declaration metaclass, and the pattern matching expressions are added in the where clause. Specifically, to represent the variable definitions in domain and codomain, we propose to use the OCL VariableExp metaclass, an OCLExpression which consists of a reference to a typed variable (OCL VariableDeclaration metaclass) through the referredVariable association (see Fig. 4). Then, after analyzing the OCL 2.0, we conclude that maximizing the use of this language it is necessary to define neither new stereotypes nor new metaclasses to the proposed profile.

As far as queries are concerned, they specify instance transformations, in the same way as relations, but with a simpler format. The idea is that they are side-effect-free functions, with formal parameters and an OCLExpression as body.

Second stage: Metamodel stereotypes definition

For each metaclass of the step above, a new stereotype has been defined. Unlike in UML, in Infrastructure the stereotype's taggedValues are properties called ownedAttribute. The following subsection shows the new stereotypes with its extended metaclasses.

Regarding the relationships among the proposed metaclasses, we have decided not to stereotype them to avoid using an excessive notation when instancing the profile. We have chosen to add constraints to the connecting concepts. Association is used to relate

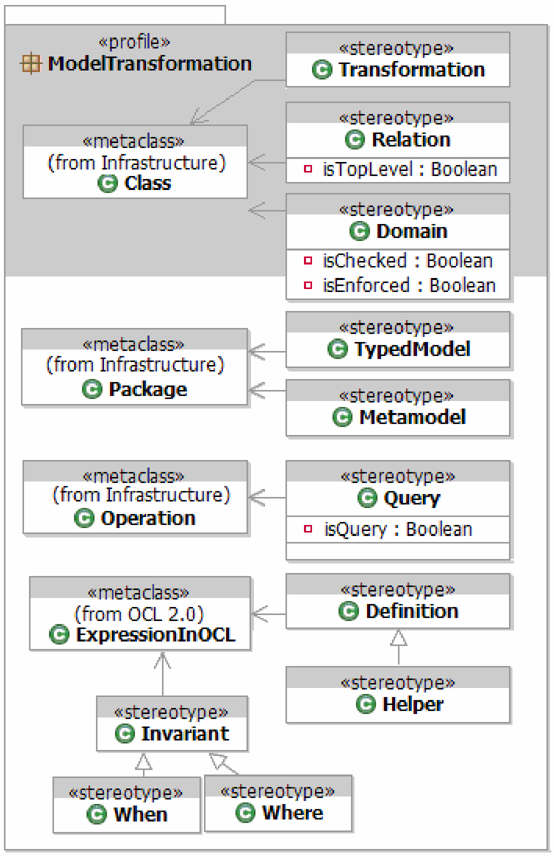

Figure 5. ModelTransformation profile

Classes, and DirectedRelationship[1] when relating Packages (as packages are not Classifiers, they cannot relate via Association).

Third stage: Identification of the proposed stereotypes's extended metaclasses.

In this subsection, the Infrastructure metaclasses that could be extended to represent the proposed metamodel are identified.

According to this metamodel, a transformation could be defined as a Class formed by the models it transforms, and by the relations and queries defined among them. Hence, all models are referred to without indicating to which one transformation is not applied, nor in which direction it is performed. This declarative style best fits for our proposal. In this sense, a relation can also be represented extending the metaclass Class, made up of the intervening domains, and the conditions and helpers constraining it. Domain adds features to models and also has Class as extended metaclass.

The query stereotype extends the metaclass Operation. Its Boolean attribute isQuery already exists in Operation.

In general, models can be represented in packages. Therefore, TypedModel and Metamodel are defined extending the metaclass Package.

The relationship between transformation and query concepts is represented by defining the owner of the <<query>> operation as a <<transformation>> class.

Finally, let us analyze the expressions (when, where and helpers) which constrain a relation of our metamodel. In MOF-based modeling languages, in many of the cases where the term expression is applied, an expression written in OCL can be used. In the OCL abstract syntax, an OCLExpression is defined recursively. For this reason the metaclass ExpressionInOCL is needed to represent the root of this recursive notation. It has a body attribute (which would be the OCLExpression) and a language attribute which should have the "OCL" value for this case. An existing stereotype for an ExpressionInOCL is <<definition>>, which constrains the ExpressionInOCL indicating that it should be defined in the context of a Classifier; it could also define additional operations and have Let expressions. This makes an OCL <<definition>> expression appropriate for modeling helpers.

