Building a DSL for Interactive TV Applications with AMMA

Mikaël Barbero
ATLAS (INRIA & LINA)
2 rue de la Houssinière
44300 Nantes, Cedex 3, France
Mikael.Barbero@univ-nantes.fr

Jean Bézivin
ATLAS (INRIA & LINA)
2 rue de la Houssinière
44300 Nantes, Cedex 3, France
Jean.Bezivin@univ-nantes.fr

Frédéric Jouault
ATLAS (INRIA & LINA) / UAB
2 rue de la Houssinière
44300 Nantes, Cedex 3, France
Frederic.Jouault@univ-nantes.fr

ABSTRACT

In this paper, we propose a solution to build a Domain Specific Language (DSL) for interactive television applications with the Atlas Model Management Architecture (AMMA) platform. We demonstrate how to give a textual concrete syntax to a metamodel and how to build an Eclipse editor automatically. Finally, we discuss the need of a megamodel for managing all of those modeling artifacts.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]:.

General Terms

Model Management, Experimentation, Languages

Keywords

Keywords are your own designated keywords.

1. INTRODUCTION

Domain-Specific Languages (DSLs) are increasingly used. They are defined by opposition to General Purpose Languages (GPLs). One advantage of DSLs against GPLs is their user friendliness. They directly represent domain concepts so that non-programmer domain experts are able to quickly understand those DSLs and use them.

However, the cost of designing a new language is very high. As DSLs only aim at a small number of experts for their domain specific uses, their development cost cannot be as expensive as GPLs' development cost. Moreover, we often need many DSLs for building complex systems while one GPL is usually enough. We give an answer to this problem by a quick prototyping methodology atop a well-defined framework. We reckon that the ease and low-cost of incrementally developed DSLs by prototyping will be the main success criteria of this approach.

We tackle here the high cost and low development speed problems of implementing DSLs. Our proposal is to use DSL building frameworks [1]. A typical example of such a framework is GME [2][3] (Generic Modeling Environment) or the Microsoft DSL Tools [4], In the current paper we'll use AMMA [1][5] (ATLAS Model Management Architecture) developed in Nantes. We report on an experiment consisting of the implementation of a languages specific to the domain of interactive television (iTV) applications. The outcome of this experiment provides an interesting example of DSL building.

This paper is organized as follow. Section 2 provides a quick introduction to the AMMA platform. Section 3 is an overview of the implementation details of the iTV DSL and the transformation to XML. Section 4 describes the tooling available for manipulating the previously build DSL. Finally, Sect. 5 discusses the need of global model management facilities to coordinate all of those modeling artifacts.

2. ATLAS MODEL MANAGEMENT ARCHITECTURE

This section briefly presents AMMA and three of its core DSLs: KM3 (Kernel MetaMetaModel), ATL (ATLAS Transformation Language), and TCS (Textual Concrete Syntax). A more complete description can be found in [1].

2.1 Overview

AMMA is built on a model-based vision of DSLs, which is presented in [1]. A DSL is considered as a set of coordinated models. Each of these models represents one facet of the language. For instance:

• Abstract Syntax. Domain concepts and their relations are captured in a metamodel called a Domain Definition MetaModel (DDMM).

• Concrete Syntax. Concrete syntaxes of DSLs can be represented as models. One possibility is to specify a transformation from concrete to abstract syntax. Another possibility is to represent a concrete syntax as a model conforming to an EBNF metamodel (i.e. a grammar) or to the TCS metamodel (see below).

• Semantics. DSLs have several kinds of semantics that may be captured by models. For instance, dynamic semantics can be captured as a mapping to an Abstract State Machine [6] model. Alternatively, the semantics of DSL_A may be implemented in terms of the semantics of DSL_B by writing a transformation from DSL_A to DSL_B.

