ATL Transformation Examples

The Microsoft DSL to EMF ATL transformation

- version 0.1 -

October 2005

by

ATLAS group

LINA & INRIA

Nantes

Content

Figures List

1 Introduction

This document provides you a complete overview of a transformation chain example between two technical spaces: Microsoft DSL (Domain Specific Languages) Tools [1] and EMF (Eclipse Modeling Framework) [2]. The aim of this example is to demonstrate the possibility to exchange models defined under different technologies. In particular, the described bridges demonstrate that it should be possible to define metamodels and models using both Microsoft DSL Tools and Eclipse EMF technologies. Note that the bridges described in this document have to be considered as preliminaries prototypes. As such, they focus on a subset of all possible transformation scenarios. Moreover, it may appear that some of the defined transformations still contain some errors.

The bridge between MS/DSL and EMF spans two levels: the metamodel and model levels. At the level of metamodels, it allows to transform MS/DSL domain models to EMF metamodels. At the level of models, the bridge allows transforming MS/DSL models conforming to domain models to EMF models conforming to EMF metamodels. At both levels, the bridge operates in both directions. A chain of ATL-based transformations is used to implement the bridge at these two levels. The benefit of using such a bridge is the ability to transpose MS/DSL work in EMF platform, and inversely.

The next sections explain the different steps to realize the bridge. Section 2 explains the operation of the bridge at the metamodel level; Section 3 shows the operation at the model level. Finally, Section 4 explains why and how an extension could be implemented.

2 The metamodel bridge

2.1 Explanation

We can use ATL [3] to transform domain models into KM3 models [5] and EMF metamodels [2] with an additional transformation from KM3 to Ecore [6]. An overview of the M2-level bridge is given in Figure 1.

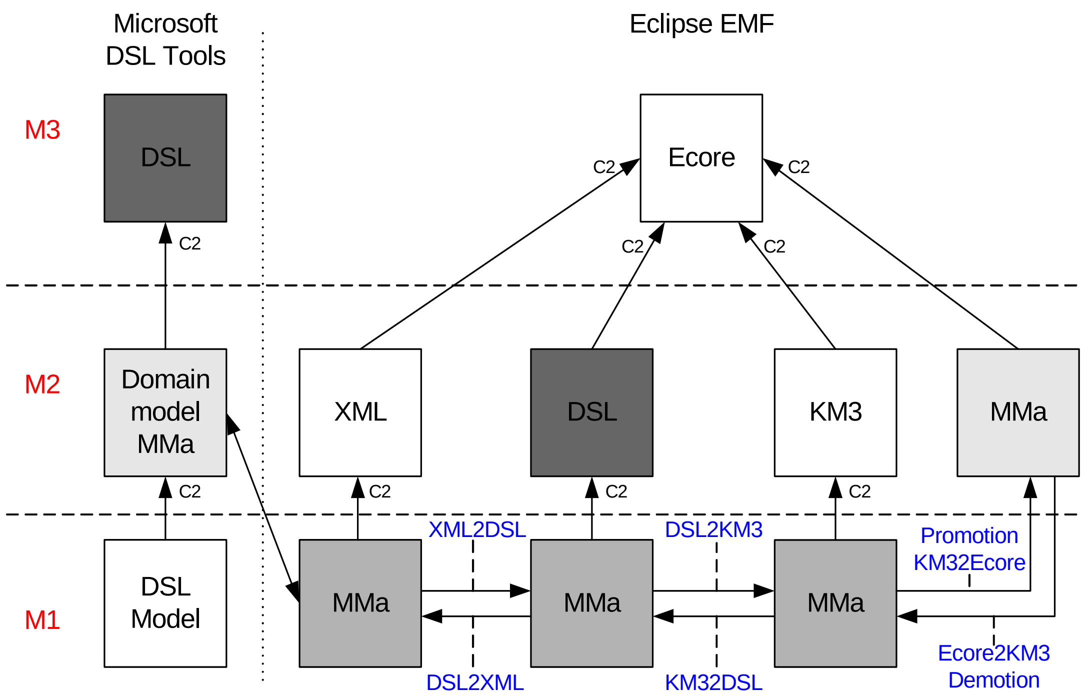


Figure 1. Overview of the M2-level bridge

A metamodel MMa (defined in DSL Tools, as in Figure 4) is injected into an XML model using an XML injector (included with ATL). After that, the model is transformed into a model conform to the DSL metamodel considered in this transformation example, and then into a KM3 model. The final step is the promotion of this model using the KM32Ecore transformation, which creates the MMa metamodel conforming to Ecore (the inverse transformation is a demotion). The inverse transformation chain from Ecore to DSL is also defined.

In fact, this transformation chain mainly consists in building transformations from DSL to KM3 and KM3 to DSL. Indeed, since KM3 acts as a pivot between the metametamodels, transformations to and from Ecore are already available.

2.2 M3-level mapping

To enable mapping between MS/DSL and EMF, a definition of each system at M3 (i.e. metametamodel) level is required. KM3 is used as a metametamodel. Since Microsoft does not specify any explicit metametamodel for DSL designers, a metametamodel has been designed by observation.

2.2.1 KM3: Kernel MetaMetaModel

KM3 is a metametamodel close to Ecore and EMOF 2.0 [7]. A class diagram version is presented in Figure 3. It is used rather than Ecore because because this example also deals with several other metametamodels such as MOF 1.4 [8]. KM3 is used as a pivot between these metametamodels as illustrated on Figure 2. Additionally, it provides a textual concrete syntax to specify metamodels, which has some similarities with the Java notation.

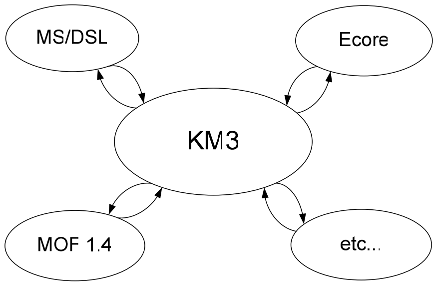


Figure 2. Use of KM3 as a pivot

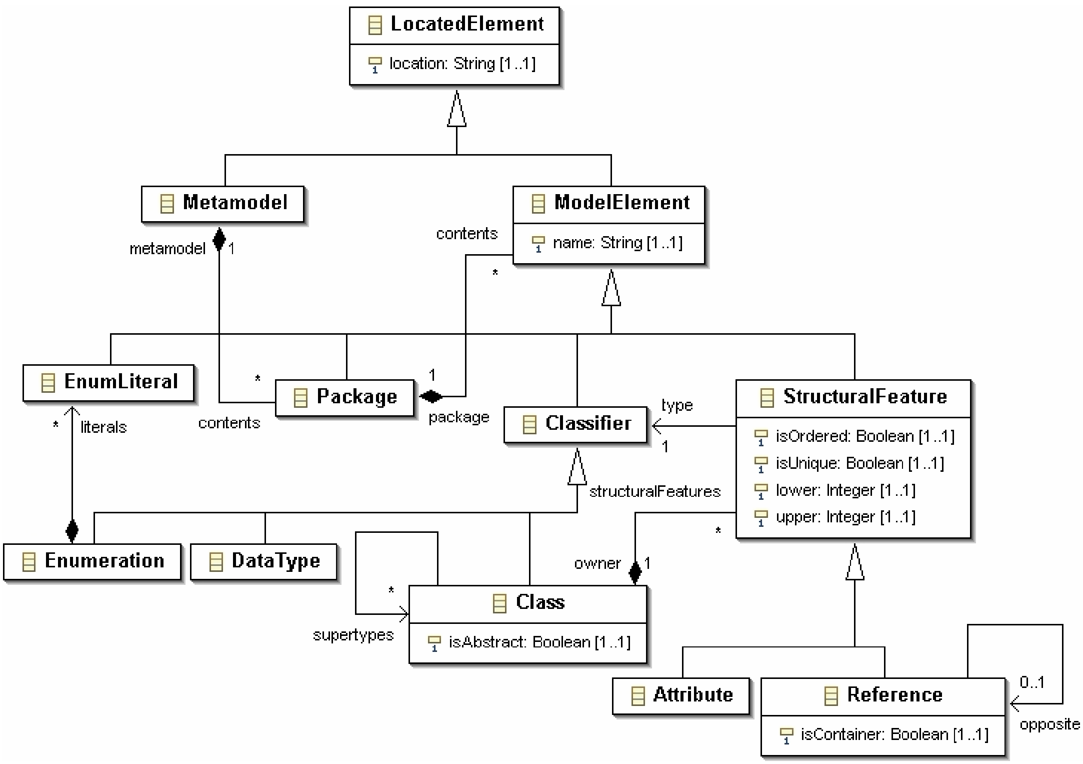


Figure 3. The KM3 metamodel

2.2.2 Microsoft DSL metametamodel

The Microsoft Tools for Domain-Specific Languages is a suite of tools for creating, editing, visualizing, and using domain-specific data for automating the enterprise software development process. These new tools are part of a vision for realizing software factories. The version considered here is May 2005 CTP release for Visual Studio 2005 Beta 2.

The equivalent of a metamodel in the Microsoft world is called a “domain model”. It is composed of a class hierarchy and relationships. A DSL metametamodel has been extracted from experience with Microsoft’s tools. Figure 4 provides a domain model example.

Classes and relationships are viewed at the same level. A relationship may be a simple reference or an embedding. It has properties and may have a super type like a class. A relationship has two roles, but a future version of DSL Tools may propose relationships with n roles. A role can be viewed as an association end in UML. It has a name as well as multiplicity max and min properties.

Domain models contain two different types: enumeration and simple type. The latter can be Boolean, String, Integer, Double, Date, etc.

The DSL metametamodel is presented in Figure 5.

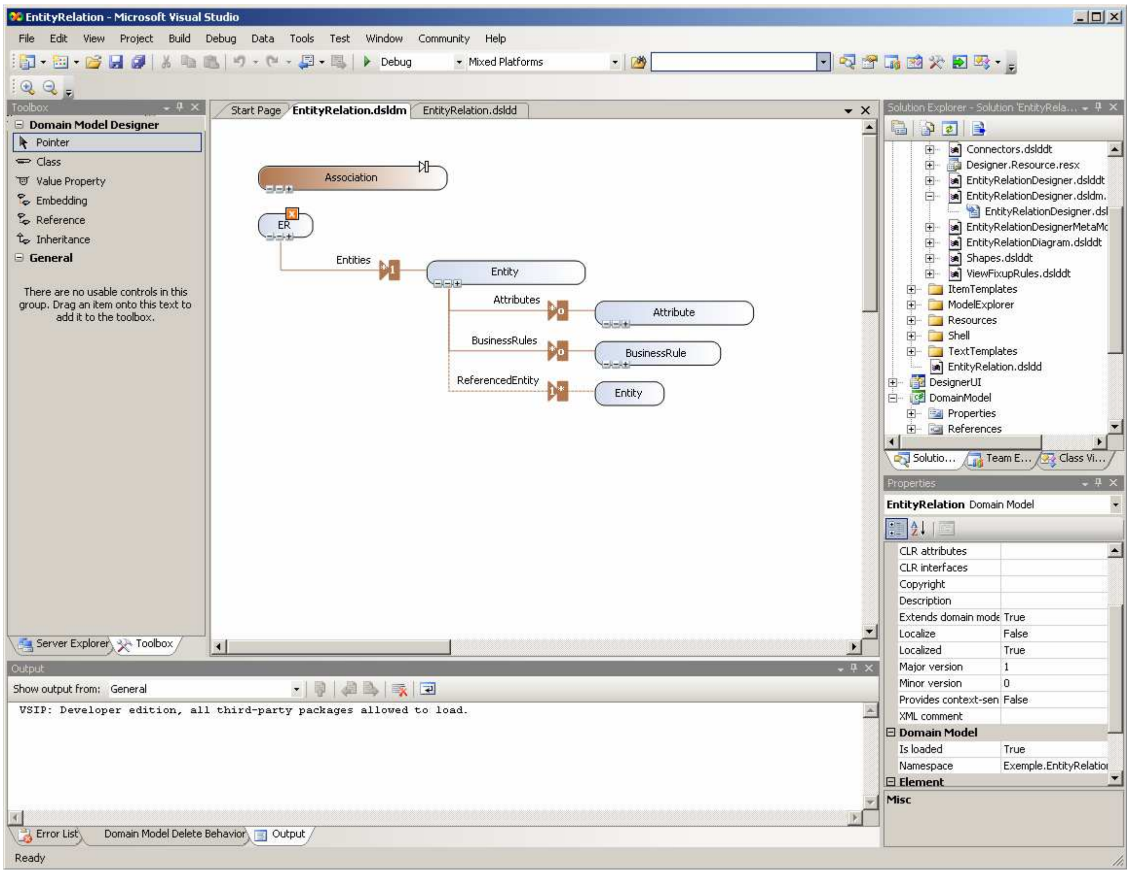


Figure 4. A Domain model example

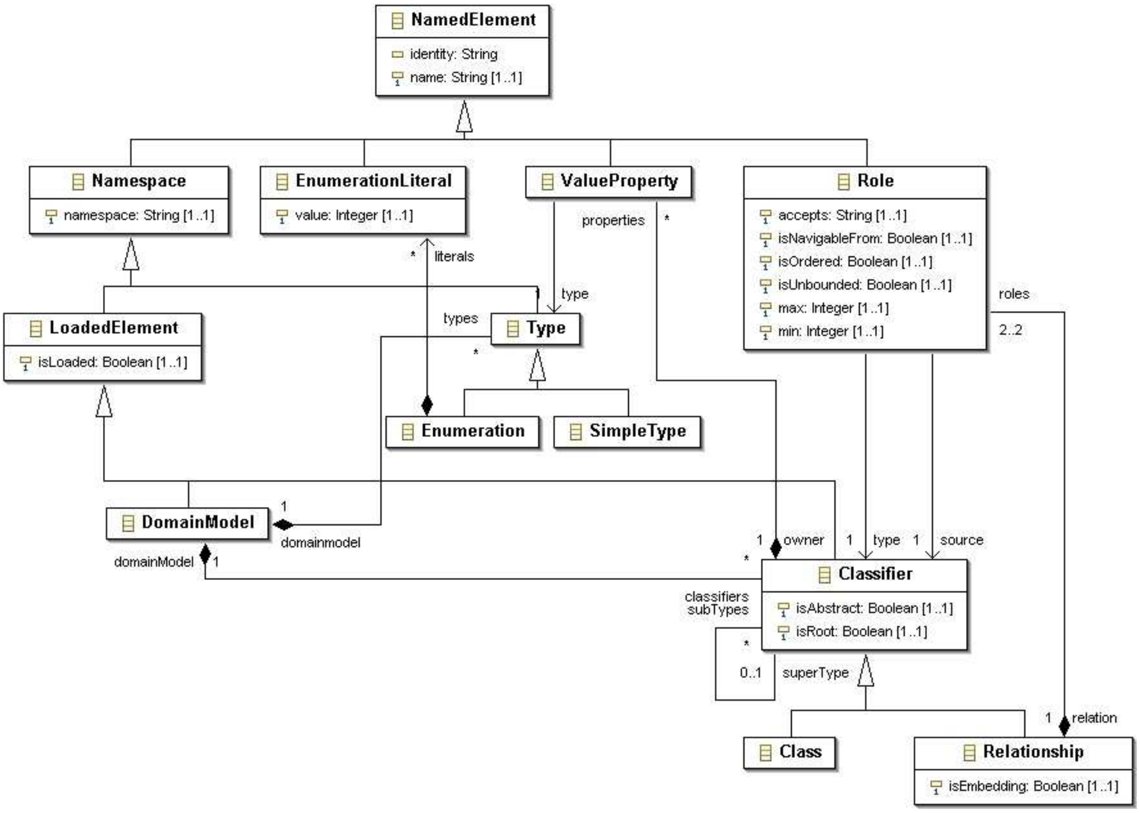


Figure 5. The DSL metametamodel

2.2.3 Comparison between KM3 and DSL

With those metamodels, we can compare KM3 and DSL with each other (simplified versions of them are showed in Figure 6), it appears that:

- KM3 and DSL Classes are almost equivalent, and have the same characteristics, except supertypes: KM3 allows multiple inheritances whereas DSL does not.
- A KM3 Attribute is equivalent to a DSL ValueProperty.
- DSL roles can be mapped to KM3 References, but those are not affiliated with a Relationship like in DSL. There are simply contained in their owner and linked to their opposite. When the owning relationship of a pair of roles is an embedding, one of the associated KM3 references is a container.
- DSL Relationships and DSL Classes have the same properties: relationships may be linked to each other, have a supertype, attributes, etc. There is no direct equivalent in KM3. We can simply consider that simple relationships (with no supertype or attribute) correspond to a pair of references whereas complex relationships correspond to classes.

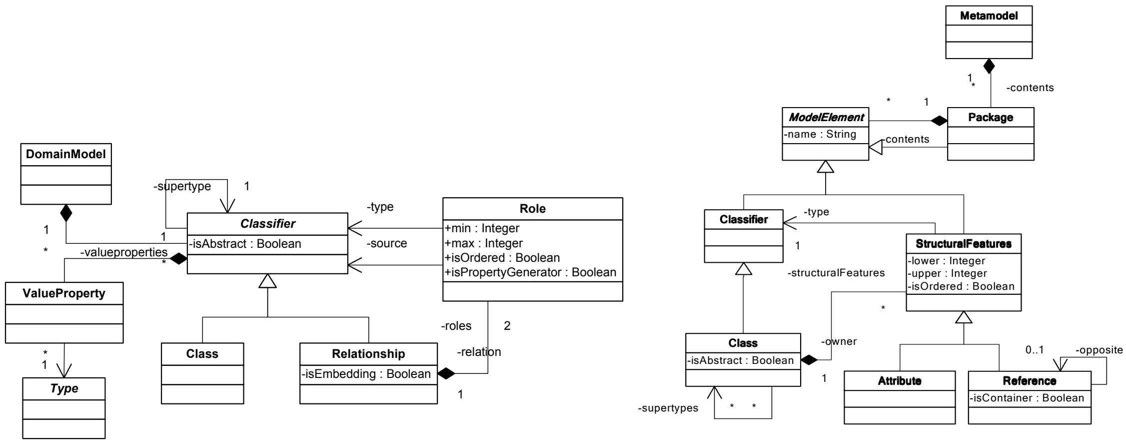


Figure 6. Simplified versions of DSL (left) and KM3 (right) metametamodels

2.3 First ATL Transformation chain: DSL to Ecore

The aim of this transformation chain is to convert a DSL metamodel, contained by a *.dsldm* file, into an Ecore metamodel, which could be used with EMF. The transformation is defined in three steps which are detailed in the following sections.

