RESEARCH ARTICLE OPEN ACCESS

Zeta - A Set of Textual DSLs to Define Graphical DSLs

Markus Gerhart*, Marko Boger**
*(Applied computer science, University of Applied Sciences, Konstanz)
**(Applied computer science, University of Applied Sciences, Konstanz)

----------**********----------

Abstract:

Domain-specific modeling is increasingly adopted in the software development industry. While textual domain-specific languages (DSLs) already have a wide impact, graphical DSLs still need to live up to their full potential. In this paper, we describe an approach to automatically generate a graphical DSL from a set of textual languages. With our approach, node and edge type graphical DSLs can be described using textual models. A set of carefully designed textual DSLs is the input for our generators. The result of the generation is a graphical editor for the intended domain. The development time for a graphical editor is reduced significantly. The whole project is available as open source under the name "Zeta". This publication focuses on the explanation of the textual DSLs for defining a graphical node and edge editor.

Keywords — **Model-Driven Software Development (MDSD), Domain-Specific Language (DSL), Metamodel, Model-Driven Architecture (MDA), Graphical Online Editor, Language design**

----------**********----------

I. INTRODUCTION

The creation of domain-specific graphical languages and editors will be very important in the near future. This arises from the common requirements regarding efficiency, speed of development and the need for domain-specific solutions. To meet these requirements, the existing approach for the development of graphical modelling languages and editors should be reconsidered.

Editors for graphical modelling languages currently come in three flavors: stencil-based drawing tools, domain-specific modelling tools, and general purpose modelling tools. Stencil-based drawing tools can be tailored to depict a certain domain-specific modelling language, but it lacks the semantic understanding of the model and can in general not be used for model-driven approaches. While they have their use cases, we will not consider them further in this paper. General purpose editors or cross-domain editors exist as tools on the market and have been very successful. Examples for languages in this category are UML or BPMN. While they have been very successful in their use cases to describe a cross-domain concern, like describing a software architecture or a business process, it has been very difficult to adapt them to domain-specific use cases, especially in a model-driven approach. There are ways to extend such general purpose languages to domain-specific concepts, in the case of UML this would be a profile. But such approaches tend to make the development of a generator for a model transformation very complicated. We will also not focus on these in this paper. The focus of the paper will be on languages and tools developed specifically for one domain with the goal of using the models as input to a model transformation. This can lead to very elegant solutions, both in terms of the modelling tool as well as the model transformation. But the development of such graphical editors is currently a very costly undertaking.

In the research project "**Pro**gress in **Gra**phical **Mo**deling **F**rameworks" (ProGraMof) we have developed a set of **d**omain-**s**pecific **l**anguages (DSL's) for the description of node and edge (or box and arrow) type graphical editors, as well as the right toolset for the use of these languages. The developed DSLs are exclusively used to describe the content (element's, behavior and rules) of the

editor and not the general infrastructure. The tooling infrastructure is provided by our framework.

Our goal is to enable domain experts to create their own graphical modelling language in a simple and cost effective way in the context of a model-driven approach. They should gain the freedom to design their own notation elements and to continually adapted them to the changing needs.

The result presented in this paper is a collection of three textual languages, each describing a separate concern of a graphical modelling language. They can be defined in any appropriate grammar definition language such as EBNF, Xtext or Ace-Grammar. In this paper, we use Ace-Grammar which is based on the **J**ava**S**cript **O**bject **N**otation (JSON) Metamodel approach.

All three languages are based on their own metamodel, which can easily be accessed from a generator language such as Xtend or Scala. We use it for the creation of a graphical editor for the defined modelling language. We have generators for several different platforms under development. The most prominent target platforms are Eclipse and a web-based editor.

The paper first gives a brief overview of the technologies used in the Section **II**. Subsequently the paper reviews related work in the field in Section **III** which are mostly other tools or languages and techniques for the generation of modelling tools. Our general approach for the model driven creation of the modelling editors and the general architecture of our framework with the detailed description the developed languages is described in Section **IV**. The core contribution of this publication are the developed DSLs to define graphical elements, styles for the graphical elements and the graphical editor itself, which are described in the subchapters of the general approach. Section **V** shows a small implementation Guide for the presented DSLs. Section **VI** illustrates the results of our approach from different angles. Finally, we summarize the limitations of our research and draw conclusions in Section **VII**.

II. BACKGROUND

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

The **E**xtended **B**ackus-**N**aur **F**orm (EBNF) is a set of metasyntax notations, which can be used to express a context-free grammar. The core is to create a formal description of a formal language. The EBNF extends the Backus-Naur Form (BNF).

A **D**omain **S**pecific **L**anguage (DSL) is a formal language which is exactly tailored to a specific domain, a specific task or problem area.

Java**S**cript **O**bject **N**otation (JSON) is an open-standard and language-independent format that uses human-readable text to transmit data objects consisting of attribute–value pairs.

Scalable **V**ector **G**raphics (SVG) is an XML-based vector image format for two-dimensional graphics with support for interactivity and animation. The SVG specification is an open standard developed by the World Wide Web Consortium (W3C).

A **c**ontext-**f**ree **g**rammar (CFG) is a set of recursive rewriting rules (or productions) used to generate patterns of strings.

The **H**yper**T**ext **M**arkup **L**anguage (HTML) is the standard markup language used to create websites or web.

Cascading **S**tyle **S**heets (CSS) is a style sheet language used to describe the presentation of a document written in a markup language like HTML.

Extensible **M**arkup **L**anguage (XML) is a markup language that defines a set of rules for encoding documents in a format which is human and machine readable.

III. RELATED WORK

Projects which have need for a domain-specific graphical editor currently only have two possibilities. On the one hand, they can use complex tools for the Model-driven generation of a domain-specific graphical editor or on the other hand they can do the implement manually. In the first case, an expert for model-driven software development is required who implements the

metamodel and the generators. In the second case a developer is required to implement the project. However, in both cases is always an expert required. The generative approach is probably the pioneering approach and the bases for the presented approach.