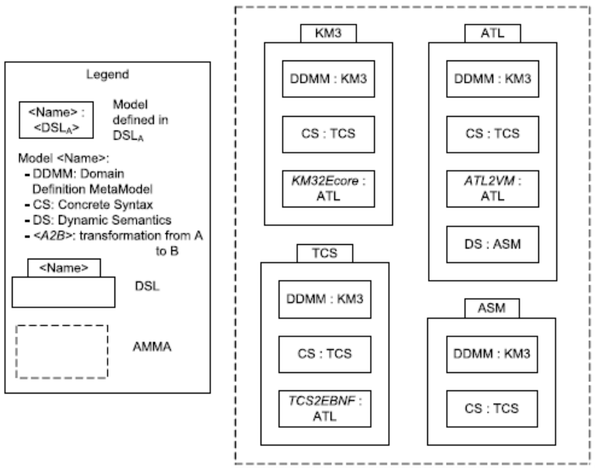


Figure 1. AMMA core DSLs

AMMA provides a set of core DSLs that are used to specify each model of a DSL. Figure 1 shows four of these core DSLs: KM3, ATL, TCS, and ASM (Abstract State Machines).

Each DSL defined with the AMMA platform (as the iTV one) is represented by a set of models defined using AMMA core DSLs. The first three core DSLs are described in the next section. As for the last one (i.e. ASM), please refer to [7], which describes how we extended AMMA with it.

2.2 KM3

The KM3 language is intended to be a lightweight textual metamodel definition language. It enables easy creation and modification of metamodels. The metamodels expressed in KM3 have good readability properties as seen in Figure 2. These metamodels may be easily converted to/from other notations like Emfactic or XMI. KM3 has a clear semantics, partially presented in [8].

Figure 1 shows that the DDMM of KM3 is expressed in KM3 (box DDMM: KM3). Its concrete syntax is defined in TCS (box CS: TCS). Its semantics is implemented by a transformation to Ecore [9] written in ATL (box KM32Ecore: ATL). Other models are not shown here. For instance, a mapping to MOF 1.4 also exists, and formal semantics of KM3 is also expressed in Prolog. A library of KM3 metamodels and their translations to various formats like Ecore, MOF 1.4, and pictures is available on [10].

2.3 ATL

ATL is a hybrid model transformation DSL. Its declarative part enables simple specification of many problems, while its imperative part helps in coping with problems with higher complexity. Informal semantics of ATL is presented in [11] along with a non-trivial case study. More than forty different scenarios accounting for more than a hundred individual transformations are available on [10]. Figure 1 shows that the DDMM of ATL is expressed in KM3 (box DDMM: KM3). Its concrete syntax is defined in TCS (box CS: TCS). Its semantics is implemented by a transformation to ATL Virtual Machine (see [12]) written in ATL (box ATL2VM: ATL). Partial formal semantics of ATL is also expressed in ASM [7] (box DS: ASM).

2.4 TCS

TCS is a DSL aimed at specifying context-free textual concrete syntaxes of DSLs. From such specifications, models can be serialized into their textual equivalent and text can be parsed into models. In other words, a TCS specification defines a bidirectional translation between a textual representation of a model and its internal representation. The choice of context-free languages was mainly motivated by the observation that programming languages use them extensively. TCS models provide a way to attach syntactic elements, such as keywords and symbols, to elements of the DDMM of a DSL. A detailed description of TCS is out of the scope of this paper. Figure 1 shows that the DDMM of TCS is expressed in KM3 (box DDMM: KM3). Its concrete syntax is defined in TCS (box CS: TCS). Its semantics is implemented by a transformation to EBNF written in ATL (box TCS2EBNF: ATL).

3. IMPLEMENTING THE iTV DSL

The goal of this work is to implement an interactive television DSL with the AMMA platform. The abstract syntax is expressed in KM3, the concrete syntax is expressed in TCS and we give the semantic by generating XML conforming to implicit schema as expressed in the reference document titled *Interactive Television Applications*. Proceeding in this way allow us to send this XML document along with the broadcast stream and the interpreter on the end-user side gives a kind of semantic. Next sections will cover this implementation beginning with the abstract syntax.