2.3.1 XML to DSL

2.3.1.1 Principle

This first transformation has to extract information from a *.dsldm* file into a DSL metamodel. The *.dsldm* file is injected into an XMI file that conforms to an XML metamodel, using an XML injector. Next step is based on an ATL transformation that captures information from the XML file to create a model which is conform to the DSL metamodel previously described.

The main work of this transformation is to achieve a mapping between *.dsldm* features (concepts, relationships, roles, enumerations, properties…) and the considered DSL model.

Four types are recognized with the developed bridge: String, Boolean, Integer and Double. The corresponding DSLs SimpleTypes are created by default in the “main” rule, DomainModel. Then when an attribute is encountered in the model, its type is linked to one of those previously created, using the helper findType(). This helper, using a resolveTemp() function, retrieve the type previously created.

In the *.xml* file, a reference to an object is symbolized by an XML Text field containing the id of the referenced feature. To put this information in the output model, a correspondence is established a between objects and their id. A specific helper creates a table with two rows (in ATL, a map) containing all the ids and the XML Element they correspond to. Then, when encountering any id in the file, it is possible to retrieve the class, relationship, role… it corresponds to by using the helper dslElementsById(), and link it in the output model.

2.3.1.2 Limitations

The only limitation is the type of the properties: as previously stated, String, Double, Integer and Boolean are the only recognized types.

2.3.1.3 Use

The first step is to create a DSL tools project (or take an existing one) that is going to be turned into an EMF project. The file that contains the Metamodel, which is usually located at this file path:

…\Visual Studio 2005\Projects\ProjectName\DomainModel*ProjectName.dsldm*

This file can be imported into an ATL project by renaming it into *ProjectName.xml*. The XML injector (right-click → "import XML model") then enables to get an XML model from the *.xml* file. This produces a file named *ProjectName.xmi*, which can be used as input of the first transformation. We must apply Executed on this file, the XML2DSL transformation provides it into an *.xmi* file containing the Metamodel conforming to the DSL Metamodel.

Figure 7 illustrates the transformation configuration: there is one input (XML) and one output (DSL) metamodel. In Path Editor, the path of the DSL and XML metamodels are respectively associated with DSL and XML. The IN field contains the path of the *.xmi* model example previously generated, and the OUT one the path for the results.

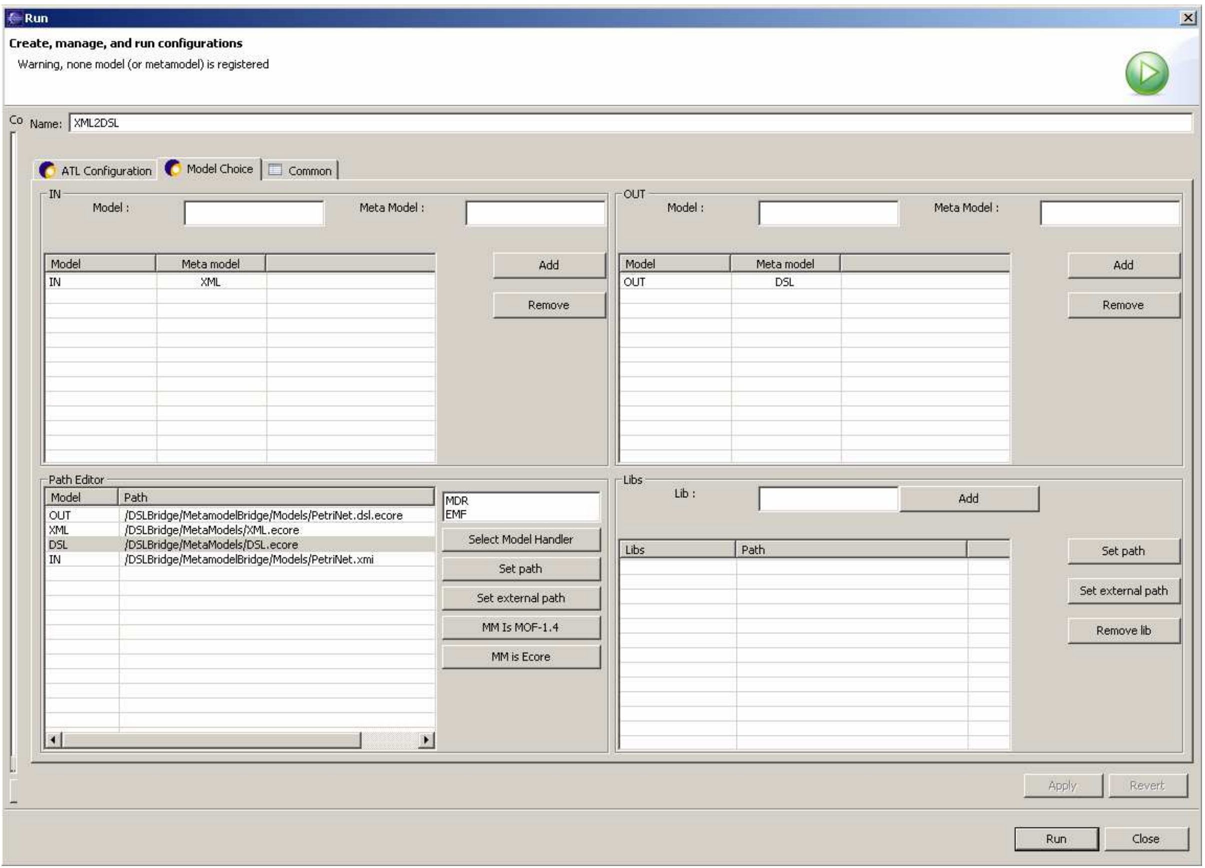


Figure 7. Configuration for XML2DSL

2.3.2 DSL to KM3

2.3.2.1 Principle

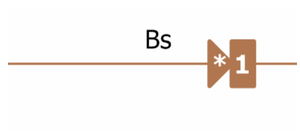
In this step, the previously generated DSL model is transformed into a KM3 metamodel using another ATL transformation. DSL classes are simply mapped to KM3 classes, like simple types and properties. Some problems have however to be considered: there exist some important differences between the

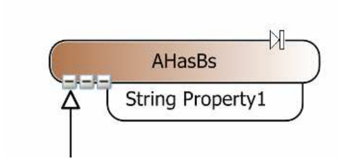
DSL and KM3 metamodels. The most important is that, in DSL, a Relationship is defined like a Class with the same properties, whereas in KM3, a Relationship between two Classes is symbolized by two references into adjoining classes. There are two solutions:

- The specific properties of the Relationship, except roles and type of containment, may be ignored; this however leads to an important information loss;
- The Relationship can be turned into a KM3 class, and keep the attributes, supertype, and other features.

Relationships can be classified into two types:

- **Simple Relationship:** If there is no supertype, attributes… it is possible to ignore the name and transform the relationship into a couple of KM3 references, using the first solution: the roles of a relation are mapped to KM3 references.
- **Complex Relationship:** If the relationship has attributes, supertypes, or subtypes, it is turned into a KM3 class (using the second solution), and two couples of references are created enabling to link it to the classes referenced by its roles, like in Figure 8.





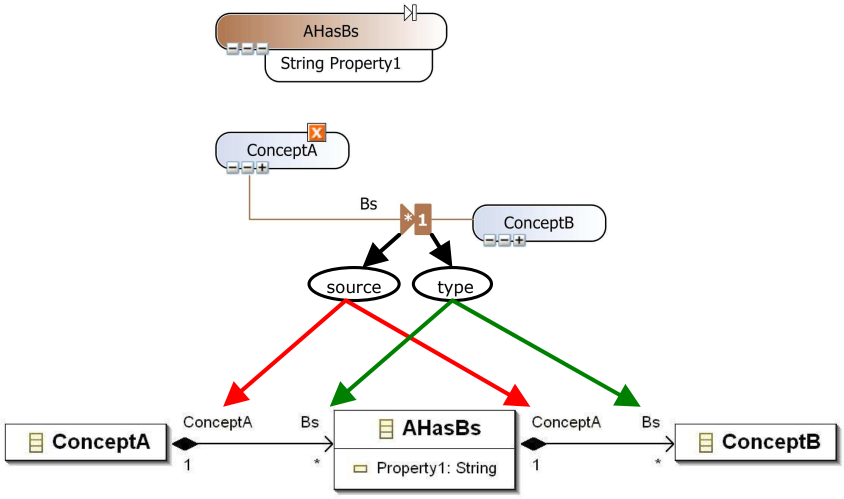


Figure 8. An example of Complex Relationship treatment

2.3.2.2 *Limitations*

There are no limitations in this transformation, but important information is lost: it is impossible to know if a class was previously a Relationship that has been turned into a class by the transformation. This problem is discussed in Section 4.

2.3.2.3 *Use*

Figure 9 provides a screenshot of the transformation configuration: there is one input (DSL) and one output (KM3) metamodel. In Path Editor, the DSL and KM3 metamodels are respectively associated with DSL and KM3. In IN field contains the path of the *.xmi* file previously generated by the XML2DSL transformation, and the OUT one the path for the results.

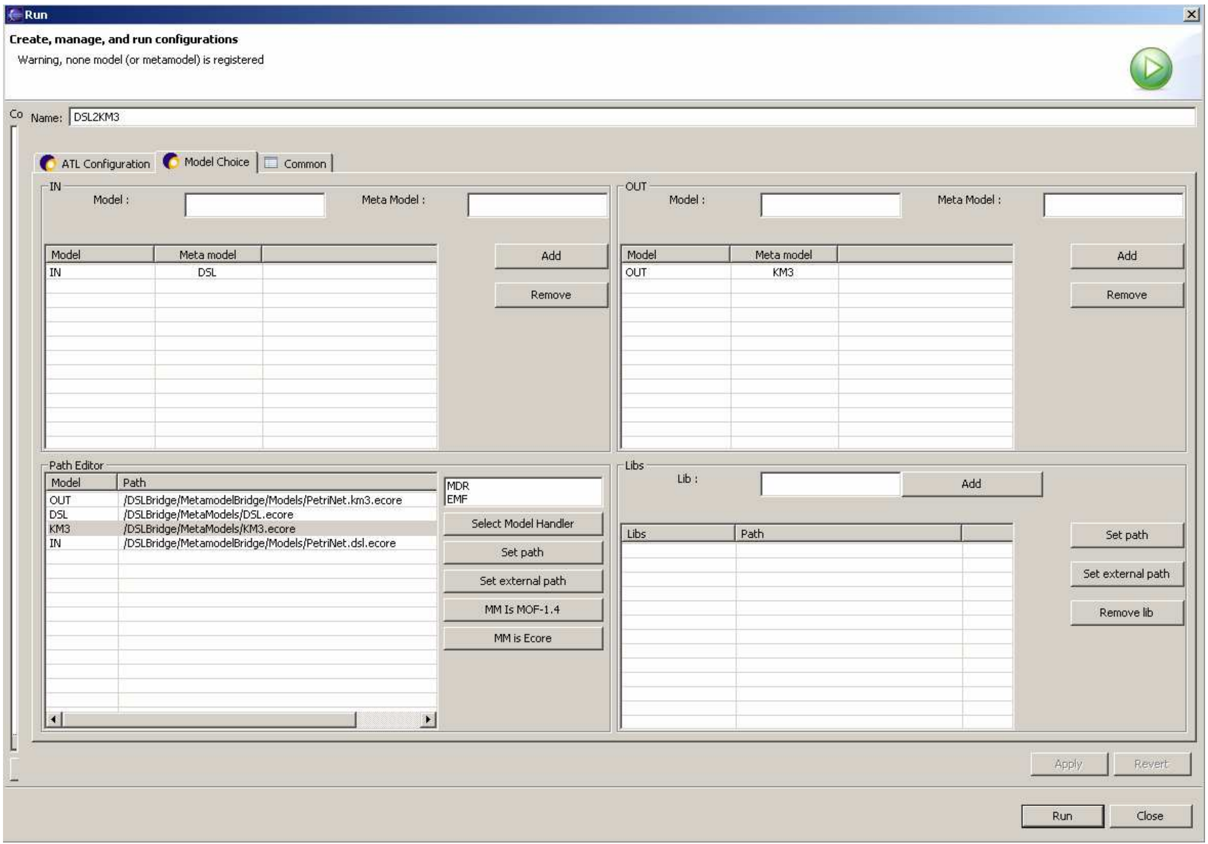


Figure 9. Configuration for DSL2KM3

2.3.3 Third step: KM3 to Ecore

The model produced at the previous step can now be used with, for instance, Omondo's EclipseUML plugin [9], by simply injecting the KM3 file into an Ecore model using the existing KM3 injector.

2.4 Second ATL Transformation chain: Ecore to DSL

This transformation starts with a model that conforms to the KM3 metamodel, and aims to transpose it into a DSL tools model. This transformation chain is composed of three steps that are detailed in the following sections.

2.4.1 First step: KM32DSL

2.4.1.1 Principle

While transforming the initial model (which is expressed in KM3) into a model which conforms to the DSL metamodel, several problems have to be considered:

- KM3 does not implement Relationships. They therefore need to be created from couples of KM3 References. To this end, the helper list stores the references which need to be turned into relationships. Each time this helper encounters a reference that has no opposite or that has an opposite reference which has never been stored, it stores it into a sequence.

```
-- This helper get a list of references which need to be turned
-- into relationship
```

```
-- CONTEXT: thisModule
-- RETURN: Sequence(KM3!Reference)
helper def: list:Sequence(KM3!Reference) =
self.getRefs()->iterate(e; seq : Sequence(KM3!Reference) = Sequence{} |
 if e.opposite.oclIsUndefined() then
       seq.append(e)
 else if (seq->includes(e.opposite) or seq->includes(e)) then
              seq
    else if e.isEmbedding() then
-- EMBEDDING
              if e.isContainer -- e is the first role
                     then seq.append(e)
                     else seq.append(e.opposite) -- e.opposite is the first role
              endif
              else
-- REFERENCE
                     seq.append(e)
              endif
       endif

  endif);
```

- When creating the relationship, the reference and its opposite are mapped to the two roles of the relationship. If the source reference did not have an opposite, it is generated.
- Some attributes of DSL do not have correspondence in KM3. They are therefore created using default values, and single identities are generated.

2.4.1.2 Use

The aim is now to turn an EMF Metamodel into a DSL project. For this purpose, a Metamodel conforming to Ecore has to be created. It is then turned into a KM3 Metamodel by using the Ecore2KM3 extractor (right-click → "extract Ecore Metamodel to KM3"). At this stage, the KM3 injector (right-click → "inject KM3 file to KM3 Model") enables to get a KM3 model from the *.km3* file. This model (an *.xmi* file) is then used as input of the KM32DSL transformation in order to generate the corresponding Metamodel in DSL.

Figure 10 provides a screenshot of the transformation configuration: there is one input (KM3) and one output (DSL) metamodel. In Path Editor, the path of the DSL and KM3 metamodels are respectively associated with DSL and KM3. The IN field contains the path of the *.xmi* previously generated, and the OUT one, the path for the results.

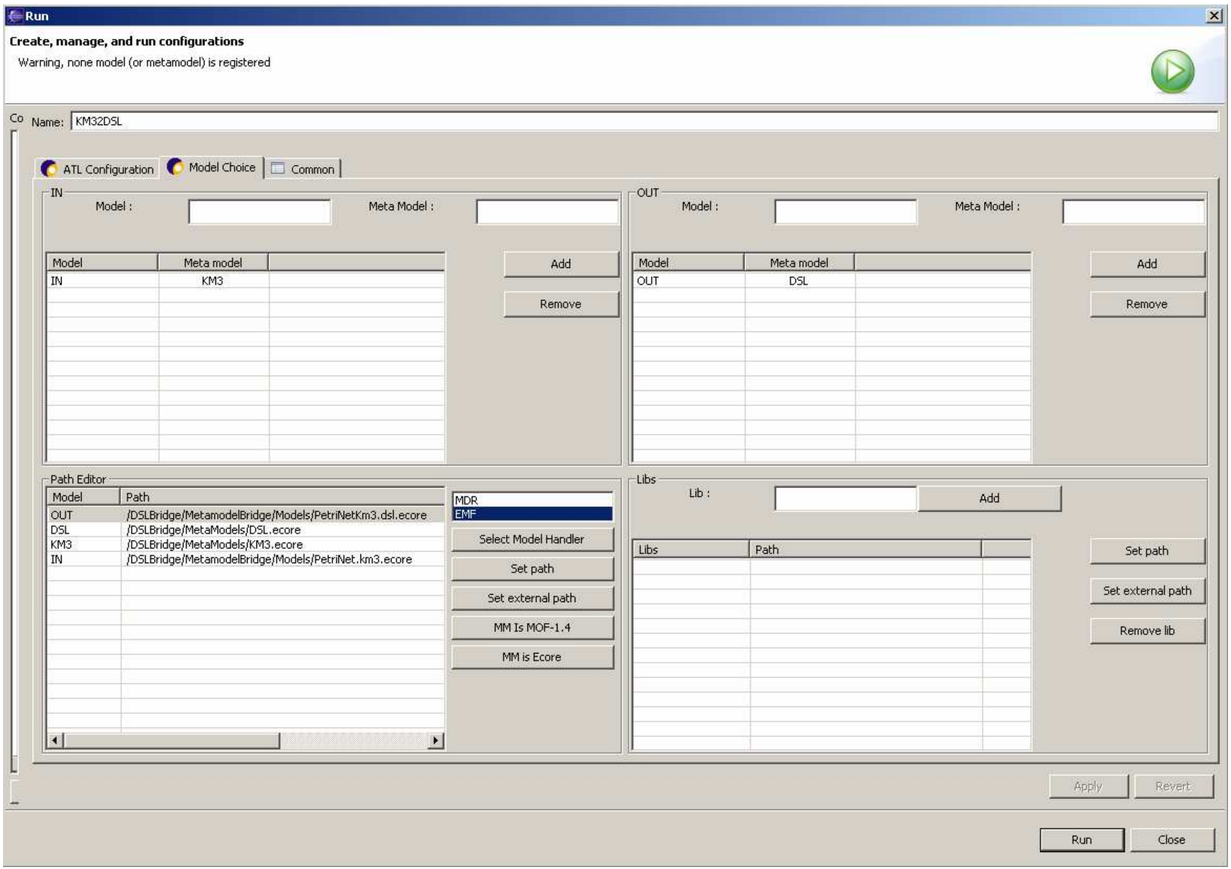


Figure 10. Configuration for KM32DSL

2.4.2 Second step: DSL to XML

2.4.2.1 Principle

This step transforms the previously obtained result into a model conforming to XML, which defines a *.dsldm* like file. In Figure 11, which is a simple metamodel of *.dsldm* file, the white entities correspond to what is supported by the considered DSL metametamodel, whereas grey entities correspond to what has been added from scratch for the purpose of this transformation.

