International Journal of Recent Technology and Engineering (IJRTE)
ISSN: 2277-3878, Volume-3 Issue-4, September 2014

MDA Approach for Transforming a Class Diagram Conform UML 2.0 to a Relational Schema Conform CWM

Maroua Aulad Ben Tahar, Souhaib Aammou, Az-eddine Nasseh, Mohamed Khaldi

Abstract*: *the objective of this study was to determine the suitability of the use of language processing ATL model and the need to implement a process of translating a UML model into a CWM model. Through the experiment here, we could provide more answers. First, the main effort of implementation is a clear and precise definition of source and target metamodels used in processing. Second, we will show that the upstream and downstream operations of calculation could be made of the same language.

Index Terms*: *ATL, CWM, MDA

I. INTRODUCTION

The problem of the automatic generation of database from a class diagram has been widely used in computer engineering and modeling. Often, we simply examine the different scenarios that can arise in terms of multiplicity of associations between classes, and to determine in each case the relational schema suited to implement these associations. In practice, however, the process appears to be much more complex when specific cases are addressed and that we are given the demands on the quality of the resulting database. Whether in terms of the standardization process, conservation expert knowledge injected by the designer throughout the build process, and / or more practical ergonomics tools available on the market, much remains to be done. In this paper we propose a complete process in addition to addressing the issues identified above, takes advantage of a new approach full of promise in the world of software engineering: the model-driven architecture. To validate the benefits of this process and the method it represents, it was decided to try some of its implementation in the ATL language specially designed for transforming models.

Revised Manuscript Received on 30 September 2014.
* Correspondence Author
Maroua Aulad Ben Tahar*, LIROSA, Abdelmalek Essaadi University, Faculty of Sciences, Tétouan, Morocco.
Prof. Souhaib Aammou, LIROSA, Abdelmalek Essaadi University, Faculty of Sciences, Tétouan, Morocco.
Prof. Az-eddine Nasseh, LIROSA, Abdelmalek Essaadi University, Faculty of Sciences, Tétouan, Morocco.
Prof. Mohamed Khaldi, LIROSA, Abdelmalek Essaadi University, Faculty of Sciences, Tétouan, Morocco.

II. UML AND CWM METAMODEL

A. *UML 2.0 Metamodel*

UML [1] (for Unified Modeling Language) is now the standard languages object wherever it is used modeling. This language is one of the central requirements of the OMG and is now in its version 2.0.

It presents the semantic and practical qualities of great use, and in this design database case, UML has long been used to specify a high level of abstraction the structure and articulation data form respectively of classes and associations. Furthermore, an advantage of this technology both cause and consequence of its popularity, is the profusion of tools on the market more or less ergonomic dedicated to the creation and manipulation of UML diagrams. That's why he was chosen in the implementation of the algorithm presented here, using a UML class diagram as a means of dialogue with the designer, to capture its requirements and business knowledge. We will not use the full complexity of UML. We expect the designer to express only information structural type. Thus, among the many types of pre-defined by the UML [6] specification diagrams, we will use only the format classes diagram. We can even meet us with a subset of predefined classes in the Kernel package specifying the superstructure of the UML 2.0. These classes are shown in the figure 1.

B. *CWM*

CWM [2] for Common Warehouse Model, is a language like UML and has the same grammar and specializes semantics. This language is a specification adopted by the OMG, and attempts to meet the challenges posed by the following simple observations. Statistics show that the average amount of information stored by a data doubles every five years organization. In most cases, the management of this data is made extremely difficult by their superabundance, their redundancy and their heterogeneity. Storage in data warehouses structured is essential to make them usable. But a key aspect of data storage in warehouses is the description of these warehouses, so among other applications to know the structure and thus to be able to extract information. This description takes the form of metadata, which is to say in the general case of data that describe the data. To complicate the situation, we know that in practice many different storage systems data exist and are used. And therefore a need for data exchange will occur between two systems. Exchange of metadata is then used to describe the two systems (interchange of warehouse metadata).

This is the interest of the CWM language. It offers a standard for the exchange of metadata, but also to express their transformation, analysis, their creation and management. In practice, CWM is actually made up of part of the UML (the package Object Model), all the elements that are not relevant in scenarios storage data has been deleted.