As regards the when and where clauses, they are Boolean expressions, therefore we specialized the existing <<invariant>> stereotype (which metaclass extended is ExpressionInOCL), for modeling them. An <<invariant>> ExpressionInOCL is an expression whose body is a Boolean OCLExpression constraining a Classifier. In our metamodel, these clauses constrain a <<relation>> class and can refer to other relations and helpers. Figure 5 shows the stereotypes defined with their extended metaclasses, making up the ModelTransformation profile.

[1] There is neither Dependency, nor any other Relationship in the Infrastructure connecting elements different from a Classifier.

Fourth stage: Stereotype attributes definition.

For each attribute in the metamodel proposed, an attribute is defined in the corresponding stereotype. Figure 5 shows the stereotypes with their attributes: IsTopLevel of Boolean type in Relation stereotype; isChecked, isEnforced, both of Boolean type in Domain stereotype. In Section 5, we present a transformation instantiation using the defined stereotypes and OCL.

Fifth stage: Complete definition of the proposed extension

The proposed extension is a Profile class instantiation within the Infrastructure 2.0 Profile package.

For those new stereotypes which add restrictions to its base metaclass, we present below a formal definition indicating which metaclasses they stereotype, their attributes and the constraints expressed in OCL that the extended metaclasses should be fulfilled when the profile is being used. To graphically emphasize some of the concepts defined in the profile, we introduce new icons. This is the case of the transformation and relation concepts.

STEREOTYPE Transformation

Extended metaclass: Class

Graphical Notation: a transformation occurrence is shown by an hexagonal figure containing the name of the transformation and the <<transformation>> stereotype

Constraints:

[1] A transformation should contain relations or queries and should have at least two models as parameters (one input and one output)

```
(self.relations()->notEmpty() or
self.queries() -> notEmpty()) and
self.modelParameters() -> size() > 1
```

[2] The model parameters set of the transformation must be equal to the domain and codomain type models set.

```
self.modelParameters()=self.relations
()-> collect (relation: Class |
relation. typedModels()) -> flatten()
-> asSet()
```

[3] A transformation can only be related to relations or typedModels

```
self.relatedElements()->forAll(re|
re.stereotype.name='relation' or
re.stereotype.name='typedModel')
```

Additional Operations

[1] The query relations() gives all the relations associated with the transformation

```
Class:: relations(): Set (Class)
relations =
self.owningPackage.ownedMember->
select ( r:Class | r.stereotype.name
= 'relation' and
self.owningPackage.ownedMember ->
exists (a:Association | a.memberEnd -
> includes (e: Property | e.type =
self and
e.opposite.type = r))
```

[3] The query modelParameters() gives all the typedModels associated with the transformation

```
Class:: modelParameters (): Set
(Package)
modelParameters =
self.owningPackage.ownedMember->
select ( m: Package |
m.stereotype.name = 'typedModel' and
self.owningPackage.ownedMember ->
exists
(d:DirectedRelationship | d.source =
self and d.target = m))
```

STEREOTYPE Relation

Extended metaclass: Class

Graphical Notation: a Relation occurrence is shown by a pentagonal figure containing the name of the transformation and the <<relation>> stereotype