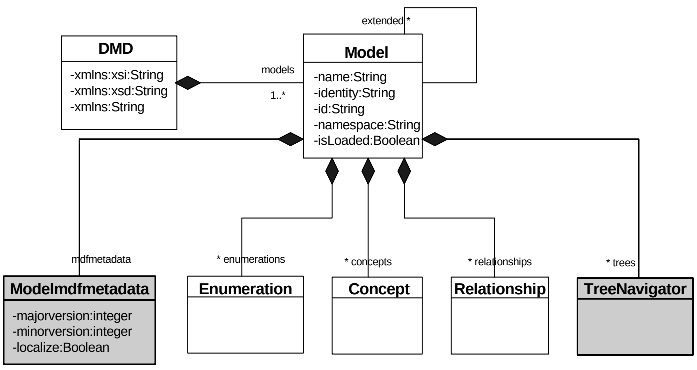


Figure 11. Simple metamodel of *.dsldm* file

Here are the the main parts of this model:

- MdfMetaData: information about the diagram (description, category);
- Extended: a reference to the model which is extended by this diagram;
- Enumerations: this part contains the enumerations and their literals;
- Relationships: this part contains the relationships and their roles and valueproperties;
- Trees: this part describes the way for DSL Tools to display the diagram as trees, with four treeNavigators.
    - treeNavigator Intrinsic: contains information about roles and inheritances.
    - treeNavigator CompleteDiagram: contains information about roles, inheritances, their definitionlevel and root classes of the diagram. These notions are explained below.
    - treeNavigator Serialization: contains a reference to the XML root element which would be used to serialize the diagram in future.
    - treeNavigator Delete: contains information about roles and inheritances., This tree is not built by the transformation (it does not seem to be mandatory).
- Concepts: this part contains the concepts and their valueproperties.

The transformation consists in mapping enumerations, concepts and relationship to their XML equivalent. For instance, a Concept would be notified as follows:

```
<concept name="ER" identity="4e10f0be... " namespace=" EntiteRelation.DomainModel"
id="i4e10f0be..." isLoaded="true" isAbstract="false">
        <mdfmetadata xsi:type="conceptorshapemdfmetadata" accessmodifier="public" category=""
description="" doccomment="" localize="false" />
</concept>
```

During this mapping, when encountering an inheritance (i.e. a concept or a relationship which has a supertype) or a role (each relationship owns two opposites roles), a ‘roleExpression’ or ‘inheritanceExpression’ has to be built in the treeNavigators.

The CompleteDiagram construction is the most important part of this transformation because two constraints have to be respected (otherwise DSL Tools can not display the diagram):

- Roots Classes must be signalized in the CompleteDiagram. In fact, those classes are the roots of the trees which appear in the model designer. In Figure 12, class1 and class3 are roots, and the designer must build two trees at least. To define if a class is root, the transformation checks whether it has no supertype and it is no pointed by any Relationship.

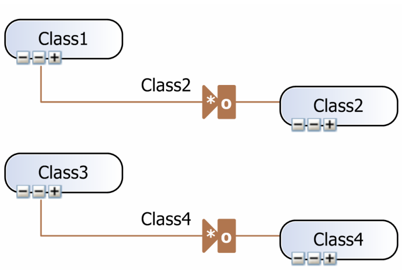


Figure 12. Roots constraint example

- When the designer needs to make a class appear several times, like in Figure 13, it must be notifoed in the treeExpression (in the CompleteDiagram treeNavigator) which corresponds to the reference or inheritance which makes necessary to display another class in the diagram. This is notified by changing the definitionlevel attribute to “use” instead of “definition”. In the transformation, when encountering a reference or inheritance which refers to a class, its definitionlevel is stored as "definition" and the class is put into a list. When this class is referred again, its definitionlevel must be store as "use" in the treeNavigator.

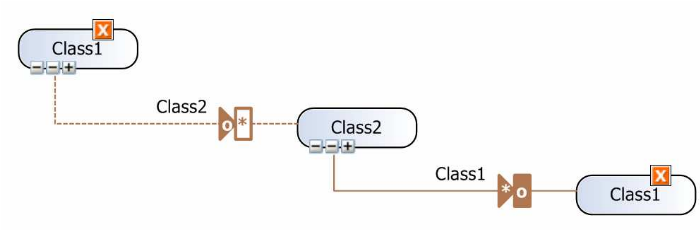


Figure 13. Definitionlevel constraint example

2.4.2.2 Limitations

The search for root classes may fail when the input model is too complex. In this case, the DSL tools can not display the model correctly and superpose some classes and relationships.

2.4.2.3 Use

Figure 14 provides a screenshot of the transformation configuration: there is one input (DSL) and one output (XML) metamodel.

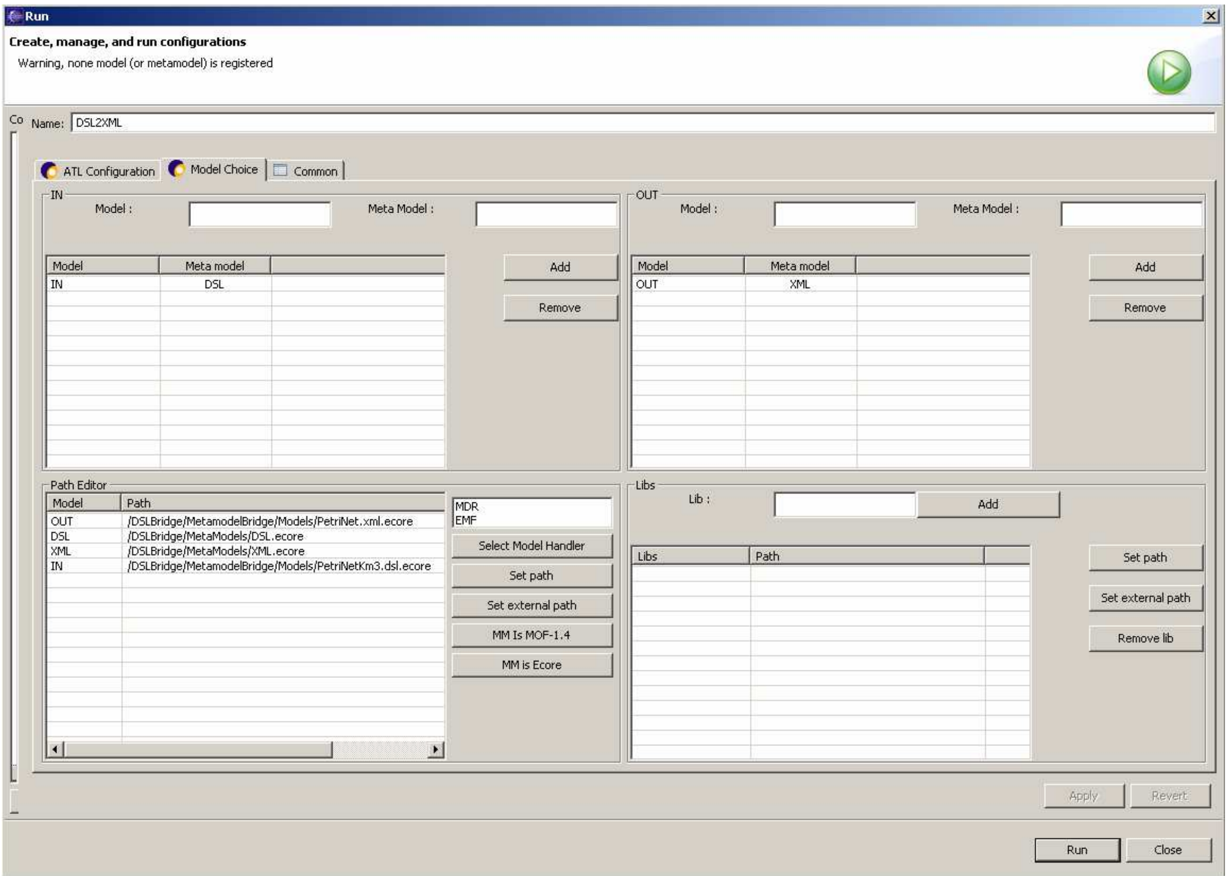


Figure 14. Configuration for DSL2XML

In Path Editor, the path of the DSL and XML metamodels are respectively associated with DSL and XML. The IN field contains the path of the *.xmi* containing the model conforming to the DSL metamodel, the OUT one, the path for the results.

2.4.3 Third step: XML2Text

The file previously generated is conform to the XML metamodel, and needs to be extracted into a real XML file.

The XML to text transformation enables to generate a *.dsldm* file that will be included into a blank project for DSL Tools, by replacing the .dsldm file. In the XML to Text ATL file, ensure that the output filepath is correct, at the top of the file, as shown below (the configuration is detailed on Figure 15).

```
query XML2Text = XML!Root.allInstances()
               ->asSequence()
               ->first().toString2('').writeTo('C:…filepath…\\DSL_Result.dsldm');
```

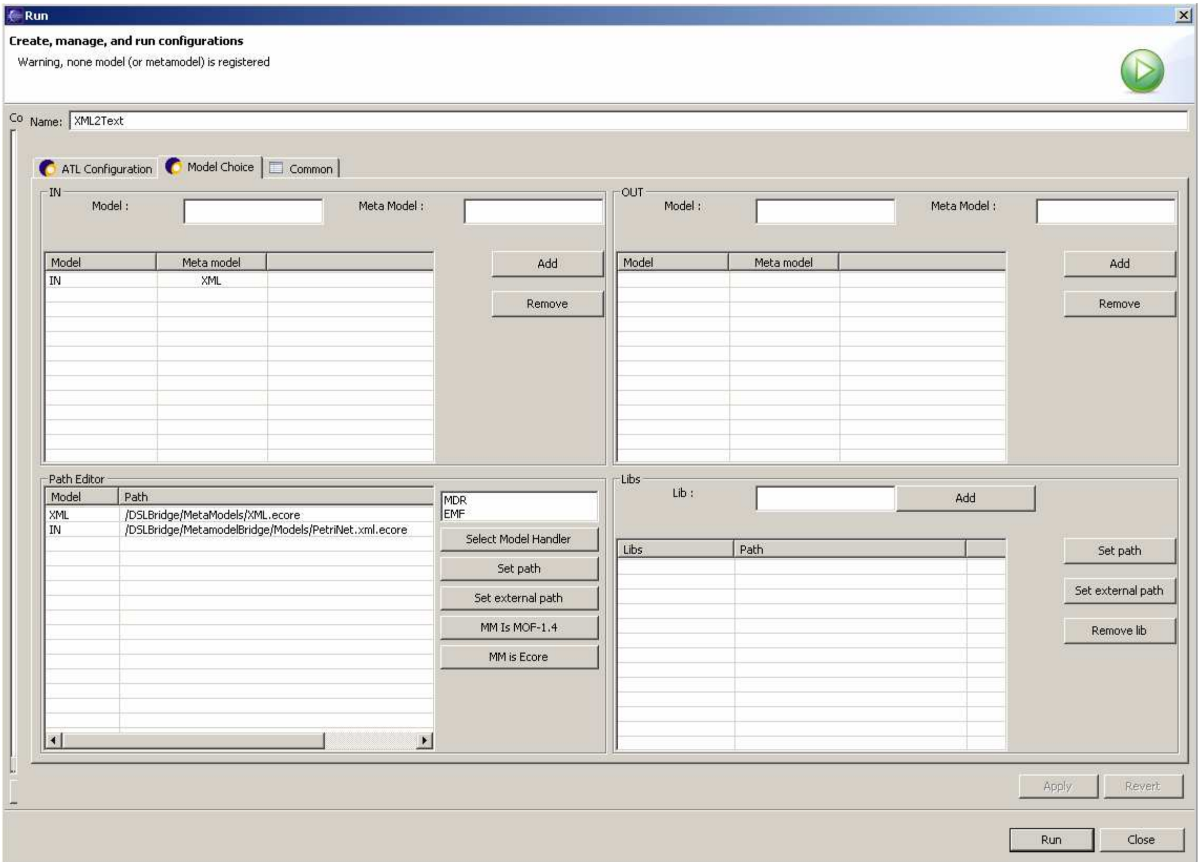


Figure 15 Configuration for XML2Text

2.5 Example: PetriNet

The aim of this section is to illustrate the metamodel bridge through the study of a PetriNets example. The first step deals with the DSL to EMF direction. For this purpose, a simple PetriNet metamodel defined under DSL Tools, is considered (see Figure 16). It is possible to note that there is one composition relationship, PlaceHasToken, which as a property ‘number’ (i.e. the number of token contained by the place). There are also two relationships, PlaceToTransition and TransitionToPlace, and both have a property ‘label’.

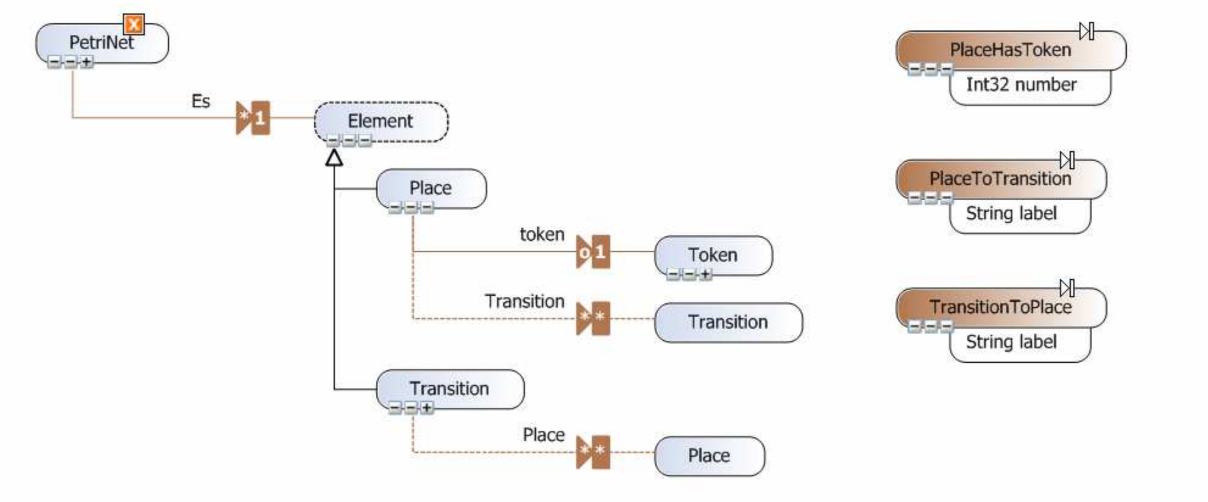


Figure 16. PetriNet view with DSL Tools

This model can be used as input of the first transformation, XML to DSL, using the configuration detailed in Figure 7. The DSL to KM3 transformation is then applied, using the configuration detailed in Figure 9. This produces a KM3 model, which can be easily turned into an Ecore model using the Ecore injector, and be displayed with the plugin Omondo's EclipseUML (see Figure 17). It is possible to note that PlaceHasToken, PlaceToTransition and TransitionToPlace become classes, linked by two references (compositions for PlaceHasToken).

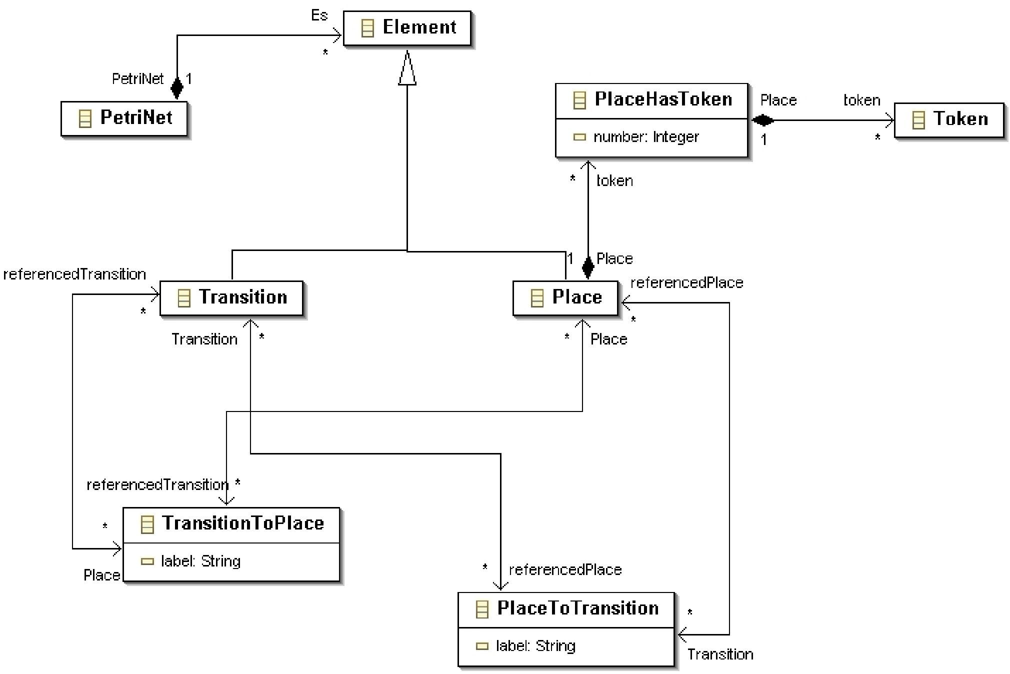


Figure 17. PetriNet view with EclipseUML plugin

In the process of turning this model into a DSL model, a KM3 model can be obtained using the Ecore to KM3 extractor. The KM3 to DSL transformation (which configuration is shown in Figure 10) produces a DSL model and then the DSL to XML transformation (which configuration is shown in Figure 14) an XML model. Finally, the XML2Text extractor generates the *.dsldm* file. Figure 18 provides the result obtained when opening the file with DSL Tools. It may be noted that

TransitionToPlace and Place to Transition concepts are displayed twice, but only one contains its properties.

