First experiments with the ATL model transformation language: Transforming XSLT into XQuery[1]

Jean Bézivin(1), Grégoire Dupé(1), Frédéric Jouault(1) (2), Gilles Pitette(2), Jamal Eddine Rougui(1)

(1) Atlas Group, INRIA and IRIN
University of Nantes,
2, rue de la Houssinière - BP 92208
44322 Nantes Cedex 3, France

(2) TNI-Valiosys
120, rue René Descartes
Technopôle Brest Iroise - BP 70801
29608 Brest Cedex, France

Abstract

ATL (Atlas Transformation Language) has been defined to perform general transformations within the MDA framework (Model Driven Architecture) recently proposed by the OMG. We are currently learning from the first applications developed with this language. The example used here is a transformation from *XSLT* to *XQuery*. Since these are two standard notations that don't pertain to the MDA space, we first need to provide some justification about this work. The global organization of technological spaces presented at the beginning of the paper is intended to answer this first question. Furthermore we propose the original characterization of a technological space as a framework based on a given unique meta-model. After having briefly presented the ATL framework, we describe the *XSLT2XQuery* transformation. We may then draw several conclusions from this experiment, suggesting possible improvements to general model transformation frameworks. ATL is still evolving since it is supposed to match the forthcoming QVT/RFP recommendation when it is ready. As a consequence, the concrete expression of the transformation presented in this paper may change, but the general ideas should remain stable.

Keywords

MDA; XML; Model Transformation; MOF/QVT; Technological Spaces; XQuery; XSLT;

1. Introduction

Within the OMG MDA initiative, a request of proposal has recently been issued on model transformation support. The MOF/QVT RFP [13] aims at defining a domain-specific language (or a family of such languages) for querying, viewing and transforming models. As part of this effort, the ATLAS group at the University of Nantes and the TNI-Valiosys Company are proposing ATL (Atlas Transformation Language) [15]. The TNI/Valiosys Company will deliver an industrial-strength version of ATL while ATLAS is working on a free software release of the current transformation engine and of a library of transformation components. Several model transformation applications are currently being developed in this language. This paper presents one of these initial applications, not the most typical but with interesting properties: transforming XSLT [24] into Xquery [25]. Many characteristics of this problem are challenging and provide a significant test to evaluate the suitability and adaptability of ATL.

The natural application domain of ATL is to express MDA-style model transformations, based on explicit meta-model specifications [5]. For example we can transform a UML model into another UML model where all the public class attributes would be privatized and corresponding accessors added. Another example would be to transform a UML 1.5 model into an EJB 1.2 implementation. This area of research and development is currently very active since we can now envision large libraries of adaptable and composable transformation components that could be acquired or developed by IT departments of various companies in order to speed up and to improve the quality of their software development and maintenance process. The main ideas behind this evolution are the consideration of models as first class entities and of transformations

[1] This work was partially supported by the MOTOR /CARROLL common project between INRIA, CEA and THALES.

as models. One of the major goals of the MDA initiative [21] [3] is to apply these model transformation frameworks to the automatic generation of platform specific models (PSMs) from the pure business expression of platform independent models (PIMs). Proceeding in this way may give some hope to be able to cope with the rapidly changing technological platforms (CORBA, XML-SOAP, Java-EJB-J2EE, ECLIPSE, C#-DotNet-Phoenix, etc.). The corresponding software development and maintenance cycle would then be viewed as a chain of successive transformations.

Many efforts are presently addressing the problem of reusability of transformation components, since this is at the hearth of the MDA proposal. We participate in this general effort by building up our own library of ATL transformation components. The motivating example we have chosen to discuss here is however less conventional. Instead of using classical meta-models like UML, SPEM [16], CWM, Java, etc., we are going to look outside the strict boundaries of MDA model engineering to experiment on how ATL could be applied to outside technological spaces as well. After having introduced this notion of technological space, sometimes abbreviated TS in this paper, we shall present the target problem situated inside the XML document space: transforming XSLT documents into XQuery documents. Of course we know that this could be done inside the XML space, by applying either XSLT or XQuery. If we import the problem into the MDA technological space, this is mainly to compare both approaches. It is also to tackle a problem more difficult than simple UML model refactoring and to assess the qualities of our ATL language. Since our ATL engine has been working for some time now, this kind of exepriment allows us to gain practical exprerience and is much helpful to evolving the language and its environment

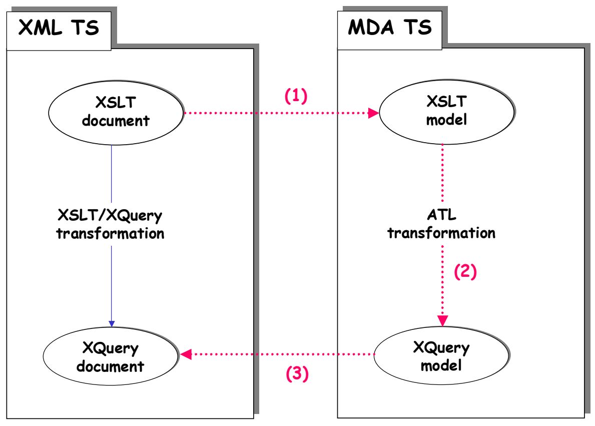


Figure 1 - Multiple ways to transform models

So the problem we are tackling is illustrated in Figure 1. We want to transform an XSLT document into a corresponding XQuery document. This should be a semantic-preserving transformation, since the XQuery program should perform similarly to the XSLT program on similar XML input files. If we were to solve completely this problem inside the XML technological space, we would probably use an XSLT transformation as suggested in the left part of Figure 1. This is not our intention here. What we propose to do is first to import the problem (i.e. the XSLT document) from the XML space to the MDA space, i.e. making this a true MDA model (1). Then we apply an ATL program to transform this model into another model corresponding to an XQuery document (2). Finally we export back the result from the MDA space to the XML space (3). We hope to learn several lessons from this work, possibly by defining a general approach that could be applied in other contexts as well.

After procedural refinement in the 70's and object composition in the 90's, model transformation seems now to follow as the dominant paradigm in software engineering. There are a lot of different approaches to model transformation and several classifications and comparisons are currently being offered [4]. The most important difficulty in these classifications is giving a precise scope to the domain of model transformation. Is XSLT a model transformation language? This brings us to answer the central question of what exactly is a model. One of the most general definitions has been given in [19]: "*A model is a formal description of a complex application artifact, such as a database schema, an application interface, a UML model, an ontology a message format*". This is a good starting point because it considers for example an XML document to be a model. It does not help much however to define a unified model transformation language or a family of such languages. Since the problem of interoperability between different transformation languages is fundamental, we need a more precise definition than the previous one. The solution we propose here is to consider that a UML model, an XML document or a Java program are all models, but models pertaining to different TSs. A UML model for example pertains to the MDA TS as standardized by OMG. Proceeding in this ways has several advantages. For example it allows considering uniformly what is usually called model-to-model, model to code or code-to-code transformations. If we manage to give a sufficiently precise and operational definition of a TS, it will then be possible to state precisely what is an internal transformation inside a given TS and what is an import-export conversion between two different TSs.
The presentation given in section 2 has the purpose of illustrating the practical importance of a precise definition of TSs. One of the original views offered is that each TS is built around the implicit or explicit assumption of a unique given meta-meta-model.
This paper expresses a view rather opposite to the one proposed in [17]. At that time, we explored the

possibility of exporting the model transformation problem from the MDA TS to the XML TS because there were no available transformation engines inside the MDA TS. Now, following the OMG MOF/QVT RFP, there are already some existing engines like the ATL one that are beginning to be available. As a consequence we may now seriously explore below the alternative approach of exporting a XML document problem for solution into the MDA technological space.

This paper is organized as follows. The notion of technological space is presented in Section 2. The ATL language is described in section 3. The motivating example of transforming an XSLT document into an XQuery document is discussed in detail in section 4. Section 5 gives the conclusions of this experiment.

2. What is a technological space?

2.1 The notion of technological space

We employ the concept of Technological Space [8] as the central concept in our analysis and comparison. A technological space is a working context with a set of associated concepts, body of knowledge, tools, required skills, and various other possibilities. It is composed of a representation system and a set of operators to access, update and process the information expressed in this common representation system. It is often associated to a given user community with shared know-how, educational support, common literature and even workshop and conference meetings. It is at the same time a zone of established expertise and ongoing research and a repository for abstract and concrete resources. Although it is difficult to give a precise definition of a Technological Space, some of them can be easily identified, for example: programming languages concrete and abstract syntax (Syntax TS), Ontology engineering (Ontology TS), XML-based languages and tools (XML TS), Data Base Management Systems (DBMS TS), Model-Driven Architecture (MDA TS) as defined by the OMG as a replacement of the previous Object Management Architecture (OMA) framework.

