# Bridging the Generic Modeling Environment (GME) and the Eclipse Modeling Framework (EMF)

Jean Bézivin(1), Christian Brunette(2), Régis Chevrel(1), Frédéric Jouault(1), Ivan Kurtev(1)
*(1)ATLAS Group (INRIA & LINA, University of Nantes)*
*(2) IRISA ESPRESSO Group, Rennes*
*{bezivin | chevrel.regis | f.jouault | ivan.kurtev}@gmail.com*
*Christian.brunette@irisa.fr*

## Abstract

*Model Driven Software Development (MDSD) is based on a number of common principles that involve the concepts of model, metamodel, and model transformation. These principles can be applied to different standards and different environments. The AMMA platform (ATLAS Model Management Architecture) is an example of an environment based on the principles of model engineering. Currently AMMA is implemented on top of EMF (Eclipse Modeling Framework). However, ignoring other environments and platforms, based on different conventions, standards or protocols like the Generic Modeling Environment (GME) would be unwise because one of the desired properties of models is their ability to be exchanged between different contexts. Due to their abstraction expression level, they should ideally be more adaptable to various operational environments than conventional code. To be able to exchange models between an EMF based system and a corresponding GME platform supposes an abstract understanding of both architectures and a precise organization of the interoperability scheme. This paper describes the first results of a project in this area and presents the lessons learnt in this work.*

## 1. Introduction

MDE (Model Driven Engineering) is a vision for developing software systems that considers models as first class entities and any software artifact as a model or a model element. The MDE principles may be implemented by different standards like MDA™ (Model Driven Architecture) [7], MIC (Model Integrated Computing) [8], MS/DSLs (Microsoft Domain Specific Languages) [12] and many others. We are particularly interested here in two of them: MDA and MIC. The OMG MDA proposal may be defined as the realization of MDE principles around a set of OMG standards such as MOF, XMI, OCL, UML, CWM and SPEM.

MIC follows the same principles and is also based on generative techniques named MIPS (Model-Integrated Program Synthesis) technology (*Figure 1*).

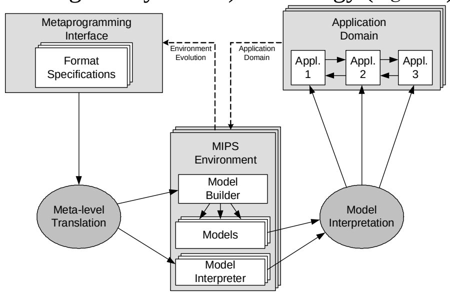

**Figure 1 MIC organization**

MDA initially claimed that one of the goals of modern interoperability support is to achieve platform independence. This is true for applications but should also be true for development tools. Our aim is to study practical interoperability between the different MDE standards. We look more precisely here at two different environments: GME (Generic Modeling Environment) for MIC and EMF (Eclipse Modeling Framework) that may be seen as one of the main operational environments for MDA. We consider EMF extended with the additional capabilities of AMMA (ATLAS Model Management Platform), a local model engineering support defined at INRIA. We consider here MIC/GME and EMF/AMMA to be two MDE technical spaces [11]. Operational bridges between different technical spaces will be called "projectors".

EMF and GME support metamodel and model management. Each model conforms to a certain metamodel. Any metamodel design supposes the existence of a metametamodel (implicit or explicit). Therefore, we have to consider bridging EMF and GME on three levels: metametamodel concepts mapping (M3), metamodel projectors (M2) and model projectors (M1) (*Figure 2*).

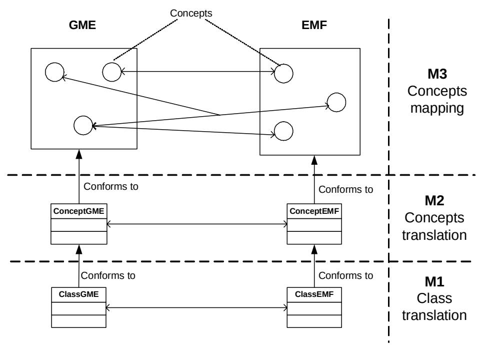

**Figure 2 Global presentation of the complete bridge**

We describe a work that has been performed at INRIA, in the ATLAS team in Nantes. We use ATL (ATLAS Transformation Language) to implement the corresponding technical space projectors. ATL allows model transformations in the EMF technical space.

One of our motivations is to study the possible migration of existing GME projects into the Eclipse technical space. As an illustrative case study we consider the SIGNAL metamodel. SIGNAL is a declarative synchronous dataflow language for modeling critical systems and embedded systems [13]. A SIGNAL GME installation was available at the beginning of the project.