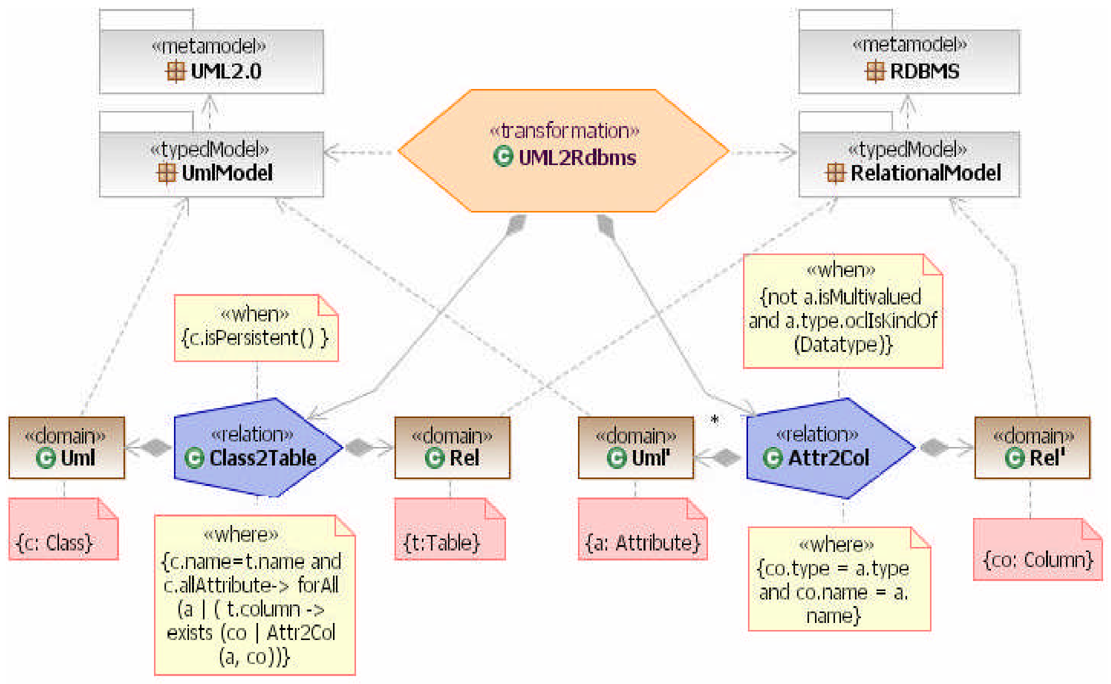

Figure 6. Instantiation of the profile for the example

[2] The query queries() gives all the queries associated with the transformation

```
Class:: queries(): Set (Operation)
queries = self.ownedOperations-
>select (o:Operation |
o.stereotype.name='query')
```

OwnedAttributes:

- isTopLevel ::Boolean - Specifies whether the relation is a top level relation. Default value is false. If isTopLevel is false, the relation must be explicitly called by another one.

Constraints:

[1]. A relation should be defined in at least one domain and one coDomain

```
self.domain()-> size()>=1 and
self.coDomain() -> size()>= 1
```

[2] A relation may have a when clause. However, it must have a where clause.

```
(self.constraint -> select(c |
c.stereotype.name = ‘when’)->
size()<= 1) and (self.constraint ->
select(c | c.stereotype.name ‘where’)
-> size()= 1)
```

Additional operations

[1] The query domain() gives all the domains associated with the relation through role name domain.

```
Class:: domain (): Set (Class)
domain =
self.owningPackage.ownedMember ->
select ( d:Class | d.stereotype.name
= ‘domain’ and
self.owningPackage.ownedMember ->
exists (a:Association |
a.memberEnd -> includes (e: Property
| e.type = self and e.opposite.type =
d and e.opposite.name = ‘domain’)))
```

[2] The query coDomain() gives all the domains associated with the relation through role name codomain.

```
Class:: coDomain(): Set (Class)
coDomain =
self.owningPackage.ownedMember->
select ( d:Class |
d.stereotype.name = ‘domain’ and
self.owningPackage.ownedMember ->
exists (a:Association |
a.memberEnd -> includes (e: Property
| e.type = self and e.opposite.type =
d and e.opposite.name = ‘codomain’)))
```

[3] The query transformation() gives all the transformation associated with the relation

```
Class:: transformation (): Set
(Class)
transformation =
self.owningPackage.ownedMember->
select ( t:Class | t.stereotype.name
= ‘transformation’ and
self.owningPackage.ownedMember ->
exists (a:Association | a.memberEnd -
> includes (e: Property | e.type =
self and e.opposite.type = t)))
```

[4] The query typedModels() gives all the type models associated with the relation (the domain and codomain typed model)

```
Class:: typedModels(): Set (Package)
typedModels =
self.domain().typedModel -> union
(self.coDomain().typedModel)
```

STEREOTYPE Domain

Extended metaclass: Class

OwnedAttributes:

- isChecked:: Boolean - Specifies whether the domain is just checked. In a checked domain, the constraints will be evaluated regarding the current model-associated values.
- isEnforced:: Boolean - Specifies whether the domain is forced. Default value is false. In a forced domain object creation and/or erasure may occur.