Figure 2 provides a global view of these five TSs. It shows that each space is defined according to a couple of basic concepts: Program/Grammar for the Syntax TS, Document/Schema for the XML TS, Model/Meta-Model for the MDA TS, Ontology/Top-Level Ontology for the Ontology TS and Data/Schema for the DBMS TS.

Each of the technologies presented in Figure 2 has basic properties and features, strong and weak points. To name some, the Syntax TS deals with executable systems; the Ontology engineering TS has some very precise definition tools like conceptual graphs and description logics, the MDA TS has received industrial agreement and backup on its Unified Modeling Language (UML) and Meta Object Facility (MOF) standard recommendations; the DBMS TS has a long record of dealing with huge volumes of structured data; the XML TS has also wide industrial acceptance in the field of semi-structured data representation; the Syntax TS is a very rich and stable technology that has been maturing for more than fifty years with formal tools (e.g. context-free or attribute grammars) and mapping these onto executable machines; the XML TS has some interesting and widely available tools such as XSLT transformation engines; the MDA TS has made available many industrial CASE tools like Rational ROSE, Argo UML, etc. supporting model creation and browsing.

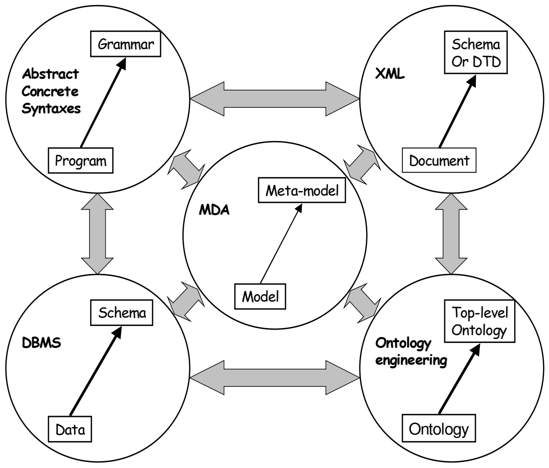


Figure 2 - Some technological spaces with some relations among them

Another idea, illustrated by Figure 2 is that no TS is an island. There are bridges among the spaces and these bridges also have particular properties. In Figure 2 we do not represent all the bridges between the various TSs and the figure does not suggest any superiority for any one of them.

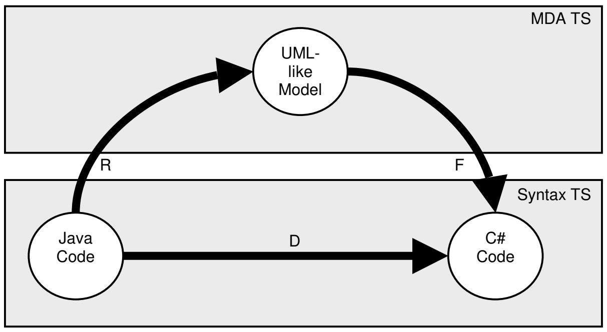


Figure 3 - Code migration

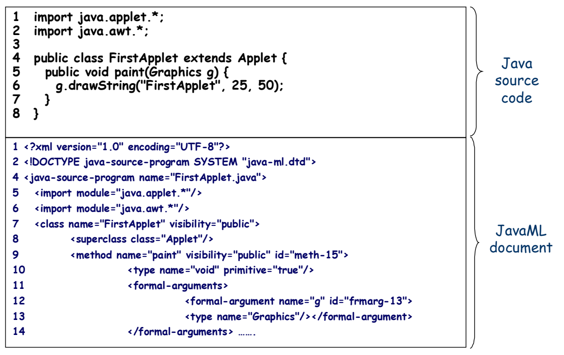
Bridges allow import-export of artifacts from one space to another. Some bridges may be bi-directional and some may be one-way bridges. Bridges are especially useful when a given operation may be performed more easily in one space and the result may then be imported into other space. For such an operation, software engineers need to compare the facility of achieving it in one space compared to the other one and also to evaluate the import/export facilities between the spaces. For illustration purposes, we take an example mentioned in [18] and illustrated in Figure 3. Recently an approach to migrate Java applications to C# applications was proposed as a set of tools by Microsoft (direct D transformation). The corresponding JUMP framework (Java User Migration Path) is entirely situated inside the Syntax TS. Alternatively, it is also possible to perform a reverse engineering operation from Java to a UML-like model (R operation), followed by a forward engineering operation to generate the C# program (F operation). According to the details of this operation (the precise pivot meta-model used), the direct D operation will be different from the combined R+F operation.

We have mentioned some TSs in the previous descriptions. The number of these spaces is obviously much more important and their structure and mutual relations are quite complex. The space of abstract and concrete syntax covers for example executable programming languages. As we know, there are many such different languages (imperative, declarative, procedural, functional, object-oriented, etc.). Each has its own properties that may accordingly be defined as sub-spaces.

Table 1- Strong and weak points of some TSs

	XML	MDA	Syntax TS	Ontologies
Executability	Poor	Poor	Excellent	Poor
Aspects	Good	Excellent	Poor	Fair
Formalization	Poor	Poor	Excellent	Fair
Specialization	Fair	Good	Poor	Fair
Modularity	Good	Good	Good	Poor
Traceability	Good	Fair	Poor	Excellent
Transformability	Excellent	Fair	Fair	Fair

Table 1 summarizes how some technologies do better on some problems than others. Once again, this is only for illustration of the proposed approach. Stating that the XML technology performs very efficiently on transformation (with the help of XSLT) and that programming language systems have much less to offer is a very abrupt simplification that should be completed.



```
import java.applet.*;
import java.awt.*;

public class FirstApplet extends Applet {
  public void paint(Graphics g) {
    g.drawString("FirstApplet", 25, 50);
  }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE java-source-program SYSTEM "java-ml.dtd">
<java-source-program name="FirstApplet.java">
  <import module="java.applet.*"/>
  <import module="java.awt.*"/>
  <class name="FirstApplet" visibility="public">
        <superclass class="Applet"/>
        <method name="paint" visibility="public" id="meth-15">
                  <type name="void" primitive="true"/>
                  <formal-arguments>
                            <formal-argument name="g" id="frmarg-13">
                            <type name="Graphics"/></formal-argument>
                  </formal-arguments> …….
```

Figure 4 - Java standard source code and JavaML representation

In order to be more precise about the distinction between TSs, let us look at another example. The same Java program may be represented in three different technological spaces: programming language syntax, XML document and MDA. The former two are external models and the latest is internal from the point of view of the MDA technological space. This is illustrated in Figure 4 and Figure 6. The standard source text of a given program, the XML document of the same program expressed in the JavaML [2] DTD for example and finally the model of the same Java program expressed in a given Java meta-model may be converted one from the other by bridges between these technological spaces. However, inside each technological space, there are tools to handle the specific representation. It would be a pity to rebuild all these tools inside the MDA space if they perform well in other spaces. It seems much better to propose well-engineered bridges between the different spaces.

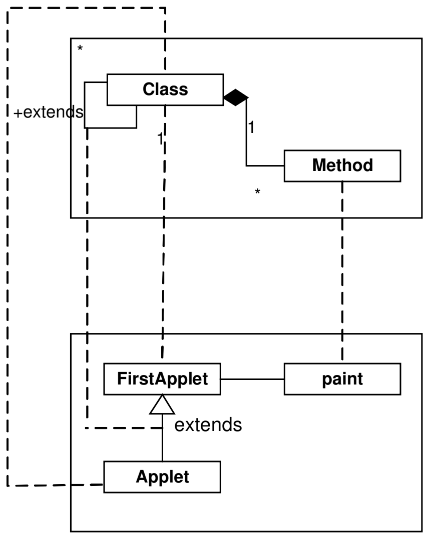


Figure 5 - Java meta-model and model representation

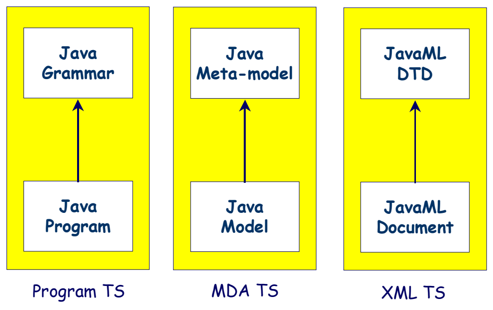


Figure 6 - Representing the same data in three different technological spaces

2.2 On the pertinence of TSs

The notion of TS allows us to deal more efficiently with the ever-increasing complexity of evolving technologies. There is no uniformly superior technology and each one has its strong and weak points. The "my technology is superior to yours" attitude is generally counter-productive.