This paper is organized as follows. Section 2 briefly describes the environments and tools mentioned in this paper. Section 3 presents our case study: a SIGNAL metamodel built with GME. Section 4 presents the metametamodels of both environments in order to map their concepts. Sections 5 and 6 respectively describe metamodel and model bridges. Section 7 gives figures corresponding to the application of the bridge to our case study. Section 8 gives conclusions.

## 2. MDE Platforms

Two of the main industrial concrete references of MDE platforms are EMF and GME. AMMA (ATLAS Model Management Architecture) is another example that extends the EMF facilities in a number of ways. By design, AMMA integrates the notion of technical space, i.e. the possibility to interoperate with other MDE and non-MDE environments (XML, EBNF, etc.). This section provides some introduction to EMF, AMMA and GME. More information may be found in the referenced documentation.

### 2.1. EMF

EMF is a modeling framework and code generation facility for building tools and other applications based on a structured data model. The following description is adapted from [3]. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. Models can be specified using annotated Java, XML documents, or modeling tools like Rational Rose, then imported into EMF. Most important of all, EMF provides the foundation for interoperability with other EMF-based tools and applications. EMF consists of three fundamental parts:

- **EMF** - The core EMF framework includes a metamodel (Ecore) for describing models and runtime support for the models including change notification, persistence support with default XMI serialization, and a very efficient reflective API for generically manipulating EMF objects.
- **EMF.Edit -** The EMF.Edit framework includes generic reusable classes for building editors for EMF models.
- **EMF.Codegen** - The EMF code generation facility is capable of generating everything needed to build a complete editor for an EMF model. It includes a GUI from which generation options can be specified, and generators can be invoked. The generation facility leverages the JDT (Java Development Tooling) component of Eclipse.

Three levels of code generation are supported:

1. **Model** - provides Java interfaces and implementation classes for all the classes in the model, plus a factory and package (meta data) implementation class.
2. **Adapters** - generates implementation classes (called *ItemProviders*) that adapt the model classes for editing and display.
3. **Editor** - produces a properly structured editor that conforms to the recommended style for Eclipse EMF model editors and serves as a starting point from which to start customizing.

### 2.2. AMMA

Currently built on top of EMF, AMMA brings additional functionalities. It could be implemented on top of other MDE environments as well. AMMA has both local and distributed implementations and is based on four blocks providing a large set of model processing facilities:

- the Atlas Transformation Language (ATL) defines model transformation facilities for a QVT-like language: a transformation virtual machine, a metamodel-based compiler, a debugging environment, etc.
- the Atlas ModelWeaver (AMW) makes it possible to establish links between the elements of two (or more) different models;
- the Atlas MegaModel Management (AM3) defines the way the metadata is managed in AMMA

(distributed registry on the models, metamodels, tools, etc.);

- the Atlas Technical Projectors (ATP) defines a set of injectors/extractors enabling to import/export models from/to foreign technical spaces (XML, EBNF, etc.).

## 2.3. GME

The Generic Modeling Environment (GME) was developed at the Institute for Software Integrated Systems (ISIS) at Vanderbilt University. It is a metaprogrammable, domain specific, graphical editor supporting the design, analysis and synthesis of complex software-intensive systems. GME is based on MultiGraph Architecture (MGA) [8], which is a part of MIC. The toolset has been applied to modeling and synthesizing several real world applications for both American government and industry organizations.

- GME has an architecture based on MS/COM technology (Component Object Model).
- The modelling paradigm contains all the syntactic, semantic, and presentation information regarding the domain – which concepts will be used to construct models, what relationships may exist among those concepts, how the concepts may be organized and viewed by the modeller, and rules governing the construction of models. [4]
- The modelling paradigm defines the family of models that can be created using the resultant modeling environment. [4]

## 3. Example: the SIGNAL metamodel

SIGNAL is a declarative synchronous dataflow language [13] for modeling critical systems and embedded systems for avionics, automotive, etc.

SIGNAL handles an unbounded series of typed values implicitly indexed by discrete logical time, called signals. At a given instant, a signal may be present, and then hold a value; or absent. The set of instants where a signal is present is called its clock. A SIGNAL program, also called process, is a system of equations over signals. The mathematical foundations on which SIGNAL is built provide formal concepts that favor the trusted design of embedded real-time systems. The toolset to program in SIGNAL, called POLYCHRONY [14], provides a formal framework to validate a design at different levels, to refine descriptions, to abstract properties and to assemble predefined components.

