MODIGEN: MODEL-DRIVEN GENERATION OF GRAPHICAL EDITORS IN ECLIPSE

Markus Gerhart[1] and Prof. Dr. Marko Boger[2]

[1]Department of Applied computer science, University of Applied Sciences, Konstanz
[2] Department of Applied computer science, University of Applied Sciences, Konstanz

ABSTRACT

Domain-specific modeling is more and more understood as a comparable solution compared to classical software development. Textual domain-specific languages (DSLs) already have a massive impactin contrast tographical DSLs, they still have to show their full potential. The established textual DSLs are normally generated from a domain specific grammar or maybe other specific textual descriptions. And advantage of textual DSLs is thatthey can be development cost-efficient. In this paper, we describe asimilar approach for the creation of graphical DSLs from textual descriptions. We present a set of speciallydeveloped textual DSLs to fully describe graphical DSLs based on node and edge diagrams. These are, together with an EMF meta-model, the input for a generator that produces an eclipse-based graphical Editor. The entire project is available as open source under the name MoDiGen.

KEYWORDS

Model-Driven Software Development (MDSD), Domain-Specific Language (DSL), Xtext, Eclipse Modeling Framework (EMF), Metamodel Model-Driven Architecture (MDA), Graphical Editor

1. INTRODUCTION

Domain-**S**pecific **L**anguages (DSLs) have a crucial importance in **M**odel-**D**riven **E**ngineering (Also known as **M**odel **D**riven **S**oftware **D**evelopment (MDSD) or **M**odel **D**riven **D**evelopment (MDD)) (MDE) and **M**odel-**D**riven **A**rchitecture (MDA). A survey of MDE practitioners [24] shows that MDE users make use of multiple modelling languages. Nearly 40% of participants had used specially design custom DSLs, 25% had used off-the-shelf DSLs and only UML had been used more widely than DSLs (used by 85% of participants). Despite the significant assumption of specially designed DSLs for MDE, we think that existing MDE/MDA solutions for the definition of custom DSLs areunsuitable. In particular, for the design and generation of a node-and-edge based graphical DSL.

Despite the significant adoption of custom DSLs for MDE, we think that existing MDE/MDA solutions for implementing custom DSLs and supporting tool chains are unsuitable for designing and developing a new node-and-edge graphical DSL. This is due to the high complexity and the range of applications of existing solutions. There is currently no solution which uses MDSD for the specific domain of the generation of domain-specific graphical editors.

Eclipse has become one focus point of tooling for model driven approaches. Projects like **E**clipse **M**odeling **F**ramework (EMF) [2] and Xtext [3] have become very popular in the modelling community. But most of the current success in Eclipse modelling has been centered around textual modelling environments. While Eclipse is designed for textual languages it can play out its textual nature very nicely for this approach. But it is a difficult environment to create graphical

DOI:10.5121/ijcsit.2016.8506

modelling tools. The basic APIs provided by the Eclipse framework; Graphical**E**diting **F**ramework (GEF) [5] and Draw2D [4]; are low level and grant no direct connection to the semantic and abstract level of a model.

The Eclipse Modelling Frameworkis often used for the semantic part, but offers no specific support for graphical modelling. A project that attempt to bridge the gap between GEF and EMF is the **G**raphical **M**odeling **F**ramework (GMF) which is now part of the **G**raphical **M**odeling **P**roject (GMP) [6]. It proposes a model-driven approach for the development of domain specific graphical modeling tools on the basis of GEF and EMF. But the domain specific models including thedescription of the editor are to be so complex that projects evolved to create these needed models from a higher perspective (abstract models). The resulting generated artifacts (program code) was very complex and very difficult to extend for additionally custom enhancements which could not be designed inside the model. The development environment was brittle and cumbersome. The GMP provides currently no specific textual or graphic language to associate a Metamodel with a predefined textual description with the aim to generate a domain specific Graphical editor.

We present in this paper a generative approach for the creation of graphical modeling tools in Eclipse. Instead of using the low level APIs of GEF and Draw2D directly, we use the relatively young Java framework Graphiti [7] which is in the meantime also part of the Graphical Modeling Project. Graphiti provides a API to build graphical editors in Eclipse and hides the complexity of the low level tools such as GEF and Draw2D. However, while it is nearly perfectly possible to create a graphical modeling environmentwith the Graphiti framework, we also think that the API is well design to generate against it from a higher abstraction level. We developed a set of textual modeling languages, which serve as input for a generator. The output of the generator is nearly plain Java code for the API provided by Graphiti as well as all other needed files for the Eclipse plugin mechanism. The result is a graphical modeling tool in Eclipse for custom-designed node-and-edge-type diagrams.

With this approach we can reduce the effort to develop a graphical modeling tool so much that the development of a graphical DSL becomes cost-efficient. Acomparable graphical modeling environment that is manually developed with the Graphiti framework might require about 20.000 lines of code, our approach generates the needed code-artefacts from only about 500 lines of the different textual model descriptions. Thus the cost to develop a graphical modeling tool is reduced so much, that it may be a viable alternative to a textual DSL. Overall we hope to foster the use of graphical DSLs in MDE-projects and to make the model-driven approach to software development more attractive in general.

The paper first provides a short overview of the used technologies in the Section 2.Subsequently the paper reviews related work in the field in Section 3, which is mostly other tools and techniques for the generation of modeling tools. Our general approach for the model driven creation of the domain specific editors and the overall architecture of our framework is described in Section 4. The core contribution of this publication are the developed DSLs to define our own graphical elements, styles for the graphical elements and the graphical editor itself which are described in the sub chapters of the general approach. Section 5 illustrates the results of our approach from different angles. Finally, we summarize the limitations of our research and draw conclusions in Section 6.

2. BACKGROUND

This section explains shortly the referred techniques, libraries and frameworks.

The **M**odel-**D**riven **E**ngineering (MDE) is concerned with the automation of software production. This means that as many artifacts of a software system are derived from formal generative models. (inspired by [32])

Model-**D**riven **A**rchitecture (MDA) is a concrete approach of the **O**bject **M**anagement **G**roup (OMG) [14], which describes a model-driven approach using its own standards (e.g. UML, MOF, XMI).

A **D**omain **S**pecific **L**anguage (DSL) [34] or application-specific language is a formal language, which is designed and implemented for a specific problem area (the so-called domain). The main goal of the design is to achieve a high degree of problem specificity. The DSL should be able to represent all the problems of the domain and nothing outside of the domain.

The **E**clipse **M**odeling **F**ramework (EMF) [35] is a framework based on the Eclipse platform, that is primarily used for model-based code generation. It ranks as the most important component of the Eclipse Modeling Project, which covers the top level Project from Eclipse the field of model-based development.

Graphical **E**diting **F**ramework (GEF) [36] is a low level framework based on the Eclipse platform for creating graphical editors within Eclipse. Furthermore, GEF helps to create graphical editors based of an existing data model, which could be design inside of the Eclipse Modeling Framework. GEF uses the Draw2d-Toolkit for the graphical representation.

Draw2d[37] is a lightweight framework for displaying graphical elements within the Eclipse environment based on SWT Canvas.Lightweight means in this case that all graphical elements, which Draw2d calls *figures*, are simple java objects with different functions for modification and no corresponding resource in the operating system.The design goal of Draw2d is the creation of vector graphics withinthe Eclipse ecosystem. The framework is part of GEF, but can also be used independently.

