Supporting Tool Reuse with Model Transformation

Zekai Demirezen, Yu Sun, Jeff Gray
Department of Computer Science
Univ. of Alabama at Birmingham
Birmingham, AL, 35205, USA
{ zekzek, yusun, gray } @ cis.uab.edu

Frédéric Jouault
AtlanMod (INRIA & EMN)
Nantes, France
frederic.jouault@inria.fr

Abstract

Software components provide a wide range of functionality that can be used across several domains. In some cases, reuse at a very coarse level of granularity (e.g., reusing functionality provided within an existing tool) is desirable, but challenging to realize due to the interface boundaries of the tool and the unanticipated level of reuse. This paper describes our results in applying domain-specific modeling and model transformation to the tool reuse problem. Specifically, we describe our results in reusing the graphical layout functionality provided by one tool (called GraphViz) that does not exist natively in another tool, such as the Eclipse Graphical Modeling Framework.

Keywords: Domain-Specific Modeling, Tool Interoperability, Model Transformation

1. INTRODUCTION

Tools provide a wide range of functionality in the context of their domain. However, in some situations a tool may lack a specific desirable operation needed by a user. Other tools in the same domain may include these missing operations, or may provide an improved implementation of the specific desired functionality in a different context. This paper describes our results in reusing the functionality of an open source tool called GraphViz[1] to embed the layout functionality that we needed in Eclipse's Graphical Modeling Framework (GMF)[2]. We implemented an Eclipse plug-in that enables the execution of the GraphViz Layout algorithm during modeling operations in the GMF editors. The main challenge we encountered was how data could be exchanged between the tools. Because both GraphViz and GMF operate on their own unique data structures, we designed an exchange mechanism to adapt their diverse formats. The specific solution to these challenges involved the application of domain-specific modeling (DSM) [1] and model transformation to specify each tool format and to convert between the representations.

[1] http://www.graphviz.org/

[2] http://www.eclipse.org/modeling/gmf/

In tool interoperability, data exchange is accomplished over well-defined data structures and exchange formats [2]. However, different standards and formats make software interoperability a challenge. There are several existing approaches that can be adopted to realize tool interoperability. For example, a general approach that is based on traditional parsing and interpreting activities can be implemented with general-purpose programming languages (e.g., Java, C#). In addition, XML-based interoperability has emerged as a popular choice for a generic exchange format among software tools. However, although this works relatively well when all considered tools use some form of XML, it is not a convenient solution when being applied in other contexts (e.g., when context-free parsing is necessary to read from a proprietary file format). A new approach to interoperability, which is advocated in this paper, is based on domain-specific modeling and model transformation techniques, where the different formats are captured as: 1) abstract definitions of data structures (i.e., metamodels) and 2) concrete syntax definitions (e.g., context-free, XML), and finally 3) transformation rules mapping from one representation to another.

The majority of tool interoperability research has focused on the *data exchange* aspect of integration [3], [4], [5]. In fact, we recently investigated this type of tool interoperability by applying domain-specific modeling to enable sharing of information between a set of code clone tools [6]. The sharing of information among similar tools enables intellectual assets to form a homogenized collection of information. Although this issue of tool interoperability is important, there are other cases of interoperability that can yield additional benefit. In particular, the concept of *sharing functionality* (as an alternative to sharing just date) between tools can enable a tool chain whereby features not available in one tool can be realized through collaboration with other tools. The challenge of such sharing of functionality is generally twofold: 1) transforming the representation of a source tool into that expected by the target tool providing the desired functionality; 2) transforming the result back to the first tool (or possibly to a third one). This is challenging because reuse at such a coarse level of granularity transcends the boundaries of the tools in a way that was likely not anticipated when each tool was designed.