Constraints:

[1] A domain should make reference to only one TypedModel (there is a DirectedRelationship between Domain and TypedModel)

```
self.owningPackage.ownedMember->
select(d:DirectedRelationship |
d.source = self and
d.target.stereotype.name =
‘typedModel’)-> size()=1
```

[2] A Domain must contain at least one variable declaration

```
self.variables() -> size()>0
```

Additional operations

[1] The query relation() gives all the relation associated with the domain .

```
Class:: relation(): Set (Class)
relation =
self.owningPackage.ownedMember->
select ( d:Class | d.stereotype.name
= ‘relation’ and
self.owningPackage.ownedMember
-> exists ( a:Association |
a.memberEnd -> includes (e :
Property| e.type = self and
e.opposite.type = d))
```

[2] The query variables() gives all the variable declarations associated with the domain .

```
Class:: variables(): Set
(VariableDeclaration)
variables = self.constraint -> select
( c:Constraint |
c.body.bodyExpression.oclIsKindOf(Var
iableExp))-> collect (c:Constraint |
c.body.bodyExpression.referredVariabl
e)
```

STEREOTYPE TypedModel

Extended metaclass: Package

Constraints:

[1] A typed model should be an instance of one metamodel, that is, it should be the source of a DirectedRelationship having as target a <<metamodel>> package

```
self.metamodel() -> size() = 1
```

[2] A typed model should only contain instances of the metamodel it relates to.

```
self.ownedMember-> forAll (e|
self.metamodel().ownedMember() ->
includes (e.type))
```

Additional operations

[1] The query metamodel () gives all the metamodel packages associated with the typedModel .

```
Package:: metamodel (): Set (Package)
metamodel =
self.owningPackage.ownedMember
->select (d:DirectedRelationship |
d.source = self and
d.target.stereotype.name =
'metamodel') -> collect
(d:DirectedRelationship | d.target) )
```

STEREOTYPE When/Where (subclass of Invariant)

Extended metaclass: ExpressionInOCL

Constraints:

[1] A when/where clause must constraint a relation.

```
self.constraint.constrainedElement.
stereotype = 'relation'
```

5. Profile Application to the Example

In this Section, we show an instance of the metamodel using the proposed profile for the redefined example in Section 3. Figure 6 illustrates this example using the proposed graphic notation and stereotypes. We can see that a new instantiation of this transformation will be upon particular elements of concrete models at the M1 level of the 4-layer architecture. This model presents a transformation instantiation that defines transformation patterns upon the generic instances of the metaclasses (e.g. classes and tables, attributes and columns). The transformation has two models as parameters (umlModel and RelationalModel) and two relations (class2Table and Attr2Col). When and where clauses are OCL expressions stereotyped with <<when>> and <<where>> respectively, which constrain the relations Class2Table and Attr2Col. Both relations have one domain (Uml) and one codomain (Rel). Because of domains and codomains are structurally identical and the only difference between them are just its role into the relation, it is possible to represent both by a class stereotyped with <<domain>>. Variable declarations within the domains are also expressed in OCL via a VariableDeclaration metaclass that, as stated in subsection 4.2, matches the TemplateExp metaclass of the QVT metamodel.

6. Conclusions and Future Work

Model Transformation is the MDD engine; in this way, models, from being mere contemplative entities, become productive entities within the software development process. Model transformations require specific languages for their definition. These languages must have a formal base, for instance, a metamodel that supports them, and allows for an automated treatment.

In this paper, we have proposed a pure declarative language to express model transformations whose initial metamodel is inspired in the QVT language. Regarding the construction of a language, there are two options: One of them is create a new language based in a metamodel implicating new metaclasses definition. Another option is using the standard extension mechanism of UML that consists in new stereotypes definition extending existing metaclasses.

Our proposal is based on specifications already existing in OMG; on the one hand, by means of the stereotype definition, we extend Infrastructure 2.0, an independent and more abstract specification than UML; on the other hand we use the OCL language to express transformation patterns, since after analyzing these concepts, we concluded that it is not necessary to create new elements for modeling them.