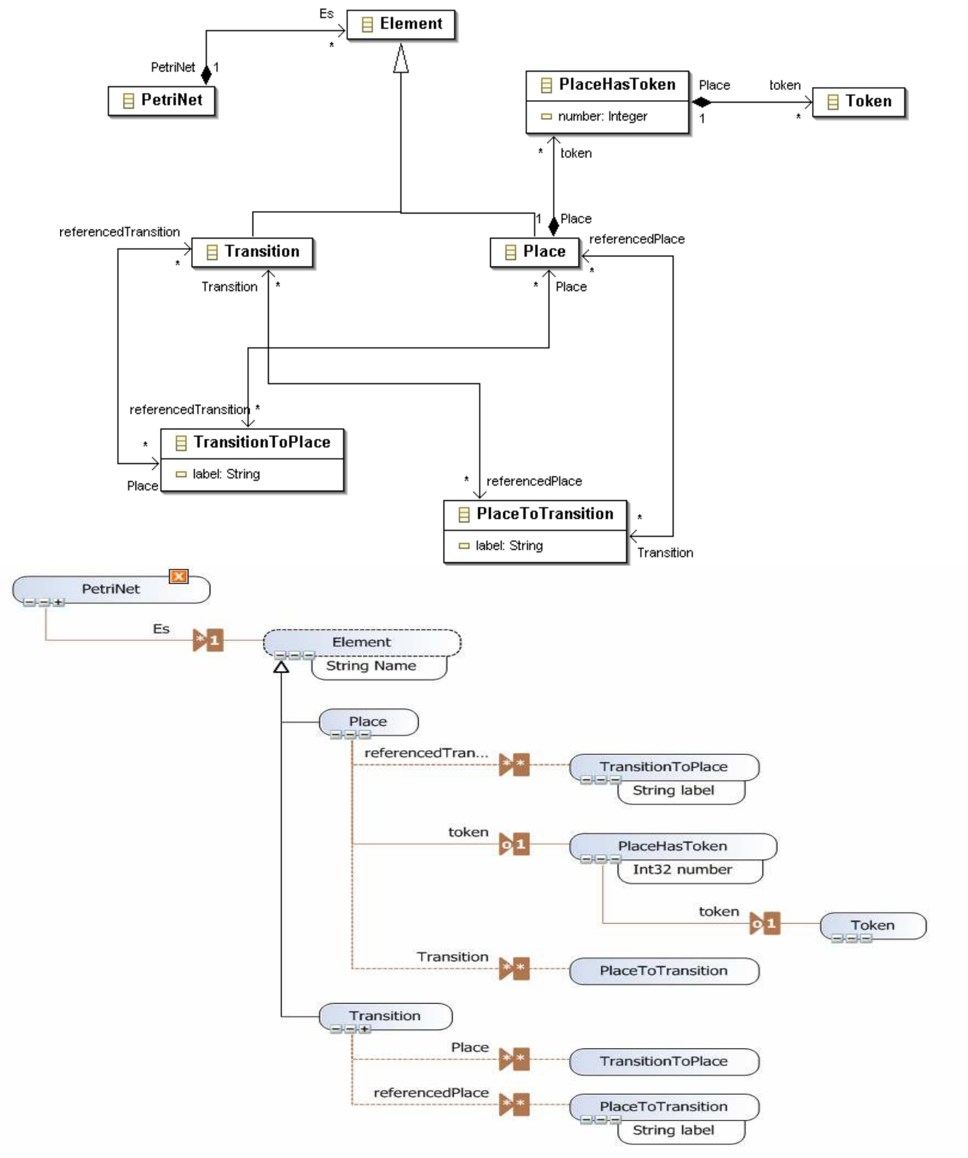


Figure 18. DSL Tools view of the final result

3 The model bridge

3.1 Introduction

The transformation chains between a DSL domain model and an EMF model (and inversely) have been defined in Section 2. However, there is still a lack for a tool enabling to transform DSL models to and from EMF models. Such a tool is described in the present section.

The first step is to consider how models are viewed by both technologies. Basically, a model has to conform to a domain model for MS/DSL and to an Ecore model for EMF.

To store its models, Microsoft uses an XML schema, which does not directly map to the domain model. With EMF, the models are stored in the XMI format and are explicitly conform to a metamodel.

To implement this bridge, information has to be grabbed from the DSL model file and transformed so that it conforms to an EMF metamodel. To this end, a metamodel that represents DSL models has to be built. DSL models conforming to this metamodel shall then be transformed into a models conforming to a metamodel defined under EMF.

Figure 19 summarizes the different steps of the model bridge.

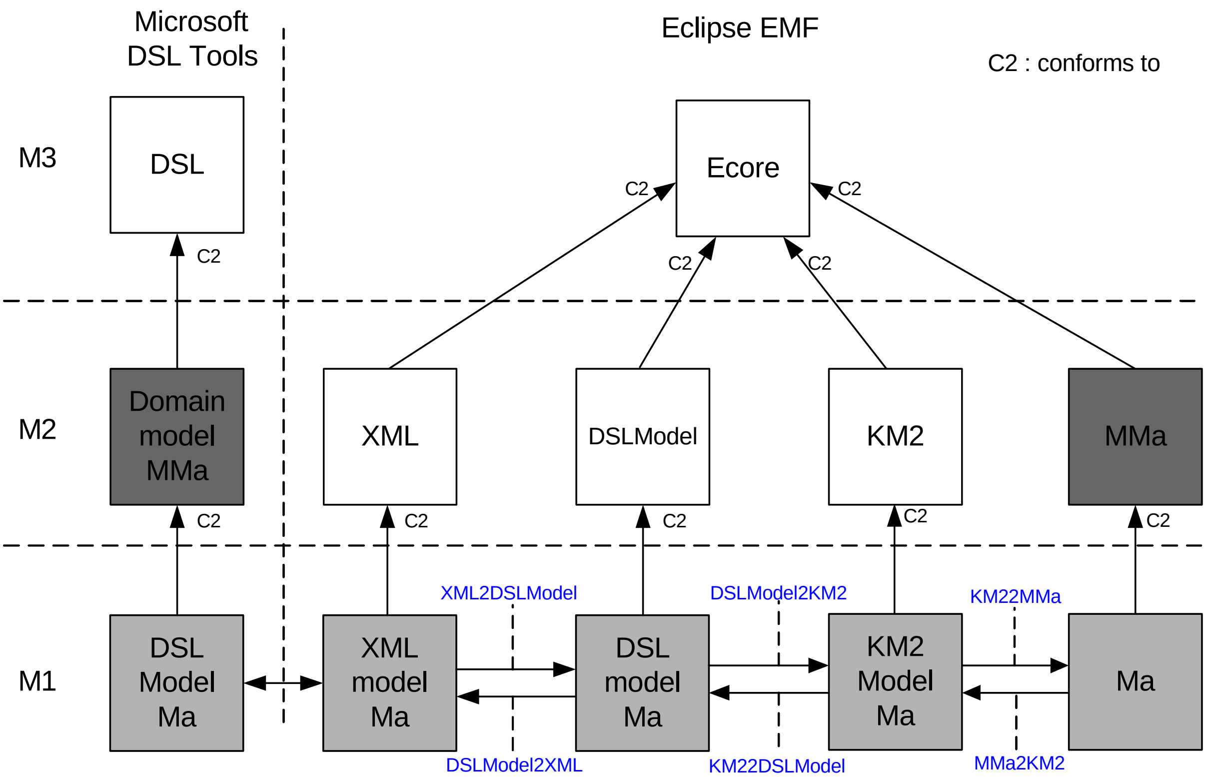


Figure 19. Model bridge overview

The first step consists in injecting the DSL model file to an XML model like for the *.dsldm* file in the metamodel bridge.

The second step consists in transforming the XML model into a model conforming to the models DSL metamodel (DSLModel). This one was defined to match Microsoft's schema as closely as possible. The transformations are XML2DSLModel and DSLModel2XML.

The third step relies on KM2 (a model representation) which is used like KM3 as a pivot between technologies. The transformations are DSLModel2KM2 and KM22DSLModel.

The last step takes a KM2 model and the metamodel MMa defined under EMF as inputs and as output a model that conforms to MMa. This is made by two transformations, the first one takes MMa has input and generates the second ATL transformation with the specific rules for MMa, defining by this mean a generic method.

3.2 Microsoft DSL models

3.2.1 Models in Microsoft DSL Tools

From a DSL domain model, Visual Studio creates a specific editor for the models. A model is stored in an *.xml* file, corresponding to an XML Schema. The Ms/DSL models can be represented by a unique metamodel, close to the XML Schema.

The models files can be found in directory Visual Studio 2005\Projects*ProjectName*\Debugging *ProjectName* Debugging.

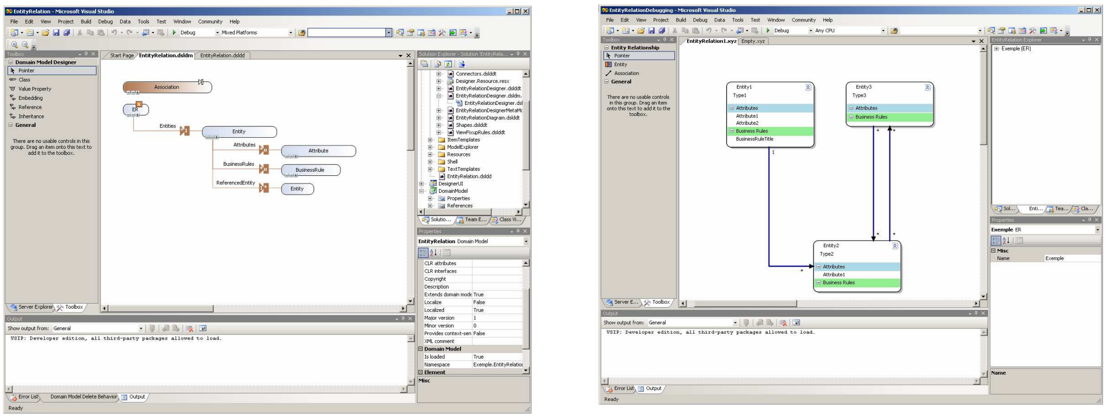


Figure 20. To the left a domain model and to the right a model from this domain model

A model is composed by model elements and links between them. A model element is an instance of a domain model classifier. This latter is known by the attribute Type of model element. Links correspond to relationships in domain model.

Figure 21 provides *.xml* schema viewed as a class diagram. The models DSL metamodel considered in this study has been created this class diagram.

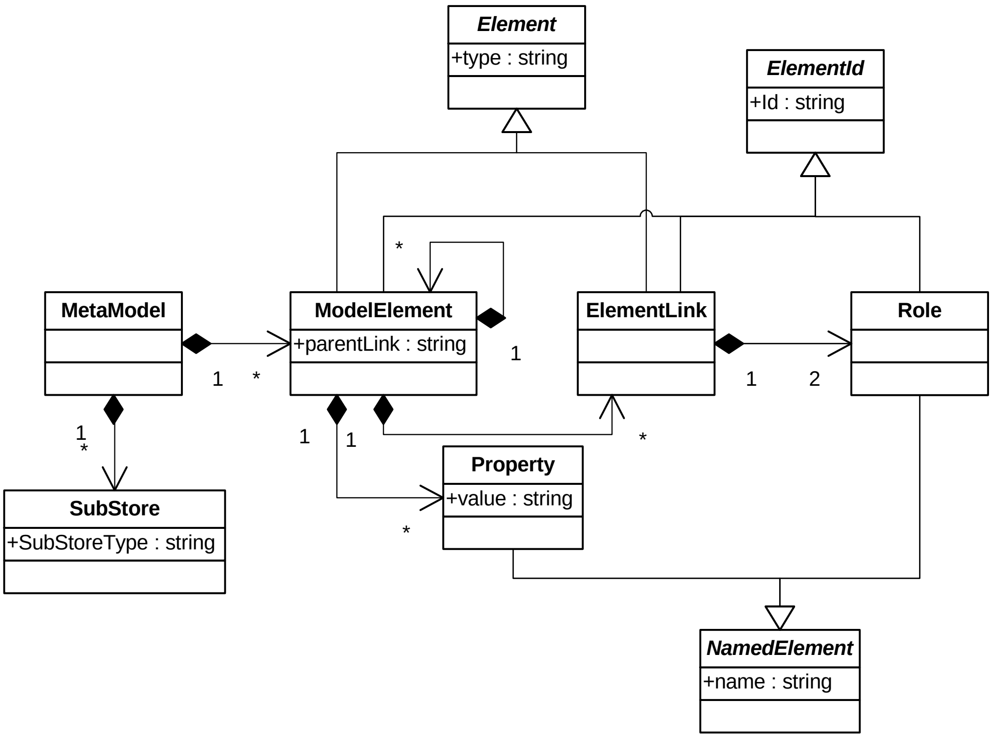


Figure 21. XML Schema for MS DSL models representation

3.2.2 Models DSL metamodel

The considered models DSL metamodel is provided in Figure 22.

Some concepts of this metamodel differ from the XML schema:

- The root class is named Model instead of MetaModel because it applies on models and may be confused.
- In the XML schema, a ModelElement may be composed of ModelElement. This case corresponds to an embedding in the domain model. The type of relationship is known by the attribute parentLink in ModelElement. The model elements which have this attribute are those which are in the composition and not the one which contains them.
  - In the metamodel, this is changed by adding a class EmbeddingLink between the container and the contained ModelElement.
- The ElementLink in the XML schema are contained by ModelElement. An ElementLink corresponds to a reference in the domain model. This latter is known by the attribute Type.
  - In the metamodel, an ElementLink is named a ReferenceLink to be close to the domain model.
- A ReferenceLink has two roles. In XML schema, a role has an attribute Id that corresponds to a ModelElement Id. As a consequence, in the metamodel a role has a reference to a ModelElement.
- The first role referenced the ModelElement that has the ElementLink and the second referenced the ModelElement to the opposite.

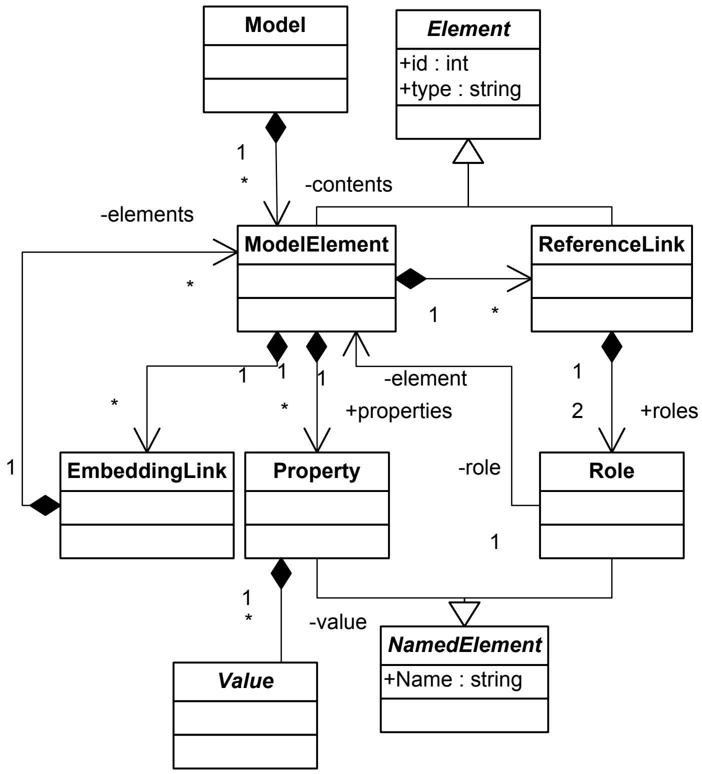


Figure 22. The models DSL metamodel

3.3 Models in Eclipse EMF

Like in Visual Studio DSL Tools, models can be created from a metamodel defined under Eclipse EMF with a specific models editor. The difference is that the models serialization is here done by applying the metamodel's structure, so the model explicitly conforms to its metamodel.

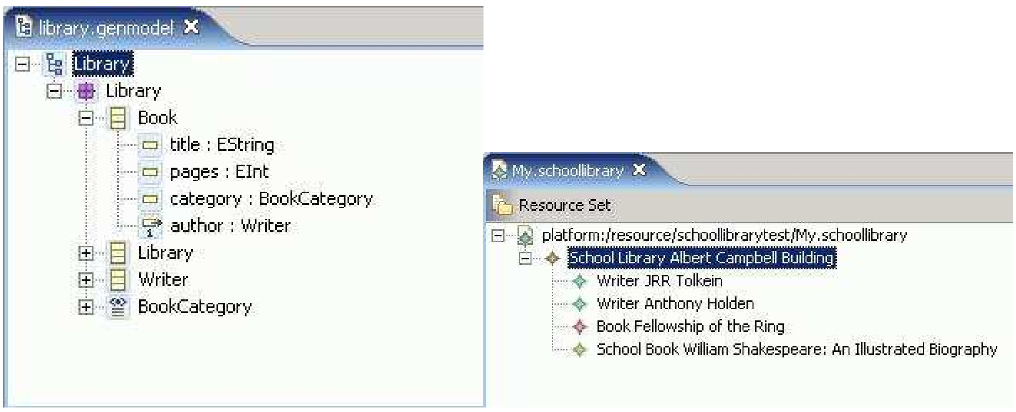


Figure 23. In left a metamodel defined under EMF and in right a model example

At this stage, it is possible to note that the transformation from the models DSL metamodel to the metamodel defined under EMF raises a problem, since the metamodel defined under EMF is variable. The solution is to have a transformation that is generic for any metamodel.