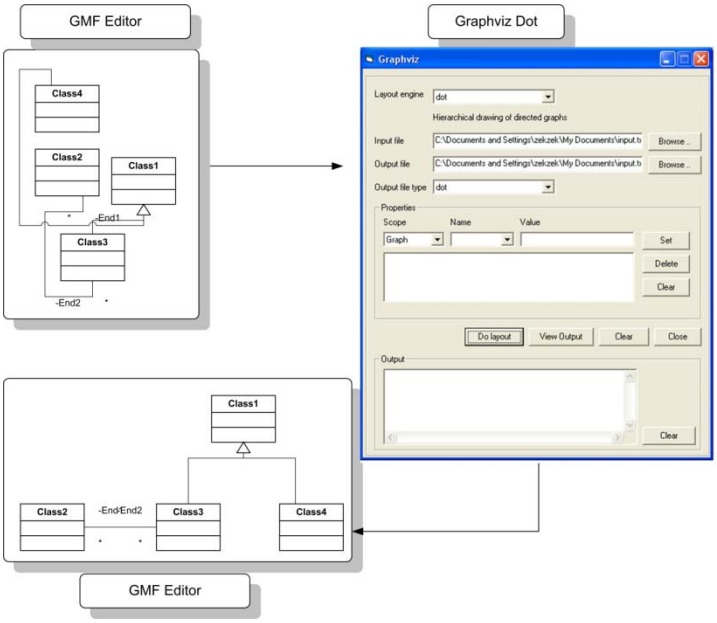

Figure 1. Overview of DSM-enabled tool interoperability applied to layout arrangement

We utilized DSM and model transformation to overcome interoperability issues between GraphViz and GMF, which provided a robust and maintainable solution that minimized tool adaptation effort. There are three main reasons that contributed to this reduction in effort:

1. In this specific context, both the source and final target information were already expressed as models. The GMF metamodel already exists in the format we needed for this integration process.
2. There is available tool support to translate between models and context-free syntax, such as the intermediate format corresponding to the input and output of the GraphViz tool. However, there is the additional cost of specifying this syntax in an appropriate formalism.
3. Moreover, our choice of domain-specific modeling and model transformation for this task is also motivated by the modularity and reusability such an approach provides. For instance, the GraphViz syntax specification may be reused in other contexts, such as providing graph layout capabilities to editors other than GMF. This is possible because the syntax specification is completely decoupled from the mapping rules.

The basic idea of the model-driven solution is to define the abstract syntax (represented as a metamodel) and the concrete syntax for GraphViz and GMF, and then execute associated transformations to map data between the tools. Figure 1 shows the effect of exporting an unorganized design model from GMF to GraphViz and importing the results of the arrangement suggested by the layout algorithm back into GMF.

According to this procedure, we first introduce and define metamodels for the two tools in Section 2. The detailed implementation of interoperability between these two tools is given in Section 3, with a brief introduction about the AmmA platform which we applied to realize the whole process. In Section 4, we conclude the paper and present some lessons learned from this experiment.

2. DOMAIN DEFINITION AND TOOL METAMODELING

This section provides an overview of the representation formats of GraphViz and GMF, which are the two tools that are considered in this interoperability study. The metamodel for each tool is specified in this section, with the corresponding mappings described in Section 3 as model transformations.

2.1 Layout DSL in GraphViz

GraphViz is an open source graph visualization application that has several layout programs for the placement of nodes and edges in graphs. One of the layout programs of GraphViz is the Dot Layout tool, which reads directed graphs, computes layouts, and produces an output of attributed graphs which includes layout coordinates [7].

Figure 2 and Figure 3 show a sample Dot file and the layout of this graph. Before the execution of the auto-arrange algorithm, the Dot file includes only node and edge definitions without position attributes (top of Figure 2). All the position data are inserted after the execution of the layout algorithm. For example, in the “Before Layout”

fragment, the *content -> chapter* and *content -> appendix* statements define two edges between a *content* node and the *chapter*, *appendix* nodes. The "After Layout Fragment" (bottom of Figure 2) is an excerpt from the definition of the graph as re-serialized by Dot. The excerpt only shows the *content* node and its edges (all other nodes are removed for space consideration) with generated attributes (i.e., *pos*, *width*, and *height*). In this fragment the *content [pos="185,162", width="0.92", height="0.50"]* fragment specifies position values of the *content* node within the graph. Other parts correspond to the definitions of edges linked to the *content* node as well as their position values. In essence, the changes between the "before" and "after" version of this graph representation correspond to the specific functionality that we desire to reuse in GraphViz.