3.1 iTV abstract syntax

The main concept of iTV applications is *Menu*. An iTV application is rooted by a TVApp element which contains *Menu*s. *MenuElement* like *Menu* itself or *Text* belongs to one and only one *Menu*. Along with *Menu*, there are two kinds of *MenuElement*: *Poll* and *Text*. *Poll* has a set of *Choice*s while *Text* has a unique attribute called *value*. This metamodel is given for convenience on Figure 2 expressed in KM3 syntax.

```
package TVApp {
   abstract class NamedElement {
      attribute name : String;
   }
   class TVApp extends NamedElement {
      reference menus[*] ordered container : Menu;
   }
   abstract class MenuElement extends NamedElement {
      reference parent : Menu oppositeOf contents;
   }
   class Menu extends MenuElement {
      reference contents[*] ordered container : MenuElement oppositeOf parent;
   }
   class Poll extends MenuElement {
      reference choices[*] ordered container : Choice oppositeOf poll;
   }
   class Choice extends NamedElement {
       reference poll : Poll oppositeOf choices;
   }
   class Text extends MenuElement {
      attribute value : String;
   }
}
```

Figure 2. iTV metamodel in KM3 syntax (iTV.km3)

This abstract syntax lets us defining the domain of an interactive television application. For better user friendliness, we would define a textual concrete syntax to models conforming to this metamodel as part of the iTV DSL.

3.2 iTV concrete syntax

As previously mentioned, textual concrete syntaxes of DSLs are designed with TCS within AMMA. For each concrete class in the metamodel, we define an element *template* stating how a model element is textually written. For instance, on Figure 3, we can see that a *Menu* textual syntax begins with the keyword “menu” followed by its name. Then, a block is opened by a curly bracket in which the content of the menu is specified (*contents* reference).

```
template TVApp main
   : name
      [ menus ] {indentIncr = 0, nbNL = 2, startNL = false}
   ;
template MenuElement abstract;
template Menu
   : "menu" name "{" [
      contents
   ] {indentIncr = 2, nbNL = 2, startNL = true} "}"
   ;
template Poll
   : "poll" name "{" [
      choices
   ] {indentIncr = 2, nbNL = 2, startNL = true} "}"
   ;
template Choice
   : name
   ;
template Text
   : name ":" 'value'
   ;
```

Figure 3. iTV concrete syntax definition excerpt in TCS

A grammar is automatically derived from both the KM3 metamodel and the TCS model to parse iTV programs into iTV models. iTV models can also be serialized to programs using a TCS interpreter written in Java.

As an illustrative example of the grammar defined the TCS model, Figure 4 depicts a very simple iTV program named *World Cup*. This program contains one main menu, itself containing one poll, one free text and one submenu.

```
"World Cup 2010"

menu "World Cup" {
   poll "Who will win?" {
      Estonia
      Lithuania
    }
   "World Cup trivia": "Trinidad & Tobago were the first..."
   menu Teams {
   }
}
```

Figure 4. iTV concrete syntax example (sample.itv)

3.3 XML extraction

The XML abstract syntax only represents the tree structure of XML and does not take into account all the constraints and constructs that can be done with XML schema. It describes the concept of a tree of nodes. Nodes can be classical *Element*, *Text* or *Attribute* nodes.

```
package XML {
   abstract class Node {
      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 {
   }
}
```

Figure 5. XML metamodel

An extractor has been defined from this pre-defined metamodel, Any model conforming to the XML abstract syntax can be extracted to a well formed XML document. Figure 6 depicts the awaited output of our DSL to be able to insert this XML into the interactive television broadcast stream.

```
<?xml version = '1.0' encoding = 'ISO-8859-1' ?>
<TVApp name = 'World Cup 2010'>
  <Menu name = 'World Cup'>
    <Poll name = 'Who will win?'>
      <Choice name = 'Estonia'/>
      <Choice name = 'Lithuania'/>
    </Poll>
    <Text name = 'World Cup trivia'>Trinidad &amp; Tobago were the first...</Text>
    <Menu name = 'Teams'/>
  </Menu>
</TVApp>
```

Figure 6. XML document for the iTV broadcast stream

3.4 Transforming iTV to XML

We have provided a simple DSL for iTV. We defined its abstract and concrete syntax and automatically generated injector and extractor from and to the textual syntax. We also used a generic solution for XML generation from a model conforming to the XML metamodel. The heart of the work to let the end user using our DSL instead of the XML syntax is to define an ATL transformation from the iTV metamodel to the XML metamodel. This transformation is quite simple as the iTV metamodel containment tree is very close to the tree structure we want to get in the final XML document. For each metamodel element, we create an XML element with the appropriate name and attributes. An excerpt is given in Figure 7.

```
-- @atlcompiler atl2006
module TVApp2XML;
create OUT : XML from IN : TVApp;

abstract rule NamedElement {
   from i: TVApp!NamedElement
   to attribute_name: XML!Attribute (
      name <- 'name',
      value <- i.name
   )
}
rule TVApp extends NamedElement {
   from i : TVApp!TVApp
   to o1 : XML!Root (
      name <- 'TVApp',
      children <- Sequence{attribute_name}->union(i.menus)
   )
}
```