Our privileged point of view is the MDA TS, where the basic concepts are MOF-compliant models and meta-models. The present work is however situated at the boundary between the MDA TS and the XML TS (documents and schemas). We wish to solve a problem of the XML TS inside the MDA TS. As previously mentioned reverse operation was evaluated in [17]. A MDA model, serialized in XMI [14], is exported from the MDA TS to the XML TS. Then this model is applied a XSLT transformation and the result is finally imported back as a model. XMI should thus be considered as part of the bridge between the MDA TS and the XML TS.

A careful consideration of various TSs shows that each one is based on the giving of an implicit or explicit meta-meta-model. The MDA TS, as defined by OMG, is based or the MOF which is aligned to the UML infrastructure. There are other different choices in the model management field like the sNets proposal [10] or the Vanilla meta-meta-model proposed in [19]. MOF, sNets and Vanilla are three examples of meta-meta-models that define similar TSs. They are all based on directed labeled graphs, but differently constrained. The MOF additionally defines a navigation and assertion language for these graphs called OCL. If we talk about trees and not graphs, the situation is similar. There are two well-known meta-meta-models constraining two families of trees. The first one is XML as presented later in Figure 12. XML trees are special trees[2] (well-formed i.e. respecting the meta-meta-model sketched in Figure 12). The second one is EBNF, which allows dealing with another family of trees (syntactical trees) for conventional programming languages. Of course in some case the programming language may have a special form allowing defining a specific meta-meta-model (e. g; Lisp, Prolog, etc.). We see then that each TS is rooted on a specific meta-meta-model.

As a final remark we should note how active is the field of transformation in the various TS. This paper deals with transformations in the MDA TS (MOF/QVT). The application field used here mentions two important transformation formalisms of the XML space (XSLT and XQuery), but there are many other concurrent approaches being currently defined for XML. Finally, in the programming language TS (e.g. Java), there is currently a very important activity going on on the same topic, for example: [6], [7], [22], [1], and many more.

[2] The description of the XML presented in Figure 12 is only minimal. We have defined an extended OCL-decorated XML meta-model. This is outside the scope of this paper.

3. Presentation of the ATL language

ATL is a model transformation language for the MDA corresponding to the ongoing QVT RFP ([13],[9], etc.). Its abstract syntax is specified as a MOF meta-model and a corresponding textual concrete syntax has been defined. An additional graphical concrete syntax also exists, which offers a way to represent a partial view of ATL declarative transformation rules.

This section presents the current version of ATL, which is still evovlving towards OMG QVT compliance and increased functionnalities.

After a brief overview of how queries, views and transformations are handled in ATL, its abstract syntax will be described. Textual and graphical concrete syntaxes will then be presented and some properties of the language will be analyzed.

3.1 Basic description of ATL

3.1.1 Queries

In ATL a query is an OCL expression, which can return primitive values (Boolean, String, Integer…), model elements, collections or tuples, or any combination of these (collections of tuples…).

The query can navigate across model elements and also call query operations on them. These operations can be defined either in the meta-model or along with the query. Adding such operations to model elements (using OCL constraints with the <<definition>> stereotype, as defined in [11]), to be used by a query, offers interesting possibilities: recursion can be used, model visitors can be written…

3.1.2 Views

Views are a special case of transformations. However, some properties of a transformation language can help to make views more useful:

- Support for incremental transformations makes it possible to update a view from its source without executing the whole transformation again.
- Bidirectionality can be used to define changeable views, which propagate their modifications to their source models.

The present version of ATL supports neither of these functionalities. However, we think a similar result (bidirectional updates) could be achieved using traceability information generated by the execution engine (see later).

3.1.3 Transformations

An ATL transformation model can transform a set of source models to a set of target models. The meta-models of every model (source or target) must be specified and be available for the transformation to be executed. ATL is actually able to handle any model defined using a MOF meta-model[3]. This includes meta-models and the meta-meta-model, due to the reflexivity of the latter. It is for instance possible to transform a UML model into a MOF meta-model, or to query all the constraints given in the MOF meta-meta-model. Navigation over models is specified using OCL.

3.2 Detailed description of abstract syntax

3.2.1 Navigation

Only navigation over fully initialized elements is allowed. A target element can only be definitively initialized at the end of the execution of the transformation. Therefore, navigation in ATL can only be done over source elements coming from source models and source or target meta-models.
If navigation over the target elements of a transformation model is necessary, it must be done in another transformation applied to the output of the first one.

3.2.2 Functions and operations

In OCL, operations can be defined on model elements. ATL reuses this possibility and allows a modeller to define operations on elements of source meta-models and on the transformation model itself.

3.2.3 Transformation rules

Different kinds of rules exist in ATL, based on the way they are called and on how they specify their result. The part of the ATL meta-model defining rules is given in Figure 7.

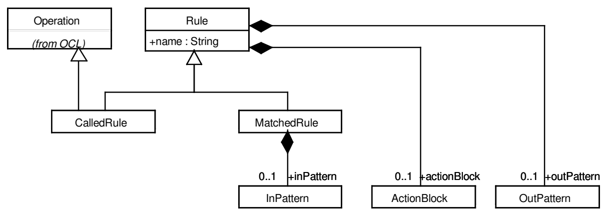


Figure 7 - Rules in ATL

A rule is either explicitly called using its name and with parameters (a *called* rule), or executed as a result of the recognition of an inPattern in the source models (a *matched* rule). The result of the execution of a rule can either be declared using an outPattern, implemented in an imperative section, or both.
A rule with an inPattern and an outPattern is called a **declarative** rule (whithout any imperative section, it is a **fully declarative** rule). A rule with a name, formal parameters, an imperative section and without any outPattern is called a **procedure**. Other combinations are simply called **hybrid** rules.
This makes ATL a hybrid language. In [4], such a language is mainly described as using a declarative approach to select rules, which imperatively specify how the work is to be done. ATL has that kind of rules, but fully declarative or fully imperative rules can also be defined.

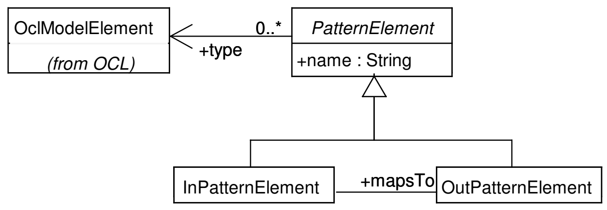


Figure 8 - Pattern elements

3.2.3.1 Source pattern

An ATL *matched* rule specifies a source pattern (or inPattern, or input pattern) as a set of types coming from source meta-models, associated to variable names and optionally filtered using an OCL Boolean expression. This filter, accesses the in elements through their variable names and returns true when a particular set is accepted by the rule. A source pattern is therefore a set of nodes from the source models, which have a specific relation, checked by the filter. A corresponding view of the ATL meta-model is given at Figure 9, pattern elements being defined at Figure 8.

[3] This includes meta-models and the meta-meta-model, due to the reflexivity of the latter.

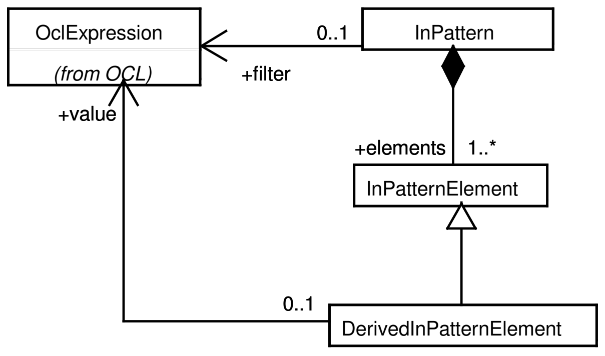


Figure 9 - Source patterns

A special case arises when all the elements recognized by the inPattern of a rule are in the same model. As a matter of fact, the subgraph formed by these elements can be and is often connected, which means that any information retrieved from one of the source elements by navigation could be obtained by navigating from any other. It seems to mean that in these cases, the additional elements are not necessary and that only one could be kept. However, several points must be considered :

- Some navigation expressions might be expressed more concisely from one node than from the others.
- Since the transformation engine must analyze every possible pattern and apply the discriminator on it, the computational gain resulting from the shortening of some navigation expressions might be lost.
- If a rule creates several elements from a single one, every generated element will be associated to it. It is sometimes necessary, however, that a part of the target elements are associated to other specific source elements.