3.4 KM2 metamodel

As shown in Figure 19, the transformation chain uses an intermediary metamodel: KM2 (see Figure 24). This one is used like a pivot between models technologies like KM3 is for the metamodels. A KM2 model is composed of model elements. A model element has an attribute name which is the type of the corresponding class in the metamodel. This one is known by the attribute metamodel in class Model. A model element has also properties which correspond to attributes and references in the metamodel.

A property contains a value that can be of different types:

- A PrimitiveVal corresponds to a simple type attribute (String, Integer, Boolean, Double).
- A ModelElementRefVal corresponds to a reference in the metamodel.
- A ModelElementVal corresponds to a composition in the metamodel.
- A SetVal contains value, it is used to represents attributes or references with cardinality > 1.

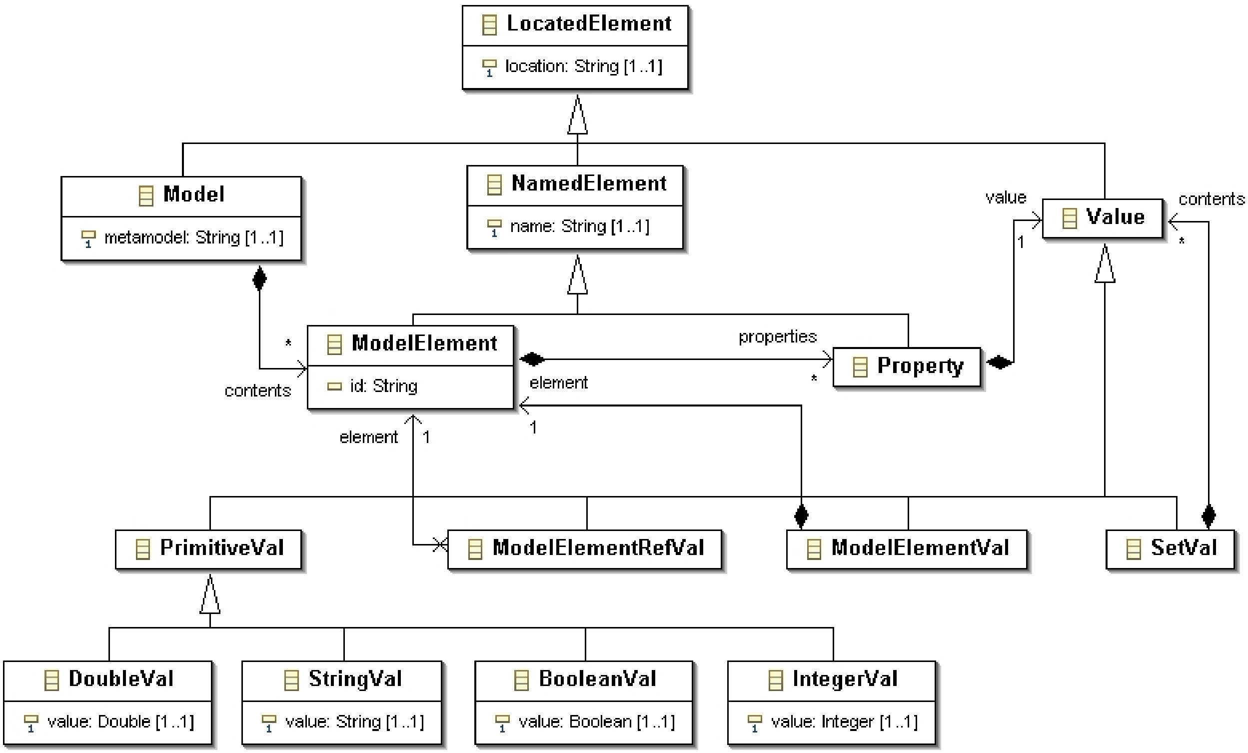


Figure 24. KM2 metamodel

Figure 25 describes an instantiation example containing a metamodel which has two classes named Class and Attribute and its corresponding KM2 model. A ModelElement corresponds to a Class type, this one has a property corresponding to the attribute name and a property attributes corresponding to the composition in the metamodel. This property has a SetVal value because attributes has multiple cardinality.

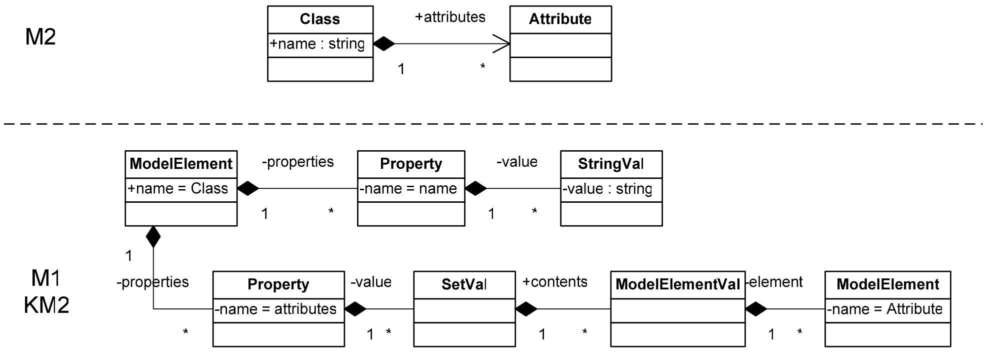


Figure 25. A KM2 instantiation example

3.5 ATL Transformations

This section is dedicated to the description of the ATL transformations that are part of the models bridge. The first transformation is XML2DSLModel that takes as input the XML model of a DSL model file and produces a DSLModel model. The second transformation is DSLModel2KM2 that generates a KM2 model from this DSLModel model. Thirdly, the transformation KM22MMa generates a model conforming to a metamodel MMa defined under EMF.

3.5.1 XML to DSLModel

3.5.1.1 Principle

DSL models are serialized with a unique XML schema (it is the same for any models). A metamodel, named DSLModel, that includes essential information from the *.xml* file has been designed. Like for the metamodel bridge, the *.xml* file is injected into an XML model. From this point, the first ATL transformation (XML2DSLModel) can be used to get a DSLModel.

The model file contains some information about model representation that is not taken into account in the metamodel. The only XML elements that are recognized are:

- om:MetaModel is mapped to a DSLModel!Model;
- om:ModelElement is mapped to a DSLModel!ModelElement;
- om:Property is mapped to a DSLModel!Property;
- om:ElementLink is mapped to a DSLModel!ReferenceLink;
- om:Role is mapped to a DSLModel!Role.

In the *.xml* file, the XML element om:Property represents a metamodel's class (or relationship) value property. It has a name and a value. It is represented in the same way in DSLModel.

The XML element om:ElementLink is contained by a om:ModelElement. It represents references between model elements. An element link contains two roles, the first one referred to the containing model element (source) and the second referred to the referring model element (type). A model element is associated to element links; they have the same attribute Id. Because, in DSL, reference relationships are viewed as classes, they can have attributes, so associating a model element to element links enables to have properties that represent the relationship's attributes. In DSLModel, these model elements are of type ModelElementLink and have reference links to the element links they are associated with.

In the *.xml* file, a om:ModelElement can have children with name om:ModelElement, it corresponds to an embedding relationship in the domain model. In this case, the children have an attribute parentLink which is a string containing the name of the embedding relationship. In DSLModel, this is represented this by a class EmbeddingLink which contains ModelElement (the children in the *.xml* file), and this class EmbeddingLink is contained by a ModelElement (the parent in the *.xml* file).

Creating an EmbeddingLink is achieved as follows: from a ModelELement, a set of String that contains the ParentLink is created with the helper getParentLinks(). After that, a Sequence of Sequence of XML!Element is created by placing in a Sequence the XML!Element that have the same attribute ParentLink with the helper SequenceOfSequence(). Then, an EmbeddingLink is created for each distinct element in the Set, and then the Sequence of Sequence named allchilds is placed into elements. See the code below for further details.

```
using {
                      allEmbeddingLinks : Set(String) =
                                 e.getParentLinks()->asSet();
                      allchilds : Sequence (Sequence(XML!Element)) =
                                 e.SequenceOfSequence(allEmbeddingLinks);
}
to
       me : DSLModel!ModelElement (
              type <- thisModule.subNamespace(e.getAttrVal('Type')),
              id <- e.getAttrVal('Id'),
              properties <- e.children->select(c | c.oclIsTypeOf(XML!Element)
                                        and c.name='om:Property'),
              embeddinglinks <- Sequence {p},
              referencelinks <- e.children->select(l | l.oclIsTypeOf(XML!Element)
                                        and l.name = 'om:ElementLink')
       ),
       p : distinct DSLModel!EmbeddingLink foreach ( pl in allEmbeddingLinks ) (
              name <- pl,
              elements <- allchilds
       )
```

In this code, it is possible to notice that, for the attribute type of the created ModelElement, we use the helper subNamespace because in the *.xml* file the type of a ModelElement is appended to the namespace of the domain model (the namespace is cut and the type kept).

As introduced previously, some XML elements are omitted. These elements are those om:ModelElement whose type ends by **Diagram** (called elementToAvoid), and the om:ElementLink with type **Microsoft.VisualStudio.Modeling.SubjectHasPresentation**. To this end, filters are defined within the rules. The code below provides a filter example: among XML!Element, those with attribute name equals to om:ModelElement are first filtered. Then, the filter verifies that the element is not an elementToAvoid or a child of this latter. Finally, it checks that the element is not a ModelElement used to describe an ElementLink.

```
rule ModelElement {
       from
       e : XML!Element (
                      if e.name='om:ModelElement'
                             then if e = thisModule.elementToAvoid()
                                    then false
                                           else if
e.isChildrenOf(thisModule.elementToAvoid())
                                                         then false
                                                         else not e.isElementLink()
                                                  endif
                                    endif
                             else false
                      endif
              )
```

3.5.1.2 Limitations

This transformation contains some limitations:

- The enumerations are not recognized;
- The simple types for a property are limited to Integer, String and Boolean.

3.5.1.3 Use

The aim of this part is to explain the use of this ATL transformation. Running this transformation requires an example of DSL model in *.xml* format. This file has to be injected into an XML model. The corresponding domain model, which is obtained after running the XML2DSL presented in the metamodel bridge is also required. Finaly, the transformation requires three metamodels: XML, DSL and DSLModel in Ecore format. Figure 26 provides a screenshot of the transformation configuration.

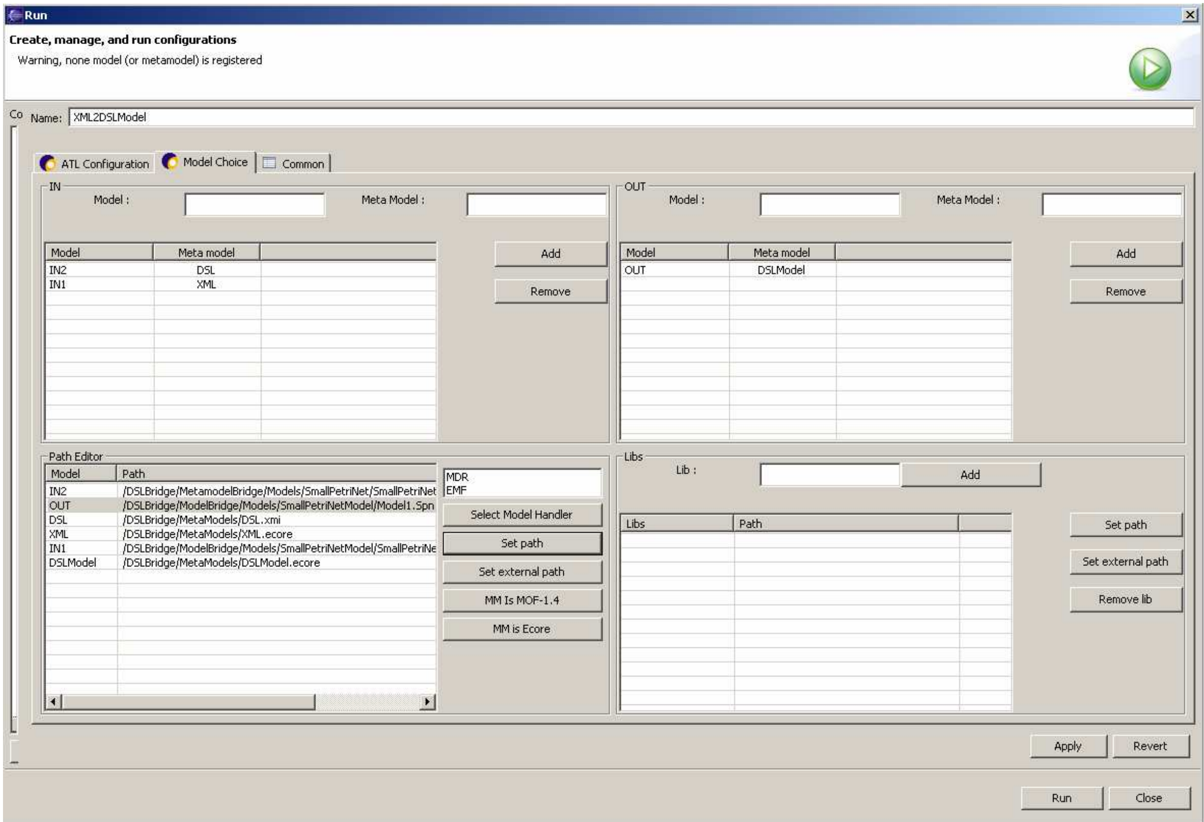


Figure 26. Configuration of the XML2DSLModel transformation

There are two input (XML and DSL) and one output (DSLModel) metamodels. In Path Editor you place in XML the path of the XML metamodel, you do the same for DSL and DSLModel. The field IN1 contains the path of the xml model, the field IN2 contains the path of the DSL model (it is the domain model that goes with the model file in xml) and the field OUT, the path for the results.

3.5.2 DSLModel to KM2

3.5.2.1 Principle

KM2 is a representation of models independent of their metamodel. It is quite the same as DSLModel that is also independent of the domain model because it is close to the XML schema.

Some elements are equivalent between both metamodels. A DSLModel!Model becomes a KM2!Model, a DSLModel!ModelElement becomes a KM2!ModelElement. The attribute type of DSLModel!ModelElement becomes the attribute name for the KM2!ModelElement.

Attributes and references correspond to properties in KM2. Whereas, they are separated in DSLModel, attributes are properties, reference relationships are ReferenceLink and embedding relationships are represented by EmbeddingLink. As a consequence, these three elements are taken for a DSLModel!ModelElement and put into KM2!Property.

```
rule ModelElement {
	from
		me : DSLModel!ModelElement (
			me.oclIsTypeOf(DSLModel!ModelElement)
		)
	to
		kme : KM2!ModelElement (
			name <- me.type,
			id <- me.id,
			properties <- Sequence {
				me.properties->asSequence(), -- Attributes
				me.embeddinglinks->asSequence(), -- Compositions
				me.getReferences() -- References
			})}
```

Creating a KM2!Property from a DSLModel!ReferenceLink si achieved by selecting the last DSLModel!Role (the type role) with the helper getReferences().

A KM2 property may have a simple value or a Set value, this one is created if the type of the ReferenceLink or the EmbeddingLink that is used to create the property corresponds to a Relationship in the DSL model that has a multiplicity > 1. The following helpers are used for this purpose:

For an EmbeddingLink:

```
-- This helper returns the role corresponding to the embedding link
helper context DSLModel!EmbeddingLink def:getRole() : DSL!Role =
	let a : DSL!Relationship =
		DSL!Relationship.allInstances()
		->select( e | e.name = self.name )->first()
	in a.roles->select( e | e.source.name = self.owner.type )->first();
```

It is called as follows in rules:

```
from p : DSLModel!EmbeddingLink (
	p.getRole().max = 0 or p.getRole().max > 1
)
```

For a ReferenceLink, the DSLModel!Role used to make the property is selected:

```
-- This helper returns a boolean which indicates if the relationship corresponding to the role
has a max cardinality > 1
helper context DSLModel!Role def: isMultiple() : Boolean =
	let a : DSL!Role = DSL!Role.allInstances()
		->select ( e | e.relation.name = self.owner.type
			and e.name = self.name )->first()
	in if ( a.max = 1 ) then false else true endif;
```

3.5.2.2 Limitations

In the metamodel bridge, if a DSL relationship has attributes or supertypes, it becomes a class in KM3. As a consequence, when passing from a DSLModel to KM2, it is necessary to check that the KM2!ModelElement with type of the relationship that becomes a class in KM3 are linked. These ModelElement can be easily recognized because, in DSLModel, they are ModelElementLink. So a ModelElementLink with properties becomes a ModelElement in KM2. Current implementation does not handle this case, so it is recommended not to use DSL domain models that does not contain relationships with properties.

3.5.2.3 Use

Running this ATL transformation requires two models and their metamodels:

- The first model is the previous output from XML2DSLModel transformation and the DSLModel metamodel.
- The second model is the corresponding DSL model like in the previous transformation and the DSL metamodel.
- The KM2 metamodel for output.

Figure 27 provides a screenshot of the transformation configuration.