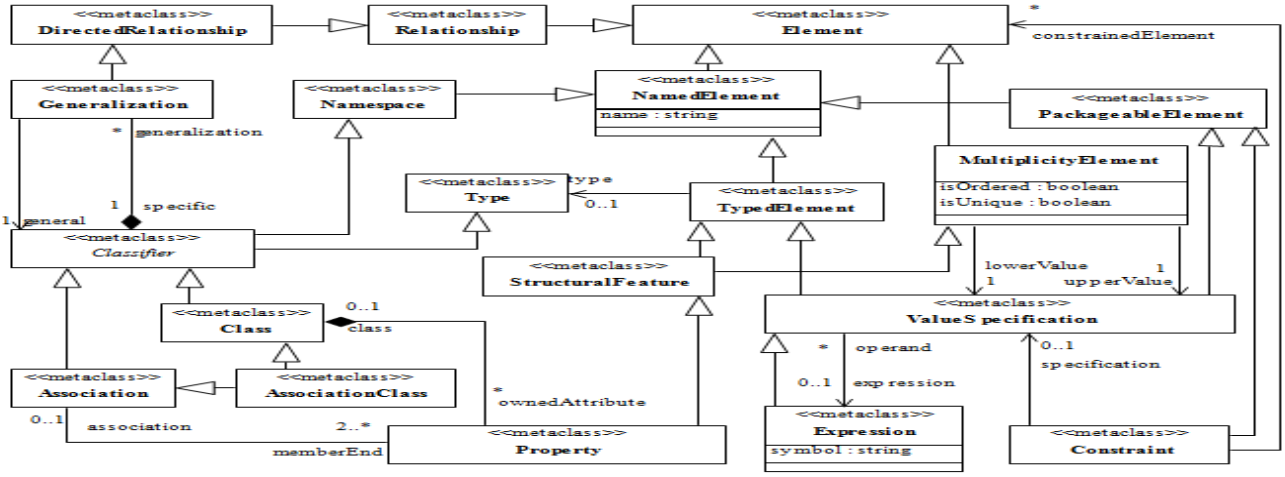

Fig. 1 UML 2.0 Metamodel

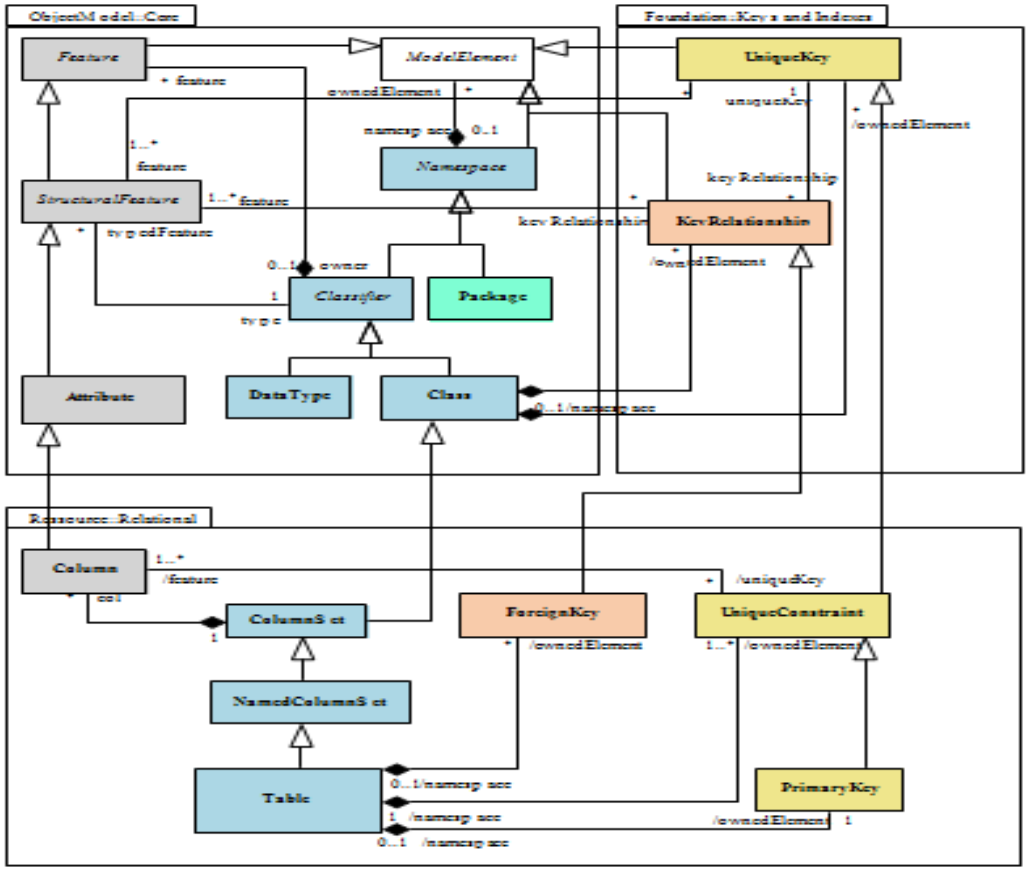

Fig. 2 CWM Metamodel

III. ATLAS TRANSFORMATION LANGUAGE (ATL)

A. Overview

ATL (Atlas Transformation Language) is a model transformation language and toolkit developed by the Atlas group (INRIA & LINA). In the field of Model-Driven Engineering (MDE), ATL provides ways to produce a set of target models from a set of source models [3]. It is developed on the Eclipse platform and particularly on its branch EMF (Eclipse Modeling Framework). It comes with many tools to facilitate its use. The formatting language keywords is provided in the ATL code editor, a debugger is provided, and a simple textual notation called km3 (for Kernel MetaModel) allows the specification of metamodels.

Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)

To achieve our original goal to automate the calculation of a relational schema CWM from UML class diagram, we use the transformation capabilities of ATL, which are also the most developed. However, this language also allows you to write simple queries on models, as provided in the Query pane QVT (for Query, View and Transformation). Moreover, it is clear that making a model transformation would be difficult without the ability to read (and therefore querying) the source model. So for transformations as read-only operations models, ATL extensively uses the writing and semantic keywords of OCL (for Object Constraint language) standard mode.

B. The Basic Operation of Transformation: the Rule ATL

A transformation program written in ATL consists of rules that specify how the elements of source model are recognized and driven to create and initialize the elements of the target model. These rules are of the general form (figure 3):

```
rule ForExample {
from
        i : InputMetaModel!InputElement
to
        o : OutputMetaModel!OutputElement(
        attributeA <- i.attributeB,
        attributeB <- i.attributeC + i.attributeD
        )
}
```

Fig. 3 An Example of Rule ATL

- *Forexample* is the name of the transformation rule.