The proposed language is expected to be minimal in the set of metamodels permitting us to express relations and queries of model transformation. To maintain simplicity and to reduce user training time are the main advantages of this minimal approach. Furthermore, since the proposal maximizes the use of OCL and there are several modeling tools that support OCL, the development work of a tool to support this transformation language is much easier.

As future working lines, in the near future, we will implement this language integrating it to the Case tool [15] that our research group is developing. This tool, oriented to formal modeling, includes the model refinement specification which can be seen as a particular model transformation case. On this topic, we have published, among other, papers [16, 17] which provide a formal base to this tool. Some of the works about model refinement that have helped us as inspiration are [13, 14]. Furthermore, we propose ourselves to incorporate to the language elements to facilitate traceability among models, consistency checking (when the transformation is forced), testing integrated to transformation, and transformation composition.

7. References

[1] MDA Guide, v1.0.1, omg/03-06-01, June 2003.

http://www.omg.org

[2] OMG (Object Management Group) http://www.omg.org

[3] Meta Object Facility (MOF) 2.0. OMG Adopted Specification. October, 2003. http://www.omg.org

[4] MOF 2.0 Query/View/Transformations - OMG Adopted Specification. March 2005. http://www.omg.org

[5] OMG. The Object Constraint Language Specification – Version 2.0, for UML 2.0, revised by the OMG, http://www.omg.org . April 2004.

[6] The Unified Modeling Language Superstructure version 2.0.,OMG April 2004. http://www.omg.org

[7] The Unified Modeling Language Infrastructure version 2.0, OMG March 2005. http://www.omg.org

[8] Jouault F., Kurtev I. Transforming Models with ATL Workshop in Model Transformation in Practice at the MODELS 2005 Conference. Montego Bay, Jamaica, 2005

[9] Akehurst D., Howells W., McDonald-Maier K. Kent Model Transformation Language. Workshop in Model Transformation in Practice at the MODELS 2005 Conference. Montego Bay, Jamaica, Oct 3, 2005

[10] Lawley M., Steel J. Practical Declarative Model Transformation with TefKat. Workshop in Model Transformation in Practice at the MODELS 2005 Conference. Montego Bay, Jamaica, Oct 3, 2005

[11] Dimitrios S. Kolovos, Richard F. Paige and Fiona A.C. Polack. The Epsilon Object Language (EOL). In Proc. Model Driven Architecture Foundations and Applications: Second European Conference, ECMDA-FA, volume 4066 of LNCS, pages 128– 142, Bilbao, Spain, June 2006.

[12] Anneke Kleppe. MCC: A Model Transformation Environment. A. Rensink and J. Warmer (Eds.): ECMDA-FA 2006, LNCS 4066, pp. 173 – 187, Spain, June 2006.

[13] Hnatkowska B., Huzar Z., Tuzinkiewicz L. On Understanding of Refinement Relationship. Workshop in Consistency Problems in UML-based SoftwareDevelopment III – Understanding and Usage of Dependency Relationships at the 7th International UML Conference.. Portugal, 2004.

[14] Liu Z., Jifeng H., Li X., Chen Y. Consistency and Refinement of UML Models. Workshop in Consistency Problems in UML-based Software Development III at the 7th International Conference on the UML.Portugal, 2004

[15] Pons C., Giandini R., Pérez G., et al. Precise Assistant for the Modeling Process in an Environment with Refinement Orientation. In UML 2004 Satellite Activities, Revised Selected Papers". Lecture Notes in Computer Science number 3297. Springer, Oct., 2004.

[16] Pons, C., Kutsche, R. Traceability Across Refinement Steps in UML Modeling. 3rd WiSME at the 7th UML Conference. October 11, 2004. Lisbon, Portugal .

[17] Pons, C., Pérez,G., Giandini, R., Kutsche, R. Understanding Refinement and Specialization in the UML. 2nd International Workshop on Managing SPEcialization/Generalization Hierarchies (MASPEGHI). In IEEE ASE 2003, Canada.

[18] Giandini, R., Pons, C. Un lenguaje para Transformación de Modelos basado en MOF y OCL. Proceedings of XXXII CLEI. Santiago, Chile. August, 2006.