Graphiti[38] combines the functionality of EMF and GEF, but hides the complexity of using GEF behind a lightweight but not so powerfulAPI.The diagrams are described by a metamodel and the diagram data is strictly separated from the actual data. This enables the opportunity of rendering an existing diagram in different environments without having direct access to the actualmetamodel instance data.

The Eclipse **G**raphical **M**odeling **P**roject (GMP) [39] provides a set of generative components and runtime infrastructures for developing graphical editors based on EMF and GEF.Thus, the Graphical Modeling Project represents the collection of all provided frameworks and technologiesby Eclipse, which are required for the generation of a model-driven solution. This project combines all model driven approaches by eclipse in one single project.

Xtext[40] is aeclipse plugin for the development and creation of textual domain-specific languages or simple programming languages.The Pluginusesparts from the Eclipse Modeling Project such as EMF, GMF, M2T and parts of EMFT. Thedevelopment with Xtext is easy to learn and provides a smooth start into the model driven engineering.

3. RELATED WORK

Graphical Modeling has received a lot of attention with the standardization of UML and BPMN as a way to design or document technical systems. However, they pose the problem that the systems tend to change independent from the models and eventually diverge. Model-driven

approaches alleviates this dilemma by making the model source, generating code from the model. Thus model-driven software engineering has been an active field of research and development.
An obvious approach is to use UML or BPMN directly, but it turns out that these general modeling languages are often not specific enough to catch all cases of a domain. For this reason, the UML version 2.0 included a mechanism to extend the metamodel through stereotypes and profiles. Such extensions have to be built on the basis of the predefined UML metamodel elements. This makes it difficult to address the model instances from the generator and makes the generator difficult to maintain. The tools most used for this approach are Enterprise Architect [9] and Magic Draw [11]. However, most generator technologies are centered around Eclipse, and building a solid tool chain can be challenging.

Tools that allow for the development of domain specific graphical modeling languages include MetaEdit+ [31] and Poseidon for DSL [8]. The integration into an Eclipse-based tool chain is achieved by an export of the resulting model into a serialized form, usually as XMI.

Eclipse has traditionally been a difficult environment to integrate graphics. Tools and frameworks to build graphical modeling tools based on Eclipse exist. The fundamental frameworks are GEF and Draw2D. Building such tools directly on GEF and Draw2D provides very little infrastructure support and is accordingly very work intensive. The aforementioned GMF improves this through a generative approach against these APIs. However, the DSLs, the generator and the generated code of GMF have themselves high complexities. Extending GMF beyond the default generated behavior can become particularly painful. Tools like Kybele [33] or Eugenia [27][28] also use a generative approach to build graphical editors for Eclipse. The generated code of these solutions is difficult to expand and to understand as these build on the low-level APIs GEF or Draw2d. Their cost-effectiveness drops if the desired functionality is not provided by the default behavior.
The non-commercial tool **M**eta **P**rogramming **S**ystem (MPS) [12] of the manufacturer Jet brains provides a range of opportunities and freedoms in the design of their own meta-models. However, due to the variety of possibilities the complexity of the tool is very high.

Another approach is to drop the use of a graphical modeling environment and instead use a textual language to express the DSL models. This approach seems to be very successful in practice. Most known tools are Xtext [3], Spoofax[26] and EMFtext [25].

This paper advocates the approach to build graphical editors directly in Eclipse. We also use a generative approach, similar to GMF, Kybele or Eugenia, but in contrast to these we avoid to generate directly against the fundamental libraries GEF and Draw2D. Instead we use a framework called Graphiti. It is relatively young and has been developed by SAP [13]. The framework hides the complexity of GEF and Draw2D and adds a high level graph-oriented API. It was intended for the manual creation of graphical editors inside Eclipse, but it naturally lends itself to a generative approach. This is what our project proposes. The focus of this paper is on the input to the generator. This consists of a set of languages to describe node-and-edge-type graphical modeling languages.

We have identified a similar approach with the project called IMES [10]. The aim of the publicly funded project is to build graphical editors based on Graphiti for functional nets and other systems. The project also uses DSLs and model-driven development (MDD) for the generative creation of the graphical editors. Currently the project is not open source, wasn't presented to the public and there was no further development since 2011, so it is difficult to compare our approach with the IMES project.

4. APPROACH

The MoDiGen project consists of several DSLs, a generator and a runtime environment. The development environment is Eclipse with a set of plugins. All essentialtools can be found on the homepage of the project, together with installation instructions. For the main part of the project, the generator, we use a language which is similar to Javaand was specifically designed and developed for the purpose of generating code artefacts, called Xtend[3]. The runtime consists of Graphiti and some extensions which we have to develop, because this functionality iscurrently not part of Graphiti. The core contribution of this paper is on the designed textual DSLs, which are needed for the development of a domain specific graphical editor and not directly on the generated code or the runtime libraries. The DSLs are developed using Xtext.

MoDiGen [1] currently contains three different DSLs- *MoDiGen Core*, *Shape* and *Style* as shown in Figure 1, with different design goals. The central DSL is the MoDiGen Core language. For very simple domain specific graphical editors, this DSL is sufficient. It defines the mapping of simple shapes, styles and the behavior of elements to metamodel classes. For shapes that are more complexthan a rectangle or circle shape, the *Shape* DSL is used. The Shape DSL is developed todesign complex shapes which consist of primitive forms like rectangles and ellipses. Additionally, it's possible to configure placings, resizing policy's and nesting. The actual design of shapes, like color and font, can be defined inline in the Shape or separated in a style description. In both cases the Style language is used to design the layout of the elements.

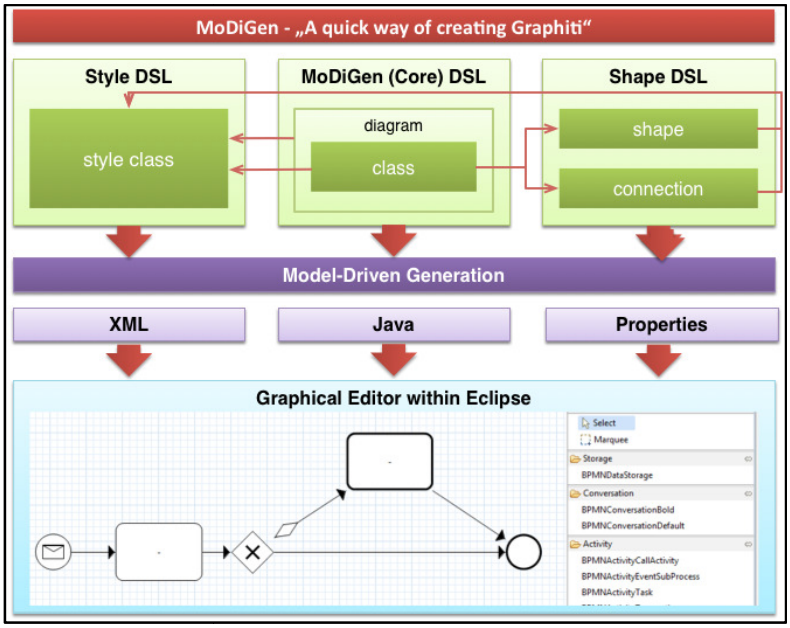

Figure1.A Model-Driven approach for Graphical Editors

The role of the Styles DSL is similar to how **C**ascading **S**tyle **S**heets (CSS) relates to **H**yper**t**ext **M**arkup **L**anguage (HTML). Defined style descriptions can be referenced from all other DSLs. Every time any of these models is saved, the *Model-Driven Generation* is triggered and generates all neededartefacts and configurations –*Java Code*, *XML files* and *Properties* - for the Eclipse Plugin mechanism. The *Graphical Editor within Eclipse* is then ready to use as plugin in Eclipse. The three designed languages are described below in detail.