The GME metamodel of SIGNAL, called *Signal-Meta*, is based on the grammar of the language. Each operator (numericals, booleans, clock relations or constraints, etc) is represented by a GME graphical component, called FCO (First Class Object, semantic), which is an *MgaAtom* instance (structural). The specification of a program with SIGNAL can be divided into two main parts. The first one is the dataflow part in which all computations, calls of process, etc. are done. The second part concerns the specification of the clock of each signal. In *Signal-Meta*, these two parts are separated in two aspects. An aspect is a partial view of a Model (a GME container) in which we can see only some kinds of FCO predefined in the metamodel. It is equivalent to a view. To quantify *Signal-Meta*, we can say that it contains 425 FCOs:

- 192 FCOs (Atoms, Models, etc) for the diagram class
- 5 ‘Aspects’
- 52 ‘Constraints’ or ‘Constraint Functions’
- 45 ‘Attributes’
- 192 ‘Proxies’.

Modeling a SIGNAL program with *Signal-Meta* consists in creating signals and linking it to the SIGNAL operators to create equations. An example of a program design (the watchdog) using *Signal-Meta* is represented on Figure 3 and Figure 4. The first one represents the ‘Dataflow’ aspect of the program and the second the ‘Clock relations’ one.

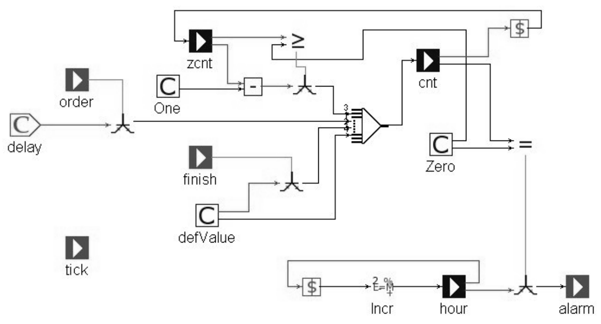

**Figure 3 ‘Dataflow’ aspect of the Watchdog**

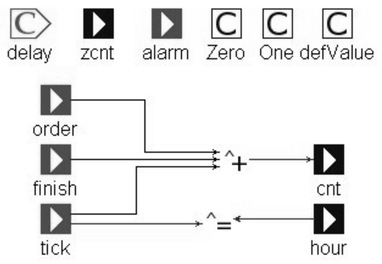

**Figure 4 ‘Clock relations’ aspects of the Watchdog**

On these screenshots, there are several concepts of the metamodel. For example on a coloured original screen presentation, red boxes correspond to input signals of the process, green boxes to output signals and blue ones to local signals. Boxes containing a ‘C’ correspond to constant values, red boxes to static input parameters and blue ones to local constant declarations. SIGNAL equations corresponding to the

'Clock Relations' aspect on Figure 4 is equivalent to the following equations:

*(| cnt ^= order ^+ finish ^+ tick*
*| tick ^= hour*
*|)*

These equations means that the clock of signal 'cnt' is equivalent to the union of the clock of signals 'order', 'finish' and 'tick'; and that the clock of signal 'tick' is equivalent to the clock of signal 'hour'.

The SIGNAL metamodel constitutes a good example to test our transformations and to give, to Eclipse users, the access to SIGNAL formal methods.

Moreover, there is a serializer to translate models described with *Signal-Meta* under GME to SIGNAL program files. So, all specifications done with *Signal-Meta* can be tested, compiled, verified and/or generated into C/C++/Java code.

## 4. M3-level mapping

To enable mapping between GME and EMF we need a definition of each system at the metametamodel level. In the AMMA platform, we use KM3 [6] as a metametamodel. ISIS specifies a metametamodel for GME design: MetaGME (we can find it on [4]).

### 4.1. KM3: Kernel MetaMetaModel

KM3 is a metametamodel close to Ecore and EMOF 2.0. A simplified and adapted version is presented on Figure 5. We use it rather than Ecore because we need to work with several other metametamodels such as MOF 1.4 and many more. KM3 is used as a pivot between these metametamodels as illustrated on Figure 6. Additionally, it provides a textual concrete syntax to specify metamodels, which has some similarities with the Java notation.

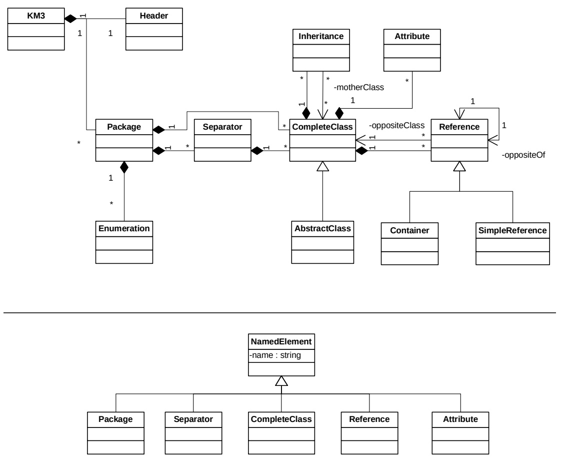

**Figure 5 Simplified version of the KM3 metametamodel**

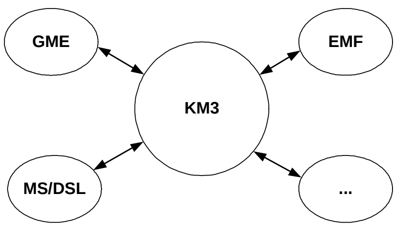

**Figure 6 Using KM3 as a pivot**

### 4.2. MetaGME

MetaGME is the metametamodel for GME model design (the library which implements MetaGME is the MGA[8] library).