Before Layout:

```
digraph G {
   book [shape=box];
   book -> title [style=dotted];
   book -> author ;
   book -> content [weight=8];
   content  -> chapter;
   content  -> appendix;
   chapter  -> section[color=red];
   section [shape=box, style=filled,
            color=blue];
}
```

After Layout:

```
digraph G {
...
<for space consideration, only the
content node is shown here>

content [pos="185,162", width="0.92",
         height="0.50"];
content -> chapter
           [pos="e,151,107 174,145
            169,137 162,126 156,116"];
content -> appendix
           [pos="e,218,107 196,145
           201,136 208,126 213,116"];
...
}
```

Figure 2. Sample graph with position attributes

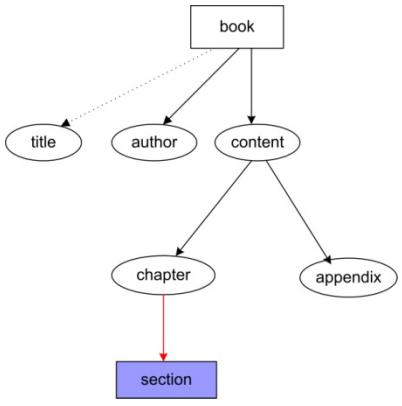

Figure 3. Layout of Sample Graph

To enable the reuse of the GraphViz functionality, the abstract representation of GraphViz is specified as a metamodel. Figure 4 shows the Dot metamodel, which we inferred by reading the user manual of Dot. The primary concept in Dot is a Graph consisting of Graph Elements that can be Nodes or Edges. Both of these kinds of elements can have attributes such as shape, label, position, width and height. The Attribute meta-element represents these attribute values.

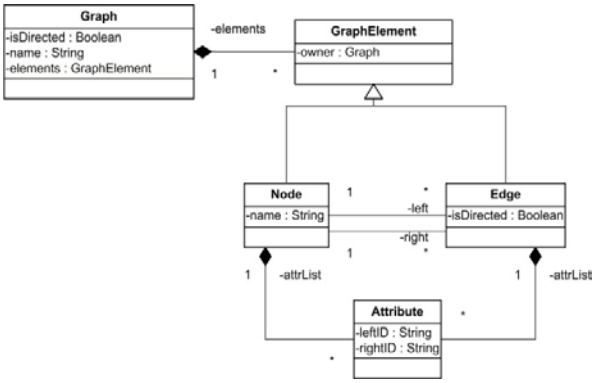

Figure 4. Metamodel for GraphViz Dot

2.2 Representation of GMF Concepts

The Eclipse Modeling Framework (EMF)[3] assists in building applications that are described by a data model that supports code generation. EMF provides tools and runtime support for manipulating models to produce a set of Java classes for the model. The meta-metamodel of EMF is Ecore. A subset of Ecore is shown in Figure 5. The Graphical Editing Framework (GEF)[4] can be used to design a graphical editor for a specific data model. The Eclipse Graphical Modeling Framework (GMF) provides a supporting infrastructure for developing graphical editors based on EMF and GEF. Models in GMF are defined by an Ecore metamodel.

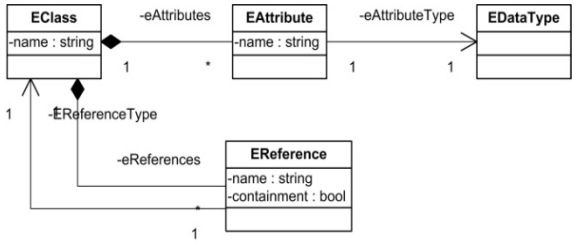

Figure 5. Subset of the EMF Metamodel [8]

[3] http://www.eclipse.org/modeling/emf/

[4] http://www.eclipse.org/gef/overview.html

3. ATL TRANSFORMATIONS SUPPORTING TOOL REUSE

We defined the required metamodels and implemented model transformations using the AtlanMod Model Management Architecture (AmmA) [9], which enabled us to accomplish all modeling steps in one complete environment. AmmA is a model engineering framework that captures domain-specific languages (DSLs) as coordinated sets of models. Model engineering uses metamodels to represent abstract syntax by describing domain concepts and their relations. The Kernel MetaMetaModel (KM3) [10] is the metamodeling language of AmmA. KM3 is comparable to the Meta Object Facility (MOF) metamodeling language proposed by the Object Management Group. KM3 is also similar to Ecore, which is used in the Eclipse Modeling Framework (EMF). Metamodels defined in KM3 may be used directly or be transformed into their MOF or Ecore equivalents. After the abstract syntax for different languages or tools is defined, it is often necessary to transform a source model into a separate target model. Model transformation enables the specification of executable mappings between metamodels. AmmA provides ATL (AtlanMod Transformation Language) [11] to specify transformations declaratively. Rules are used to specify the kind of target elements that have to be created for each of the kinds of source elements.