4.1. THE MODIGEN CORE LANGUAGE

The most fundamental properties of a graphical editor are defined in the MoDiGen Core language. This is a simple text file with the ending.*modigen*. The MoDiGen project wizard creates such a file initially with some example code. Due to the file extension the Xtext environment offers features like code completion, syntax highlighting and different other conveniences, which makes the languages easy to learn.

At the beginning of the MoDiGen model description the *diagram* keyword initializes the declaration of the diagram type, followed by aunique identifier of the EMFmetamodelEClass (see Figure 2). This class serves as the root class for all further defined classes. The diagram type is used in the generated code as the name of the Graphiti diagram. Afterwards the EClass and EReferenceof the metamodel are mapped to their defined appearance and associated behavior on the diagram. The mapping is initiatedby the keyword *class* followed by the fully qualified EClass or EReferencedescription. This is mapped to a shape, which can be done in one of two ways. For simple cases the MoDiGen Core DSL allows the definition of shapes consisting of rectangles, text and lines. For more elaborate shapes the mapping can reference a shape defined in the Shape DSL. These defined shapes can be referenced by their unique description. This will be explained later in the paper. It is recommended to define the shapes in the appropriate shape language. As a result, the clear separation between behavior (MoDiGen core DSL), form (shape DSL) and design (style DSL) is observed. Only for very small projects, it is useful to break this rule.

```
import BPMN.*
diagram bpmn for BPMNModelElement style BpmnDefaultStyle {
   class BPMNActivityCallActivity {
      shape BPMN_Activity_CallActivity {
         name into shapeName
      }
      behavior {
         create into modelElements
         palette "Activity" askFor name;
      }
   }
   class BPMNConditionalFlow {
      connection BPMN_ConditionalFlow {
         from formElement;
         to toElement;
      }
      behavior {
         create into modelElements
         palette "Connections";
      }
   }
}
```

Figure2.MoDiGen File Example content

After the definition of some shapes for the new graphical editor, it's possible to define somebehaviors of the different shapes. Therefore there are different types of behaviorsfor example the creation behavior, which defines in which metamodel Attribute the instance is saved or the palette behavior which specifics the categorization of the elements in the generated editor. Also custom behaviors can be defined. For this, the necessary structural code is generated, the code defining the behavior needs to be written in Java manually.

4.2. DEFINITION OF SHAPES

The core MoDiGen DSL allows the definition of simple (basic) forms, but if figures with a higher complexity are needed there is a special DSL for the creation of shapes. The shape DSL has to some degree a similarity to SVG, but the design goal differs. Similar to SVG is, that the Shape DSL allows also primitive shapes like polygons,ellipses, rectangles, lines and polylines which can be combined to complex shape structures. The biggest difference is that SVG is based on XML. While the shape DSL is based on a context-free grammar. This gives us more freedom in design respectively more opportunities for future enhancements. The two main goalsare readability and fast learning and not machine readability for the fast programmatic processing. Furthermore, we had requirements which SVG currently cannot satisfy, like the definition of the passing of parameters totext fieldsor resizing policies or the programmatic analysis of properties. The re-sizing can be influenced with the attribute stretching in horizontal and vertical orientation (true or false). In addition, it is also possible to only allow proportional resizing. Shapes can be nested, allowing for complex graphical elements. The behavior of nested elements in regards to scaling can be defined by the attribute *layout*. It can have the values fixed (no adjustment), vertical, horizontal, and fit (according to an algorithm). Shapes can make use of nesting by re-using existing shapes or by in-place definition. Opening and closing curly brackets and are used to mark nesting consistently throughout the DSL. All forms except for lines, polylines and text can arbitrarily be nested. The Shape DSL is consistently used for the description of node elements as well as for connections. The following section explains the definition of shapes in detail.

4.2.1. DEFINITION OF SHAPES

The definition of a shape starts always with the predefined keyword *shape*and is followed by a unique shape identification (in Figure 3 the Shape is called *BPMN_Event_Mail*). This shape identification is used as name of the generated Java class, therefore is recommended to use no special characters. This class implements a marker interface *IShape* and extends the *DefaultShape* class. The interface description provides two different*getShape()*functions: the first one returns the *ContainerShape* and the second one the *PictogramElement*. The *ContainerShape*serves as a container which is added as a "placeholder" to the diagram. It contains aconcrete*PictogramElement* and all its possible nested elements. It is also possible to save more configuration information such as anchor definitions for the connection anchor options. The functionthatoffers the *PictogramElement* always contains on the root level an invisible rectangle, because an*ContainerShape* does not allow to save more than one element on the top layer. So the top layer always contains an invisible rectangle which is able to save multiple child's in it. The actual size of the invisible rectangle is calculated from contained shapes on every different layer.

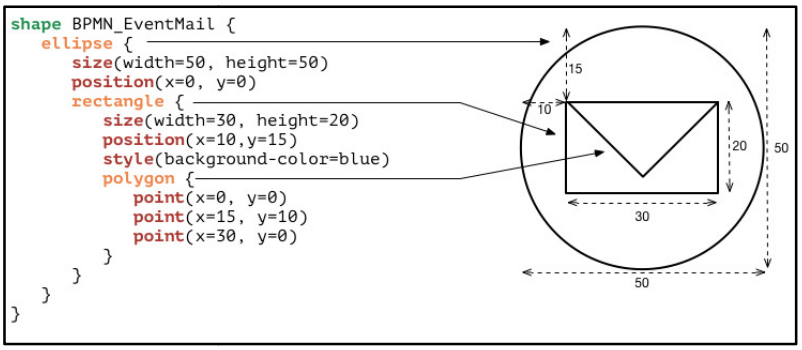

Figure3.Creation of a Shape as *mail event* (BPMN)

The example in Figure 3 describes how to create an envelope for the *mail event* of the **B**usiness **P**rocess **M**odel and **N**otation (BPMN) [29] with the Shape DSL. The entire figure is inside a circle. The circle is built with an ellipse with the samevalue for width and height. Nested into this circle is a rectangle, which defines the border of the envelope. The position of the rectangle is expressed relative to the surrounding ellipse. Finally, a polygon completes the entire envelope.

4.2.3. ANCHORAGE ON SHAPES

Anchor points are an important aspect for the design of shapes. Anchor points serve as points where connections can be attached to a shape. The Shape DSL offers four different options for the definition of anchor points. There are two predefined anchor types which are called *center* and *corners* (see Figure 4). The keyword *center* is comparable to the Chopbox anchor in [5]. This definition attaches the connection end point always to the center of the figure/shape.But the connection line and its decoration ends on the outermost shapes border. The second predefined keyword of an anchor type is *corners*.This predefined value creates anchor points on all corners of the invisible rectangle as well as on the middle point of the four border sides of the shape. Figure 4 shows the different anchor types.