A solution provided in ATL is the possibility to declare source elements as derived from others. In this way, if two elements are connected, one of them is computed from the other by navigation but can still be associated to a specific target element.

3.2.3.2 Target pattern

An target pattern (or outPattern, or output pattern), as specified on Figure 10, is a set of types coming from target meta-models associated to variable names and bindings. When the rule containing the target pattern is executed (either for a *called* or for a *matched* rule), the target elements of the specified types are created. A binding specifies the value used to initialize a specified property of an instance. A target pattern is consequently a set of nodes, which can be linked together by the bindings.

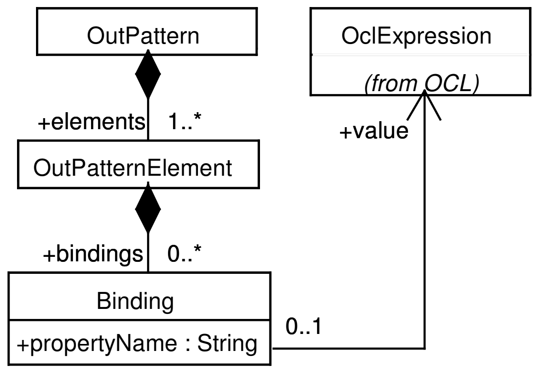


Figure 10 - Target patterns

Edges of target models, viewed as graphs, are created by bindings and can be set in a rule or between rules, using the target-from-source resolution algorithm described later as part of execution semantics in 3.2.4.

3.2.3.3 Imperative block

The imperative block of an ATL rule specifies a sequence of instructions that are to be executed after the application of the outPattern (if present). The language used is designed to be compatible, through an appropriate transformation, with Action Semantics.
As the latter is already standardized and there is a direct mapping between ATL abstract and concrete syntaxes for imperative instructions, these are not described here but only in the presentation of the concrete syntax in 3.5.2.

3.2.3.4 Rule inheritance

A rule can extend another one. The new rule can specify additional elements or restrictions on its source pattern. A restriction can be in the form of a new filter (which will be logically anded with the old one) or of the redefinition of a source element with a type extending the old one. Additional target elements or bindings can also be specified.

3.2.3.5 Abstract rules

An abstract rule is a rule that cannot be executed as such but must be extended to be useful.

3.2.4 Execution semantics

Executing an ATL transformation model requires several steps. If the model is given in textual format, it is first parsed and transformed to a model defined using the ATL meta-model. This model is then statically checked for semantic errors against the ATL meta-model and source and target meta-models. The next step can either be an interpretation or a compilation followed by an execution. In both cases, the application of an ATL transformation follows the semantics described in this section.

If there is a *called* rule marked as *entrypoint* (only one is allowed) it is executed first. This rule can call any number of *called* rules until it reaches its end.

Then, the *matched* rules are executed. In a first time, the output elements are instantiated. For each rule, every combination of elements matching the types of its inPattern is tried and checked against its filter. It the latter returns `true`, a pattern has just been recognized and the rule is matched for a specific set of elements.
Each time a declarative rule is matched, its target elements are instantiated. This is simply done by instantiating each element of its outPattern. A run-time link between the rule, the recognized and newly generated elements is created. This link associates one output element to each input element, which becomes the default element for implicit target-from-source resolution. A given source element cannot participate to more than one inPattern, otherwise a runtime error occurs. As a matter of fact, this cannot be statically detected in the general case; some filters would have to be checked for not being simultaneously `true`.
In a second time, the bindings are applied to initialize every output element. Depending on the type and multiplicity of the property to be set, different actions can be done:

- If the type is primitive (String, Integer, Boolean, Double) and the multiplicity has an upper bound of 1, the result of the evaluation of the right part is simply used to set the property (which is an Attribute).
- If the type is complex (a Class of the meta-model) and the multiplicity's upper bound equals 1, the right operand of the binding must evaluate to a model element of one of the source models (navigation over target elements being prohibited). The value, which will be used to set the property, cannot be this model element, which is not in the same model as the property's owner. However, at this point, a link might exist between this source element and some target elements, if it belongs to a matched subgraph of a rule (the same or another). If it does not, there is an execution error because the binding cannot properly initialize the property. If it does, however, the default element, associated to this source element by the run-time link, is used to set the property. If another element is explicitly specified, it is used in place of the default one. This is the target-from-source resolution algorithm.
- If the upper bound of the multiplicity is greater than 1, there are two possibilities:
  - The right operand of the binding evaluates to a single element (primitive or complex), which type matches the type of the property (otherwise, it would have been statically plotted as a semantic error). This element is added to the collection of elements of the property.
  - The right operand evaluates to an OCL Collection of elements (primitive or complex), which types match the type of the property. The size must match the multiplicity, or there is an execution warning (an imperative section can correct this). Every element is added to the collection of elements of the property.

In a third time, the imperative blocks of the *matched* rules are executed.

Eventually and provided it exists, the *called* rule marked as *endpoint* is executed.

Note: *called* rule containing an outPattern are not executed as *matched* rules. The target elements are simply created and initialized before the execution of the imperative block, but are not automatically linked to any source element.

3.3 Reflection

During the execution of a transformation, source models, source and target meta-models are navigable. The ATL meta-model and the presently executed transformation model itself are also navigable.
Rules can be explored from their names and the whole model can be accessed by its name.

3.4 Traceability in ATL

Traceability is achieved in ATL by having the transformation engine storing runtime information on the transformation in a model based on a specific meta-model.
Rather than defining such a meta-model and fixing traceability model generation rules, we think it would be preferable to define a customizable mechanism using reflection.
The specification of this mechanism is out of the scope of this paper. It will be defined in an ulterior document.

3.5 Textual concrete syntax

3.5.1 Declarative constructions

A concrete syntax has been defined and mapped to the ATL meta-model in order to make it possible to textually express transformation models.
Type names are prefixed by their meta-model names to prevent name collision in case of multiple source meta-models. For instance, with UML as a source meta-model, the following source pattern applies the rule to each pair composed of a Class and one of its Attribute:

```
  from
    c : UML!Class,
    a : UML!Attribute
    (a.owner = c)
```

Two nodes, a Class and one of its Attribute, are recognized by this pattern. A performance issue appears here: this inPattern forces the execution engine to check every pair of class and attribute against the filter. Declaring the class as derived would solve this problem:

```
  from
    a : UML!Attribute,
    derived c : UML!Class = a.owner
```

Note that deriving the attribute from the class would not be equivalent.

An outPattern is defined in a similar way. The operator used in bindings is <-. Here is how the instantiation of an UML!Class and of one of its UML!Attribute is declared:

```
  to
    c : UML!Class,
    a : UML!Attribute (owner <- c)
```

Declarative rules are defined by associating a target pattern to a source pattern. For instance, here is how a simple mapping from UML to a relational database meta-model (RDBMS) could be defined:

```
rule Class2Table {
  from class : UML!Class
  to
    table : RDBMS!Table
      mapsTo class (
      name <- c.name
    ),
    pk : SimpleRDBMS!Key (
      name <- class.name,
      owner <- class,
      column <-
        class.attribute->select(e|
          e.kind = 'primary'
        )
    )
}

rule Attribute2Column {
  from attr : UML!Attribute
  to
    col : RDBMS!Column mapsTo attr (
      name <- a.name,
      owner <- a.owner
    )
}
```

The source element to which a target element is (optionally) associated is specified using the `mapsTo` keyword.

In the case of a target-from-source resolution, which does not use the default target for a specific source, the target element variable name of the rule must be given. For instance:

```
rule Association2ForeignKey {
  from asso : UML!Association
  to
    fk : RDBMS!ForeignKey
      mapsTo asso (
      refersTo <-
[Class2Table.pk] ia.destination,
      owner <- ia.source
    )
}
```

A runtime error occurs when a source element has not been transformed by the given rule.
A problem arises when several rules can have generated the specific target element from the source one. For instance, if the source element is a Classifier, the target could have been created either by a Class2Table or by an Inteface2Table rule. In this case, both rules must inherit from a single rule declaring an abstract output element.

3.5.2 Imperative instructions

The optional imperative block of a rule is composed of a sequence of instructions, which are to be executed in the given order. Several kinds of instructions exist and are presented thereafter.