- *i* (resp. *o*) is the name of the variable in the body of the rule will represent identified source (resp. Created the target element) element.
- *InputMetaModel* (resp. *OutputMetaModel*) is the metamodel to which the source model (resp. The target model) processing complies.
- *InputElement* means the metaclass elements of the source model to which this rule will apply.
- *OutputElement* means the metaclass from which the rule will instantiate the target items.
- The exclamation point is used to specify how a metaclass metamodel belongs in case disambiguation.
- *attributeA* and *attributeB* are attributes of the metaclass *OutputElement*
- Their value is initialized with values *i.attributeB*, *i.attributeC* and *i.attributeD* metaclass *InputElement* attributes.

In addition to the rules, the ATL language has the keyword "helper", which allows you to define macros outside the rules for factoring code portions commonly used. An ATL program, called a module, is essentially a grouping of rules and helpers. Outside the module itself, the fixed elements of the translation are both source and target metamodels. The source model can in turn be seen as the parameter of the transformation, and the target model outcome. It is through the configuration of the program execution the translation engine you specify concretely what files should look for meta environment, the source model, the program file and in which file expected that he writes the result model. The source and target models of the transformation must conform to their respective metamodels provided by the user. Note that an ATL module can also be represented as a model, because like any language ATL has its own metamodel. Finally, note ATL natively used as meta-meta version barely different from the MOF meta-metamodel defined by the EMF Ecore branch of the Eclipse platform. The following diagram (figure 4) illustrates these principles through the example of an ATL transformation named *Author2Person*, for transforming a model named *Author* in a model named *Person*.