MetaGME mixes the structural and semantic aspects. This metamodel describes the GME project organization (structural) which contains the semantic in some specific fields (kind). We want to translate the semantic information.

We now present the structural organization. The root element of a metamodel in GME is called a "project" (*MgaProject*). It is composed of a folder hierarchy (*MgaFolder*). Folders contain paradigm sheets (*MgaModel*), which are a part of the metamodel.

A *MgaModel* is composed of *MgaAtom* which are equivalent to UML classes, inheritances, associations and other. All relationships between atoms are instance of *MgaConnection*. Finally, references allow *MgaAtom* references from a paradigm sheet to another one. For a metamodel design, we call it a proxy.

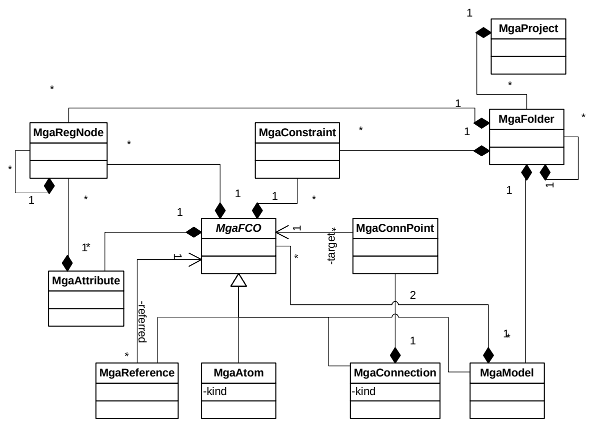

**Figure 7 Simplified version of the MetaGME metametamodel**

All the elements have a graphical presentation and, therefore, a spatial position. They are placed by *MgaRegNode* instances.

The *kind* attribute of *MgaAtom* or *MgaConnection* describes the semantic of the project, i.e. the metamodel atom role (atom, model, attribute, inheritance, etc.) or connection role (association, composition, etc.). Elementary types that do not contain other objects are defined as atoms, while models are composite classes.

### 4.3. Comparison between KM3 and GME

With those diagrams and some practice, we can compare KM3 and GME. It appears that:

- GME *MgaModel* and KM3 *Separator* are almost equivalent.
- GME *MgaAtom* includes KM3 *CompleteClass*. Only *MgaAtom* having kind "*Atom*" or "*Model*" are equivalent to these.
- A KM3 *Attribute* is equivalent to an *MgaAtom* having *kind* equals to "Attribute".
- KM3 *supertype* is a combination of *MgaAtom* having kind equals to "Inheritance" and connected *MgaConnection* (the other *MgaConnection*'s end is the superclass or the derived). It is the same for associations that are *MgaAtom* with kind "Connector" and correspond to KM3 *SimpleReference*.
- Contrary to KM3, GME has three inheritance types: implementation inheritance, interface inheritance and classical inheritance. KM3 can only express the last one.
- GME associations are *MgaAtom* connected to an other *MgaAtom* having kind equals to "Connection" with some *MgaConnection* (see Figure 8)

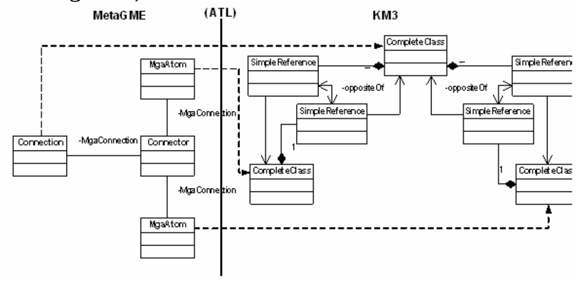

**Figure 8 Transformation of GME associations in KM3**

- GME has constraints.
- GME has proxies.
- GME has graphical organisation.

### 4.4. XML metamodel

We use XML projectors to inject or extract data from different environments (GME, Eclipse, etc.). GME has a project to XML injector/extractor and AMMA has an XML to XMI injector/extractor.

This may be viewed as using the XML technical space to help bridging the GME and EMF technical spaces by using specific projectors. Each projector consists in a pair: injector and extractor.

The model resulting of an XML file injection by the AMMA injector conforms to the XML metamodel presented in Figure 9.