3.5.2.1 Expressions

Expressions are written in OCL and can be used as instructions. This would be useless in case of simple query expressions, but the call to non-query operations, such as an imperative rule, is allowed.

3.5.2.2 Variables

Variable declaration use an adapted OCL syntax:

```
let varName : varType = initialValue;
```

Variables are therefore typed and must be defined before first use.

3.5.2.3 Assignment

The binding operator <- is reused with the same semantics (including automatic target-from-source resolution) as an assignment operator. The simple assignment operator := can be used when automatic resolution is not required. Its left operand simply takes the value of its right one.
An assignment is an instruction but not an expression. The following instruction is therefore illegal:

```
myFunction(myVar <- 'a string');
```

3.5.2.4 Instances handling

Whereas instances are automatically created in declarative rules, it is possible to explicitly create or delete an object in an imperative block.
Instance creation uses the `new` operator:

```
let myClass : UML!Class =
      new UML!Class();
```

Removal of an instance is performed by the `delete` operator:

```
delete myClass;
delete myClass.contents->select(
      e|e.oclIsTypeOf(UML!Attribute)
 );
```

The parameter of `delete` must be an element of a target model or a collection of such elements, in which case every element of the collection is deleted. Note that the collection itself cannot be explicitly deleted since it is only a runtime instance, which will not persist in any target model. Such instances are often created in OCL (such as in certain iterators), which assumes the existence of a mechanism such as a garbage collector.

3.5.2.5 Conditional statements

OCL already provides an if-then-else construction. It is however an expression and could be compared to the ternary operator (`condition ? if-true : if-false`) of languages such as C and Java.
Yet, we need a conditional instruction close in meaning to the if statement in C or Java. The if statement in ATL has the same syntax than in these languages:

```
if(condition1) {
  -- if condition1
} else if(condition2) {
  -- if (not condition1)
  -- and condition2
} else {
  -- if (not condition1)
  -- and (not condition2)
}
```

A switch statement is also defined, with broader semantics (in comparison to its C or Java equivalent): the expression on which the condition is tested can be non-scalar. The first `case` with an expression having the same result has the first one will have its instruction block executed.

```
switch(expression) {
  case expression1:
  -- if (expression = expression1)
    break;
  case expression2:
  -- if (expression <> expression1)
  -- and (expression = expression2)
    break;
  default:
  -- if (expression <> expression1)
  -- and (expression <> expression2)
    break;
}
```

3.5.2.6 Loop statements

`while` and `do while` loop statements are defined using the same syntax found in C or Java:

```
while(condition) {
-- while condition is true
}
```

and

```
do {
-- executed at least once and while
-- condition is true after that
} while(condition);
```

A loop statement iterating over the elements of a collection is also defined:

```
foreach element in collection {

}
```

3.6 Graphical representation

ATL transformation models can be partially represented using a graphical concrete syntax. Not every language construction has a graphical counterpart, only main ones. The primary idea behind the existence of this syntax is that patterns of declarative rules are often best apprehended when one can actually see which elements of a meta-model are involved.
However, the best way to represent OCL expressions seems to be the standardized concrete syntax. Consequently, neither filters of source patterns nor bindings of target patterns can be drawn.
Figure 11 shows how the first of the two rules presented in the previous section would be drawn. It is a graphical representation of the types of the elements composing the pattern recognized by the rule (a Class here) as well as the types of the target elements (a Table here) and the link between source and target elements. The names of the variables holding the references to these elements at runtime are also shown inside the pins of the rule connected to the meta-model elements. It is immediately obvious here that a Class will be transform into a Table.
This syntax can be used to help understanding a particular transformation, or even to assist its developer who can then have a more global view of what he/she

is modelling. Besides, a modelling tool could provide a GUI allowing a hybrid development alternating graphical rule overview inner mechanism specification.

3.7 Some properties of the language

3.7.1 Directionality

In ATL, a transformation is unidirectional. We believe a language designed so that every transformation model can be applied in both directions would have limitations.
However, when a transformation model contains only fully declarative rules, it should be possible to derive a part of the symmetric transformation automatically, depending on the complexity of the expressions used in the bindings.
For instance, in a simple one-to-one mapping, the rule:

```
rule Class2Class {
  from mc : MOF!Class
  to
    uc : UML!Class mapsTo mc (
      uc.name <- mc.name
  )
}
```

could be automatically derived into:

```
rule Class2Class {
  from uc : UML!Class
  to
    mc : MOF!Class mapsTo uc (
      mc.name <- uc.name
  )
}
```

The conditions on a transformation model allowing an automatic symmetrical transformation derivation are yet to be defined. Once they are, a set of OCL constraints on ATL transformation models could be specified to test whether a particular model is reversible.

3.7.2 In-place transformations

The target model is always a new one in ATL, but a special kind of ATL transformations has been given a very close semantics to in-place transformations:
The source model is first copied to the target model and then transformation rules are applied. Such a transformation model can be thought as having a set of implicit rules copying all elements (meta-model dependant), and a set of explicit rules written by the modeller.

3.7.3 Incremental transformations

ATL offers no direct support for incremental transformations. However, we think some cases needing such transformations could be implemented using traceability. This is an area of ongoing resarch.

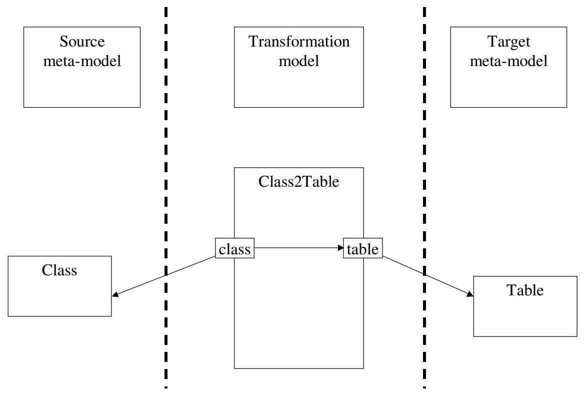


Figure 11 - Example of ATL graphical syntax

4. Transforming XSLT into XQuery

For the first experiments of ATL, we chose a transformation between two languages of transformation in the XML technological space. The goal of this experiment is to prove the feasibility of the XSLT2XQuery transformation in the MDA technological space with ATL. In practice, we will require import/export facilities and a composition of transformations. The presentation of the XSLT2XQuery transformation begins by presenting the different meta-models used in the transformation. The second sub-section presents the transformation process. Finally the last section describes the XSLT2XQuery transformation in the MDA technologic space.

4.1 Meta-models presentation

4.1.1 The XML meta-model

The XML meta-model is part of the bridge between the XML TS and the MDA TS. We chose to use a meta-model similar to the one proposed by NetBeans [23]. The XML meta-model presented on Figure 12 describes an XML document (*Document*) composed of one root node (*RootNode*). *Node* is an abstract class having two direct children : *ElementNode* and *AttributeNode*. *ElementNode* represents the tags, for example a tag named *xml*: `<xml></xml>`. *ElementNodes* can be composed of many *Nodes*. *AttributeNode* represents attributes, which can be found in a tag, for example the *attr* attribute: `<xml attr="value of attr"/>`. *ElementNode* has two sub classes : *RootNode* and *TextNode*. The *TextNode* is a particular node, which does not look like a tag; this is only a string of characters.

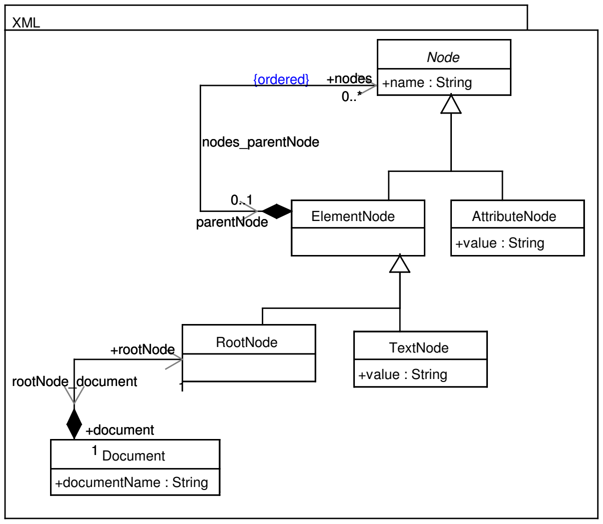


Figure 12 - XML meta-model

4.1.2 The XSLT meta-model

The XSLT meta-model that we wrote to perform the transformation is an extension of the XML meta-model (cf. Figure 12). The extension consists of classes represented in grey on Figure 13. The main class is called *XSLTNode* inheriting from *ElementNode*. The *XSLTNode* class has sub classes representing XSLT elements: *xsl:apply-templates, xsl:template, xsl:if, xsl:value-of*. To keep the explanation simple, we ignore several features like *xsl:for-each, xsl:choose, xsl:sort, xsl:copy-of* elements; thit is why they are neither in the meta-model nor in the transformation code.