There are many representatives of development environments and approaches of model driven development and model driven architecture in the market. Almost all of them allow the creation of DSL's which are based on a metamodel and the implementation of a generator that generates the required artefacts. This solution works perfect for the development of a DSL for the creation of a approach which is presented within this paper but is not suitable for the actual users how wants in a simple way created his own graphical editor. This is due to the fact that the existing tools are made for software developers and not for the different users of the domains.

There are currently no comparable solutions to create a domain-specific graphical node and edge editor with reference to a metamodel. Therefore, no direct comparison to existing solutions can be provided. Only the existing solutions for model driven software developers could be used as a comparative criterion, but these are not suitable for the above mentioned reasons.

The design of the developed languages is influenced by many different languages and research papers. The following list of references had the main design influences. Zeta adopts some ideas of the concepts and syntactic conventions of Java [5]. From Beta [7] and common diagram types like UML or the **B**usiness **P**rocess **M**odel and **N**otation (BPMN) comes the idea that closed shapes should be nestable. Zeta's design of key value pairs to describe properties of the predefined objects is adopted from the **J**ava**S**cript **O**bject **N**otation (JSON) [1] and **C**ascading **S**tyle **S**heets (CSS) [6]. Maybe there are other allusions to other languages but which are not known to us.

Zeta provides a powerful set of constructions for creating specific elements which can be linked to an existing metamodel described in Zeta [2]. The aim is that with this set of languages the user has the option to develop his own domain specific diagram editor in an easy and fast way.

IV. APPROACH

In the research project "ProGraMof" we have developed a complete tool set for the fast and efficient creation of graphical modelling languages as well as their efficient use in graphical editors. The tool set as a whole is called Zeta. The name is inspired by the internet culture to use a z to express a "twist" of some sort, like in "gamez", "warez" etc. In our case the fundamental twist is on the concept of "meta", which regularly leads to headaches among apprentices of model-driven technologies. We needed to twist our meta-metamodeling language - originally Ecore - to achieve better results and came up with the name Zeta-Core. This then inspired the name of the entire project. We use the Greek letter ζ (zeta) in such combinations, i.e. ζ-Core, to express the relation between the family of languages in the Zeta project.

We have developed frameworks for graphical editors on several platforms. These frameworks contain the domain independent concerns of graphical editors, such as manipulating graphs and user interaction. The domain-specific parts are expressed in three domain-specific textual languages. Our generators, framework code and runtime are based on the Scala platform by Lightbend. For the web-based platform we use the Ace Editor with the plugin Ace-Grammar. The core contribution of this publication are the textual DSLs.

Figure 1 shows the three developed languages " ζ-Elements", " ζ-Style" and " ζ-Diagram". The aim of this strict division of the languages is the separation of concerns for presentation (style), structure (element) and the definition of the actual Diagrams which reference elements and style.

The ζ-Style language offers the functionality to express design for elements. This is similar to the "Cascading Style Sheets" short CSS which is, for example, used to design websites. The ζ-Style-DSL does not offer the full potential of the CSS but provides a subset of the CSS options. Through the use of the CSS approach, it is possible to expand the style DSL at any time.

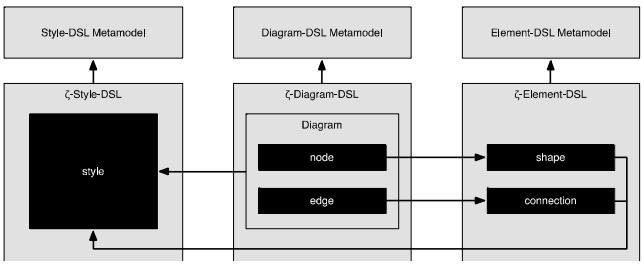

Fig. 1 Dependencies of the developed languages

The ζ-Elements language is used to define any type of shapes or connections. This may be on the one hand common shapes such as ellipses, rectangles or polylines or on the other hand connections which consist of a line on which different forms can be placed. A shape is not equivalent to a form. A shape consists of at least one form, but forms can be nested as many times as needed, so it's possible to produce very complex shapes. The design of the shapes is made possible through the association of a style or of defining the style within the shape.

The link which connects the style and element language is the diagram language. This defines the real content of the new diagram type to be defined. A diagram exists of nodes (elements) and edges (connections) which is a known concept of the graph Theory [9]. It is assigned to every node an element type and every edge a connecting type. In addition, the diagram language can contain a reference to a predefined style what opens the possibility to define a "Corporate Identity" style which is applied to all elements of the diagram.

The clarity is guaranteed by the clear and clean separation of presentation (the definition of the appearance of elements) and contents (the definition of forms). A mixture of these both different areas of responsibility leads to confusing textual descriptions like it is to be found for example with **S**calable **V**ector **G**raphics (SVG). Another advantage of this approach is that with style adaptations merely the style description must be changed and the other languages remain unchanged.

Each of these three languages owns her own corresponding metamodel in the form of a Zeta-core file. These are essential for the code generation. By the use of a metamodel, it is possible to use different generating languages for the code production.

The developed languages follow a compact construction, which orientates itself by current object-oriented programming languages. For the clarity of sub-elements and properties of elements, they are enclosed in curly brackets "{" (at the beginning) and "}" (at the end). This approach is comparable to classes and methods of object-oriented programming languages like java, Scala etc.. Properties of an element can be passed with round brackets "(" (at the beginning) and ")" (at the end). These parameters are directly comparable to an input parameter of methods. A clear and easy structure of the developed languages originates from these both easy rules. In the following, the metasyntax notation of the developed languages is explained in detail and displayed on the basis of examples.

The **E**xtended **B**ackus-**N**aur **F**orm (EBNF) [10] is nowadays the most rigorous, comprehensible and clearest way to define the metasyntax of programming languages. For this reason, the representation of the languages is realized by the EBNF. Listing 1 describes the EBNF notation of the data types which can be used within the languages.

```
identifier  ::= (digit | letter) {digit | letter | "-" | "_"}.
integer     ::= digit {digit}.
double      ::= digit {digit} ["." {digit}].
digit       ::= "0" | . . . | "9".
letter      ::= "a" | . . . | "z" | "A" | . . . | "Z".
boolean     ::= "true" | "false".
```

Lis. 1 EBNF representation of the Data Types

Only six data types are currently defined and can be used. This is mainly due to the fact that in JSON only these data types are intended. Currently, no further data types are needed. Should this be the case, they can be created from these basic types. Because the complete data exchange shall be done on the basis of JSON, it makes sense to define the same data types. Table I shows examples for each of the data types.