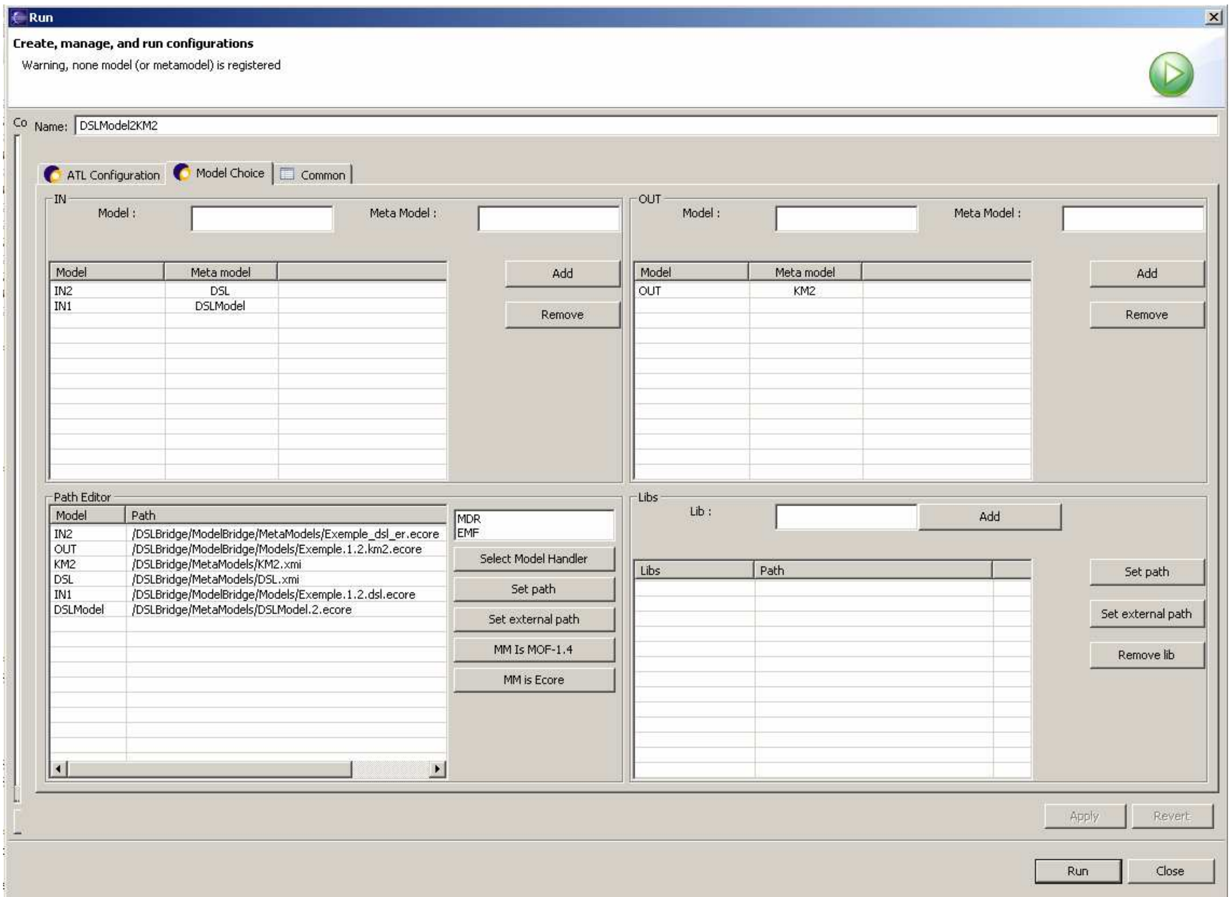


Figure 27. Screenshot of the DSLModel2KM2 configuration

There are two input (DSLModel and DSL) and one output (KM2) metamodels. In Path Editor, the path of the DSL, DSLModel and KM2 metamodel are respectively associated with DSL, DSLModel and KM2. The field IN1 contains the path of the DSLModel model created with the XML2DSLModel transformation, the field IN2 contains the path of the DSLModel and the field OUT, the path for the results.

3.5.3 KM2 to DSLModel

3.5.3.1 Principle

This transformation is the inverse of DSLModel2KM2, it takes a KM2 model as input and produces a DSLModel model. As noticed previously, KM2 and DSLModel have many similarities, so the major difficulty is here to recognized whether KM2!Property will be a DSLModel!Property, DSLModel!EmbeddingLink or a DSLModel!ReferenceLink. Another difficulty is to make for DSLModel!Reference the corresponding DSLModel!ModelElementLink that corresponds to a relationship in the metamodel (the DSL one).

The first problem is addressed using the three following helpers:

- getProperties() to recognize which KM2!Property will be DSLModel!Property;
- getPropertyContainer() to recognize which KM2!Property will be DSLModel!EmbeddingLink;
- getPropertyReference() to recognize which KM2!Property will be DSLModel!ReferenceLink.

Those helpers are shown below, the first is getProperties and the other one is getPropertyReference. The last helper is shown here because it is the same as the latter except that it recognizes composition.

```
-- This helper returns a Sequence of KM2!Property that corresponds to DSLModel!Property
-- From the name (type) of the CONTEXT it makes a Sequence of KM3!Attribute present in
-- the metamodel and then select in the CONTEXT properties the corresponding KM2!Property.
-- CONTEXT: KM2!ModelElement
-- RETURN: Sequence(KM2!Property)
helper context KM2!ModelElement def: getProperties() : Sequence(KM2!Property) =
let a : Sequence(KM3!Attribute) =
        KM3!Class.allInstances()->select( c | c.name = self.name )
        ->collect(p | p.structuralFeatures)->flatten()
        ->select( a | a.oclIsTypeOf(KM3!Attribute))->asSequence()
in
        a->iterate(e;acc : Sequence(KM2!Property) = Sequence{} |
        if      self.properties->select( p | p.name = e.name )->first().oclIsUndefined()
                then acc
                else acc -> including(self.properties->select( p | p.name = e.name )->first())
        endif);

-- This helper returns a Sequence of KM2!Property that corresponds to references in the
metamodel
-- CONTEXT: KM2!ModelElement
-- RETURN: Sequence(KM2!Property)
helper context KM2!ModelElement def: getPropertyReference() : Sequence(KM2!Property) =
        let a : Sequence(KM3!Reference) =
                KM3!Class.allInstances()->select( c | c.name = self.name )->collect(p |
p.structuralFeatures)
                ->flatten()->select( a | a.oclIsTypeOf(KM3!Reference) )->select(b | not
b.isContainer )->asSequence()
        in
                a->iterate(e;acc : Sequence(KM2!Property) = Sequence{} |
                        if      self.properties->select( p | p.name = e.name )-
>first().oclIsUndefined()
                                then acc
                                else acc -> including(self.properties->select( p | p.name =
e.name )->first())
                        endif);
```

3.5.3.2 Rules specifications

Here is a description of the rule composing the transformation:

- Rule **Model**: From a KM2!Model, a DSLModel!Model is created.
    - The metamodel in KM2 corresponds to the domain model.
    - They got the same contents.
- Rule **ModelElement**: From a KM2!ModelElement, a DSLModel!ModelElement is created.
    - The type is the ModelElement name in KM2.
    - Their Id corresponds.
    - The ModelElement properties are selected with getProperties helper.
    - The ModelElement embeddingLinks are selected with the getPropertyContainer helper.
    - The ModelElement referenceLinks are selected with the getPropertyReference helper.
- Rule **ReferenceLink**: This one is used to create ReferenceLink from a KM2!Property that corresponds to a reference in the metamodel. A KM2!Property may contain several ModelElementRefVal (with a SetVal). However, in DSLModel, a ReferenceLink contains two roles and a DSLModel!Role can refer to only one ModelElement, so a ReferenceLink has to be created for each KM2!ModelElementRefVal in the KM2!Property. This is achieved by using the helper getRefVal() that returns a Sequence of KM2!ModelElementRefVal and for each element in this sequence a DSLModel!ReferenceLink is created.
    - The ModelElementRefVal from the sequence is also used to create the roles.
- Rule **Role**: From a KM2!ModelElementRefVal, two DSLModel!Role are created (a DSLModel!ReferenceLInk contains two roles). In order to create a DSLModel!Role, it is required to know its name and what element it refers to. This latter is the one in KM2!ModelElementRefVal, and the name is the name of the KM2!Property that got the ModelElementRefVal. In order to create the second role, the metamodel (the DSL model) has to be searched to retrieve the opposite by using the helper getOpposite().
    - The referred element in the opposite role is the KM2!ModelElement that have the KM2!Property used to create the roles.
- Rule **EmbeddingLink**: From a KM2!Property which is a container (a composition), a DSLModel!EmbeddingLink is created.
    - The name is the name of the DSL!Relationship from the domain model.
    - The elements are the ones in the values of the KM2!Property.
- Rule **Property**: From a KM2!Property that corresponds to an Attribute in the metamodel, a DSLModel!Property is created.
    - Their names correspond.
    - Their values correspond.

3.5.3.3 Limitations

This version does not create DSLModel!ModelElementLink.

3.5.3.4 Use

This transformation requires three metamodels and their models in input and one in output.

- The three inputs are:
  - The DSL metamodel and the model example;
  - The KM3 metamodel and the model example;
  - The KM2 metamodel and the model example.
- The output is the DSLModel metamodel.

Figure 28 provides a screenshot of the transformation configuration: there are three input (DSL, KM3 and KM2) and one output (DSLModel) metamodels. In Path Editor, the path of the DSL, KM3, KM2 and DSLModel metamodels are respectively associated with DSL, KM3, KM2 and DSLModel. The field IN1 contains the path of the KM2 model example (the one created with DSLMode2KM2 can be used), the field IN2 contains the path of the KM3 model, the field IN3 contains the DSL model, and the field OUT, the path for the results.

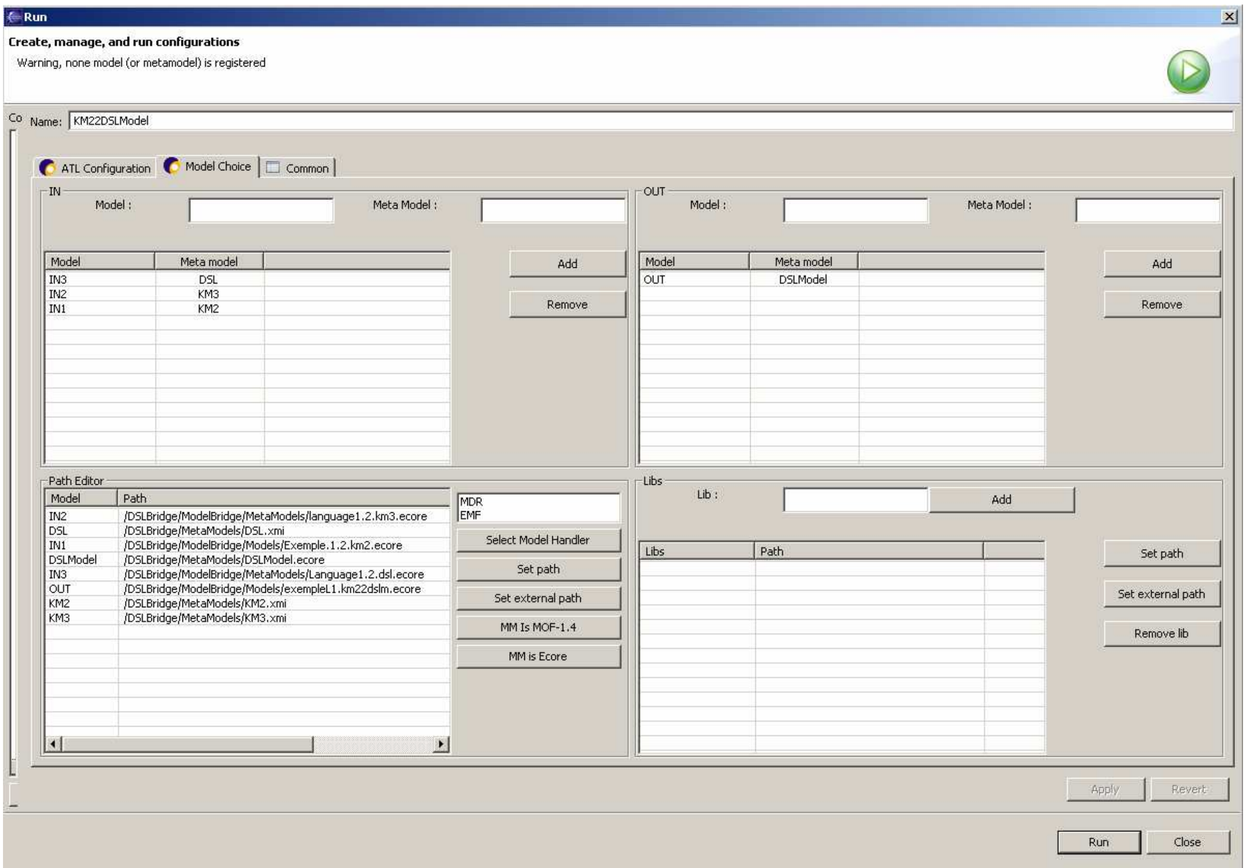


Figure 28. Configuration for KM22DSLModel

3.5.4 KM2 to Metamodel

3.5.4.1 Principle

The last transformation must lead to a model that directly conforms to its metamodel defined under EMF. This last is variable, so a transformation written between KM2 and the metamodel will work for only one metamodel. As a consequence, the transformation must be decomposed in two steps, as shown in Figure 29.

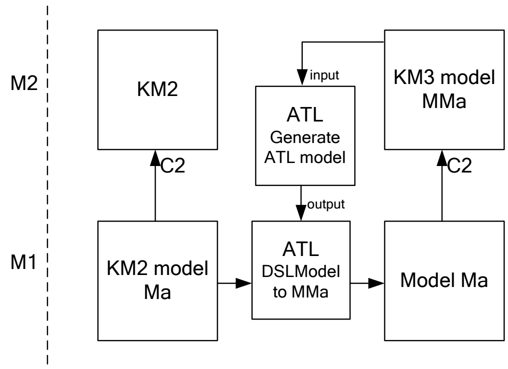
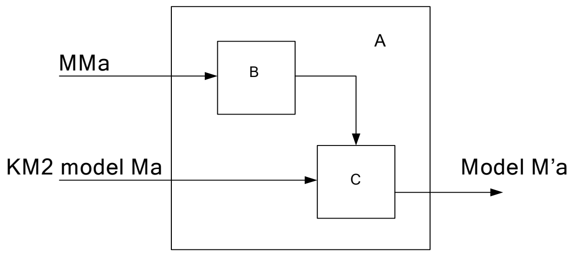


Figure 29. Overview of KM22Metamodel

The first step is to write a transformation that takes in input a model in KM3, it is the metamodel that corresponds to the models, and generates an ATL transformation which contains the necessary rules corresponding to the metamodel for making the models transformation. This solution is generic for any metamodel.

Figure 30 illustrates the process for obtaining this. From a metamodel MMa, the transformation B creates an ATL model C (i.e. a transformation). This one takes in input a KM2 model and outputs a model Ma directly conforms to MMa.



M'a = A(Ma,MMa) = C(Ma)
C = B(MMa)

Figure 30. Process to generate a transformation

3.5.4.2 KM32ATL_KM22MM

The transformation KM32ATL-KM22MM takes in input a KM3 model that is the model's metamodel to be transformed, and generates an ATL model that can be injected into an ATL file. This one is a transformation that takes in input a KM2 model and produces a model that explicitly conforms to the input's metamodel in KM3.

The principle is to create an ATL model from the input KM3 model. To this end, an ATL!Module is created from the KM3!Metamodel (this is achieved by the rule Module). An ATL!Module has a name, made with the String KM2 attached to the name of the KM3!Package, is composed of inModels (that is

KM2), outModels (that is the KM3!Package), elements (that correspond to theKM3!Class) and the library KM2Tools (which contains the helpers for the generated transformation).

For each KM3!Class from the input model KM3, a corresponding rule is created. This rule has an inPattern of type KM2!ModelElement with a filter with the name of the KM3!Class (that is also the name of the KM2!ModelElement). The type of the outPattern is the one of the KM3!Class. This outPattern contains bindings that correspond to the attributes and references of the KM3!Class.

For each KM3!Attribute and KM3!Reference, a binding containing an ATL!OperationCallExp to an helper from the library KM2Tools is created.

3.5.4.3 *Use*

Running this transformation requires the KM3 model's metamodel in Ecore format, as well as the ATL metamodel in MDR (MOF 1.4) format. The result of this transformation has to be serialized by using TCS. This step produces an ATL file that corresponds to the transformation KM2 to Metamodel. Figure 31 shows the transformation configuration.

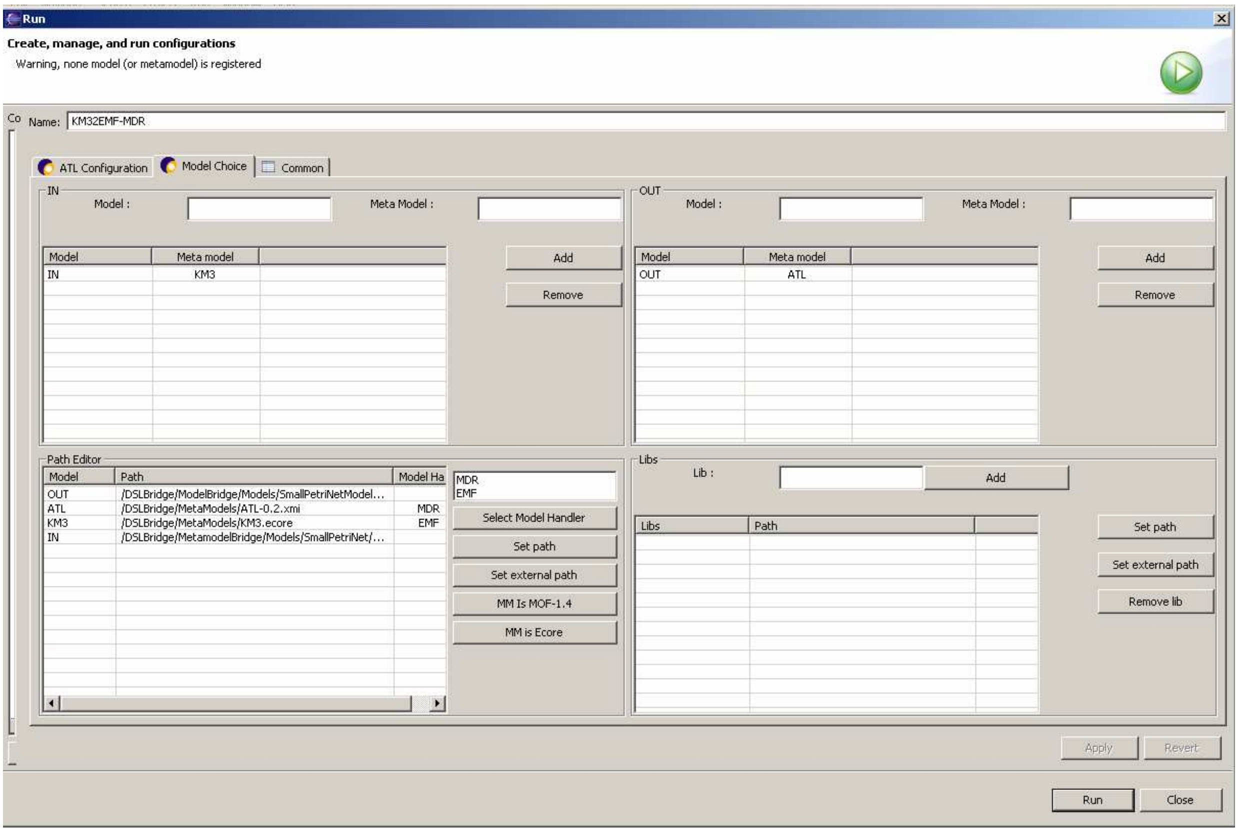


Figure 31. KM32ATL_KM22MM configuration

In this configuration, the KM3 metamodel, a KM3 model, and the ATL metamodel in MDR format are required. In the path editor, the path of the KM3 metamodel is associated with KM3 and the field IN contains the path of the KM3 model. The field ATL contains the path of the ATL metamodel (be care to select model handler MDR), and the field OUT, the path for the result.