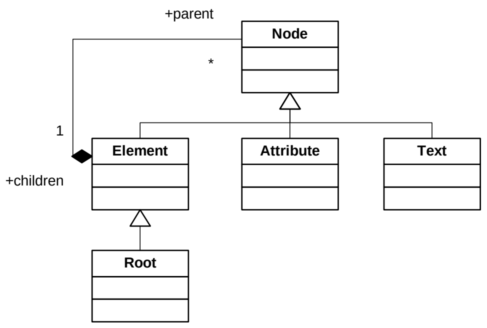

**Figure 9 The XML metamodel**

## 5. M2-level bridging

Now that we know the correspondences between GME and KM3, we can use ATL to transform GME projects into KM3 metamodels. KM3 to Ecore (i.e. EMF metametamodel) is then handled by a specific AMMA transformation.

ATL can transform a model conforming to a source metamodel into a model conforming to a target metamodel. The transformation is itself a model conforming to the ATL metamodel (Figure 10).

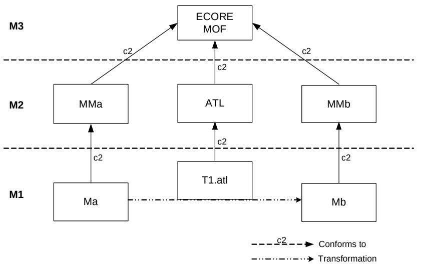

**Figure 10 Overall organization of ATL transformations**

So, we consider our KM3 and GME metametamodels like metamodels and the metamodels conforming to these like models. Some promotion or demotion mechanisms are used to ensure the translation between levels.

An overview of the M2-level bridge is given on Figure 11.

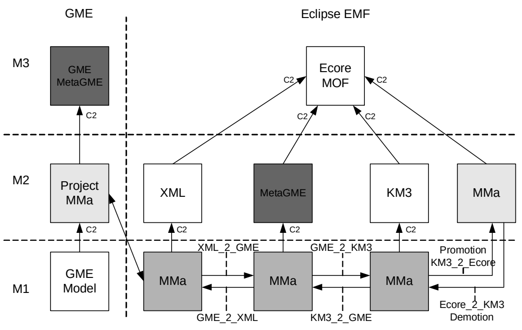

**Figure 11 M2-level bridge overview**

For a GME to Ecore transformation, we can see that a metamodel *MMa* (defined in GME) is injected into an XML model using the XML injector. After that, the result of injection (a model) is transformed using ATL into a model conforming to *MetaGME* in the Eclipse technical space, and later, to a model conforming to KM3. The final step is the promotion of this model using a *KM3_2_Ecore* transformation, which creates an isomorphic *MMa* metamodel conforming to Ecore (EMF).

The inverse transformations from Ecore to GME lose some information (e.g. GME graphical information). An additional model containing the information that KM3 cannot represent could be used to prevent this information loss.

## 5.1. First transformation: GME to Ecore

To make the transformation we proceed in four steps. There are detailed below.

### 5.1.1. First step: GME to XML extractor

GME has a project to XML extractor.

### 5.1.2. Second step: XML_2_GME

AMMA has a XML to XMI injector. Then we can use an ATL transformation to translate the result of injection in our MetaGME metamodel defined in KM3.

The main work of this transformation is a mapping between the XML file features (atom, model, reference, etc.) and our MetaGME metamodel concepts (*MgaAtom*, *MgaModel*, *MgaReference*, etc.).

### 5.1.3. Third step: GME_2_KM3

This is the heart of the bridge. In this step we transform the previously obtained MetaGME into a KM3 metamodel, using another ATL transformation. Some information is lost, like graphical organisation, proxies, implementation inheritance, etc. but most of the logical information is kept (except OCL constraint).

Some element transformations are simple: an *MgaProject* becomes a KM3 metamodel, an *MgaFolder* becomes a KM3 Package and an *MgaModel* becomes a KM3 Separator. *MgaAtom* having kind equals to “Atom” or “Model” are mapped to KM3 CompleteClass. All the attributes attached to these *MgaAtom* become KM3 *Attribute* contained in the produced classes.

There are, however, some problems about association translation. Indeed, these relations are quite complicated in GME. For an association between two classes, GME needs seven elements. It includes association classes. For the translation, in order to keep all the information about this association class, we create an intermediate *CompleteClass*. See Figure 12 below.