TABLE I
DATATYPES EXAMPLES

Keyword	Example
identifier	A1234-B12234

Integer	123123
Double	1.21 or 143.121
Digit	6
Letter	A
boolean	True or false

A. ζ-Element DSL

The ζ-Elements language has, to a certain degree, a similarity to SVG, but the design goals are different. The commonality to SVG is that the ζ-Elements DSL uses primitive forms like lines, curves, rectangles, polygons and ellipses to define more complex forms. Nevertheless, a decisive difference between these both languages is that SVG is based on XML. The ζ-Element DSL bases on a context-free grammar, what permits more freedom in the design. The principal purpose of the language is a light legibility and a quick understanding for human users. The machine readability or fast machine processing plays only a subordinate role. Furthermore, there are requirements which SVG did not meet, for example, the definition of scaling rules and the handling of in-/output parameters for in-/output fields for the programmatic validating or processing.

The nesting of forms is essential in order to create complex shapes. To initiate a nesting, the curly brackets "{" (start) and "}" (end) are used. This rule applies to all developed languages. All forms except lines, curves and texts can be nested in any depth. The element DSL is used exclusively for the definition of forms and connections. To improve the legibility of the language for the user, a syntax highlighting is designed. We defined that the keyword "shape" is used to define a form and "connection" to define a connection between two forms. Both are always displayed in green. The basic forms are always highlighted in orange and the properties of a basic form always in red. By this go forward also, nested elements are easy to recognize. Furthermore, the definition or the reworking of already existing forms is made easier. In the following, the general construction of the different elements is explained.

In every of the following described figures are optional information displayed italic. If a property is optional, this is also valid for the parameters. Besides all values to be defined by the user, optionally with a data type, are marked in boldface.

SHAPES

Listing 2 shows the basic structure of a form. The definition of a form always starts with the keyword *shape* followed by a unique name. Optionally an already defined style can be referenced within round brackets directly behind the unique name. This style is inherited to every sub-element and can be overwritten by every separate sub-element. The actual definition of a form begins always with curly bracket. The properties sizeMin and sizeMax, in each case with the parameters width and height, restrict the expansion of a form. This is important for nested forms to prevent a too small or too big representation of a sub-element. The form can be limited concerning the dimension's adaptation by defining the attribute "resizing". The restriction can be defined horizontally and vertically as well as proportionally. The attribute "resizing" must be checked at runtime using an algorithm. Should a size update be permitted and the properties minimum and maximum size are defined, a size update is possible only in this range.

The visibility or the Z-Index of elements is fixed by their arrangement. This means that the lowest element in the textual description is the element with the highest Z-Index respectively the element covers all other elements. Furthermore, implies that all nested elements overlay their respective parent elements.

To attach connections to different points of a shape, the definition of anchor points has been integrated. An anchor point is created with the keyword anchor. The position of an anchor point can be defined by assigning one of three predefined constants (corner, center or edges) to the property position. The "corner" constant defines that all corners of a shape are anchor points. "Center" defines that an anchor point at the center of the shape is present. "Edges" defines that on the four edges an anchor is defined. Apart from the pre-defined anchor points they can be freely positioned relative or absolute. Absolute anchors are realized by handing over the parameters "x" and "y" to the property "position". The value range is between the

maximum width respectively height of the shape. If the parameters "xoffset" and "yoffset" are handed over, relative anchors can be created. Relative anchor points automatically scale by resizing a shape at runtime, which is not intended for absolutely defined anchor points. The relative position parameters of an anchor point can range between 0 (upper left corner) and 1 (completely right respectively completely below).

```
shape       ::= "shape" identifier [ "(" style ")" ] "{" shapeBlock "}".
shapeBlock  ::= [sizeMin] [sizeMax] [resizing] [geoMetricFigure] [anchors].
sizeMin     ::= "sizeMin" "(" "width" ":" integer "," "height" ":" integer ")".
sizeMax     ::= "sizeMin" "(" "width" ":" integer "," "height" ":" integer ")".
resizing    ::= "resizing" "(" "horizontal" ":" boolean ","
                "vertical" ":" boolean ","
                "proportional" ":" boolean ")".
anchors     ::= "anchor" "{" [anchor] "}".
anchor      ::= "position" "(" (relative | absolute | predefined) ")".
relative    ::= "xoffset" ":" double "," "yoffset" ":" double.
absolute    ::= "x" ":" integer "," "y" ":" integer.
predefined  ::= "predefined" ":" ("corner" | "center" | "edges").
```

Lis. 2 EBNF representation of a Shape element

All following described forms/output fields can be extended with the optional parameter style. By referencing an existing style name in rounded brackets after the form name, the style is applied to the whole form. Besides even single attributes of a style can be overridden or be created the first time. This regulation is valid for all forms and is not explained again for the single forms.

The definition of a line starts with the keyword *line* and the entry of two points within curly brackets (see Listing 3). The points are properties of the line. A point exists of an obliging coordinate (x and y) and the optional entry of a curve before and after the point. The coordinate of a point is looked relatively to the parent element, based on the upper left corner. This means, with increasing X value the point walks to the right or down with increasing Y value.

```
line       ::= "line" [ "(" "style" ":" identifier ")" ] "{" lineBlock "}".
lineBlock  ::= point point [style].
point      ::= "point" "(" "x" ":" integer "," "y" ":" integer ")".
style      ::= "style" "{" styleBlock "}".
```