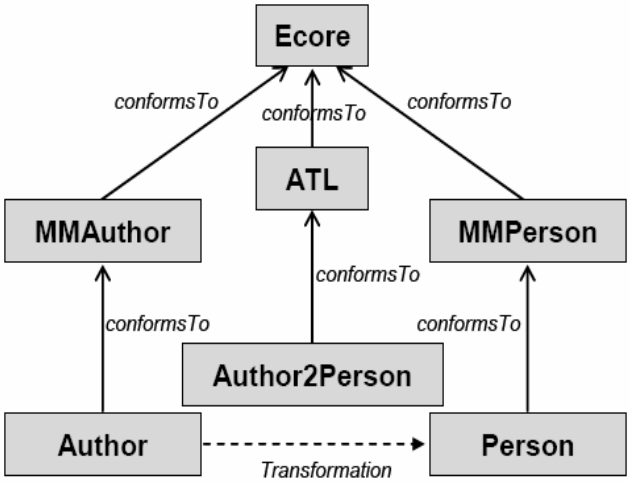

Fig. 4 ATL Transformation Author 2 Person

In practice, the models and their targets and take in the meta source Eclipse development environment as text files in XMI format with .ecore extension.

IV. METAMODEL AND MODEL SPECIFICATION

Consider our problem "Generate normalized relational model from UML class diagrams." To do this, we must provide the following information to the system in order to feed the engine ATL translation:

- The UML 2.0 metamodel
- The CWM metamodel
- The UML model (the domain model conforms to the UML 2.0 metamodel)
- The ATL transformation code

These would be possible to automatically generate

- The CWM model (the relational schema compliant metamodel CWM)

A. Metamodel Specification

The UML 2.0 metamodel and CWM are written in the language km3 in two separate text files:

- MDC.km3 (for Domain model) in which we specified two package, both part of the UML 2.0 metamodel that interested us, and a package where we grouped specific metaclass in our field.
- MLRnL.km3 (for Relational Logic Model) in which we extract the packages and metaclasses that interested us the CWM specification OMG.

Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)

Here (figure 5) is a sample file and specifically MD.km3 Package UML2.0.

```
package UML20 {
class Element {
}
abstract class NamedElement extends Element {
attribute name[0-1] : String;
}
abstract class Namespace extends NamedElement {
}
abstract class Classifier extends Namespace, Type {
reference generalization[*] container :
Generalization oppositeOf specific;
}
class Class extends Classifier {
reference ownedAttribute[*] container : Property
oppositeOf "class";
}
…
}
```

Fig. 5 Excerpt from a Definition File UML 2.0 Metamodel in km3 Format

From these two separate text files, obtained using an "injector" accessible through the interface of the platform eclipse the two corresponding XMI files (figure 6).