After the keyword *anchor* the predefined anchor values*center* or *corners* can be declared or a curly bracket is used to define custom anchor positions. Any number of anchor points can be defined inside the curly brackets. The custom anchor points can be set to fixed coordinates (using *x* and *y*) or to a relative position (using *xoffset* and *yoffset*). Fix point anchors are placed at exactlythe defined position within the created shape. The relative anchor definitions are placed in relation to the size of the shape with values between 0.0 (*xoffset* - left or *yoffset* - top) and 1.0 (*xoffset* - right or *yoffset* - bottom). The lower part of Figure 4 shows the creation of anchors on the edges of the rhombus for the *XOR-Gateway* in BPMN. As shown in the Figure 4 on the right side the same anchor point can be defined fixed or relative.

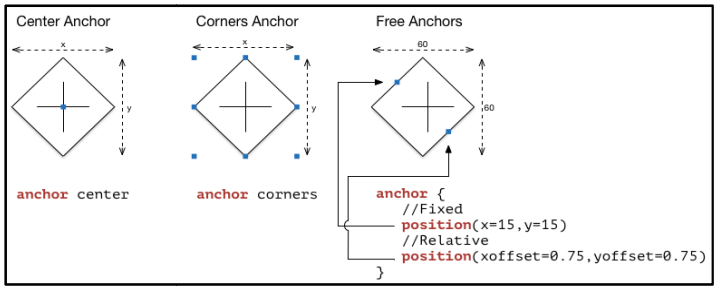

Figure4. Anchorage on Shapes

4.2.4. CONNECTIONS AND PLACINGS

A connection always contains a line, which connects two different or also the same shape(s). Any connection must have always a source and a target anchor point, which attaches the connection to a shape. Additionally, connections can have decorationse.g. arrowheads or any kind of a shape. The example of BPMN describes a conditional connection, shown in Figure 5. This connection has two different decorations. At the one end the arrowhead and the rhombus at the other end. It's also possible to place text fields directly on the connection to define diagram types which needs e.g. cardinalities or connection descriptions.

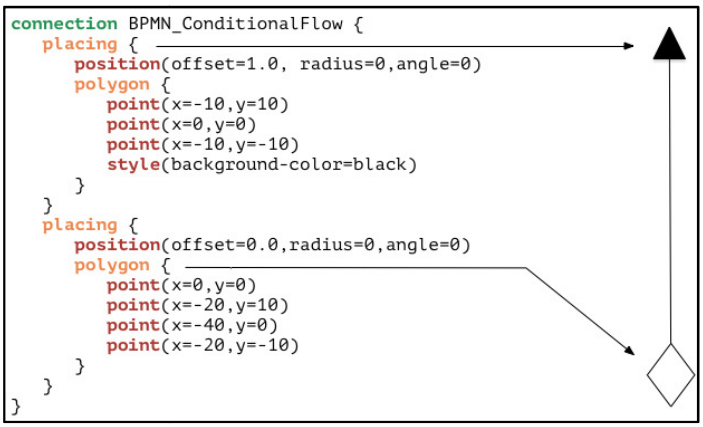

Figure5.Connection-Shapes and their Placings

In Shape DSL connections are defined using the keyword *connection*. It can be followed by a routing type. Two routing types are currently predefined, the free form and the Manhattan routing. The Manhattan routing is auto-layouted in vertical and horizontal lines and uses rounded corners. The free form connection is the easiest routing option because it's just a straight line, which can be customized by the user by adding bend points to the connection.

Placings can be definedon or around a connection. Therefore, a placing is a definition for a decoration on the connection. The actual position of a placing on a connection depends on three different required attributes (offset, angle and radius). The offset defines the relative position on the line where the decoration owns it's starting point. This relative value ranges between 0.0 (beginning) and 1.0 (end). Additionally, the placing can be positioned around the connection with the definition of the angle(0-360 degrees)and radiusattribute. For graphical decorations directly on the line, these values have to be zero.But for the positioning of text fields or maybe other shapesaround the line, this is very helpful. The combination of angle and radius define a vector starting at the offset on the line. This vector points to the place where the placing is drawn. The Figure 6 shows such a vector.

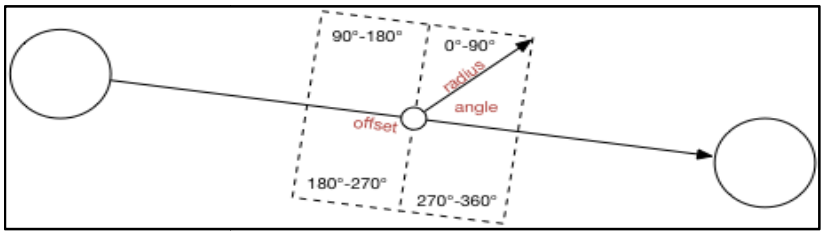

Figure6. Positioning of placings on Connections

The example in Figure 5 shows a*conditional sequence connection*which is part of the BPMN. There are two placings on the connection. The first is located at the end (offset=1.0) and is an arrowhead, drawn as a filled polygon. The other is a rhombus placed at the beginning (offset=0.0).

4.2.5. DEFINITION OF STYLES

Alayout definition with the style DSL allows to declare a number of attributes for shapes and makes such a style reusable via a unique identifier. Common attributes such as background-color

or line-style can be defined. It's possible to reference a defined stylefromany of the three DSLs or the style attributes can be set individual for shapes or single shape elements such as ellipse, rectangle or polygon. If no style definition is referenced or specified, the generator usesa default style declaration. For each level of nesting the style information is inherited from the next level above. It's possible to override the individual style attributes on each different nesting level. This offers the possibility of a very detailed definition on the look and feel of the shape. Furthermore, it makes it easy to change or extend the general appearance of the diagram very quickly.

```
style BlackAndWhiteStyle {
   description = "White background and Black foreground."
   transparency = 0.95
   background-color = white
   line-color = black
   line-style = solid
   line-width = 1
   font-color = black
   font-name = "Tahoma"
   font-size = 10
   font-italic = yes
   font-bold = no
}
```

Figure 7. Style File Example content

A style definition is described in a text file with the suffix.*style*. The description of a style begins with the keyword*style* and a unique style name see Figure 7 (in the example *BlackAndWhiteStyle*). This style is generated to a Java class with the same unique style name. This class implements the interface *ISprayStyle*. This specified interface forces the generated class to implement the function*getStyle()* which returns an instantiation of the Graphiti style class. This procedure maps all defined values of the style definition to the Graphiti Style class. A style definition can contain many different attributes e.g. line attributes like line-width or line-color. The line style attribute is responsible for the visual appearance of a line and the following options are predefined *solid*, *dot*, *dash*, *dash-dot* and*dash-dot-dot*. The font section of a style definitionspecifies attributes like font-size, font-color and font-name.There are a variety of other attributes of each section, this is just a selection.

```
style BlackAndYellowStyle extends BlackAndWhiteStyle {
  description = "Yellow background and Black foreground."
  background-color = yellow
}
```

Figure 8. Example description of style Inheritance

The underlining of fonts is currently not implementedin the Graphiti Framework. Therefore, this feature is at the moment not been reflected in the DSL as well, but this is an important enhancement which should be provided in the near future. Another problem of Graphiti is the definition of colors, because Graphiti does not allow to differentiate between a font- and a line-color. Graphiti contains only one color definition - the *foreground-color*. The *foreground-color*defines, according to Graphiti, everything which is drawn like the line border and the fonts. In MoDiGen we solved thisproblem with a workaround, by generating a different methods for every style called *getFontColor()*. This functionreturns a different *foreground-color* as the *line-color* for the font. The usage of the colors is quite simple, because there are just two different possible options to use them. The first option is to use the predefined color values like white,

black, gray and many more. The second option is to use the *RGB*-structure. This possibility provides the definition of the intensity of the red, green and the blue color individually with a value range between 0 and 255. Therefore, the designer is able to create nearly any color. In addition to the color attribute there is an attribute for transparency. A valid value range for the transparency is between1.00 and 0.00. 1.0 means that the object is fully visible and 0.00 is exactly the opposite. If no *background-color* is intended, the color can be set to *transparent*.