Lis. 3 EBNF representation of a line

The definition of curves or lines which exist on several points is important and often used aspect to be able to display complicated forms. The general construction of a *polyline* is comparable with the *line*. The difference consists in the fact that a polyline can exist of arbitrarily many points and a curve can be defined for each point which has an inbound and outbound edge (see Listing 4). The specification of curves is realized with the optional parameters curveBefore and curveAfter. The curve of the edge leading to the point can be influenced with the parameter curveBefore. The value of the curvature specifies by which length before the point (a maximum of half the distance) the curvature starts. The same approach is also valid for the parameter curveAfter, but for the outgoing edge. Nevertheless, the definition of curves causes that the defined points do not lie on a straight line.

```
polyline       ::= "polyline"
                   [ "(" "style" ":" identifier ")" ] "{" polylineBlock "}".
polylineBlock  ::= point {point} [style].
point          ::= "point" "(" "x" ":" integer "," "y" ":" integer ","
                   "curveBefore" ":" integer "," "curveAfter" ":" integer ")".
style          ::= "style" "{" styleBlock "}".
```

Lis. 4 EBNF representation of a polyline

The ability to create closed forms, is a basic requirement for the element DSL. A simple representative is a rectangle. The general definition of a rectangle is described in Listing 5. A rectangle needs the specification of width and height, which is described by the property size. To create a valid representation of a rectangle, another mandatory information is not required. To allow more freedom in design and creation, there are other properties and attributes. To shift the reference point (the standard is defined in the upper left corner (0,0)) the property position must be defined with the parameters x and y. This helps particularly with nested elements so that these can be freely positioned. It is important, that this specification is always defined absolutely. The specification of the property curve allows the change of the corner points to a curve. The parameter width influences the width of the area to be rounded of the corner points on the X axis. This means that the maximum value of the width is half of the rectangle width. The parameter height immediately behaves like the width on the Y axis. Within a rectangle, any other elements can be nested or a compartment container (see Listing 9) can be integrated.

```
rectangle       ::= "rectangle"
                    [ "(" "style" ":" identifier ")" ] "{" rectangleBlock "}".
rectangleBlock  ::= [position] size [curve] [style]
                    [compartment] [geoMetricFigure].
position        ::= "position" "(" "x" ":" integer "," "y" ":" integer ")".
size            ::= "size" "(" "width" ":" integer "," "height" ":" integer ")".
curve           ::= "curve" "(" "width" ":" integer "," "height" ":" integer ")".
style           ::= "style" "{" styleBlock "}".
geoMetricFigure ::= {line | polyline | rectangle |
                    polygon | ellipse | textfield}.
```

Lis. 5 EBNF representation of a rectangle

Another simple representative of closed forms is the ellipse. The definition of an ellipse is introduced with the keyword ellipse. An ellipse requires the specification of width and height, which are described on the size property. The width or height defines the diameter in the horizontal (width) and vertical (height) extent relative to the center. Thus, it is possible to create a circle by specifying the same value for width and height. The position defines the reference point of the element which lies with an ellipse beyond the real element. This is in the left upper corner of an invisible rectangle with the width and height of the size property defined value. Within an ellipse, as with the rectangle, any other elements can be nested or a compartment container (see Listing 9) can be integrated.

```
ellipse         ::= "ellipse"
                    [ "(" "style" ":" identifier ")" ] "{" ellipseBlock "}".
ellipseBlock    ::= [position] size [style]
                    [compartment] [geoMetricFigure].
position        ::= "position" "(" "x" ":" integer "," "y" ":" integer ")".
size            ::= "size" "(" "width" ":" integer "," "height" ":" integer ")".
style           ::= "style" "{" styleBlock "}".
geoMetricFigure ::= {line | polyline | rectangle |
                    polygon | ellipse | textfield}.
```

Lis. 6 EBNF representation of a ellipse

With a polygon, it is possible to create complicated forms in one step without nesting other forms. A polygon exists of a number of points and is always a closed area. The keyword polygon initiates the definition of a polygon and contains at least 3 up to n points. The definition of the points is identical with polyline and is explained in Listing 4. It is important that the points are connected in the given order (top to bottom). The determining difference between a polyline and a polygon is, that a polygon is always a closed area. Furthermore, a polygon can contain other elements. There is merely the restriction that a polygon cannot include a compartment. This is due to the fact that in a polygon can be no assurance that it is always sufficient free space available within the polygon. This has the background that with complicated forms a correct arrangement within the polygon is hard to be realized up to impossibly. Nevertheless, this premise can be lifted by the nesting of a rectangle or an ellipse. This is clearly made easier by the use of a rectangle or ellipse and is easier with the use of the programmatic arrangement.

```
polygon         ::= "polygon"
                    [ "(" "style" ":" identifier ")" ] "{" polygonBlock "}".
polygonBlock    ::= point point {point} [style].
point           ::= "point" "(" "x" ":" integer "," "y" ":" integer ")"
                    "curveBefore" ":" integer "," "curveAfter" ":" integer ")".
style           ::= "style" "{" styleBlock "}".
```

Lis. 7 EBNF representation of a polygon

The keyword outputField generates an output field for a string representation of a form. The number of output fields is not limited, merely the id must be unique. The positioning and the size of the field are steered about the keyword position or size and behaves the same as in the forms. The alignment of the text within the output field can be adjusted with the respective keyword align. The alignment can be done horizontally and vertically. For this reason, the constants (left, center and right) were predefined for the horizontal and (top, middle and bottom) for the vertical adjustment. This is the only spot which breaks the rule of the strict separation of layout and presentation. However, it does not make sense to define an own style because of alignment indication.

```
textfield       ::= "textfield"
                    [ "(" "style" ":" identifier ")" ] "{" textfieldBlock "}".
textfieldBlock  ::= identifier [position] size [align] [style].
position        ::= "position" "(" "x" ":" integer "," "y" ":" integer ")".
size            ::= "size" "(" "width" ":" integer "," "height" ":" integer ")".
align           ::= "align" "(" "horizontal" ":" ("left" | "center" | "right")
                    "," "vertical" ":" ("top" | "middle" | "bottom").
style           ::= "style" "{" styleBlock "}".
```