Figure 7. iTV to XML transformation excerpt

The complete process is organized as follow. First, we take the .itv program conforming to the iTV grammar (which has been automatically generated from the TCS and KM3 models of iTV domain). We apply the EBNF Injector operation. It uses the generated parser to create a model conforming to the iTV abstract syntax.

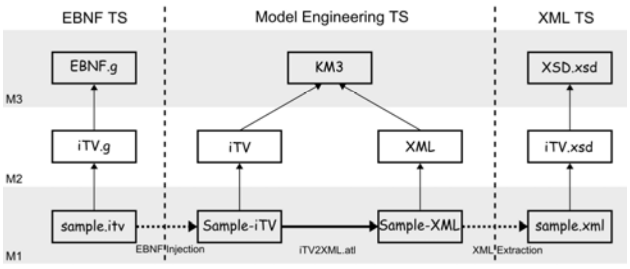


Figure 8. Complete process: from textual program to XML document

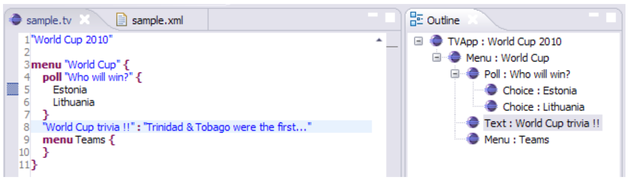
Then we apply the iTV to XML transformation written in ATL to extract (thanks to the generic XML extractor) this XML model to an XML document. The complete process, i.e. from the textual representation to the XML document is depicted on Figure 8.

4. TOOLING AVAILABLE WITH AMMA

Apart from the ability to generate the necessary XML document from the textual DSL of iTV, AMMA provides facilities to use this DSL. The main one is the Textual Generic Editor (TGE). TGE is an Eclipse editor that gives a full-featured editor for DSL building with AMMA. It provides color highlighting, source level debugging, outlines, etc. Figure 9 is a screenshots of such an editor. Actually, TGE does not take the abstract and concrete syntax models for dynamically building such a text editor. It rather takes an editor model and an outline one.

The editor model describes how text blocks are organized, what are symbols, keywords, and comments. For each kind of symbol, you can associate a style: bold face, blue color, etc.

The outline model describes which elements are to be displayed in the outline. Maybe you don't want to show every detail of models. For instance, when editing a Java file, the outline never shows outline for statements but only high level constructs as methods, attributes, etc. Moreover, an outline model can specify how to display outline's elements. For instance, you may want to use a name attribute, or the concatenation of many strings, etc. Finally, you may want to specify specific icons for specific model elements.



```
"World Cup 2010"

menu "World Cup" {
  poll "Who will win?" {
    Estonia
    Lithuania
  }
  "World Cup trivia !!" : "Trinidad & Tobago were the first..."
  menu Teams {
  }
}
```

```
TVApp : World Cup 2010
  Menu : World Cup
    Poll : Who will win?
      Choice : Estonia
      Choice : Lithuania
    Text : World Cup trivia !!
    Menu : Teams
```

Figure 9. TGE for editing iTV programs

Since everything is customizable, we can generate those two models from the abstract and concrete syntax models automatically. ATL transformations named TCS2Editor and KM32Outline do this. Editor and outline depicted on Figure 9 are default ones using models generated from those transformations.

5. THE NEED OF MEGAMODEL

After the description of the implementation of the iTV DSL with AMMA, we would like to introduce the next important issue in the Model Driven Engineering and DSL field: managing the complexity. With a model based view of DSLs, the number of modeling artifacts will increase in such a way that we need something to manage induced complexity. The idea of megamodeling is proposed in order to cope with this accidental complexity that has been observed in the building of real-life model-driven engineering solutions.