3.1 Representing Abstract and Concrete Syntax

Metamodeling and model transformation are not enough to deal with concrete cases. The reason is that in most cases source and target models are not already specified as abstract graphs, but rather as concrete files using specific formats (e.g., based on grammars or on XML schemas). AmmA offers several tools (called projectors) to deal with concrete syntax. Such tools perform projections, which are two special kinds of transformations: 1) an injection produces a model from a concrete file; 2) an extraction takes a model as source and creates a concrete file as target. One such tool is TCS (Textual Concrete Syntax) [12], which enables the specification of context-free concrete syntax. Once a TCS model has been specified, it is possible to inject programs as models, and to extract models as programs. AmmA also provides an XML metamodel and its associated projectors. The XML injector takes any XML document as input and produces a corresponding model, whereas the XML extractor does the opposite by producing XML documents from XML models. The examples presented in this paper are based on AmmA.

At the beginning of our project we defined metamodels for each tool (Figure 4 and Figure 5; shown here as UML models, but their specific definition is in KM3). An existing GMF metamodel definition allowed us to reuse this definition as input to ATL transformations without further specification, because ATL is compatible with Ecore. However, we required the Dot metamodel definition in KM3 to provide a target for the required transformations.

3.2 Defining Model Transformations for Reuse

In AmmA, transformation operations require models to be loaded before the transformation executes. Because Dot files have a specific format, there is a need to parse and load them into models that conform to the metamodel of Figure 4. TCS injection and extraction methods facilitate these activities. We defined the concrete syntax of the Dot format in TCS. An excerpt from the TCS definition of Dot is shown in Figure 6. Each template provides the concrete syntax specifications of a meta-element. For example, the Graph template enables Graph attributes to be parsed and also pretty-printed (i.e., TCS works both ways). Graph elements are listed in a Dot file between the brackets and are separated by semi-colons.

We defined two transformation specifications for this case study (GMF2Dot.atl and MergeDotPositionIntoGMF.atl). Figure 7 shows the GMF2Dot definition that specifies the EMF Ecore (Figure 5) to Dot (Figure 4) meta-element mappings.

```
template Graph main context :
         (isDirected ? "digraph":"graph") name "{" elements {separator = ";"}
         [[ ";" | ]] "}";
template GraphElement abstract;
template Attr_Stmt abstract;
template Node addToContext : name attr_list;
template Edge :
         left{refersTo = name, autoCreate = ifmissing, createIn = '#context'.elements}
         (isDirected ? "->" : "--")
         right{refersTo = name, autoCreate =ifmissing, createIn = '#context'.elements}
         attr_list;
template Edge_Attr_Stmt context : "edge" attr_list;
template Graph_Attr_Stmt context : "graph" attr_list;
template Node_Attr_Stmt context : "node" attr_list;
template A_Item_List context : "[" a_item_list {separator = "," } "]";
template A_Item context : leftID isDefined(rightID)? "=" rightID);
```

Figure 6. Excerpt of the Dot TCS

Each rule defines mapping an element from a source metamodel to an element from a target metamodel. For example, the Node2Node mapping provides a transformation between Ecore Node elements and Dot Node elements. In this rule, Node type corresponds to the Class Node in the GMF metamodel. Edge-to-edge transformation is accomplished by checking the edge type because of the fact that the association Edge is an Edge in the GMF metamodel, which has type AssociationLink. Finally, edge source and target meta-elements, which are Node elements, are mapped to Dot Node elements to complete the Ecore Edge to Dot Edge Transformation.

```
rule Node2Node{
  from n : GMF!Node
         (n.type = 'ClassNode')
  to   c : Dot!Node
         (name <- n.element.name)
}
rule Edge2Edge{
   from n : GMF!Edge
          (n.type =AssociationLink')
   to   c : Dot!Edge
   (left <- n.source, right <- n.target)
}
```

Figure 7. Excerpt of the GMF2Dot.atl

Figure 8 shows an excerpt of the MergeDotPositionIntoGMF transformation. This transformation is a merging activity, which combines a Dot and GMF model into a refining model to generate the resulting GMF model. Imperative statements are implemented to copy elements in entrypoint rule Main. During the copy, generated position values (*x, y, width* and *height* attributes) are assigned into corresponding node elements of GMF attributes. In this rule, the GMF!Bounds helper function is used to query position information from the Dot model.