Lis. 8 EBNF representation of a textfield

A crucial quality of graphic editors is to integrate elements into other elements at runtime. This important feature is reached by the specification of the property compartment see Listing 9. A "Compartment" is a container which allows the storage of the defined shapes at runtime. This

allows an interaction between shapes at runtime. If the property compartment is defined, the entire element serves as a storage container. A compartment can be integrated into all closed forms apart from polygons. To influence how the objects are stored graphically, the compartment has the property layout. It defines how elements are arranged in the compartment. This property is used only for more than one stored element and offers four arrangement possibilities. By the predefined parameter fixed, the elements are stored at the place where the user has dropped the element. This makes possible that elements overlap and are not directly visible. The defined values vertically and horizontally arrange the elements among each other respectively side by side. Through the constant fit, a layout algorithm is triggered, which calculates the optimum position for each element of the compartment area. But it is conceivable that elements are exchanged in their order by the Algorithms and cannot be assigned directly any more. The property stretching depends directly on the constants vertical and horizontal of the layout property. If stretching is prevented horizontally or vertically, the stored elements are arranged up to the defined width of the element. An automatic enlargement of the element does not occur. Otherwise, the element is increased according to the stored elements. Particularly in the case of a horizontal and vertical arrangement, the distance between the elements plays a certain role. This can be controlled by the property spacing which defines the spacing of pixel. Nevertheless, spacing refers only to the distance between the stored elements. To adjust the space between the outermost elements and the compartment border the property margin is used. The distance is also given here in pixel. Finally, a compartment must always contain a unique id with which the compartment can be referenced. Specifying an id for a compartment is authentic in order to establish a reference to the metamodel.

```
compartment       ::= "compartment" "{" compartmentBlock "}".
compartmentBlock  ::= identifier [layout] [spacing] [margin] [stretching].
layout            ::= "layout" ":" ("fixed" | "vertical" | "horizontal" | "fit").
spacing           ::= "spacing" ":" integer.
margin            ::= "margin" ":" integer.
stretching        ::= "stretching" "(" "horizontal" ":" boolean ","
                      "vertical" ":" boolean ")".
```

Lis. 9 EBNF representation of a compartment

CONNECTIONS

Listing 10 describes the basic structure of a connection. The keyword connection initiates the definition of a connection which is complemented by a unique connection id. In round brackets, an already defined style can optionally be referenced. This style is inherited to everybody sub-element and can be overwritten or enhanced by every sub-element by specifying the property styles, but only the relevant style information for the connection are applied. This means that a style can be used at the same time for the definition of a connection and a shape. The attribute connection-type defines whether the connection is layouted independently or by an algorithm which has to be implemented within the generator. The predefined option for the free adjustment is freeform or manhatten for simple manhatten layout algorithm. The property placing takes over the positioning of shapes on or at a connection. A placing always consists of the property's position, an element or predefined shape. For the element applies the same rules and the same structure is necessary as discussed in the subsection "Element DSL".

```
connection       ::= "connection" identifier
                     [ "(" "style" ":" identifier ")" ] "{" connectionBlock "}".
connectionBlock  ::= connectionType [placing] [style].
connectionType   ::= "connection-type" ":" ("freeform" | "manhatten").
placing          ::= "[" position (geoMetricFigure | shape) "]".
position         ::= "position" "(" "offset" ":" double "," "radius" ":" integer
                     "," "angle" ":" integer ")".
geoMetricFigure  ::= (line | polyline | rectangle |
                     polygon | ellipse | textfield).
shape            ::= "shape" ":" identifier.
```

Lis. 10 EBNF representation of a connection

Through this, it is possible to define complicated forms on a connection, which cannot be created with just one basic form like an arrow head.

The free positioning of a placing is allowed with the property position. A position requires the parameters offset, radius and angle. The parameter offset defines on which relative point of the connection the anchor point is defined. The value range of the offset moves between 0 (0% origin of the connection in draw direction) and 1 (100% end

of the connection). The radius specifies in which distance from the anchor point the placing will be placed in relation to the specified angle. The radius can assume any integer value, the angle is always limited to the 360° of a circle. With this procedure, the user can define shapes above or below the connection, which is necessary for a lot of diagram types like the **E**ntity **R**elationship **D**iagram (ERD).

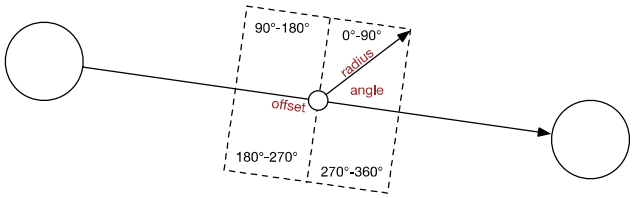

Fig. 2 Explanation of a Placing

B. ζ-Style DSL

The presentation of shapes is an independent area and should be separated from the definition of shapes. Besides it makes sense, to define a "Corporate style" for a developed diagram to make changes to the whole diagram as easy as possible. This causes, with the creation of a new diagram, a separation of structure and presentation. The approach of separation of structure, content and presentation is already a big part of the web development with the **H**yper**t**ext **M**arkup **L**anguage (HTML) (structure and content) and **C**ascading **S**tyle **S**heets (CSS) (presentation). The Style DSL uses ideas of the CSS, which have proven in the context of a style language for diagram types for graphic editors. A style is initiated with the keyword of the same name and is identified by a unique style name. An existing style can be extended or overwritten with the keyword extends and the unique style name. All attributes of the referenced style are inherited by the use of the keyword extends.

```
style           ::= "style" identifier
                    ["extends" identifier] "{" styleBlock "}".
styleBlock      ::= ["transparency" ":" double]
                    ["background-color" ":" allCol]
                    ["gradient-orientation" ":" ("horizontal"|"vertical")]
                    ["line-color" ":" allCol]
                    ["line-width" ":" integer]
                    ["line-style" ":" lineType]
                    ["font-name" ":" string]
                    ["font-color" ":" hexCol]
                    ["font-size" ":" integer]
                    ["font-italic" ":" boolean]
                    ["font-bold" ":" boolean]
                    ["highlighting" ":" highlightingBlock].
highlightingBlock ::= "(" "selected" ":" hexCol "," "multiselected" ":" hexCol
                    "," "allowed" ":" hexCol "," "unallowed" ":" hexCol ")".
allCol          ::= hexCol|"transparent"|identifier.
hexCol          ::= hex|color.
lineType        ::= "solid"|"dot"|"dash"|"dash-dot"|"dash-dot-dot".
hex             ::= "#" 6 * (digit | letter).
color           ::= "red"|"blue"|"green".
```