In this case the filled attribute value of the Graphiti Style class is declared to false. For the line-visibility is the same approach as for the *background-color* used. Styles offer the possibility to inherit from each other. The root element of the style inheritance hierarchy is the interface *ISprayStyle* followed by a default implementation called *DefaultSprayStyle*. This default implementation defines all default style attributes if no style is specified by the user. But this approach can be used in the DSL as well. It's also possible that a style inherits from another style and only just a few attributes are changed. Figure 8 describes exactly this case. The style BlackAndYellowStyle inherits from the *BlackAndWhiteStyle* and just the background-color is changed to yellow.

This inheritance mechanism gives the freedom that all attributes of the style DSL are optional. In addition, it's also possible to program a style definition manually in Java. It just needs to implement the interface the*ISprayStyle* and the style is available for referencing in MoDiGen or Shapes and for inheritance. Some nice enhancements as color gradients and element shadows will be added in the near future and will share the same properties.

4.2.6. Integration Of Styles Into Shapes/Connections

The style definitioncan be defined in aspecific external style description or can be part of a shape definition. For this purpose, the style keyword is used and the different style attributes are defined inside the round brackets. With this approach it's possible to override or define one or more style attributes. This opportunity has only an effect on the corresponding element and not to the nested elements or other forms on the same layer. This behavior is shown in Figure 9. There the *background-color* is set to the blue color for the rectangle.

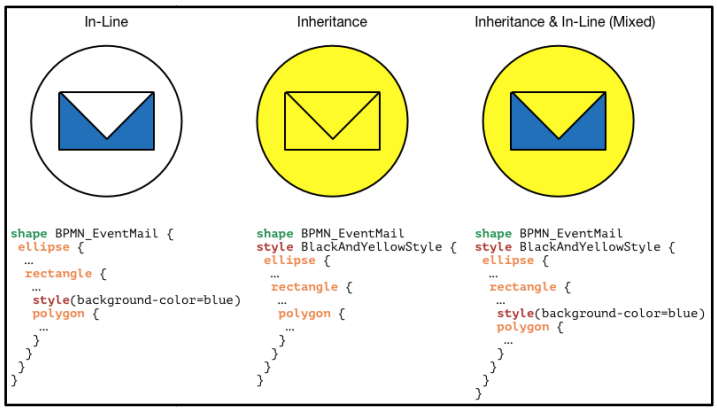

Figure 9. The different types of Style Integration and inheritance within shapes

The second optionoffers the possibility to inherit a style definition for a whole shape. In order to use this option, the style class must be referenced by its unique identifier after the keyword style. The middle part of the Figure 9 uses the style class BlackAndYellowStyle for the whole shape.

The complete shape gets with this definition a yellow background. If the same approach is used at the beginning of the rectangle, then the entire envelope would become yellow, but not the circle. As shown in the right part of the Figure 9 these two methods can be mixed. The inline defined attributes of a shape will always overwrite the attributes of the referenced style definition. In Figure 9 the whole shape is yellow apart from the rectangle, which overrides the color attribute with the value blue. In our experience, this mixing is very useful.

4.3. THE GENERATED GRAPHICAL EDITOR

The generated graphical editor is a set of fully functional Eclipse Plugin's. The editor consists of four divisions. The left area contains the package explorer which contains the different created diagrams with the option to create new diagram instances. The right area contains the elements described with the shape DSL in the previous step. In the lower area is the Property View, which displays properties of elements according to the defined properties in the MoDiGen core DSL. The area in the center is the main area to draw the new diagram according to the previous defined Metamodel. Figure 10 shows the editor with an example Diagram of the BPMN.

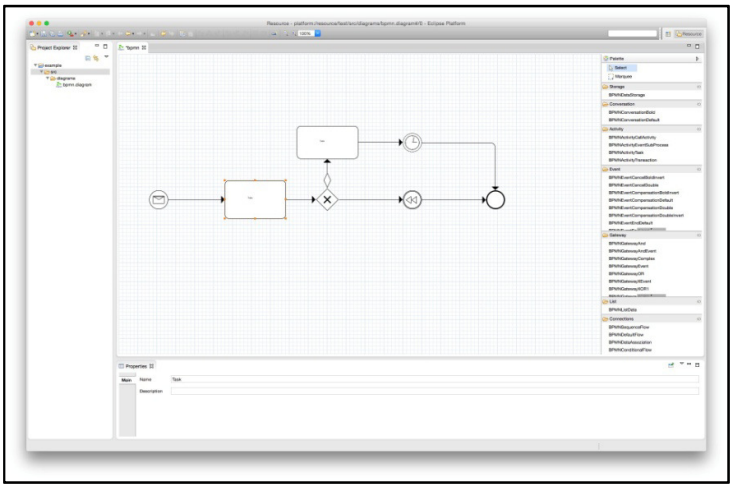

Figure10. The generated domain-specific graphical editor in Eclipse

5. EVALUATION

The presented generative approach reduces the needed effort to develop a domain specific graphical modeling tool for the Eclipse framework considerably compared to coding such a solution manually in Java against the Graphiti API. For every domain object requires the Graphiti framework a set of features. The generator creates this required Add-, Create-, Layout- and an UpdateFeature (a lot more planned), which altogether consist of at least 400 lines of code per domain object. These can be generated from MoDiGen from about 10 lines of code. The factor in terms of code between the MoDiGen DSLs (10 lines) and the generated Graphiti Code (350 lines) is approximately 35. For several implemented examples we consistently observed this factor, with exception of very small diagrams. Figure 11 shows the number of required lines of code in MoDiGen respectively Java (generated) based on the five diagram types "Petri-Net", "BPMN", "**E**vent-driven **P**rocess **C**hain" (EPC), "**P**iping and **i**nstrumentation **D**iagram" (P&ID) and "Basic Class Editor". The biggest difference of generated Java-Code and additional files (60154 lines of code) to MoDiGen (1784 lines of code) is evidenced in the business process modeling language "BPMN". This can be mainly attributed to the variety of different graphical elements. The

smallest factor of about 30 results for the very simple chart type “Basic Class Editor”. It consists of only two Graphical elements ‘(shape and connection) and does not have any complex logic.