3.3 Overview of Transformation Process

Figure 9 shows the complete GMF to Dot transformation scenario. The GMF metamodel defines GMF models and the Dot metamodel defines GraphViz models. To summarize, the steps of the required transformations are as follows:

- Defining merge transformation of source and output models (arranged model) with ATL (MergeDotPositionIntoGMF.atl)
- Defining the Dot metamodel (Dot.km3) (Note: GMF metamodel already exists)
- Defining the Dot concrete syntax (Dot.tcs)
  - Required to extract Dot models in AmmA to Dot models in GraphViz
  - Used to inject Dot models in GraphViz to Dot models in AmmA after the execution of auto layout
- Defining the transformation of GMF models to Dot models with ATL (GMF2Dot.atl)

The GMF model is transformed using GMF2Dot.atl. The generated Dot model is extracted and manipulated in Graphviz. The Dot.exe program, which is in the Graphviz tool, is executed to arrange the model. The manipulated model is injected into the AmmA platform. The model that has been auto-arranged and the existing model are merged to produce the final GMF model (using MergeDotPositionIntoGMF.atl).

```
helper context GMF!Bounds
def: dotPosition : TupleType (x:String,y:String,w:String,h:String) =
let data:Set(Dot!Node) = Dot!Graph.allInstances()
    ->first().elements
      ->select(e|e.oclIsTypeOf(Dot!Node))
        ->select(e|e.name = self.refImmediateComposite().element.name)
in
let data2:Set(Dot!A_Item) = data->first().attr_list->first().a_item_list
in
    Tuple{
     x = data2->select(e|e.leftID='pos').first().rightID->first(),
     y = data2->select(e|e.leftID='pos').first().rightID->last(),
     w = data2->any(e|e.leftID='width').rightID,
     h = data2->select(e|e.leftID='height').first().rightID
    }
entrypoint rule Main() {
do {
for(b in GMF!Bounds.allInstances()) {
   b.x <- b.dotPosition.x;
   b.y <- b.dotPosition.y;
   b.width <-(b.dotPosition.w.toReal();
   b.height <- (b.dotPosition.h.toReal();
 }
}
```

Figure 8. Excerpt of the MergeDorPositionIntoGMF.atl

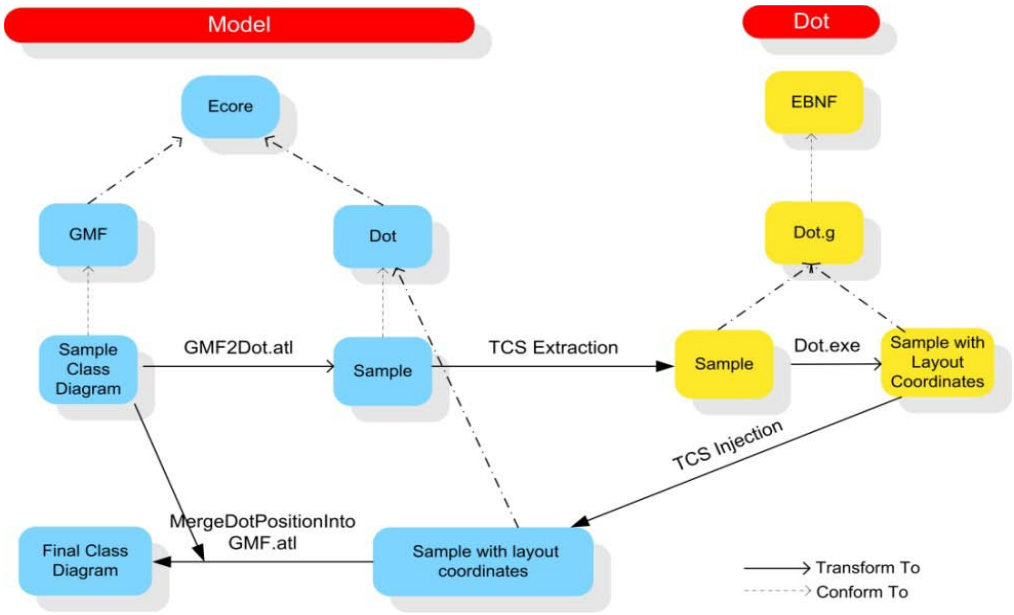

Figure 9. Overview of the Dot-GMF Interoperability Implementation

Imperative statements are implemented as mappings to copy elements. During the copy, generated position values are assigned to corresponding node elements of GMF. At the end of these steps, a GMF model has been auto-arranged using Dot algorithms. With ATL, by defining formal transformation rules, this case study has demonstrated that an algorithm available from the Dot tool can be used in the GMF tool. Details about the complete implementation described here is available at our project website at:

http://www.cis.uab.edu/zekzek/gmf-dot/

4. CONCLUSION

At the beginning of the project we knew that it would be difficult to reuse the functionality of tools like GraphViz due to the coarse grained nature of such reuse. Because such tools operate as stand-alone applications, we needed to understand their interfaces and assumptions to enable the reuse of functionality. Domain-specific modeling, coupled with model transformation, enabled us to focus on the clear and organized mapping structure between meta-elements of the tools. The declarative specification of models and transformation rules were the primary representation that enabled the reuse of tool implementation, rather than implementing conversion tools implented in a general-purpose language. This raised the abstraction of the problem description to a more appropriate level.

Although this paper is focused on the interoperability of two very specific tools (i.e., GMF and GraphViz), the general principles of metamodeling and model transformation are applicable to many different domains and tool integration efforts. Because the realization of interoperability also depends on the data used by each tool, parsing the data to inject into a model is an indispensible step for model transformation to enable tool interoperability. However, not every tool provides a strutured data format that can be parsed. Some tools may not have an explicit data model that can be ised; some others may offer a special format that is not even context-free, which is challenging to process. Therefore, how to parse and process these kind of data formats and support such types of tools is the key to future enhancements of our approach. In addition, we also encountered support and usage limitations of our chosen modeling environment. AmmA is one of the most mature model engineering/DSL tool suite. However, it is not as mature as IDEs for general purpose languages like Java or C#. Although it provides most of the basic model activities such as load, edit, build, store, and execution, lack of debugging makes it hard to identify errors in either the metamodel definition or the transformation specification.

5. Acknowledgements

We would like to thank the National Science Foundation (NSF-CAREER-0643725) for supporting this work.

6. REFERENCES

[1] Gray, J., Tolvanen, J., Kelly, S., Gokhale, A., Neema, S., Sprinkle, J., Handbook of Dynamic System Modeling, (Paul Fishwick, ed.), CRC Press, Chapter 7, pp. 7-1 through 7-20, ISBN: 1584885653 (2007)

[2] Benguria, G., Larrucea, X.: Data Model Transformation for Supporting Interoperability. In: International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems, pp. 172-181, Alberta, Canada (2007)

[3] Del Fabro, M., Bezivin, J., Valduriez, P.: Model-Driven Tool Interoperability: An Application in Bug Tracking. In: International Conference on Ontologies, Databases, and Applications of Semantics, pp. 863-881, Montpellier, France (2006)

[4] Karsai, G., Gray, J.: Component Generation Technology for Semantic Tool Integration. In: IEEE Aerospace Conference, pp. 491-499, Big Sky, Montana (2000)

[5] Tratt, L.: Model Transformations and Tool Integration. Software and Systems Modeling 4, 112-122 (2005)

[6] Sun, Y., Demirezen, Z., Jouault, F., Tairas, R., Gray, J.: A Model Engineering Approach to Tool Interoperability. In First International Conference on Software Language Engineering (SLE), Springer-Verlag LNCS 5452, pp. 178-187, Toulouse, France (2008)

[7] Gansner, E., North, S.: An Open Graph Visualization System and Its Applications to Software Engineering. Software: Practice & Experience (Special Issue on Discrete Algorithm Engineering) 30, 1203-1233 (2000)

[8] Budinsky, F., Steinberg, D., Ellersick, R., Merks, E., Steinberg, D., Grose, T.: Eclipse Modeling Framework. Addison-Wesley, Reading (2003)

[9] Kurtev, I., Bézivin, J., Jouault, F., Valduriez, P.: Model-based DSL Frameworks. In: International Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 602-616, Portland, Oregon (2006)

[10] Jouault, F., Bézivin, J.: KM3: A DSL for Metamodel Specification. In: Proceedings of the International Conference on Formal Methods for Open Object-Based Distributed Systems, pp. 171-185, Bologna, Italy (2006)

[11] Jouault, F., Allilaire, F., Bézivin, J., I. Kurtev: ATL: A Model Transformation Tool. Science of Computer Programming. 72(1-2): 31-39 (2008)

[12] Jouault, F., Bézivin, J., Kurtev, I.: TCS: A DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In: International Conference on Generative Programming and Component Engineering, pp. 249-254, Portland, Oregon (2006)