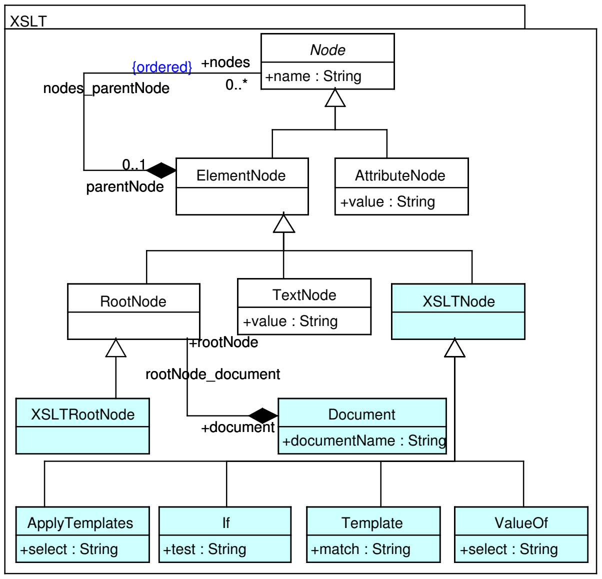


Figure 13 - XSLT meta-model

The *name* attributes of XSLT classes are set with the name of the tag corresponding to the owner class. The XML attributes of XSLT tags, for example *select* in an *xsl:value-of* tag, are represented by UML class attributes having the same name and typed as *String*.

4.1.3 The XQuery meta-model

An *XQueryProgram* is composed of *ExecutableExpression* which can be FLWOR expressions, function calls (*FunctionCall*) and function declarations (*FunctionDeclaration*). The main class is *FLWOR*, it represents *FLWOR* expressions which are composed of *For, Let, Where, Order by* and *Return* statements. *For* is composed of an XPath expression representing the value stored by the variable defined by the *var* attribute. *Let* is also composed of an XPath expression representing the value stored by the variable defined by the *var* attribute. *Where* is composed of a boolean XPath expression used to do a selection on the variables of the *For* statements. *OrderBy* is composed of an XPath expression defining how to order the output. *Return* is composed of expressions representing the output data. Those expressions are *ExecutableExpressions*, XML *Nodes*, *ReturnXPath* expressions. The *Node* class and its sub classes are copied from the XML meta-model. We choose to use two different XPath classes, because the expressions used in the *return* part are between braces in the textual format of XQuery.

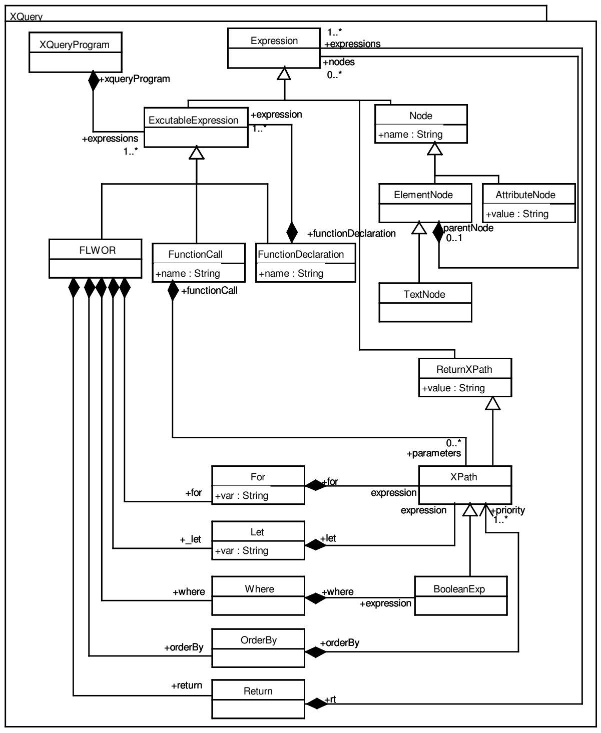


Figure 14 - XQuery meta-model

4.2 XSLT2XQuery transformation

In this sub section we define each step in the process of the XSLT2XQuery transformation. We therefore offer a process framework and the meta-model defining the representation of mappings with associated semantics.

4.2.1 The transformation process

This section summarizes the different steps of the transformation process illustrated in Figure 15 and Figure 16. The XSLT2XQuery transformation is carried out on instances of XSLT meta-model, and produces a corresponding model based on the XQuery meta-model (cf. Figure 14).

Before to tackle the XSLT2XQuery transformation in the MDA technological space, it is necessary to perform two first steps to get a usable XQuery model.

The first step (1) consists in bringing back the context of work in the MDA technological space by importing an XSLT document to a model based on the XML meta-model. An ATL importer able to import XML document into the MDA TS is used to perform this import. Indeed an XSLT is an XML document. We then have a first form of the XSLT in the MDA TS.

The second step (2) produces the wanted form of the XSLT document. This form is a model based on the XSLT meta-model. The second step is a transformation called XML2XSLT written in ATL. XML2XSLT consists of mapping each source model based on XML meta-model into a target model based on XSLT meta-model (cf. Figure 15). Firstly, it starts by copying the XML elements, which are not XSLT tags. Secondly, the transformation seeks and extracts the *ElementNode* instances, which correspond to the XSLT nodes. Those XSLT elements will be transformed into XSLT instances (in grey on the meta-model of Figure 14.

The third step (3) is the key transformation: XSLT2XQuery. This transformation consists of the definition of relationship between expressions over the two models, such as respecting the semantics and the functionality of their formalisms. XSLT2XQuery is detailed in the next subsection.

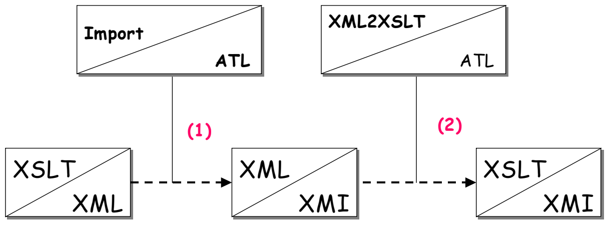


Figure 15 - Steps (1) and (2) of the transformation process

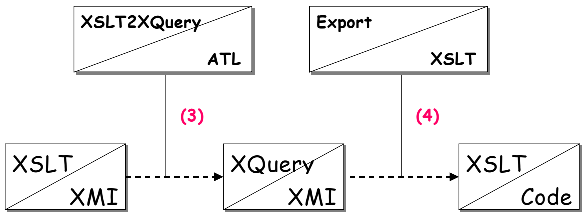


Figure 16 - Steps (3) and (4) of the transformation process

The last step (4) exports the model resulting from the XSLT2XQuery transformation into an XQuery document (in the XML TS). In our tests we used XSLT to perform the export. This XSLT transforms the XMI serializations of the XQuery models into XQuery expressions. Such transformations are really easy to write when the meta-model is a good representation of the grammar of the output.

The ATL framework automatically generates the XMI serialization of each model present in the transformation process. XMI is the format chosen to store models read and written by transformation. This is why XSLT can be used to perform an export. In fact the export to the XML TS is done at the end of each transformation by serializing models in XMI documents. The last step of the XSLT2XQuery transformation process is then a transformation in the XML TS.

In the ATL framework, it is now possible to write exporters to XML and textual technological spaces. Those exporters are particular ATL transformations composed of OCL expressions describing the way to get the representation of the exported model. In this case the exportation is really placed in the MDA space: the XMI files, which store the exported models, are loaded in the ATL engine.

4.2.2 The XSLT2XQuery transformation

To have an easily understandable transformation we simplified a bit the input XSLT documents by adding constraints.

The first one is that all the *template* tags must be direct children of the root node. This constraint simplifies the behavior of templates.

The second constraint is that the value of a *select* attribute of an *apply-template* must only be a tag name (it can not be an XPath expression). This constraint hides the main difference between a template and a function call. An *apply-templates* tag applies all available templates to a set of elements and each template treats only the elements that it is dedicated to. Whereas a function call applies a function to a set of elements; the test of the type of the elements must be explicitly described in the function declaration. The second constraint is useful to avoid describing this test.