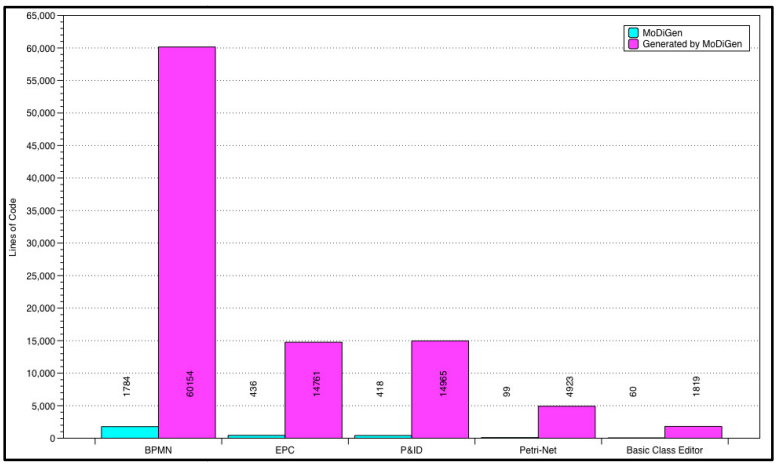

Figure11. Lines of Code: MoDiGen vs. Java (Generated)

The division of the lines of code on the three developed languages is shown in Figure 12. It can be observed that the main effort lies in the description of graphical elements, which is demonstrated in the use case of “BPMN” which consists of about 70 Shapes and connections. The number of lines of code for the styles cannot be directly measured, since this depends on the level of detail that the domain expert wishes to reach. For simple styles as few as 10 lines can be sufficient, more complex styles can take up several hundred lines of code. In these examples we merely created quite basic styles, which is usually inherited to all elements. The real benefit can be represented by the ratio of the lines of code necessary in MoDiGen and the generated Java code, which is shown in Figure 13.

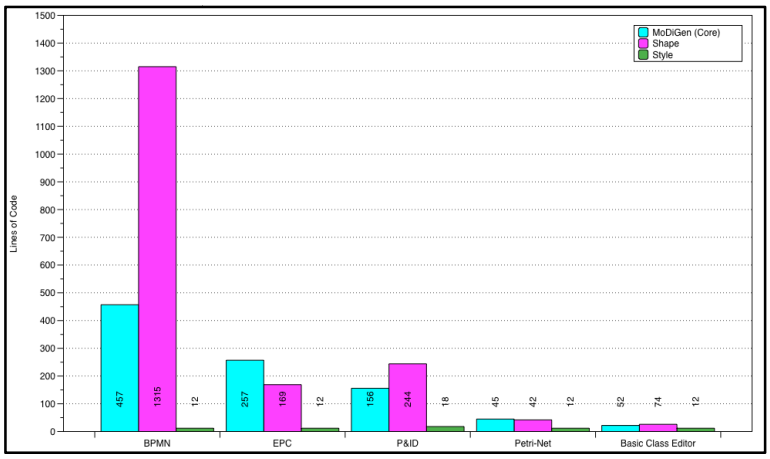

Figure 12. Lines of Code in different DSLs

The smallest factor is observed for the Basic Class Editor with a factor of 30. The highest factor within our set of examples is observed for the Petri-Net with a factor of 49. If all five examples are considered, this results in a mean factor of 36 (Blue line). The green and red lines show the mean factor without the highest and lowest value included, respectively. If the largest and

smallest values are not considered, there is a mean factor of above 34. This is indeed a huge benefit. We believe the generated code to be quite comparable to code that is hand written against the Graphiti API directly. In addition, the generated code solves a number of issues with the Graphiti code, that would need to be solved individually in each project otherwise, resulting in additional complexity.

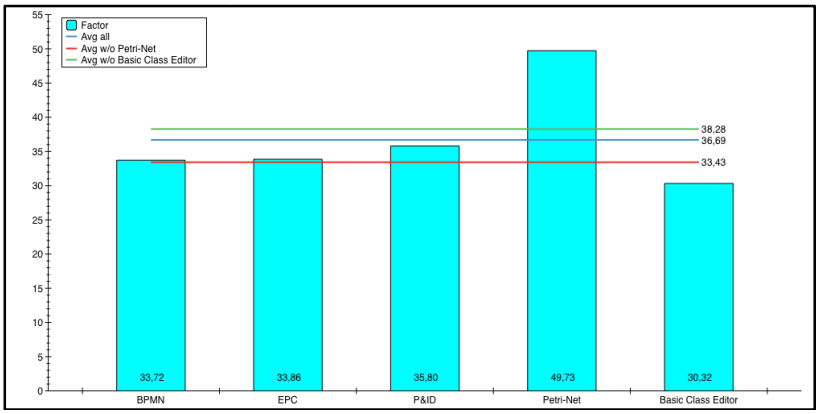

Figure13. Ratio Lines of Code MoDiGen vs. Java(Generated)

A quite representational DSL is the P&ID, describing pipes and other plumbing for the domain of water or gas pipes. It was developed for the Modelling Challenge of the Language Workbench Challenge in 2012. It consists of 20 domain objects and was described in MoDiGen in about 400 lines of code. This generates about 80 Java Classes with altogether about 15000 lines of code. Thus we argue that our approach reduces the development time for developing a graphical DSL in Eclipse by a significant factor. This should make the development of graphical modellingplugins for the Eclipse framework much more attractive and useful than in the past.

The applications BPMN Modeler 2.0 [15] [16] , EPC Tools [17], ePNK[18] and Basic Class Editor were used as comparable applications and analyzed with respect to lines of code. These applications were both generated and written by hand, slight differences in the functionality of the hand written and generated tools are possible. Table 1 displays the Lines of Code (LOC) per application. The number of lines of code was determined by the tool CLOC [19] [20]. The shown Lines of Code include on the one hand only the JAVA files (black) and, secondly, all the Lines of Code of the different Eclipse plugins (red). The difference is quite severe for the ECP Tools application. This is because many parts of the application are written in HTML, which is not considered in the calculation of the JAVA files. In the other cases, the difference is very small, because only configuration files are added.

Table 1. Lines of Code – CLOC

	MoDi Gen	Generated by MoDiGen	BPMN Modeler 2.0	EPCTools	ePNK	Basic Class Editor
BPMN	1.784	56.263 (60.154)	117.255 (128.170)	-	-	-
EPC	436	14.165 (14.761)	-	8.868 (85.151)	-	-
P&ID	418	13.659 (14.965)	-	-	-	-
Petri-Net	99	4.056 (4.923)	-	-	36.322 (37.406)	-
Basic Class Editor	60	1.680 (1.819)	-	-	-	1.732 (1.872)

The tools BPMN Modeler 2.0 and Basic Class Editor were developed with the help of the Graphiti Framework and are therefore a very good comparison of generated and manually written Java (Graphiti) code. The Basic Class Editor is of particular importance. This editor only includes a graphical element and a connection. From Table 1 it can be seen that the generated and the manually written code are very close together with respect to the number of lines of code. This shows that the generated and manually written code are comparable. The eclipse projects of the generated and handwritten code can be found at [21].

In addition, the developed languages were evaluated with 40 participants. The different participants were divided into four groups (Bachelor and Master Student, Developers and Others). Each participant of the different groups received a 4-hour introduction to meta-modeling and in MoDiGen. After the short workshop and a lunch break, each participant had to complete 4 different tasks sequentially with increasing degree of difficulty. The tasks were to use the diagram (Task 1), shape (Task 2) and style (Task 3) language separately and in total in the final task.