3.6 Example: Small Petri Net

This section describes a use case of the model bridge. In this scope, a simple Petri Net metamodel has been considered. However, due to the limitations of the model bridge, this metamodel is not the same as in the metamodel bridge example: it does not define any DSL relationship containing a property since this one would be transformed into a KM3 class and this case is not implemented (see Section 3.5.2.2.).

The domain model used for this example is presented in Figure 32. This domain model only contains Place and Transition. A Token is represented by an Integer property in Place.

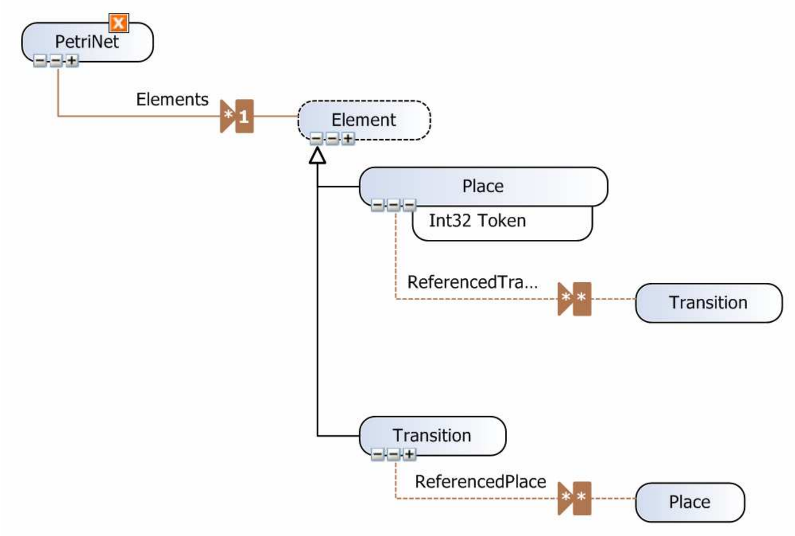


Figure 32. Small Petri net domain model

A simple Petri net example that contains three places and one transition, with Place1 containing two tokens, has been designed using the DSL Tools model editor.

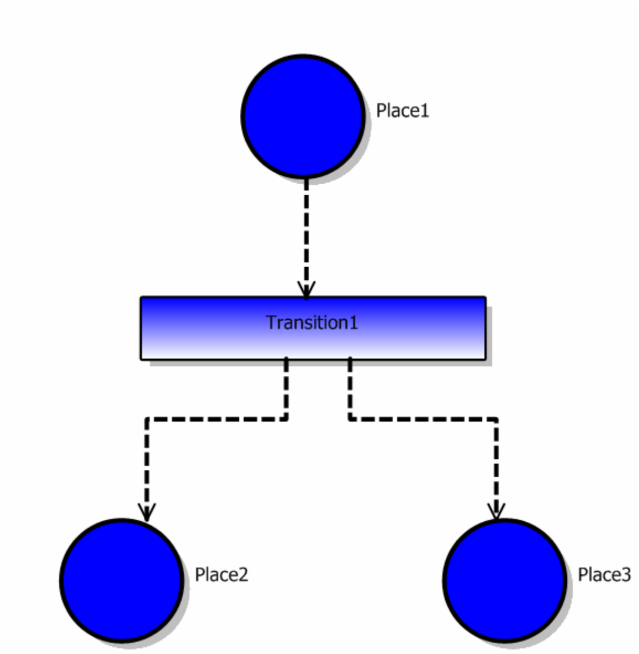


Figure 33. A simple Petri net model *SmallPetriNet1.xml*

The first step is to use the metamodel bridge to transform the domain model into a KM3 model and then inject this one into an Ecore model (see Section 2.5).

The file *SmallPetriNet1.xmi* is then used with the XML2DSLModel transformation (see Section 3.5.1.3) to produce a DSLModel model (*Model1.Spn.dslmodel.ecore*). This model now has to be transformed into a KM2 model by using the DSLModel2KM2 transformation (see Section 3.5.2.3): this generated result is the file *Model1.Spn.km2.ecore*.

Once a KM2 model has been built, it is possible to use the KM22SmallPetriNet transformation. This one is generated by the KM32ATL_KM22MM transformation. Figure 34 provides a screenshot of the configuration of the KM22SmallPetriNet transformation.

To use this transformation, the *SmallPetriNet.ecore* file, which corresponds to the SmallPetriNet metamodel in Ecore format, is required. It is obtained by using the Ecore injector on the *SmallPetriNet.km3* file. The injector is available with ATL Development Tools (ADT) [10]. The KM2 metamodel and the KM2 model *Model1.Spn.km2.ecore* are also required.

In Path Editor, the KM2 field contains the path of KM2 metamodel, and the field IN, the path of the file *Model1.Spn.km2.ecore*. The metamodel in output is SmallPetriNet (corresponding to *SmallPetriNet.ecore*). The library KM2Tools path also requires to be filled.

The result is the file *Model1.Spn.ecore*.

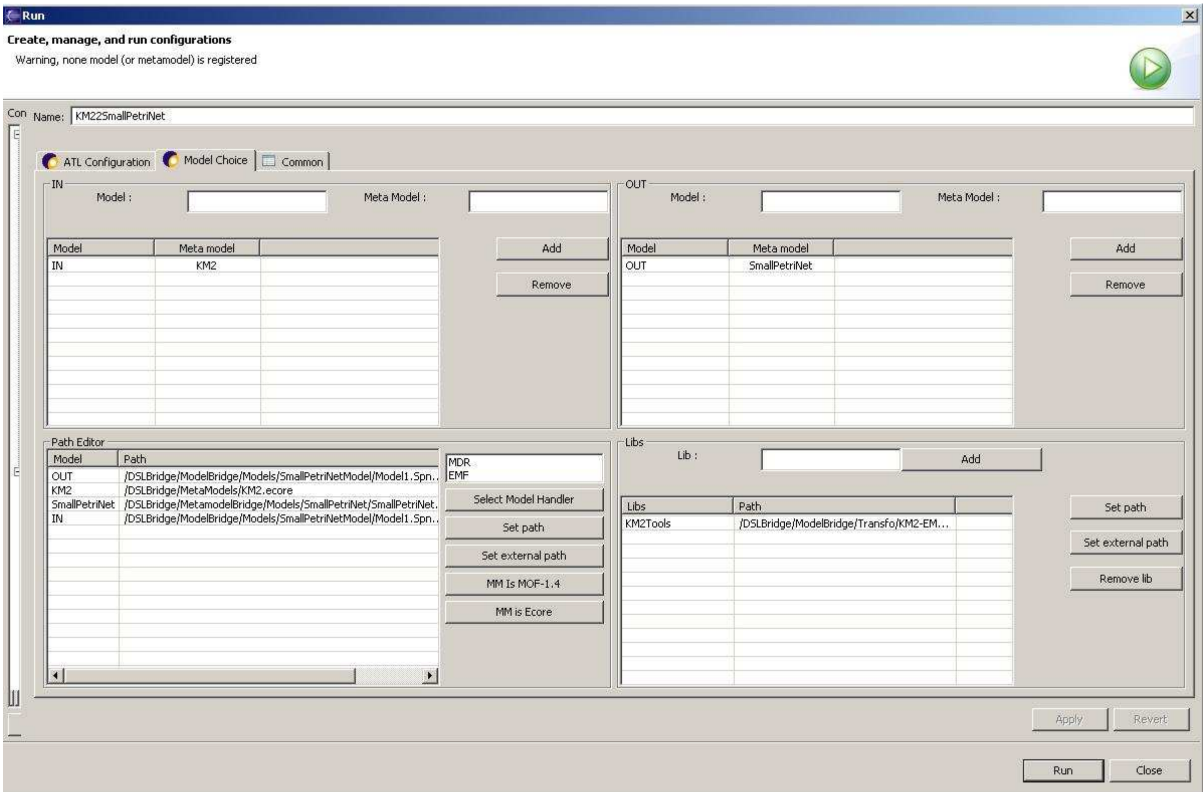


Figure 34. Configuration for KM22SmallPetriNet

4 Extension

This section provides explanations on how to keep the information that is lost during the transformation in the metamodel bridge from a DSL model into a KM3 model.

In a domain model, classes and relationships are placed at the same level. Relationships can be viewed as association classes like in UML. Figure 35 represents a simple domain model example with a relationship BReferencesC between a ConceptB class and a ConceptC. This relationship has an attribute Property of type String. Figure 36 shows this domain model in UML representation. The relationship is viewed as an association class with an attribute Property.

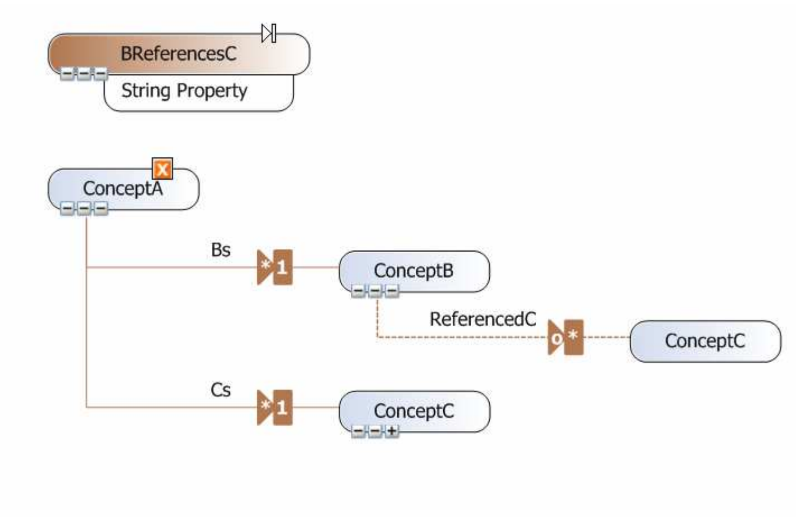


Figure 35. Simple domain model example

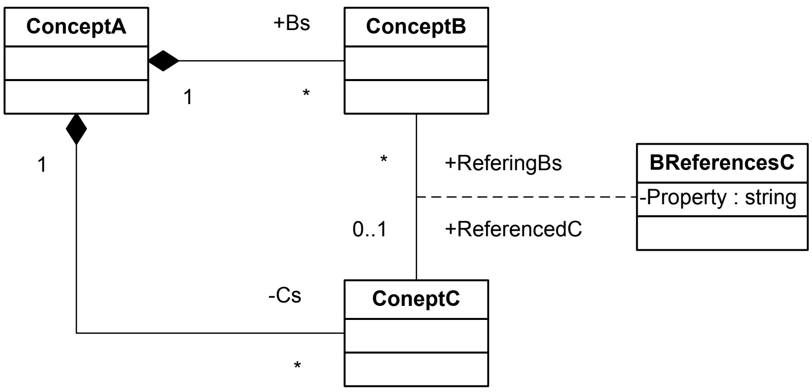


Figure 36. The same domain model view with UML representation

During transformation into KM3, the implemented solution is to transform relationships that have class characteristics (inheritance, properties) into classes. The obtained result is provided in Figure 37.

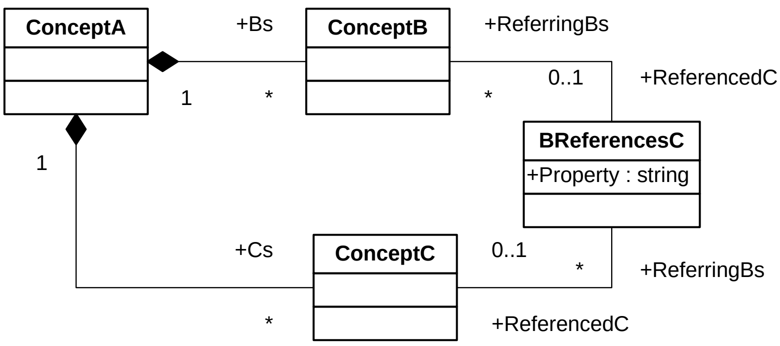


Figure 37. The result after DSL2KM3 transformation

One loses the information that encodes that BReferencesC is a relationship and not a class. Saving these data may be achieved by enlarging the KM3 so that it takes this information into account. However, in this case, this will lead to have a large metamodel that be not use correctly.

Another solution would be to keep this information into a model independent of metamodel DSL and metamodel KM3. This model must keep information lost during the DSL to KM3 transformation and allows finding them during the KM3 to DSL transformation.

Such a model can be called KM3Annotations, and has to conform to a metamodel. It must be able to store the information contained in the metamodel source (here DSL) which is not retranscribed in KM3.

DSL = KM3 + KM3Annotations

In the scope of the considered example, the KM3Annotations model must contain the following information:

- BReferencesC is a relationship.
- BReferencesC links a class of type ConceptB to a class of type ConceptC.
- BReferencesC has two association ends: ReferringBs and ReferencedC with their cardinality.

This information has to be stored in KM3Annotations conforming to its metamodel (it is not a text format representation). However, it would be interesting to be able to specify this information manually in some cases: for instance, when starting with a KM3 model that has to be transformed into a DSL model, it should be possible to define what class has to be a relationship.

Implementing this extension therefore requires defining a KM3Annotations metamodel which contains information that cannot be present in the KM3, thus allowing to exchange models between various technical spaces without losing the intent of the initial representation.

5 References

[1] Microsoft DSL Tools web site, http://lab.msdn.microsoft.com/vs2005/teamsystem/Workshop/DSLTools/.

[2] The Eclipse Modeling Framework (EMF), http://www.eclipse.org/emf/.

[3] The ATLAS Transformation Language (ATL), http://www.eclipse.org/gmt.

[4] The Eclipse project, http://www.eclipse.org/.

[5] KM3 User Manual. The Eclipse Generative Model Transformer (GMT) project, http://eclipse.org/gmt/.

[6] F. Budinsky, and D. Steinberg, and E. Merks, and R. Ellersick, and T. J. Grose: Eclipse Modeling Framework, Chapter 5 *Ecore Modeling Concepts*, Addison-Wesley.

[7] OMG/MOF. *Meta Object Facility (MOF)*, v2.0. OMG Document formal/03-10-04, April 2004. Available from www.omg.org.

[8] OMG/MOF. *Meta Object Facility (MOF)*, v1.4. OMG Document formal/02-04-03, April 2002. Available from www.omg.org.

[9] The Omondo EclipseUML plugin. Available at http://www.omondo.com/download/index.html.

[10] The ATL Development Tools (ADT). The Eclipse Generative Model Transformer (GMT) project, http://eclipse.org/gmt/.