The third constraint forces the XSLT programmer to write a template matching to '/'. To write this template will force the XSLT programmer to explain how the transformation must start. This information is necessary to the XQuery program, because XQuery is partly an imperative language: it describes the order of the program execution.

If we would not respect the previous constraints the transformation would also be writeable. We would just have to do some more tests. For example to remove the second constraint, we would have to generate XQuery code that defines which function to call. The choice will be done in function of the node names.

The transformation can be divided in three types of rules:

- the rule used to create the XQuery expression container which is an *XQueryProgram* instance (the first rule),
- the rules used to copy XML elements (the two last rules),
- the rules used to transform *xsl* elements into XQuery expressions (the other rules).

The source code of the transformation is in the appendix of this document.

The first rule (cf. lines 4 to 30) creates five kinds of elements and one instance for each kind: *XQueryProgram*, *FLWOR*, *For*, *XPath* and *Return*. Each of those instances is described by one output pattern (respectively at lines: 7, 10, 16, 21, 25). The first output pattern creates only an *XQueryProgram* instance. The second describes the *FLWOR* instance. This output pattern owns three ATL bindings. The first one explains that the FLWOR instance is an expression of the previously created *XQueryProgram*. The second and third output pattern specifies that the *for* property and the *return* property will be set with the instances generated by the output pattern creating respectively *For* and *Return* instances. As you can see on line 18, the *For* has its *expression* property connected to an *XPath* instance. This XPath expression defines the sub nodes of the root one of the transformed XML document. This expression is used to specify that the transformation begin at the root of the documents. This start point is the equivalent of the template matching to '/'. The *Return* output pattern owns a binding describing that the *expression* property correspond to the output elements created by the transformation of sub nodes of the *XSLTRootNode*. (The *_XSLT* variable represents the *XSLTRootNode*.)

The following parts describe the XQuery equivalent expression of each *xsl* tag. The two first rules (cf. lines 32 to 76) are the more important. They describe how template mechanisms can be converted into XQuery expressions. To be brief, we can just say that *xsl:template* tags are converted into function declarations and *xsl:apply-templates* tags into function calls.

We transform *xsl:template* tags into function declarations because a template is a set of ordered instructions called from different parts of the program. This definition is close to the definition of a function. To imitate the mechanism of the template, the created functions have a parameter, which describes the set of elements on which the template performs.

The *xsl: apply-templates* tags are converted into function calls. An *xsl: apply-templates* owns an attribute describing the set of the elements on which the template will apply. This information is stored in the function parameter.

The rule starting at the line 32 describes the elements, which are generated by the *xsl:template* transformation. The main elements are instances of *FunctionDeclaration*. The function name is the *match* attribute value of the *xsl:template* prefixed with `fct`. The *expression* property of *FunctionDeclaration* instance refers to a *FLWOR* expression described in the second output pattern. At the line 39, the right part of the binding is a sequence of *FLWOR* elements from the *_Template* rule. The specification of the sequence is necessary because *_Template* represent an only element and the meta-model specify that the *expression* property is a set of elements. The lines 40 and 43 describe the fact that the *FunctionDeclaration* instances are owned by the instance created by the transformation of *XSLTRootNode* instance. The right part of the binding describes *XSLTRootNode* instance with an OCL expression getting the first element of the set of all *XSLTRootNode* instances. We choose the first element of the set because in an XSLT document there is only one root node.

The rule on lines 64 to 71 describes the creation of a function call: the name of the function is the value of the *select* attribute (of the *apply-templates* node) prefixed with `fct` (thanks to the second constraint previously presented at the beginning of the sub section) and the *parameters* property is bind to an *XPath* instance, which is set with the value of the *select* attribute prefixed with `$var/`. This is necessary because the current node is not implicit in XQuery that is why we need to use a variable. The *$var* variable is used in every FLWOR expression to simulate the current node. It and its value is defined in the *For* (or in the *Let*) statement of each FLWOR expression.

The rule on lines 78 to 108 describes the transformation of *xsl:if* tags into *FLWOR* expressions. The *where* statement is used to do the *if* work: the *test* expression of the *xsl:if* is copied in the XPath expression connected to the *Where* instance. The `$var/` prefix is added to the XPath expression to simulate the current node as explained previously. The connection to the parent elements is done in parent creation rule.

The rule starting at line 110 describes the transformation of *xsl:value-of* tags into *ReturnXPath* expressions. This rule consists of a copy of string expressions and an addition of the `$var/` prefix. This is possible because navigation languages of the input transformation and of the output transformation are the same (XPath). If navigation languages had been different the transformation would be more complicated, because we would have to treat navigation expressions as compositions of meta-model instances and not as strings.

The two last rules (cf. line 118 to the end) describe the transformation of XML elements from the XSLT documents. Those elements are just copied by the first rule while the second one copies their attributes. The discriminators of those rules test if the nodes are *xsl* elements not to copy them. The helper called *isNotPredefined* does this work.

4.2.3 An illustrative example

Figure 17 and Figure 18 present a test of the transformation. On Figure 17 we can see the input XSLT which extracts the employees having a salary greater than 2000$ from an XML document storing employees data. Figure 18 presents the output of the transformation, which has been run and does the same work than the XSLT with Qexo [20] implementation of XQuery.

```
<xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >
        <xsl:template match="/">
                <emps>
                  <xsl:apply-templates select="employee"/>
                </emps>
        </xsl:template>
        <xsl:template match="employee">
          <xsl:if test="salary&gt;2000">
            <emp>
              <xsl:value-of select="name"/>
              <xsl:value-of select="firstname"/>
            </emp>
          </xsl:if>
        </xsl:template>
</xsl:stylesheet>
```

Figure 17 - Test input : an XSLT file

```
define function fctemployee($paramVar)
{
  for $var in $paramVar
  return
    let $var := $var
    where $var/salary>2000
    return
      <emp>{$var/name}{$var/firstname}</emp>
}
for $var in document("xmlFile.xml")/*
return
  <emps>{fctemployee($var/employee)}</emps>
```

Figure 18 - Test output an XQuery file

5. Conclusion

Why inventing new transformation languages since we already have Prolog or XSLT or even Java, Perl, and Python that could be used for this purpose ? We hope the material presented in this papermay partially hepl answer some of these questions.

As announced at the beginning of this paper, the application presented here is atypical of the model engineering technology because not strictly situated inside the MDA TS. This has been done on purpose because we believe the MDA TS is not an island and should be considered in relation to other TSs. Since the XSLT processing of XMI serialized model had already been tried, we have tackled here the reverse problem. This shows that bridges between various TSs are not symmetrical, but this does not come as a surprise.

Working at the boundary between two TSs is important because we believe a transformation system must be wider than a simple language and corresponding engine. It should encompass a large library of transformation components and also a set of import-export facilities from/to other TSs.

We have learnt a lot during this work. First, since it was a difficult and unconventional problem; it obliged us to push the ATL language to some of its limits and several aspects were improved. ATL is still evolving because one of our main objectives is to make it compliant with the result of the MOF/QVT convergence recommendation when it is ready.

But we also hope to add many original features to the ATL transformation system. We already knew that being able to deal with a large hierarchical library of transformation components is a must and that corresponding browsing features should be available in any practical transformation framework. We presently know that we also need to provide a corresponding library of import-export component between the MDA and other TSs. Furthermore we understand that these import-export components should absolutely be specializable. An XML import-export component will serve to deal with any other XML schemas as has been done here for XSLT and XQuery. Similarly other generic import-export components are being built, for example another family based on EBNF to deal with the TS of conventional programming languages.

One of the roles that should be played by any QVT-compliant transformation language is to deal with transformation legacy i.e. working transformations expressed in other formalisms. The present work illustrates one possibility to deal with this.

It is of paramount importance to compare and classify the different transformation systems that are being proposed for various emerging technologies. The features, possibilities, characteristics, scope of applicability, scalability, etc. of these systems should be put in correspondence and seriously evaluated (composability, extensibility, modularity mechanisms, organization structure, tracing facilities, directionality, etc.). An interesting initial evaluation is provided in [4]. Unfortunately this does not take into account proposals that are outside the strict MDA QVT proposals. XML-based, Java-based, Prolog-based, graph-transformation systems, and many others should be compared because they may often be used for the same tasks. This paper has shown how important the notion of technological space is, if we want to avoid sterile discussions. Furthermore we have pushed the idea to its application, by building a new bridge between the MDA and the XML TS, in the reverse direction of the old bridge presented in [17] that was operating in the opposite direction. We are presently convinced that the notion of TS is not only an interesting discussion idea, but an operational concept very useful to the engineer that has a problem to solve and that does not know beforehand which technology is the most suitable to solve it.