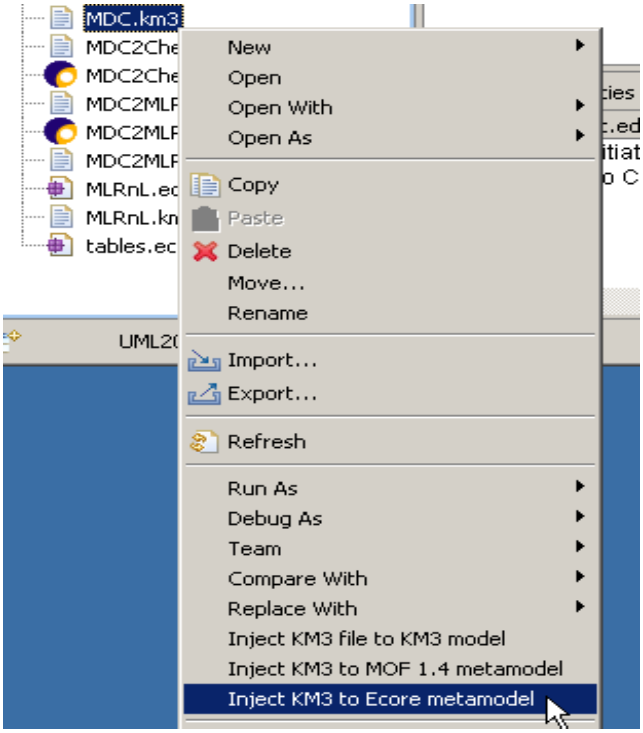

Fig. 6 "Injection" of km3 File in the XMI Format that Conform Ecore

In these XMI files, there are the classes originally defined in km3 in text translated in terms of metaclasses (since it is a metamodel) in XMI, so also in text, and consistent with the meta-metamodel Ecore (very close to the MOF). Here are the same metaclasses as in the previous example this time expressed in XMI format (figure 7).

```
<?xml version="1.0" encoding="ISO-
8859-1"?>
<xmi:XMI xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XML
Schema-instance"
xmlns:ecore="http://www.eclipse.org/e
mf/2002/Ecore">
<ecore:EPackage name="UML20">
<eClassifiers xsi:type="ecore:EClass"
name="Element"/>
<eClassifiers xsi:type="ecore:EClass"
name="NamedElement" abstract="true"
eSuperTypes="/0/Element">
<eStructuralFeatures
xsi:type="ecore:EAttribute"
name="name" ordered="false"
unique="false" eType="/2/String"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass"
name="Namespace" abstract="true"
eSuperTypes="/0/NamedElement"/>
<eClassifiers xsi:type="ecore:EClass"
name="Classifier" abstract="true"
eSuperTypes="/0/Namespace /0/Type">
<eStructuralFeatures
xsi:type="ecore:EReference"
name="generalization" ordered="false"
upperBound="-1"
eType="/0/Generalization"
containment="true"
eOpposite="/0/Generalization/specific
"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass"
name="Class"
eSuperTypes="/0/Classifier">
<eStructuralFeatures
xsi:type="ecore:EReference"
name="ownedAttribute" ordered="false"
upperBound="-1" eType="/0/Property"
containment="true"
eOpposite="/0/Property/class"/>
</eClassifiers>
…
</ecore:EPackage>
</xmi:XMI>
```

Fig. 7 Excerpt from a Definition file UML 2.0 Metamodel in XMI Format

Now the XMI format and as we have said Ecore metamodel conforming to meta included in the plugin eclipse EMF metamodel can be interpreted and displayed by the graphics editor developed as part of the same plugin (figure 8).

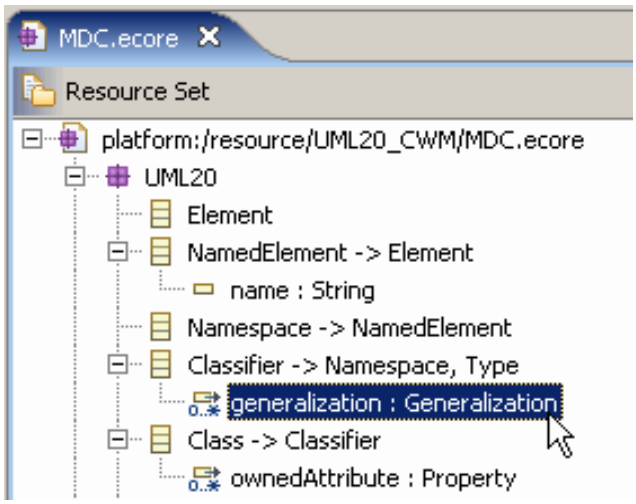

Fig. 8 The File MDC. Ecore Displayed by the EMF Editor

B. Model Specification

Published By:
Blue Eyes Intelligence Engineering and Sciences Publication (BEIESP)

Although many tools which can be difficult to implement are in principle available, writing a model is more tedious. We can no longer use the language km3 exclusively dedicated to the definition of meta. We must begin by writing directly to a text file XMI format specifying elements of the model with their opening and closing tags. In the following example (figure 9), we created a exemple.ecore file where we define a class type *Class*. The type Class is a reference to the metaclass of the same name in the metamodel UML2.0 itself referenced by the xmlns attribute of the tag xmi: XMI.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns="UML20">
                    <Class>
                      …
                    </Class>
</xmi:XMI>
```

Fig. 9 Example of a Class Definition in a Ecore File in XMI Format

As far as the XMI syntax is correct and consistent with the metamodel, the editor then makes access easier, mouse, attributes and associations items previously created (figure 10).

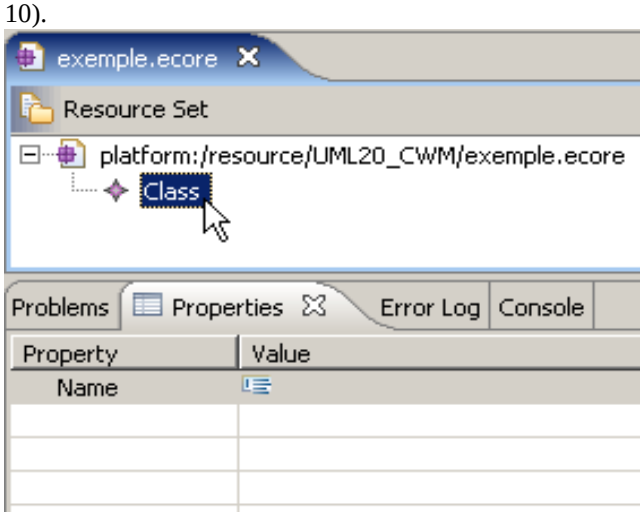

Fig. 10 A tab "Properties" Provides Access to the Properties of the Class Class

In addition to editing in a tab property values of the selected item, a menu accessible by right-clicking on the same element can in some cases cause the creation of new elements from the metamodel. For example, we see in our extract from the UML 2.0 metamodel that elements of type Class have between each other by their member *ownedAttribute* elements such *Property*. This relationship is an aggregation relationship, indicated in our original file km3 UML 2.0 metamodel with the keyword "container" (figure 11).