Appendix A The XML metamodel in KM3 format

```
package XML {
	abstract class Node {
		attribute startLine[0-1] : Integer;
		attribute startColumn[0-1] : Integer;
		attribute endLine[0-1] : Integer;
		attribute endColumn[0-1] : Integer;
		attribute name : String;
		attribute value : String;
		reference parent[0-1] : Element oppositeOf children;

	}

	class Attribute extends Node {

	}

	class Text extends Node {

	}

	class Element extends Node {
		reference children[*] ordered container : Node oppositeOf parent;

	}

	class Root extends Element {

	}
}
```

Appendix B The KM3 metamodel in KM3 format

```
package KM3{
	abstract class LocatedElement {
		attribute location : String;
	}

	abstract class ModelElement extends LocatedElement {
		attribute name : String;
		reference "package" : Package oppositeOf contents;
	}

	class Classifier extends ModelElement {}

	class DataType extends Classifier {}

	class Enumeration extends Classifier {
		reference literals[*] ordered container : EnumLiteral;
	}

	class EnumLiteral extends ModelElement {}

	class Class extends Classifier {
		attribute isAbstract : Boolean;
		reference supertypes[*] : Class;
		reference structuralFeatures[*] ordered container : StructuralFeature oppositeOf owner;
	}

	class StructuralFeature extends ModelElement {
		attribute lower : Integer;
		attribute upper : Integer;
		attribute isOrdered : Boolean;
		attribute isUnique : Boolean;
		reference owner : Class oppositeOf structuralFeatures;
		reference type : Classifier;
	}

	class Attribute extends StructuralFeature {}

	class Reference extends StructuralFeature {
		attribute isContainer : Boolean;
		reference opposite[0-1] : Reference;
	}

	class Package extends ModelElement {
		reference contents[*] ordered container : ModelElement oppositeOf "package";
		reference metamodel : Metamodel oppositeOf contents;
	}

	class Metamodel extends LocatedElement {
		reference contents[*] ordered container : Package oppositeOf metamodel;
	}
}
```

Appendix C The DSL metamodel in KM3 format

```
-- @name                    DSL
-- @version                 1.1
-- @authors                 Guillaume Hillairet (g.hillairet@gmail.com), William Piers
(willaim.piers@gmail.com)
-- @date                    2005/06/14
-- @description             This meta-model represents domain models (or metamodels in MDA)
used by Microsoft DSL Tools (May 2005 CTP release for Visual Studio 2005 Beta 2). With DSL
Tools you can create your own designer for a visual domain specific language that is represent
by a domain model. The tools generate the code of a graphical editor for the language you
defined in a domain model.
-- @see                     http://lab.msdn.microsoft.com/teamsystem/workshop/dsltools/

package DSL {

	abstract class NamedElement {
		attribute name : String;
		attribute identity : String;
	}

	abstract class LoadedElement extends Namespace {
		attribute isLoaded : Boolean;
	}

	abstract class Namespace extends NamedElement {
		attribute namespace : String;
	}

	-- @comment This class represents a domain model which contains classes and
relationships.
	class DomainModel extends LoadedElement {
		reference classifiers[*] container : Classifier oppositeOf domainModel;
		reference types[*] container : Type;
	}
	-- @begin Classifiers
	-- @comment This class represents a Classifier. It has properties, may have one super
type and can be abstract.
	abstract class Classifier extends LoadedElement {
		attribute isAbstract : Boolean;
		reference properties[*] container : ValueProperty oppositeOf owner;
		reference superType[0-1] : Classifier oppositeOf subTypes;
		reference subTypes[*] : Classifier oppositeOf superType;
		reference domainModel : DomainModel oppositeOf classifiers;
	}
	-- @comment This class corresponds to a class in DSL. It extends Classifier.
	class Class extends Classifier {}
	-- @comment This class corresponds to a relationship in DSL. A relationship is view as
a class so it extends Classifer.It has two roles, and can be an embedding or a reference.
	class Relationship extends Classifier {
		attribute isEmbedding : Boolean;
		reference roles[2-2] container : Role oppositeOf relation;
	}
	-- @comment This class represents a role. A role can be view as an association end, it
has cardinality (min, max) and can be ordered.
	class Role extends NamedElement {
		attribute min : Integer;
		attribute max : Integer;
		attribute isUnbounded : Boolean;
		attribute accepts : String;
		attribute isOrdered : Boolean;
		attribute isNavigableFrom : Boolean;
		attribute isPropertyGenerator : Boolean;
```

```

			reference source : Classifier;
			reference type : Classifier;
			reference relation : Relationship oppositeOf roles;
		}

		-- @end Classifiers

		-- @begin Types
		-- @comment This class represents a property. A property is had by a classifier, the
	type of the property is represent by the class Type.
		class ValueProperty extends NamedElement {
			reference owner : Classifier oppositeOf properties;
			reference type : Type;
		}

		abstract class Type extends Namespace {}

		class SimpleType extends Type {}

		class EnumerationLiteral extends NamedElement {
			attribute value : Integer;
		}
		-- @comment This class represents an enumeration.
		class Enumeration extends Type {
			reference literals[*] container : EnumerationLiteral;
		}
		-- @end Types
	}
```

Appendix D The KM2 metamodel in KM3 format

```
package KM2 {

	abstract class LocatedElement {
		attribute location : String;
	}

	abstract class NamedElement extends LocatedElement {
		attribute name : String;
	}

	class Model extends LocatedElement {
		attribute metamodel : String;
		reference contents[*] ordered container : ModelElement;
	}

	-- the name of a ModelElement is the name of its type
	class ModelElement extends NamedElement {
		attribute id[0-1] : String;
		reference properties[*] ordered container : Property oppositeOf owner;
	}

	class Property extends NamedElement {
		reference owner : ModelElement oppositeOf properties;
		reference value container : Value oppositeOf owner;
	}


-- Values
	abstract class Value extends LocatedElement {
		reference owner : Property oppositeOf value;
		reference set[0-1] : SetVal oppositeOf contents;
	}

	class ModelElementVal extends Value {
		reference element container : ModelElement;
	}

	class ModelElementRefVal extends Value {
		reference element : ModelElement;
	}

	class SetVal extends Value {
		reference contents[*] ordered container : Value oppositeOf set;
	}

    -- PrimitiveValues
	abstract class PrimitiveVal extends Value {

	}

	class BooleanVal extends PrimitiveVal {
		attribute value : Boolean;
	}

	class DoubleVal extends PrimitiveVal {
		attribute value : Double;
	}

	class IntegerVal extends PrimitiveVal {
		attribute value : Integer;
	}
```

```
        class StringVal extends PrimitiveVal {
                attribute value : String;
        }
    -- End PrimitiveValues
-- End Values
}
```

Appendix E The DSLModel metamodel in KM3 format

```
-- @name		DSLModel
-- @version		1
-- @domains		DSL models
-- @authors		Hillairet Guillaume (g.hillairet@gmail.com)
-- @date		2005/07/05
-- @description	This metamodel represents DSL models used by Visual Studio DSL Tools to
represents models for a domain model. Those models are stored in an xml file, this metamodel
captures informations about model's elements but not the model's graphical representation.

package DSLModel {
	-- @begin Model's Elements
	class Model {
		attribute domainModel : String;
		reference contents[*] ordered container : ModelElement;
	}

	abstract class Element {
		attribute type : String;
		attribute id : String;
	}

	class ModelElement extends Element {
		reference parentLink : EmbeddingLink oppositeOf elements;
		-- a Property is an Attribute in domain model
		reference properties[*] container : Property oppositeOf owner;
		-- a EmbeddingLink is an Embed relationship in domain model
		reference embeddinglinks[*] container : EmbeddingLink oppositeOf owner;
		-- a ReferenceLink is a Reference relationship in domain model
		reference referencelinks[*] container : ReferenceLink oppositeOf owner;
	}

	class ModelElementLink extends ModelElement {
		reference links[*] : ReferenceLink oppositeOf modelElement;
	}
	-- @end Model's Elements

	-- @begin Links

	-- EmbeddingLink represents embedding relationships
	class EmbeddingLink extends NamedElement {
		reference owner[0-1] : ModelElement oppositeOf embeddinglinks;
		reference elements[*] container : ModelElement oppositeOf parentLink;
	}

	-- ReferenceLink represents reference relationships
	class ReferenceLink extends Element {
		reference owner[0-1] : ModelElement oppositeOf referencelinks;
		reference modelElement : ModelElementLink oppositeOf links;
		reference roles[2-2] container : Role oppositeOf owner;
	}
	-- @end Links

	abstract class NamedElement {
		attribute name : String;
	}

	class Property extends NamedElement {
		reference owner : ModelElement oppositeOf properties;
		reference value container : Value;
	}

```

```
        class Role extends NamedElement {
                reference element : ModelElement;
                reference owner : ReferenceLink oppositeOf roles;
        }

        -- @begin Value
        abstract class Value {}

        class IntegerValue extends Value {
                attribute value : Integer;
        }

        class DoubleValue extends Value {
                attribute value : Double;
        }

        class BooleanValue extends Value {
                attribute value : Boolean;
        }

        class StringValue extends Value {
                attribute value : String;
        }
        -- @end Value
}
```

Appendix F The ATL metamodel in KM3 format

```
package Core {
	class Element {
		attribute location : String;
	}
}

package Types {
	abstract class CollectionType extends OclType {
		reference elementType container : OclType oppositeOf collectionTypes;

	}

	abstract class OclType extends OclExpression {
		reference definitions[*] : OclContextDefinition oppositeOf context_;
		reference oclExpression[*] : OclExpression oppositeOf type;
		reference "operation"[0-1] : Operation oppositeOf returnType;
		reference mapType2[0-1] : MapType oppositeOf valueType;
		reference "attribute" : Attribute oppositeOf type;
		reference mapType[0-1] : MapType oppositeOf keyType;
		reference collectionTypes[0-1] : CollectionType oppositeOf elementType;
		reference tupleTypeAttribute[*] : TupleTypeAttribute oppositeOf type;
		reference variableDeclaration[*] : VariableDeclaration oppositeOf type;
		attribute name ordered : String;
	}

	class StringType extends Primitive {
	}

	abstract class Primitive extends OclType {
	}

	class RealType extends NumericType {
	}

	class TupleType extends OclType {
		reference attributes[*] container : TupleTypeAttribute oppositeOf tupleType;

	}

	class SequenceType extends CollectionType {

	}

	class BooleanType extends Primitive {

	}

	class OclModelElement extends OclType {
		reference model : OclModel oppositeOf elements;

	}

	class SetType extends CollectionType {

	}

	class BagType extends CollectionType {

	}

	class OrderedSetType extends CollectionType {

	}
```

```

        abstract class NumericType extends Primitive {

        }

        class TupleTypeAttribute extends Element {
                reference type container : OclType oppositeOf tupleTypeAttribute;
                reference tupleType : TupleType oppositeOf attributes;
                attribute name ordered : String;

        }

        class IntegerType extends NumericType {

        }

        class MapType extends OclType {
                reference valueType container : OclType oppositeOf mapType2;
                reference keyType container : OclType oppositeOf mapType;

        }

}

package Expressions {
        class CollectionOperationCallExp extends OperationCallExp {

        }

        class VariableExp extends OclExpression {
                reference referredVariable : VariableDeclaration oppositeOf variableExp;
                attribute name ordered : String;

        }

        class EmptyMapExp extends OclExpression {

        }

        class RealExp extends NumericExp {
                attribute realSymbol ordered : Double;

        }

        abstract class PrimitiveExp extends OclExpression {

        }

        class IterateExp extends LoopExp {
                reference result container : VariableDeclaration oppositeOf baseExp;

        }

        abstract class PropertyCallExp extends OclExpression {
                reference source container : OclExpression oppositeOf appliedProperty;

        }

        class TuplePart extends VariableDeclaration {
                reference tuple : TupleExp oppositeOf tuplePart;

        }

        abstract class OclExpression extends Element {
                reference ifExp3[0-1] : IfExp oppositeOf elseExpression;
                reference appliedProperty[0-1] : PropertyCallExp oppositeOf source;
                reference collection[0-1] : CollectionExp oppositeOf elements;
                reference letExp[0-1] : LetExp oppositeOf in_;
                reference loopExp[0-1] : LoopExp oppositeOf body;
```

```
			reference parentOperation[0-1] : OperationCallExp oppositeOf arguments;
			reference initializedVariable[0-1] : VariableDeclaration oppositeOf initExpression;
			reference ifExp2[0-1] : IfExp oppositeOf thenExpression;
			reference "operation"[0-1] : Operation oppositeOf body;
			reference ifExp1[0-1] : IfExp oppositeOf condition;
			reference type container : OclType oppositeOf oclExpression;
			reference "attribute"[0-1] : Attribute oppositeOf initExpression;

		}

		class IntegerExp extends NumericExp {
			attribute integerSymbol ordered : Integer;

		}

		class EnumLiteralExp extends OclExpression {
			attribute name ordered : String;

		}

		class OperatorCallExp extends OperationCallExp {

		}

		class IteratorExp extends LoopExp {
			attribute name ordered : String;

		}

		class StringExp extends PrimitiveExp {
			attribute stringSymbol ordered : String;

		}

		class BooleanExp extends PrimitiveExp {
			attribute booleanSymbol ordered : Boolean;

		}

		class LetExp extends OclExpression {
			reference variable container : VariableDeclaration oppositeOf letExp;
			reference in_ container : OclExpression oppositeOf letExp;

		}

		class Iterator extends VariableDeclaration {
			reference loopExpr[0-1] : LoopExp oppositeOf iterators;

		}

		class VariableDeclaration extends Element {
			reference letExp[0-1] : LetExp oppositeOf variable;
			reference type container : OclType oppositeOf variableDeclaration;
			reference baseExp[0-1] : IterateExp oppositeOf result;
			reference variableExp[*] : VariableExp oppositeOf referredVariable;
			reference initExpression[0-1] container : OclExpression oppositeOf initializedVariable;
			attribute varName ordered : String;
			attribute id ordered : String;

		}

		class OperationCallExp extends PropertyCallExp {
			reference arguments[*] ordered container : OclExpression oppositeOf parentOperation;
			attribute operationName : String;
			attribute signature[0-1] : String;

```

```
		}

		abstract class NumericExp extends PrimitiveExp {

		}

		class BagExp extends CollectionExp {

		}

		abstract class CollectionExp extends OclExpression {
			reference elements[*] ordered container : OclExpression oppositeOf collection;

		}

		class IfExp extends OclExpression {
			reference thenExpression container : OclExpression oppositeOf ifExp2;
			reference condition container : OclExpression oppositeOf ifExp1;
			reference elseExpression container : OclExpression oppositeOf ifExp3;

		}

		class LoopExp extends PropertyCallExp {
			reference body container : OclExpression oppositeOf loopExp;
			reference iterators[1-*] container : Iterator oppositeOf loopExpr;

		}

		class TupleExp extends OclExpression {
			reference tuplePart[*] ordered container : TuplePart oppositeOf tuple;

		}

		class SequenceExp extends CollectionExp {

		}

		class NavigationOrAttributeCallExp extends PropertyCallExp {
			attribute name ordered : String;

		}

		class SetExp extends CollectionExp {

		}

		class OrderedSetExp extends CollectionExp {

		}

	}

	package ATL {
		class DerivedInPatternElement extends InPatternElement {
			reference value container : OclExpression;

		}

		class Query extends Unit {
			reference body container : OclExpression;
			reference helpers[*] ordered container : Helper oppositeOf query;

		}

		class Module extends Unit {
			reference inModels[1-*] ordered container : OclModel;
			reference outModels[1-*] container : OclModel;
			reference elements[*] ordered container : ModuleElement oppositeOf module;

```

```
	}

	class ActionBlock extends Element {
		reference rule : Rule oppositeOf actionBlock;
		reference statements[*] ordered container : Statement;

	}

	abstract class Statement extends Element {

	}

	class ExpressionStat extends Statement {
		reference expression container : OclExpression;

	}

	class BindingStat extends Statement {
		reference source : OclExpression;
		attribute propertyName : String;
		reference value container : OclExpression;

	}

	class IfStat extends Statement {
		reference condition container : OclExpression;
		reference thenStatements[*] ordered container : Statement;
		reference elseStatements[*] ordered container : Statement;

	}

	class ForStat extends Statement {
		reference iterator container : Iterator;
		reference collection container : OclExpression;
		reference statements[*] ordered container : Statement;

	}

	class Unit extends Element {
		reference libraries[*] container : LibraryRef oppositeOf unit;
		attribute name ordered : String;

	}

	class Library extends Unit {
		reference helpers[*] ordered container : Helper oppositeOf library;

	}

	abstract class Rule extends ModuleElement {
		reference outPattern[0-1] container : OutPattern oppositeOf rule;
		reference actionBlock[0-1] container : ActionBlock oppositeOf rule;
		reference variables[*] ordered container : RuleVariableDeclaration oppositeOf
rule;
		attribute name ordered : String;

	}

	abstract class OutPatternElement extends PatternElement {
		reference outPattern : OutPattern oppositeOf elements;
		reference sourceElement[0-1] : InPatternElement oppositeOf mapsTo;
		reference bindings[*] ordered container : Binding oppositeOf outPatternElement;

	}

	class InPattern extends Element {
		reference elements[1-*] container : InPatternElement oppositeOf inPattern;
		reference rule : MatchedRule oppositeOf inPattern;
		reference filter[0-1] container : OclExpression;
```

```
	}

	class OutPattern extends Element {
		reference rule : Rule oppositeOf outPattern;
		reference elements[1-*] ordered container : OutPatternElement oppositeOf outPattern;

	}

	abstract class ModuleElement extends Element {
		reference module : Module oppositeOf elements;

	}

	class Helper extends ModuleElement {
		reference query[0-1] : Query oppositeOf helpers;
		reference library[0-1] : Library oppositeOf helpers;
		reference definition container : OclFeatureDefinition;

	}

	class SimpleInPatternElement extends InPatternElement {

	}

	abstract class InPatternElement extends PatternElement {
		reference mapsTo : OutPatternElement oppositeOf sourceElement;
		reference inPattern : InPattern oppositeOf elements;

	}

	abstract class PatternElement extends VariableDeclaration {

	}

	class CalledRule extends Rule {
		reference parameters[*] container : Parameter;
		attribute isEntrypoint : Boolean;

	}

	class Binding extends Element {
		reference value container : OclExpression;
		reference outPatternElement : OutPatternElement oppositeOf bindings;
		attribute propertyName ordered : String;

	}

	class ForEachOutPatternElement extends OutPatternElement {
		reference collection container : OclExpression;
		reference iterator container : Iterator;

	}

	class RuleVariableDeclaration extends VariableDeclaration {
		reference rule : Rule oppositeOf variables;

	}

	class LibraryRef extends Element {
		reference unit : Unit oppositeOf libraries;
		attribute name ordered : String;

	}

	class MatchedRule extends Rule {
		reference inPattern[0-1] container : InPattern oppositeOf rule;
		reference children[*] : MatchedRule oppositeOf superRule;
```

```
			reference superRule[0-1] : MatchedRule oppositeOf children;
			attribute isAbstract ordered : Boolean;

		}

		class SimpleOutPatternElement extends OutPatternElement {

		}

	}

	package OCL {
		abstract class OclFeature extends Element {
			reference definition[0-1] : OclFeatureDefinition oppositeOf feature;
			attribute name ordered : String;

		}

		class Attribute extends OclFeature {
			reference initExpression container : OclExpression oppositeOf "attribute";
			reference type container : OclType oppositeOf "attribute";

		}

		class Operation extends OclFeature {
			reference parameters[*] ordered container : Parameter oppositeOf "operation";
			reference returnType container : OclType oppositeOf "operation";
			reference body container : OclExpression oppositeOf "operation";

		}

		class Parameter extends VariableDeclaration {
			reference "operation" : Operation oppositeOf parameters;

		}

		class OclModel extends Element {
			reference metamodel : OclModel oppositeOf model;
			reference elements[*] : OclModelElement oppositeOf model;
			reference model[*] : OclModel oppositeOf metamodel;
			attribute name : String;

		}

		class OclContextDefinition extends Element {
			reference definition : OclFeatureDefinition oppositeOf context_;
			reference context_ container : OclType oppositeOf definitions;

		}

		class OclFeatureDefinition extends Element {
			reference feature container : OclFeature oppositeOf definition;
			reference context_[0-1] container : OclContextDefinition oppositeOf definition;

		}
	}
```