If we go back to the iTV DSL we built in previous sections, here is the list of models and metamodels we had to deal with:

- The iTV model and the iTV metamodel,
- The TCS model for iTV models and the TCS metamodel,
- The XML model and the XML metamodel,
- The "iTV to XML" ATL transformation and the ATL metamodel,
- The "EBNF injection" and "XML extraction" operations
- Many textual iTV programs
- Etc…

All of these modeling artifacts have dependencies on each others and it may become difficult to maintain all of them in a coherent state. For instance, adding a few concepts to the iTV metamodel may imply to modify the TCS model and then rebuild the injector for its concrete syntax. The iTV to XML transformation may have to be updated also. This can become quite heavy, even if the iTV use case is simple.

MDE mainly suggests basing the software development and maintenance process on chains of model transformations (Model-to-Model, Model-to-Text or Text-to-Model). A single transformation is quite easy to handle, but as soon as we tackle real-life situations (as those described in previous sections, even if it is a very simple example) we are faced with large sets of MDE artifacts from which we have to assemble a solution. The classical programming level tools are of no significant help here to manage this kind of situation. If we want software systems to be designed from a high number of models, metamodels, transformations, converters and other similar components, we need to provide a Global Model Management (GMM) environment that will allow the user to handle this complexity without major penalties. We want to address this specific problem by describing the need for GMM.

On one hand, the building of complex software solutions involves more and more models in the broad sense. These numerous models are not only UML models and can be of very various natures (e.g., models expressed in XML documents that conform to specific XSD schemas or DTD or EMF models that conform to different metamodels expressed in Ecore format, etc). Moreover, these same models are often linked to each other and are also involved in complex chains of operations, which, most of the time consist in sequences of transformations. As a consequence, the solution providers need facilities (i.e. methodologies and tools) for managing all these models, specifying all the relationships and dependencies that may exist between them and building the complex chains of operations involving all these different modeling artifacts.

On the other hand, with such solutions as the Eclipse Modeling Framework (EMF) or the Microsoft DSL Tools being increasingly used, we may anticipate that MDE is going to be more and more present in many organizations. Applying MDE processes to complex real-life use cases, means that we need to handle a huge number of various and varied modeling artifacts. These artifacts are:

- **Numerous**. One application may use several hundreds of such components selected between large libraries of available components amounting to tens of thousands of units and even more.
- **Distributed**. Some components may be available on a Web site different from the one where the application is executed or deployed.
- **Interrelated**. The artifacts are related by strong semantic links. For example a QVT or ATL transformation should refer to its source and target metamodels. These metamodels may be themselves versions or extensions of other metamodels. A model Mb obtained from a model Ma by an ATL or QVT transformation Mt may record its origin model and its transformation model Mt. Moreover a traceability relation may relate these models Mb and Ma. These are only a few of the semantic relations that may be found in a set of MDE artifacts.
- **Heterogeneous**. The artifacts are of different natures. They should be categorized in a very systematic organization, i.e. a typing system. The simplest idea that comes to mind is to consider that all artifacts are models. Then we have a simple solution, which consists in stating that each one is typed by its metamodel. We will mainly follow this conjecture in the present work that all managed artifacts are models, conforming to a precise metamodel.
- **Complex**. The artifacts may contain a lot of internal elements. Here again, if we follow the simplifying conjecture stated above, the possible nature of elements contained in one artifact is defined by the metamodel.

In order to provide access to these artifacts in the definition of MDE solutions, we need to hide additional generated complexity here. A search into a large distributed library of metamodels or the consecutive chaining of a lot of different operations on models are examples of global model management facilities that should be provided. The simplest way is to apply general MDE principles to the handling of models. This is the main idea of megamodels described below, i.e. models which elements represent model themselves. Megamodeling is named after the idea of megaprogramming introduced by B. Boehm [13] in order to propose a solution to the construction of large-scale software systems.

We introduce the concept of megamodel [14] in order to provide some kind of registry for models in the context of Global Model Management (GMM). A megamodel is a model that contains global entities like terminal models, metamodels, transformations, etc. In our approach, a megamodel is considered as a representation of what we name a “working zone” (i.e. a context composed of various MDE artifacts). For an MDE application developer, the working zone is the complete set of MDE artifacts that he/she may potentially use to build the application or a DSL for instance.