We are also confident in the suitability of ATL for many tasks, including non-conventional ones. As part of the ongoing collaboration between ATLAS and TNI-Valiosys, similar experiment are going to be conducted, for example in domain of programming language (EBNF import-export components).

During this presentation we pointed out some open questions about transformation frameworks. Among these, the traceability problem should have a good position in the research agenda. Many other subjects should also be mentioned as areas of ongoing research. The possible expression of model weaving operations (i.e. binding a business model to a platform definition model in order to get a platform specific model) as a sequence of transformation is an interesting and difficult problem. More generally we are studying the potential applications of defining higher order transformation in ATL. Higher order transformations are transformations taking other transformations taking other transformation as input and/or producing other transformation as output. Even if these possibilities are present in other TSs (with XSLT for example), their particular interest seems yet underestimated.

6. Acknowledgements

We thank Patrick Valduriez and the members of the TRISKELL team and of the MOTOR activity in the CARROLL project for multiple advices and comments on this work.

7. References

[1] AndroMDA http://www.andromda.org/pages/whatisit.html

[2] Badros, G. J.: JavaML Documentation http://www.cs.washington.edu/homes/gjb/JavaML/

[3] Bézivin, J.: From Object-Composition to Model-Transformation with the MDA. TOOLS-USA'2001, Santa Barbara, USA2001 http://www.sciences.univ-nantes.fr/info/lrsg/Recherche/mda/TOOLS.USA.pdf

[4] Gardner, T., Griffin, C., Koehler, J., Hauser, R.: Review of OMG MOF 2.0 Query/Views/Transformations Submissions & Recommendations towards. final Standard. http://www.omg.org/docs/ad/03-08-02.pdf

[5] Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A.: Transformation: The Missing Link of MDA. ICGT 2002

[6] Jamda Java Model Driven Architecture. http://sourceforge.net/projects/jamda/

[7] Kniesel, G., Koch, H.: Static Composition of Refactorings. University of Bonn, Submitted for publication, April 2003

[8] Kurtev, I., Bézivin, J., Aksit, M.,: Technological Spaces: An Initial Appraisal. CoopIS, DOA'2002 Federated Conferences, Industrial track, Irvine, 2002

[9] Langlois, B., Farcet, N., THALES recommendations for the final OMG standard on Query / Views / Transformations. OOPSLA 2003 "Generative Techniques in the context of Model Driven Architecture" workshop, October 27, 2003 http://www.oopsla.org/oopsla2003/files/ws-3.html

[10] Lemesle, R.,: Transformation rules based on meta-modeling. EDOC'98, San Diego, 3-5 November 1998 http://www.sciences.univ-nantes.fr/info/lrsg/Pages_perso/RL/Publications/EDOC98-lemesle.pdf

[11] Object Management Group: UML 2.0 OCL 2nd revised submission. 2003 http://www.omg.org/docs/ad/03-01-07.pdf

[12] Object Management Group: OMG/MOF Meta Object Facility (MOF) Specification. September 1997 http://www.omg.org/docs/ad/97-08-14.pdf

[13] Object Management Group: OMG/RFP/QVT MOF 2.0 Query/Views/Transformations RFP. October 2002, http://www.omg.org/docs/ad/02-04-10.pdf

[14] Object Management Group: XML Model Interchange (XMI). October 1998 http://www.omg.org/docs/ad/98-10-05.pdf

[15] OMG / MOF 2.0, Query / Views / Transformation ad/2002-04-10, Revised Submission, Version 1.0, 2003/08/18, OpenQVT

[16] OMG / SPEM. Software Process Engineering Process Metamodel (SPEM). OMG Document ad/formal/02-11-14, version 1.0, November 2002

[17] Peltier, M., Bézivin, J., Guillaume, G.: MTRANS: A general framework, based on XSLT, for model transformations. Workshop on Transformations in UML (WTUML), Genova, Italy, April 2001 http://www.sciences.univ-

nantes.fr/info/lrsg/Pages_perso/MP/pdf/wtuml_2001.pdf

[18] Ploquin, N.: Tooling the MDA framework: a new software maintenance and evolution scheme proposal http://www.sciences.univ-nantes.fr/info/lrsg/Pages_perso/Publications/joop01.pdf

[19] Pottinger, R.A., Bernstein, P.A.: Merging Models Based on Given Correspondences. University of Washington Technical Report UW-CSE-03-02-03, February 2003

[20] Qexo - The GNU Kawa implementation of Xquery. http://www.gnu.org/software/qexo/

[21] Soley, R., and the OMG staff: Model-Driven Architecture. OMG document Available from www.omg.org November 2000.

[22] Velocity 13.1, The Apache Jakarta Project, http://jakarta.apache.org/velocity/

[23] XML meta-model, netBeans.org, http://mdr.netbeans.org/mdrxml.html

[24] World Wide Web Consortium: XSL Transformations (XSLT) Version 2.0. http://www.w3.org/TR/xslt20/

[25] World Wide Web Consortium: XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/

Appendix

```
module XSLT2Xquery;
create OUT: XQuery from IN : XSLT;

rule XSLTRootNode2XQueryProgram {
  from _XSLT : XSLT!XSLTRootNode (true)
  to
    XQueryProgram : XQuery!XqueryProgram mapsTo
_XSLT,

  FLWOR : XQuery!FLWOR (
    xqueryProgram <-  XQueryProgram,
    for <- for,
    return<- return
  ),

  for : XQuery!For (
    var <- '$var',
    expression <- forExpression
  ),

  forExpression : XQuery!Xpath (
    value <- 'document(\"xmlFile.xml\")/*'
  )

  return : XQuery!Return (
    expressions <- _XSLT.nodes->select(
      t|t.match= '/'
    )->collect(t|t.nodes)
  )
}

rule Template2FLOWR {
  from  _Template:  XSLT!Template  (_Template.match  <>
'/')
  to

    functionDeclation : XQuery!FunctionDeclaration
    mapsTo _Template(
      name <- 'fct'+_Template.match,
      expression <- Sequence{FLWOR},
      xqueryProgram <-
XSLT!XSLTRootNode.allInstances()->first(),
    ),

    FLWOR : XQuery!FLWOR (
      for <- for,
      return <- return
  ),

    for : XQuery!For (
      expression <- forExpression,
      var <- '$var'
  ),

    forExpression : XQuery!Xpath (
      value <- '$paramVar'
    ),

  return : XQuery!Return (
    output <- _Template.nodes,
    expressions <- _Template.nodes
  )
}

```

```
rule ApplyTemplates2FunctionCall {
  from _ApplyTemplates : XSLT!ApplyTemplates(true)
to
  functionCall : XQuery!FunctionCall
  mapsTo _ApplyTemplate(
    name <-'fct'+ApplyTemplates.select,
    parameters <- Sequence{parameter}
  ),

  parameter : XQuery!XPath(
    value <- '$var/' + _ApplyTemplates.select
  )
}

rule If2FLWOR{
  from _If: XSLT!If(true)
  to

  FLWOR : XQuery!FLWOR mapsTo _If(
    _let <- varlet,
    where <- where,
    return <- if
  ),

  varlet : XQuery!Let(
    expression <- letExpression,
    varlet.var <- '$var'
  ),

  letExpression : XQuery!XPath(
    value <- '$var'
 ),

  where : XQuery!Where(
    expression <- whereExpression
  ),

    whereExpression : XQuery!BooleanExp(
    value <- '$var/' + _If.test
  ),

  return : XQuery!Return(
    expressions <- _If.nodes
    )
}

rule ValueOf2ReturnXPath {
  from _ValueOf : XSLT!ValueOf(true)
  to
    XPath : XQuery!ReturnXPath mapsTo _ValueOf(
    value <- '$var/' + _ValueOf.select
  )
}

rule ElementNode2ElementNode {
  from _ElementNode : XSLT!ElementNode(
XSLT2XQuery.isNotPredefined(_ElementNode)
  )
  to
    elementNode : XQuery!ElementNode(
      name <- _ElementNode.name,
      output <- _ElementNode.nodes,
      nodes <- _ElementNode.nodes
  )
}

rule AttributeNode2AttributeNode {
  from _AttributeNode : XSLT!AttributeNode(
    XSLT2XQuery.isNotPredefined(_ElementNode)
  )
  to attributeNode : XQuery!AttributeNode
  mapsTo _AttributeNode (
    name <- _AttributeNode.name,
    attributeNode.value <- _AttributeNode.value
  )
}
```