```
class Class extends Classifier {
reference ownedAttribute[*] container : Property
oppositeOf "class";
}
```

Fig. 11 A tab "Properties" Provides Access to the Properties of the Class Class

This particular quality of the relationship between Class and Property, allows to have the editor of the eclipse platform elements feature for automatically generating "child" of another. It is this function that we use in the example below (figure 12).

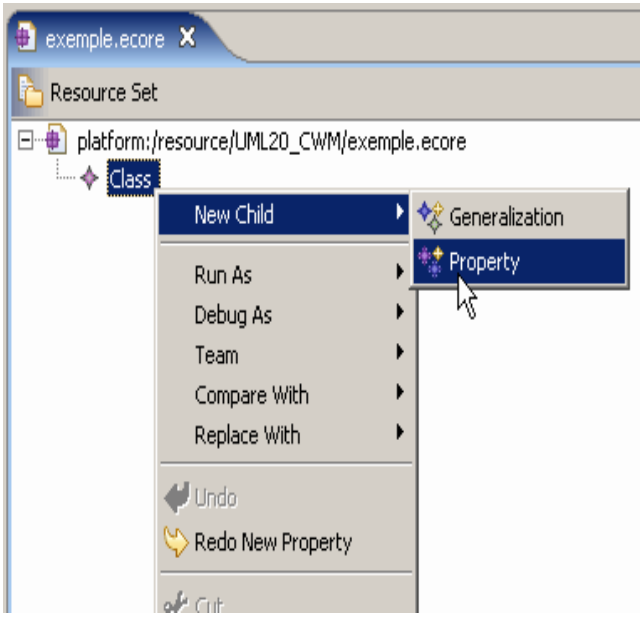

Fig. 12 Menu "New Child"

Once an item by another creates a variant of the "New Child" function is available, since it can create elements "sibling" of the same level in the hierarchy container/content. The following example shows how to create a second property in the same class, using this option.

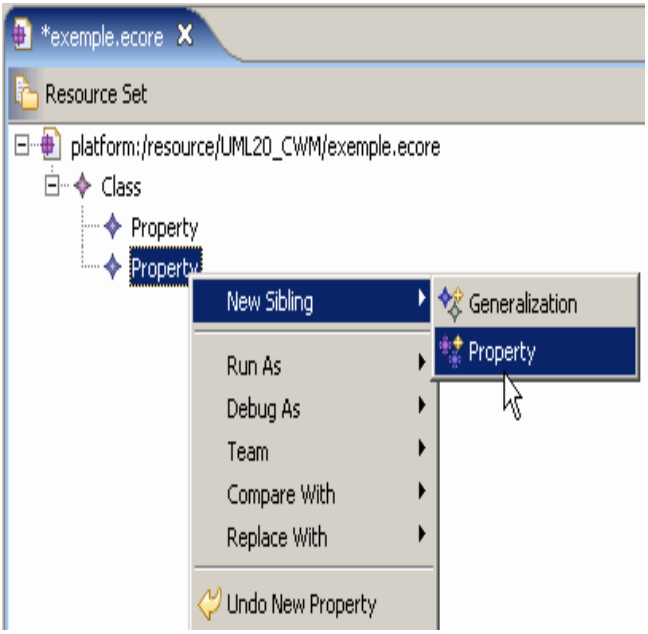

Fig. 13 Menu "New Sibling"

Using these mechanisms, and the *Properties* tab, it is thus relatively easily specify a *class*. However, experience shows that writing and complete models is fairly tedious.

C. The ATL Module

Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)

The writing of the transformation program itself does not pose any problems in practice. It simply boils down to creating a text file with the extension atl, where the transformation rules and helpers are written (figure 14).

Fig. 14 Excerpt from the Code Transformation of ATL Module

V. IMPLEMENTATION

The technical framework and concepts that are manipulated are set. We can discuss the details of the experiment we conducted there. The order in which the steps are described corresponds to the chronological order of the development of a model transformation in general. In practice however, as in our case in other research, this process is far from linear, and many feedbacks occur between phases of development.

A. Domain Model Used

This domain model (figure 15) shows the case where a user needs to record information relating to student placements. So we find in the domain model classes that should eventually allow the persistence of such information. The class name is explicit, it is seen as the designer tries to save the student who took the course, the teacher was eventually supervising the course, what proposal of any company comes from this course, and where and when is his defense.

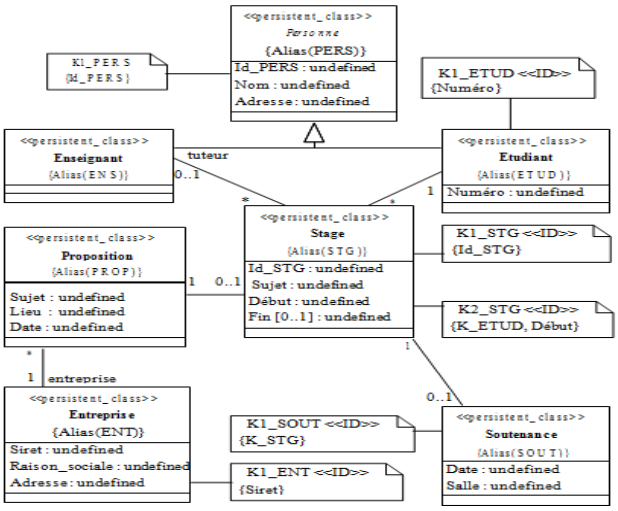

Fig. 15 Domain Model Used