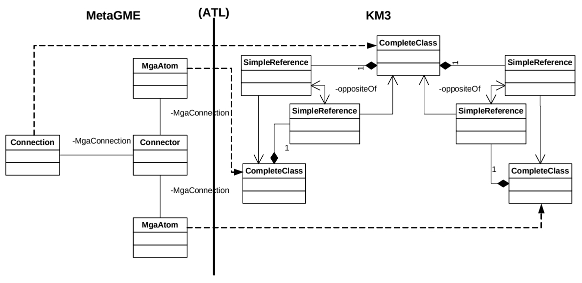

**Figure 12 GME associations transformed in KM3**

The inheritance transformation concerns only simple inheritance (not implementation or interface inheritance). It is a lossless transformation. We combine the *MgaAtom* having *kinds* equal to “*Inheritance*” and its *MgaConnection* to produce a KM3 *supertypes* relation. See Figure 13.

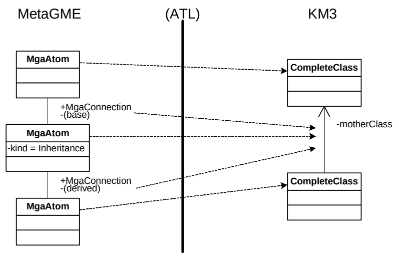

**Figure 13 GME inheritances transformed in KM3**

GME compositions become double KM3 references. See Figure 14.

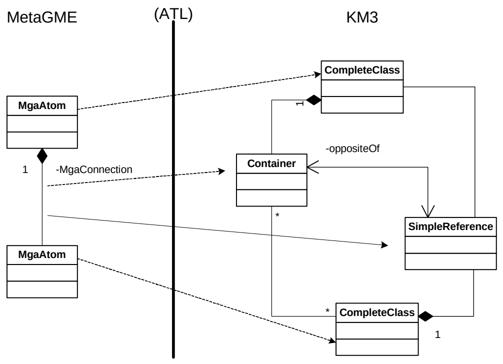

**Figure 14 GME compositions transformed in KM3**

Another difficulty is for the proxies. When we translate a metamodel, the *MgaReference* disappears. The *MgaAtom* referred to by a proxy becomes the target of the *MgaConnection*, which refers to these proxies. See Figure 15.

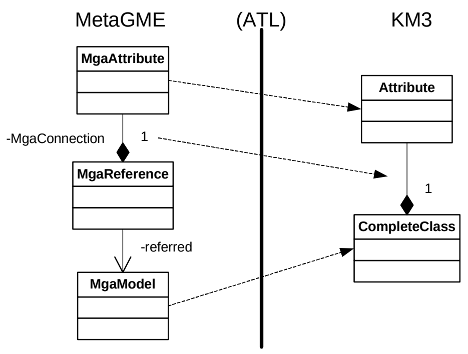

**Figure 15 GME proxy transformed in KM3**

The last point concerns the MetaGME constraints. KM3 cannot express these but we add a comment, which contains the OCL expression before the concerned *CompleteClass*, so we do not loose completely these information. They could also be translated into ATL code that would check these constraints as presented in [2]. This is, however, out of the scope of this paper.

### 5.1.4. Fourth step: KM3_2_Ecore

Using the existing transformation KM3_2_Ecore, we get a metamodel conforming to Ecore from the result of previous step. We can now use the metamodel designed in GME under Eclipse environment.

## 5.2. Opposite transformation: Ecore to GME

We start this transformation with a metamodel conforming to Ecore, and we now want to transform it into a GME project. To define the transformation we proceed in four steps detailed below.

### 5.2.1. First step: Ecore_2_KM3

This transformation is already done and is integrated in Eclipse (ATL plug-in).

### 5.2.2. Second step: KM3_2_GME

GME concepts are roughly a superset of KM3 ones. Thus, there is no major problem to translate the KM3 data in GME. But some additional information is necessary:

For KM3 association concept we have to generate an *MgaAtom* having kind equals to "Connection". More especially we have to generate a graphical organization by producing *MgaRegNode* for all the transcript elements.

We can find back some *MgaReference* by analysing the KM3 *Separator*, which contains classes and the links they use.

### 5.2.3. Third step: GME_2_XML

Now, we translate the model conforming to our *MetaGME* metamodel into a model conforming to the XML metamodel in order to generate an XML file, which will contain the GME project.

### 5.2.4. Fourth step: XML_2_Text

We use AMMA XML to text extractor to create a GME project file.

## 5.3. Limitations

When we tried to inject the XML file representing the GME SIGNAL metamodel with our AMMA XML injector, we encountered an *OutOfMemoryError* due to the size of the data. This occurred because MetaGME exports both semantics and graphical presentation to XML and the presentation part can be too verbose for the XML injector.

In order to overcome this issue, we actually used an XSLT program to filter out presentation tags from the XML file (Figure 16). The reduction in size was about 70%, which allowed us to continue with our bridge.