Lis. 11 EBNF representation of a style

An object can be influenced regarding the opacity by the use of the attribute "transparency". This causes that the element to be shown is displayed transparently in the value range between 0.0 (completely visible) and 1.0 (completely transparency). The background color can be defined with the attribute background-color. Valid values of the background color are a Hexadecimal color value, a predefined color (default RGB colors) or a reference to a defined gradient (see Listing 12). If a gradient is used, the orientation can be influenced by the parameter gradient-orientation. It can be chosen between the values, horizontal and vertical. The lines of an element can be influenced with the line attributes (color, width and styles). For the line-color are the same values valid as for the background color. The line thickness (attribute line-width) can be defined by passing the pixel information. For the line types are five kinds predefined (solid, dot, dash, dash-dot and dash dot dot). The representation of text is steered with the font attributes. These range from the font-name of the font-color by specifying a hexadecimal or a predefined color until the font size in pixels. Besides it can be defined about true/false whether the writing should be shown in italics or fat. The attribute highlighting defines with which color selected or several selected (multi-selected) elements are bordered. In addition, can be steered about the parameters allowed and unallowed which color should appear with a valid or invalid connection of two elements around the target

element. For all four parameters, a Hexadecimal color value or a predefined color can be given. The definition of gradients is initiated with the keyword gradient, followed by a unique name. Within a gradient, there are several areas. An area owns the parameters color (Hexadecimal color value or a predefined color) and offsets (values between 0.0 and 1.0). The offset specifies in percent from which place should be started with the given color. There are at least two areas mandatory to create a gradient. The area specifications can be extended to any number of areas.

```
gradient        ::= "gradient" identifier "{" gradientBlock "}".
gradientBlock   ::= ["description" ":" {letter}] area area {area}.
area            ::= "area" "(" "color" ":" hexCol "," "offset" ":" double ")".
```

Lis. 12 EBNF representation of a gradient

C. ζ-Diagram DSL

The diagram DSL establishes the link between the defined graphic elements with the developed metamodel. Thereby it is possible to connect a specific graphic representation to a Metamodel-class with the properties which are defined in the Metamodel-class. The diagram definition is initiated by the keyword diagram, followed by a unique diagram name. After that, the referenced metamodel must be specified in order to establish a relationship between them. Optional a specified style can be referenced which serves as the basic style for the whole diagram. Through this definition it is possible to create a corporate design for the whole diagram which of course can always be overwritten by an element.

In a diagram action groups can be created. An action Group represents a collection of actions that can be integrated by different nodes. An action is initiated by the keyword action, followed by a unique name. In addition, an action owns the parameters label and implementation. The label name of the action is displayed in the front-end. The real implementation of an action must be implemented by the user itself. It is crucial that a location is predefined within the generator where the custom implementations are stored. Within the action, the class name and the accompanying method name can be defined by the keywords "class" and "method". There will be no further testing of the implemented source code and it is up to the programmer that the implementation is complete and correct.

The definition of nodes is initiated with the keyword node followed by a unique node name. The link of the node with the metamodel-class is prepared by the syntax "for <nodename>". In addition, an independent style can be assigned to a specific node. The assignment of a graphical representation of a node is performed by the attributes "shape" which references a previously defined shape. The relationship of metamodel attributes to in-/output fields within a shape can be realized with the syntax "more Attribute / string -> field name". The number of relations between In-/output fields and more Attributes is unlimited. A purely static text can be achieved by passing a normal string literal. The property "palette" specifies the name under which the elements are grouped.

Furthermore, the behavior of individual nodes can be manually extended with the properties "onCreate", "onDelete" and "onChange". For this purpose, already defined action Groups can be involved by the keyword "include" or node-specific actions can be created within the property by specifying an action section. The number of definable actions is unlimited, but the user must ensure that the runtime of the application stays in a tolerable execution time.

```
diagram       ::= "diagram" identifier "for" identifier
                  ["(" style ")"] "{" diagramBlock "}".
style         ::= "style" ":" identifier.
diagramBlock  ::= {actionGroup | node | edge}.
actionGroup   ::= "actionGroup" identifier "{" action {action} "}".
action        ::= "action" identifir "(" "label" ":" identifier ","
                  "implementation" ":" identifier ")".
node          ::= "node" identifier "for" identifier
                  ["(" style ")"] "{" nodeBlock "}".
nodeBlock     ::= [shape] [palette] [container] [methods] [actions].
shape         ::= "shape" ":" identifier [ "(" [properties] [compartment] ")" ].
properties    ::= ("var" identifier "->" identifier) |
                  ("val" {letter} "->" identifier).
compartment   ::= "nest" identifier "->" identifier.
palette       ::= "palette" ":" {letter}.
container     ::= "container" ":" {letter}.
methods       ::= [onCreate] [onUpdate] [onDelete].
onCreate      ::= "onCreate" "{" [actionBlock] [askFor] "}".
onUpdate      ::= "onUpdate" "{" [actionBlock] "}".
onDelete      ::= "onDelete" "{" [actionBlock] "}".
actionBlock   ::= ["call" "action" identifier]
                  ["call" "actionGroup" identifier].
askFor        ::= "askFor" ":" identifier.
actions       ::= "actions" "{" actionsBlock "}".
actionsBlock  ::= ["include" identifier ["," identifier]] [action].
edge          ::= "edge" identifier "for" identifier
                  ["(" style ")"] "{" edgeBlock "}".
edgeBlock     ::= [connection] ["from" ":" identifier]
                  ["to" ":" identifier] [palette]
                  [container] [methodes] [actions].
connection    ::= "connection" ":" identifier [ "(" [properties] ")" ].
```

Lis. 13 EBNF representation of a diagram

V. IMPLEMENTATION GUIDE

In this section we explain, how to realize a concrete implementation with the before described languages. This is done using the **B**usiness **P**rocess **M**odel and **N**otation (BPMN) [8] as example.