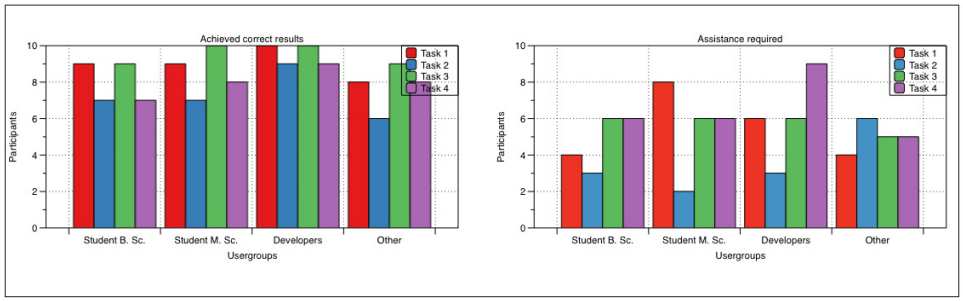

Figure14. Achieved correct results and Assistance required

Figure 14 (left side) shows that 90% of participants could solve Task 1, 72.5% Task 2, 95% Task 3 and 80% Task 4 correctly. The right diagram in Figure 14 shows that the tasks could be finished without assistance. 55% of the participants did not need help with the editing of Task 1, 35% Task 2, 57.5% Task 3 and 65% at Task 4. The second Task could only be processed correctly by approximately 75% and only 35% of the participants did not need help.This is probably because the definition of complex figures by a textual description requires more exercise and is not such

as easy with a graphical editor. This assumption is supported by the comment that a graphical live illustration of the defined element would have been very helpful.

Figure 15 shows the processing time for each user group and task with the fastest, slowest (perhaps cancelled) and average value. The values are distributed almost homogeneously. This allows the conclusion, that the languages have a certain complexity, especially the shape dsl, but almost everybody is able to learn the different languages. On average the participants needed 29.75 minutes for Task 1, 41,25 minutes for Task 2, 22,25 minutes for Task 3 and 59,5 minutes for Task 4.

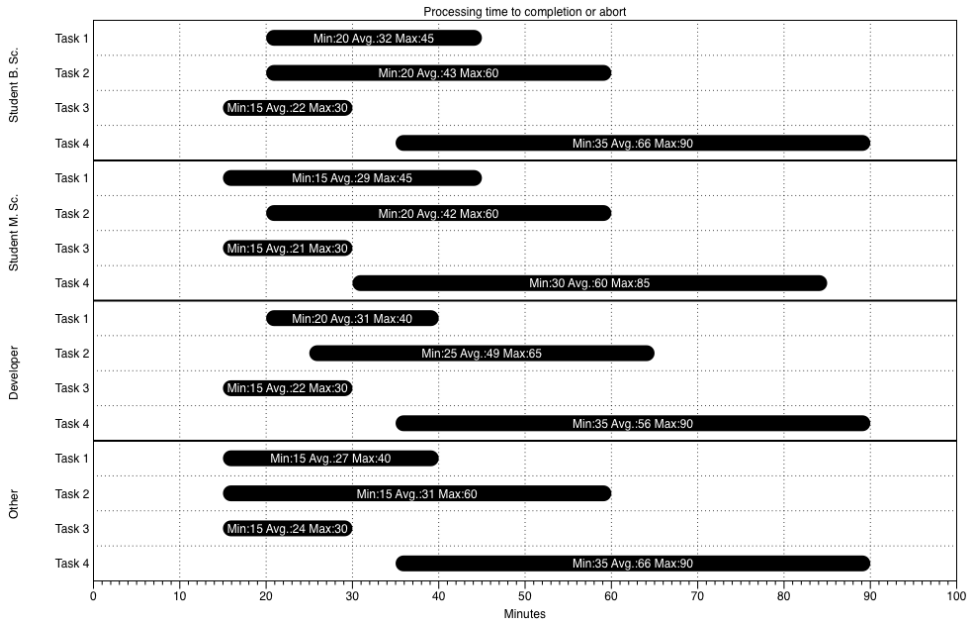

Figure15. Processing time to completion or abort

The evaluation of the dsl shows that the languages includes and combinesa lot ofuseful approaches and the users can use the languages very well separated and in combination. This is proved by the high rate of successfully completed tasks. With the integration of some approaches of CSS within the style language, the evaluation members find their way around very quickly. The creation of graphic elements with the shape dslwas initially very unfamiliar and the possibilities were not directly recognized. This problem could be solved with increasing processing of the task. After the training, the Diagram languages were considered to be very compact.

Nevertheless, the evaluation showed that some weaknesses are present within the presented approach. For example, a live preview of the defined graphical elements with the shape dsl would be very useful. However, this is not an issue of the languages but is an extension of the editor and tooling. The keyword "ask for" was not understood by a lot of participants. For this reason, it must be considered whether a renaming is meaningful. The diagram language part of the action groups was entirely ignored. Therefore, the question arises whether these are useful or the use must be described in more detail. The full evaluation plan and report is available at [22][23].
From this evaluation we conclude that the presented approach reduces the effort for the development of a graphical editor in Eclipse considerably compared to manually written code against Graphiti, under the constraint that the requirements are met by the generated editor. We

have indications that the use of Graphiti reduces the effort compared to the use of GEF and Draw2D, but have no formal evaluation for that.

6. CONCLUSIONS

In this paper we have presented that the development of a graphical DSL with the usage of the open source framework MoDiGen can be very efficient. We presented an approach for the model driven generation of graphical modelling tools as part of the Eclipse framework. The modellingenvironment can be adjusted to the needs of a specific domain. Thisresults in a set of Eclipse plugins,which supports the defined graphical domain-specific language. It's not necessary to be an expert of the field of metamodeling, to develop a domain specificgraphical modelling tool, because the presented DSLs are quite easy to learn, read and write. The meta model and the DSLs can be tailored to the specific needs of the domain, therefore the models can be very specific. In this paper we showed some example elements, but its although possible to use the same approach for various domain-specific modelling languages.

The described project is currently not finished and still lacks a number of important features before its use could be recommended in the context of productive development. But this will improve over time. Examples of such features are improved support for text, the inclusion of shadowsand the use of rapid buttons. More important is the question of the limitations of the approach. We see no constrains regarding the graphical editor itself and the provided API by Graphitiwith respect to the model-driven approach. Unfortunately, the Graphiti Framework did not provide a lot of features we had expected. For example, Graphiti currently does not support the design of text like underling. This feature is essential for a class diagram which is part of the UML. Another limitation is the standard zooming function within Graphiti diagrams and the hole eclipse environment. The eclipse ecosystemespecially Graphiti zooms by resizing all sizes of the elements, including the border-size of any element. This leads to a confusing functionality of the editor.

The generative approach is currently reduced to the cases which are often needed by domain developers (node and edge diagrams).In UML most diagram types fit into this category. But the other diagram types such as sequence and the timing diagram could not be implemented currently. They cannot be described with the presented DSLs without considerable extensions.Therefore, the generator and the metamodel fit to this cases. Corner cases could be covered by the manual extension of the generated code. The generated code is well prepared for this case.