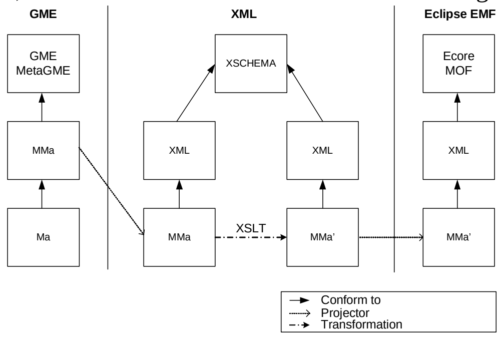

**Figure 16 An XSLT filter**

# 6. M1-level bridging

## 6.1. Overview

We now have a metamodel bridge. The next step to reach our objective is to create a model transformation in order to inject the models from an environment to the other.

To do this, we have to know the GME model representation. In fact, GME uses the *MetaGME* metametamodel like a metamodel for the models conformance. The difference lies in the kind of field (*MgaAtom*) that now corresponds to the metamodel concepts. This is because *MgaAtom* corresponds to syntax rather than semantics: it represents a graph node, which type is contained in the kind attribute.

The M1-level bridge is presented on Figure 17. The ATL transformations *GMEModel2EMFModel* and the reverse must be generated at the M2-level bridge because the element's name concordance and other elements like links are in relation with the metamodel transformation.

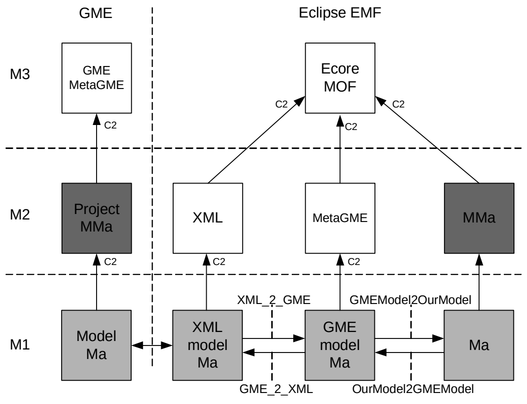

**Figure 17 M1-level bridging overview**

## 6.2. Bridge organisation

The GME model organisation is the same as for the metamodel, but the *kind* attributes now correspond to our metamodel concepts. We already know this construction for association, inheritance and we can apply the same choices for these translations.

We describe below the three steps about the GME to Ecore model injector (it is the same way for the opposite one).

### 6.2.1. First step: GME to XML extractor

We export the project to an XML file like for the metamodel bridge.

### 6.2.2. Second step: XML_2_GME

Here again, we can reuse the corresponding transformation developed for the metamodel bridge.

### 6.2.3. Third step: GMEModel2OurModel

As we have seen, the ATL transformation which translates GME models conforming to *MetaGME* in EMF to the model conforming to our metamodel obtained on the M2- level bridge must be produced. Figure 18 summarizes this third step.

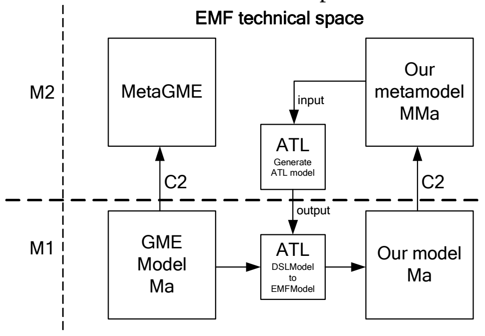

**Figure 18 Overview of step 3**

## 6.3. Summarizing the M1-level bridge protocol

Figure 19 summarizes the M1-level bridge protocol.

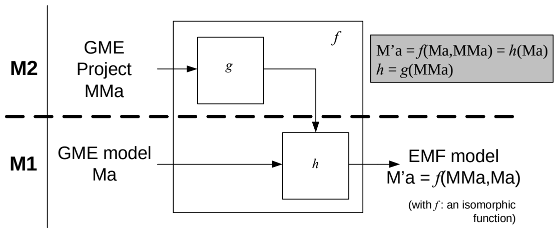

**Figure 19 Process to generate a transformation**

From metamodel *MMa* (GME), the transformation *g* creates an ATL[1] model *h*. The model *h*, is thus an executable transformation. This one takes as input a model *Ma*, which conforms to *MMa* in GME, and creates as output a model, *M'a*, conforming to *MMa* in EMF.

## 7. About the SIGNAL transformation

The metamodel bridge is already effective and we may compare the resulting KM3 file from the GME entry:

- There are 117 *CompleteClass* including 13 *AbstractClass*.
- There are 198 *Reference* including 63 *Container*.
- The KM3 file size is 29285 bytes while the GME project size is 2068 kilobytes. This corresponds to a gain of 98.58% of the original size. This can mostly be explained by the loss of graphical information and by the fact that KM3 does not suffer from XML verbosity.
- Moreover, the nested organization of KM3 replaces the GME approach which separates the elements (*MgaAtom*) and theirs links (*MgaConnection*).
- We also lost the Aspects (view).