As we said, the choice of UML language to express the domain model allows the designer to use his presupposed knowledge of this language to enrich its model. We see for example that cleverly factored some personal information for Teacher and Student classes in a common superclass Person, using the UML inheritance mechanism. We also see that the designer, combining classes with each other, specifies various cardinalities of these associations. These cardinalities are much sense, are included in the algorithm processing, and lead generation strategies different key in the final relational model. To facilitate the work of the designer when specifying the domain model, the authors of the algorithm have also enriched the vocabulary with two important UML stereotypes, stereotypes "*persistent_class*" and "*ID*".

- The stereotype "*persistent_class*" allows to explicitly specify which classes of its domain model it wants to see implemented in the form of tables in the relational model. This distinction between persistent and non-persistent classes allows him to have more freedom to create in the design of useful utility classes at that time but to sustain relevant information.
- The stereotype "*ID*" allows for him the designer to specify identifiers for classes. These identifiers and treatment are essential in the creation algorithm and the final resolution of the key relational model. They are the building blocks of the primary key and foreign key tables and references derived domain model.

While it may define one or more identifiers for table, the designer does not have to. The process then provides for their generation by the automatic system, to ensure that each table has at least one unique identifier.

B. Produce Metamodels

One of the first tasks to start programming a model transformation with ATL is the definition of the target metamodels and source. In theory, it precedes all other conditions since the writing of the source on the one hand and the other model of ATL code itself is expressed in terms of the rules of the elements handling the source and target metamodels.

1) Specify the target metamodel: the metamodel CWM.

As we have said, the purpose of the algorithm is to eventually lead to a model of relational schema. CWM language has this ability through its packages ObjectModel::Core (borrowed from UML 2.0) Foundation::Keys and Indexes and Resource::Relational (see Figure 2 CWM metamodel). We were able to rely exclusively on these three packages for our needs. We appropriate syntax and semantics described in the specification of the OMG to document data models we produce. In the eclipse platform, the concrete specification of the target metamodel was simply to rewrite the language km3 the contents of these 3 packages to inject into the platform as Ecore file figures 16 and 17).