The example in figure 3 shows how to create an envelope for the mail event of BPMN with the ζ-Elements DSL.

```
shape BPMN_EventMail {
   ellipse {
      size(width=50, height=50)
      position(x=0, y=0)
      rectangle {
         size(width=30, height=20)
         position(x=10,y=15)
         style(background-color=blue)
         polygon {
            point(x=0, y=0)
            point(x=15, y=10)
            point(x=30, y=0)
         }
      }
   }
}
```

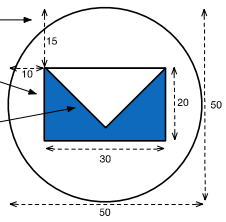

Fig. 3 Creation of the Shape „mail event“ (BPMN)

The whole figure is inside a circle, an ellipse where width and height have the same value of 50. Nested into this ellipse is a rectangle which forms the edge of the envelope. The width of the rectangle is 30 and the height is 20. The position is expressed relative to the surrounding ellipse and is shifted 15 pixels down and 10 pixels to the right. Finally, a polygon completes the envelope; also, its coordinates are relative to its container, the rectangle. The polygon points run from the top left (0,0) across the lower middle (15,10) up to the upper right point (30,0). The information of the background color of the rectangle leads to the fact that the rectangle is filled completely with blue. However, the polygon overlays a part of the parent element by which this area is let out. This is due to the fact that nested elements lie higher from the point of view of the cross section (Z-index).

```
connection BPMN_ConditionalFlow {
   placing {
      position(offset=1.0, radius=0, angle=0)
      polygon {
         point(x=-10,y=10)
         point(x=0,y=0)
         point(x=-10,y=-10)
         style(background-color=black)
      }
   }
   placing {
      position(offset=0.0, radius=0, angle=0)
      polygon {
         point(x=0,y=0)
         point(x=-20,y=10)
         point(x=-40,y=0)
         point(x=-20,y=-10)
      }
   }
}
```

Fig. 4 Creation of a Connection as Conditional Flow (BPMN)

A connection always consists of a line that connects two shapes with each other. Any connection has always a source and a target anchor, on which the attachment of the connection is done. But connections can have decorations, such as arrowheads or a shape. In the example of BPMN, there is a conditional connection, shown in Figure 4. This connection has two decorations, the arrowhead at the one end and the rhombus at the other. Text such as the connection name or the cardinality can be positioned on the connection as well. The arrowhead is realized with a definition of a placing at the end of the line. End of the line means that the offset is defined as 1.0 (100%) of the position property. The radius and the angle do not matter because the placing is on the line. Within the placing a polygon is defined. This runs through the points (-10,10), (0,0) and (-10, -10). The point (0,0) is the top of the triangle. Finally, the triangle is filled in black by the definition of the background color. The second placing, at the beginning of the line, defines a rhombus. The position of the

beginning is achieved through the definition of the offset with the value 0.0. The rhombus is a polygon, with the points (0,0) below, (-20,10) left, (-40,0) top and (-20, -10) right.

```
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:true
   font-bold:false
}
```

Lis. 14 Creation of a default Black and White Style

Listing 14 shows the definition of a Black and White Style. This style is similar to the default style of the connection "BPMN Conditional Flow" in Figure 4. Only the transparency does not match. The definition of the transparency of 0.95 would mean that the element is barely visible. The background-color of all elements of the style would be applied and would be defined with white. The lines definition with the color black, solid and width 1 applies to both connections as well as to the edges of elements. The defined font properties were in an italic illustration, not bold printed in a size of 10 pixels with the font "Tahoma" in black. These properties are applied to all texts, for which the style is used, as long as they are not overwritten by another style definition.

```
diagram bpmn for BPMNModelElement (style:BpmnDefaultStyle) {
   node BPMNActivityCallActivity for MMElement {
      shape:BPMN_Activity_CallActivity(MMAttribute -> shapeName)
      palette:"Shapes"
      onCreate {
         action askFor shapeName
      }
   }
   edge BPMNConditionalFlow for MMElement{
      connection:BPMN_ConditionalFlow {
         from fromElement;
         to toElement;
      }
      palette:"Connections"
   }
}
```

Lis. 15 Diagram File Example content

The next model is the ζ-Diagram DSL. This is of particular importance to link the defined elements and the metamodel. Listing 15 defines a diagram with the name "BPMN" which refers to a ζ-Core metamodel with the name "BPMNModelElement". Furthermore, the specified style "BpmnDefaultStyle" is applied to all elements which are referenced or defined in this diagram. The defined node "BPMN Activity Call Activity" is stored in the metamodel class "MMElement". This assignment is realized by using the keyword "for". The graphical representation of nodes is possible through the assignment of a shape to the same named attribute. In this example, the shape "BPMN_Activity_Call Activity" is referenced and the value of the text box "shape name" is stored within the metamodel attribute "MMAttribute" of the metamodel class "MMElement". The defined node is placed in the category "Shapes" of the generated editor. While the creation of a new node the user is asked directly for the name of the new element. This behavior is implemented via the method "onCreate". This uses the standard action "askFor" on the attribute "shape name". The definition of an edge behaves identically to the one of a node. Besides the fact that an edge has the attributes "from" and "to". These attributes reference the metamodel classes from or to which connections are possible.

VI. EVALUATION

An evaluation of the developed languages was conducted with a total of 38 participants. The participants were divided into the groups Bachelor and Master Student, Developers and Others. Each group received a 4-hour introduction to meta-modeling and in Zeta. After the introduction and a break, each participant had to work on 4 tasks sequentially. The tasks were to use the diagram (Task one), shape (Task two) and style (Task three) language separately and in conjunction in the final Task four.

Task one was to use the diagram language. As a basis was, the concept (meta-model), the style and the element description were provided. The aim was to describe the graphical editor and realize the connection between the metamodel and the graphical representations.

As a second task, the element language should be used to create graphic elements. For this purpose, templates were provided for the elements which should be created without layout information. The participants had to create 3 items in total, which had rising complexity. As a basis, the metamodel, the style and the diagram description had been provided. This was needed in order to examine the textual

description with the help of the generated graphical editor.

The next task was to use the style language for the design of the elements. For assistance, the meta-model, the item, the diagram description and the graphic representation of the elements was provided. It is important that the printed graphical representation of the elements contains the layout information. The correct representation could be continuously checked with the help of the generated graphical editor.