The real limitation is more the development environment in the form of the Eclipse framework. The aim of Eclipse is to develop a tool for the textual software development.An example of this is that the storage of a change in a file must be triggered explicitly.This approach is not possible for the real-time evaluation of model data.This requires a continuous storage for each change.However, this is hard to achieve in Eclipse.

Other topics, like multi-user collaborative modelling environments, evolution of Metamodels, or diffing and merging changes in versions of graphical models remain topics of research and are independent of Eclipse or a model driven approach.

REFERENCES

[1] MoDiGen - A Quick Way of Creating Graphiti (2016). https://www.modigen.de
[2] The Eclipse Foundation: Eclipse Modeling Framework Project (EMF) (2016). http://www.eclipse.org/modeling/emf/

[3] Itemis AG (2016). http://www.eclipse.org/Xtext
[4] The Eclipse Foundation: Draw2d (2016). http://www.eclipse.org/gef/draw2d/
[5] The Eclipse Foundation: Graphical Editing Framework (GEF) (2016). http://www.eclipse.org/gef/
[6] The Eclipse Foundation: Graphical Modeling Project (GMP). (2016)http://www.eclipse.org/modeling/gmp/
[7] The Eclipse Foundation: Graphiti: A Graphical Tooling Infrastructure. (2016)http://www.eclipse.org/graphiti/
[8] Poseidon for DSLs. (2016)http://www.gentleware.com/poseidon-for-dsls.html
[9] Sparx Systems: Enterprise Architect. (2016)http://www.sparxsystems.com/
[10] A DSL for Graphiti Editors (IMES), Research Project. (2016)http://5ise.quanxinquanyi.de/2011/08/18/a-dsl-for-graphiti-editors/
[11] No Magic Inc.: MagicDraw (2016). https://www.magicdraw.com/
[12] MPS Meta Programming System (2016). http://www.jetbrains.com/mps/
[13] Software Developer SAP (2016). http://www.sap.de
[14] Object Management Group (2016). http://www.omg.org/
[15] Eclipse - BPMN2 Modeler (2016). https://www.eclipse.org/bpmn2-modeler/
[16] GIT Repository - BPMN2 Modeler (2016). https://github.com/eclipse/bpmn2-modeler
[17] Uni Paderborn –EPCTools (2016). http://www2.cs.uni-paderborn.de/cs/kindler/research/EPCTools/
[18] Technical University of Denmark – EPNK (2016). http://www.imm.dtu.dk/ekki/projects/ePNK/
[19] Count Lines of Code – CLOC (2016). http://cloc.sourceforge.net/
[20] Count Lines of Code - CLOC Git (2016). https://github.com/AlDanial/cloc
[21] Implementation Basic Class Editor MoDiGen and JAVA (2016). http://www.modigen.de/BasicClassEditor/
[22] Evaluation Plan (2016). http://www.modigen.de/evaluation/Evaluierungsplan-MoDiGen-DSLs.htm
[23] Evaluation Report (2016). http://www.modigen.de/evaluation/Evaluationsbericht-MoDiGen-DSLs.htm
[24] Hutchinson, J., Whittle, J., Rouncefield, M., &Kristoffersen, S. (2011, May). Empiricalassessmentof MDE in industry. In Proceedingsofthe 33rd International Conference on Software Engineering (pp. 471-480). ACM.
[25] Johannes, J., & Aßmann, U. (2010, October). Concern-based (de) compositionof model-drivensoftwaredevelopmentprocesses. In International Conference on Model Driven Engineering Languagesand Systems (pp. 47-62). Springer Berlin Heidelberg.
[26] Kalleberg, K. T., & Visser, E. (2007, March). Spoofax: An interactivedevelopmentenvironmentforprogramtransformationwithStratego/XT. In A. Johnstone, & T. Sloane (Eds.), Workshop on Language Descriptions, Tools, andApplications (LDTA 2007) (pp. 47-50).
[27] Kolovos, D. S., Rose, L. M., Paige, R. F., & Polack, F. A. (2009, May). Raisingthelevelofabstraction in thedevelopmentof GMF-basedgraphicalmodeleditors. In Proceedingsofthe 2009 ICSE Workshop on Modeling in Software Engineering (pp. 13-19). IEEE Computer Society.
[28] Kolovos, D. S., Rose, L. M., Abid, S. B., Paige, R. F., Polack, F. A., &Botterweck, G. (2010, October). Taming EMF and GMF usingmodeltransformation. In International Conference on Model Driven Engineering Languagesand Systems (pp. 211-225). Springer Berlin Heidelberg.
[29] Object Management Group: Business Process Model and Notation, Speci_cation V2.0. http://www.omg.org/spec/BPMN/2.0/PDF, Needham, Massachusetts, VereinigteStaaten (2011). Accessed 2015-01-22
[30] Rubel, D., Wren, J., &Clayberg, E. (2011). The EclipseGraphicalEditing Framework (GEF). Addison-Wesley Professional.
[31] Smolander, K., Lyytinen, K., Tahvanainen, V. P., &Marttiin, P. (1991, May). MetaEdit—a flexible graphicalenvironmentformethodologymodelling. InInternational Conference on Advanced Information Systems Engineering (pp. 168-193). Springer Berlin Heidelberg.
[32] Pietrek, G., & Trompeter, J. (2007). Modellgetriebene Softwareentwicklung-MDA und MDSD in der Praxis; entwickler. press. Frankfurt am Main.
[33] Vara, J. M., & Marcos, E. (2012). A frameworkfor model-drivendevelopmentofinformationsystems: Technical decisionsandlessonslearned. Journal of Systems and Software, 85(10), 2368-2384.
[34] Mernik, M., Heering, J., &Sloane, A. M. (2005). Whenandhowtodevelop domain-specificlanguages. ACM computingsurveys (CSUR), 37(4), 316-344.

[35] Steinberg, D., Budinsky, F., Merks, E., &Paternostro, M. (2008). EMF: eclipsemodelingframework. Pearson Education.
[36] Rubel, D., Wren, J., &Clayberg, E. (2011). The EclipseGraphicalEditing Framework (GEF). Addison-Wesley Professional.
[37] Draw2D, E. Project Homepage.
[38] Refsdal, I. (2011). Comparisonof GMF and Graphiti based on experiencesfromthedevelopmentofthe PREDIQT tool. University of Oslo.
[39] Gronback, R. C. (2009). Eclipsemodelingproject: a domain-specificlanguage (DSL) toolkit. Pearson Education.
[40] Moritz Eysholdt and Heiko Behrens. (2010). Xtext: implement your language faster than the quick and dirty way. In Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion (OOPSLA '10). ACM, New York, NY, USA, 307-309.

AUTHORS

Markus Gerhart studied business computer science with a focus on Software Engineering at the University of applied science Konstanz. He then completed a master's degree in the study program with a focus on business processes modelling and model driven software development and is currently a PhD student of Prof. Dr. Marko Boger in the area of the model-driven software development.

Prof. Dr. Marko Boger studied in Karlsruhe (Germany) and Toulouse (France) and received his doctor degree in Aachen and Hamburg (both Germany). Then he founded the company Gentleware, which was in the area of graphical modeling one of the leading tools in 2002. He was actively involved in the standardization of UML 2 as a head of the working group. Since 2009 he is professor of software engineering and software architecture at the University of applied science Konstanz.