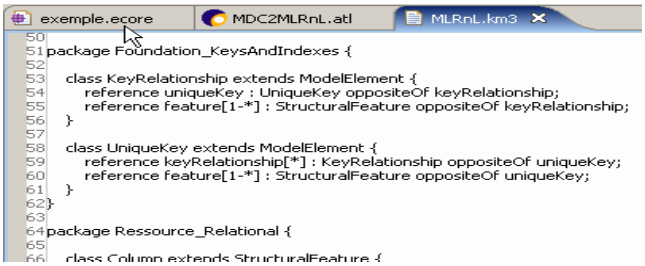

Fig. 16 Extract of the File MLRnL.km3 Published in the Eclipse Platform

Published By:
Blue Eyes Intelligence Engineering and Sciences Publication (BEIESP)

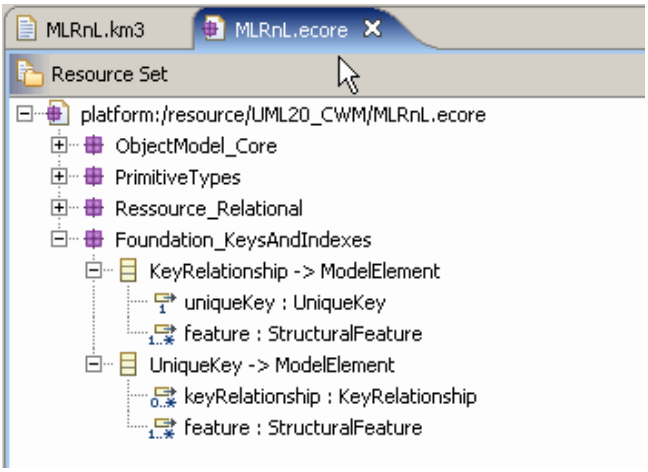

Fig. 17 "Injection" of km3 File Result

2) Result: metaMDC

A metamodel of domain model is built on the basis of existing metaclasses in the UML 2.0 standard. Ten specific classes are added to our problem in a new package with MDC inheritance and the following associations (figure 18):

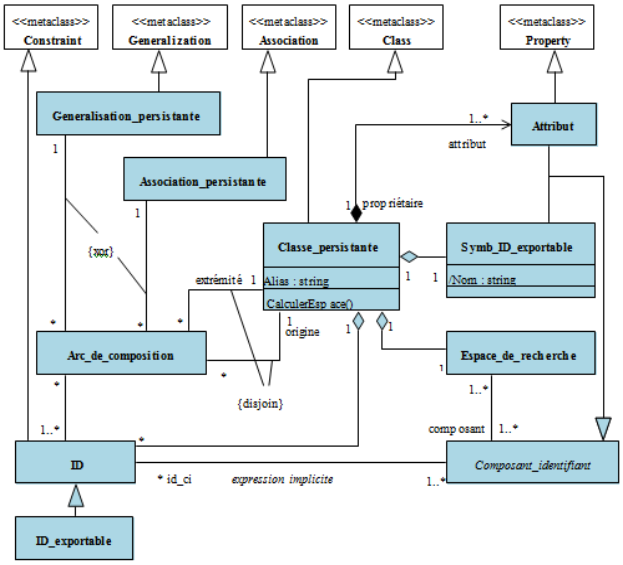

Fig. 18 Our Metamodel Domain Built on UML 2.0

The integration of this meta-model in the Eclipse environment is as previously seen through its rewriting km3. The MDC.km3 file contains the translation. Included are both UML20 package that includes selected from those of the UML 2.0 metamodel metaclasses and MDC containing her specialized metaclasses added by us. MDC.km3 The file is then injected into the XMI format conforming to Ecore metamodel of EMF eclipse plugin.

VI. CONCLUSION

The objective of this study was to determine the suitability of the use of language processing ATL model and the need to implement a process of translating a UML model into a CWM model. Through the experiment here, we could provide more answers. First, the ATL language is well adapted to a portion of the requirement (the actual transformation), the main effort of implementation is a clear and precise definition of the source and target meta-models used in the processing. Secondly, it was shown that the upstream and downstream operations of calculation could be made with the same language.

REFERENCES

1. RUMBAUGH, James, JACOBSON, Ivar, et BOOCH, Grady. Unified Modeling Language Reference Manual, The. Pearson Higher Education, 2004.
2. POOLE, John, CHANG, Dan, TOLBERT, Douglas, et al. Common warehouse metamodel developer's guide. John Wiley & Sons, 2003.
3. ATL Reference manuals: User Manual, Starter Guide, Installation Guide and the ATL Virtual Machine Specification. Available http://www.eclipse.org/gmt/atl/doc .

Published By:
Blue Eyes Intelligence Engineering and Sciences Publication (BEIESP)