Megamodels conform to a particular type of metamodels. These metamodels may be original ones but also extensions of already existing ones. This principle is summarized in Figure 10:

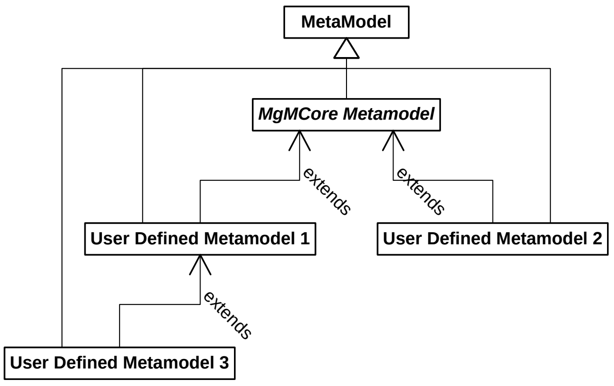


Figure 10. Megamodels metamodels

We believe that there is a basic megamodel metamodel that defines standard resources and behaviors in the context of GMM (we call it *MgMCore* Metamodel in Figure 10). However, users may want to refine this metamodel (or another already defined extension of this metamodel) if they need to specify their own GMM policy. That is the reason why we introduce the "*extends"* [15] relation (and corresponding mechanism) in order to allow them to extend an existing metamodel by creating a new metamodel that inherit the properties of the base metamodel.

6. CONCLUSION

In this paper we have described how to implement a DSL with tools available from the AMMA platform. We learnt to use AMMA core DSLs (KM3, TCS and ATL) to define abstract syntax, concrete syntax and semantic of the iTV DSL. Along with the methodology and core DSLs for defining DSLs, we saw an example of tool being dynamically build to help the use of model based DSLs.

This implementation also shows that the complexity of model driven solutions to DSL building increase very quickly. We propose to use model driven engineering principles to solve this complexity issue and introduced the concept of megamodeling to this purpose.

7. REFERENCE

[1] Bézivin, J., Jouault, F., Kurtev, I., Valduriez, P.: Model-based DSL Frameworks. (2006)

[2] GME: The Generic Modeling Environment, Reference site, http://www.isis.vanderbilt.edu/Projects/gme. (2006)

[3] Karsai, G., Gray, J.: Component Generation Technology for Semantic Tool Integration. In: Proceedings of IEEE Aerospace 2000 Conference, Big Sky, MT, March. (2000)

[4] Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley (2004)

[5] Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: Modeling in the Large and Modeling in the Small. In Uwe Amann, Mehmet Aksit, A.R., ed.: Proceedings of the European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, LNCS 3599, Springer-Verlag GmbH (2005) 33–46

[6] Börger, E.: High Level System Design and Analysis using Abstract State Machines. In: FM-Trends 98, Current Trends in Applied Formal Methods. Volume 1641. (1999) 1–43

[7] Di Ruscio, D., Jouault, F., Kurtev, I., Bezivin, J., Pierantonio, A.: Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs. (2006)

[8] Jouault, F., Bézivin, J.: KM3: a DSL for Metamodel Specification. In: Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, Bologna, Italy. (2006).

[9] Budinsky, F., Steinberg, D., Ellersick, R., Merks, E., Brodsky, S.A., Grose, T.J.: Eclipse Modeling Framework. Addison Wesley (2003)

[10] ATLAS team: ATLAS MegaModel Management (AM3) Home page, http://www.eclipse.org/gmt/am3/. (2006)

[11] Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Satellite Events at the MoDELS 2005 Conference. Volume 3844 of Lecture Notes in Computer Science., Springer-Verlag (2006) 128–138

[12] Jouault, F., Kurtev, I.: On the Architectural Alignment of ATL and QVT. In: Proceedings of ACM Symposium on Applied Computing (SAC 06), model transformation track, Dijon, Bourgogne, France (2006)

[13] Boehm, B. Sherlis, W. MegaProgramming, in Proc. of the DARPA Software Technology Conference, (Arlington, Va, April 1992.).

[14] Bézivin, J., Jouault, F., and Valduriez, P., On the Need for Megamodels. In: Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications.

[15] Barbero, M., Jouault, F., Gray, J., Bézivin, J., A Practical Approach to Model Extension. In: Proceedings of the ECMDA 2007 conference, Haïfa, Israel, June 2007.