We can now work on EMF with our GME-designed metamodel and models. We may also use ATL as a transformation language instead of GReAT (i.e. GME transformation language).

## 8. Conclusions

This paper has reported on an experiment of practical interest: building operational bridges between two major MDE platforms. The resulting tools may be of potential utility to solving practical problems. We have also learnt a lot in this practical endeavor on the way to build real life transformation systems.

We have seen that achieving model interoperability is much more complex than simply defining a local serialization format in a local context (e.g. XMI). Our work is an experiment of a possible systematic approach.

GME was a precursor to model engineering platforms and, as such, it may today pay the cost of having being built on a technology of the 90's (COM).

[1] The complete ATL metamodel may be found at [1]

However more recent proposals (like EMF) may also age very soon in a world of rapid technology obsolescence. For model engineering to achieve its platform independence promises, we must show that different MDE proposals made at some given time (e.g. in the Microsoft or in the OMG context) or at successive times may allow to migrate and evolve products resulting sometimes from significant efforts. The example chosen to illustrate this here is the Signal metamodel environment.

The lessons learned in this initial experiment show that we can probably build a systematic way to bridge similar environments instead of having each time to construct ad-hoc non-reusable solutions.

First, we must recognize the complexity of the task. ATL is a QVT-like language. One simple task that could be achieved in ATL is transforming some MOF-based and XMI encoded model into a similar model. For example an UML model may be transformed into a Java model provided all are conforming to MOF 2.0 for example and encoded in XMI 2.0. But the problem in real life is that data that we have to transform is rarely natively encoded in XMI 2.0 and conforming to MOF 2.0. Many tools may quite easily implement a simple UML to UML or UML to Java transformation, including UML-based CASE tools. This is completely different from using MDE as a basis for providing a competitive generic data-transformation framework between any kind of tools, legacy or recent tools. The real-life experiment presented here illustrates the complexity of the task and the number of involved components. At this point in time we do not yet have the MDE technology that will allow us to cope with this increase in complexity. Managing long chains of combined transformations, with all the involved metamodels, injectors and extractors for a variety of formats needs integrated model-based development environments that we have still to define and to build.

On a more positive side, we have found in this project reasons to be optimistic about MDE approaches. Considering any structured data as a model, i.e. something that conforms to a precise metamodel, is a way to provide regularity and conceptual simplicity to a basically heterogeneous world that becomes every day more complex. Several conceptual intuitions like the importance of higher order transformations (i.e. transformations that generates other transformations and/or take transformations as input) as again been validated in this project on practical grounds.

## 9. Acknowledgements

We would like to thank Freddy Allilaire, David Touzet, Patrick Valduriez as all the members of the AMMA group for their numerous contributions to this document.

## 10. References

[1] ATL, ATLAS Transformation Language Reference site http://www.sciences.univ-nantes.fr/lina/atl/ including KM3: Kernel Metametamodel definition.

[2] Bézivin, J., and Jouault, F., Using ATL for Checking Models, to appear in the proceedings of GraMoT workshop, GPCE 2005.

[3] Eclipse Modeling Framework http://www.eclipse.org/emf/

[4] GME, The Generic Environment, Reference site. http://www.isis.vanderbilt.edu/Projects/gme/

[5] A MOF-Based Metamodeling Environment, Emerson, Sztipanovits, Bapty www.isis.vanderbilt.edu/publications/archive/Emerson_MJ_10_9_2004_A__MOF_Bas.pdf

[6] GMT, General Model Transformer Eclipse Project, http://www.eclipse.org/gmt/

[7] MDA reference site http://www.omg.org/mda/

[8] Soley, R., and the OMG staff, Model-Driven Architecture, OMG Document, November 2000, http://www.omg.org/mda

[9] MIC, Vanderbilt University http://www.isis.vanderbilt.edu/Research/mic.html/

[10] MDSD http://www.voelter.de/services/mdsd-tutorial.html

[11] Kurtev, I., Bézivin, J., Aksit, M. Technical Spaces: An Initial Appraisal. CoopIS, DOA'2002 Federated Conferences, Industrial track, Irvine, 2002 http://www.sciences.univ-nantes.fr/lina/atl/publications/

[12] Microsoft Domain-Specific Language (DSL) Tools, May 2005 CTP Release for Visual Studio 2005 Beta 2. http://lab.msdn.microsoft.com/teamsystem/workshop/dsltools/default.aspx

[13] Le Guernic P., Talpin J-P., Le Lann J-C. POLYCHRONY for Systems Design. Journal of Circuits, Systems and Computers, Vol. 12, No. 3 (2003) 261-303.

[14] POLYCHRONY reference site. http://www.irisa.fr/espresso/Polychrony/