The final task was to create a fully functional graphical editor. As a basis the meta-model and the graphical representation of generated elements with layout information was provided. The aim of this task was to apply all previously performed tasks in conjunction.

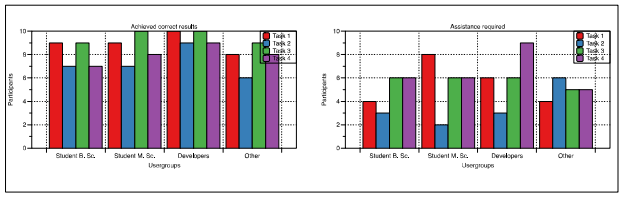

Fig. 5 Achieved correct results and Assistance required

Figure 5 on the left side shows that 90% of participants could solve task one, 72.5% could successfully solve task two, 95% task three and 80% task four correctly. The right diagram in Figure 5 shows that the tasks could be finished without assistance. 55% of the participants did not need help with the editing of task one, 35% task two, 57.5% task three and 65% at task four. Task two could only be solved correctly by about 3/4 and only 35% of the participants did not need help, it's probably because the definition of complex figures by a textual description requires more exercise. This presumption is supported by the note of the participants that a graphical illustration of the form would have been very helpful. However, it is important that a large part of the participants were able to successfully finish the tasks.

During the processing of the tasks, some usage problems have occurred. These were divided into the groups critical and non-critical. We defined that a critical usage problem is that the defined goal of the task cannot be achieved without concrete assistance by the exercise leader. Furthermore, these also include problems with the infrastructure, such as system crashes or errors which are owed to the development environment. Figure 6 shows the distribution of the usage problems for each user group and task. In black all occurred usage problems are listed. In red, only the critical use problems are illustrated. It turns out that the most common and critical problems have arisen with task 4. This is in direct connection with task 1. A large part of the problem is related to the complexity and abstractness of meta-modeling and has no direct relation to the developed languages. Task 2 and 3 show that the usage problems for the element and style language are low to very low. The usage problems for the element language are greater because the textual description of complex graphical elements, without having a direct preview, is hard to imagine.

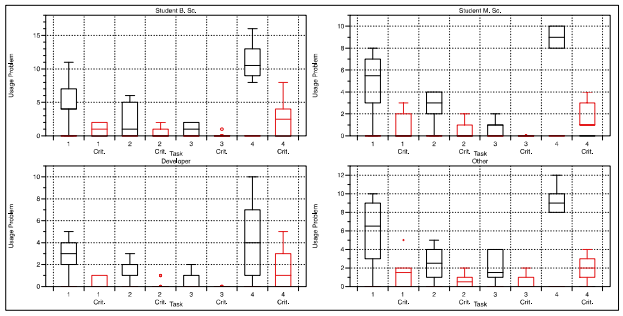

Fig. 6 Usability problems overview per task and user group

Task 4 has the highest complexity which is reflected in the required processing time (see Figure 7). This figure shows the required processing time per task and user group. It can be seen that the processing times for the different tasks and users are different. However, this was to be expected, because each person processes, receives and applies the newly acquired knowledge at different speeds. The most important finding is that there has been a significant improvement in the processing time compared to the sum of task 1 to 3 to task 4. This shows that the languages can be applied more quickly after a period of training.

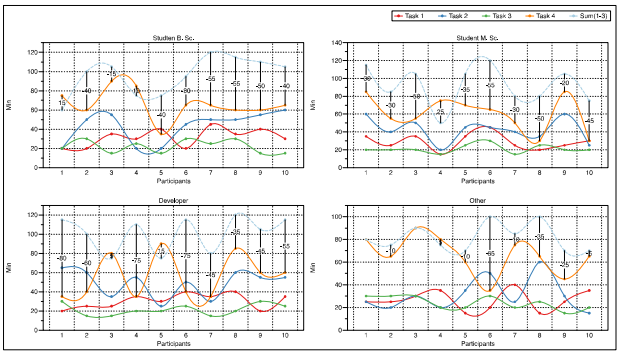

Fig. 7 Detailed processing time per task and user

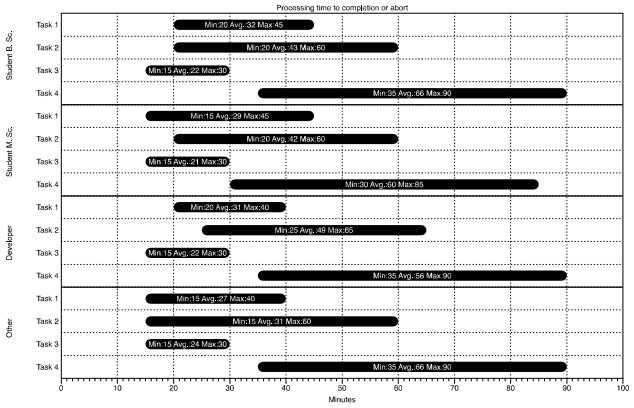

Fig. 8 Processing time to completion or abort

Figure 8 shows the processing time for each user group and task with the fastest, slowest (perhaps canceled) and average value. The values are distributed almost homogeneously. From this, it can be concluded that the languages have a certain complexity but almost every user is able to learn the languages. On average the participants needed 29:45 minutes for task one, 41:15 minutes for task two, 22:15 minutes for task three and 59:50 minutes for task four.

The performed evaluation of the developed languages shows that the languages have many good approaches and the users can handle the languages very well in general. This is demonstrated by the high rate of successfully completed tasks. By using some of the approaches of CSS within the style language, the participants find their way around very quickly. The definition of graphic elements was initially very unfamiliar and the possibilities were not recognized directly. However, this has improved 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, for example, a live preview of the created graphical elements would be very helpful. However, this is not an issue of the languages but is an extension of the editor and tooling. The keyword "ask for" has not been understood directly by many participants. Here it must be considered whether a renaming is useful. The action groups of the diagram language were 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 are available at [3] [4].

VII. CONCLUSION AND FUTURE WORK

In this paper we have shown that developing graphical DSLs can be very efficient by using the presented textual DSLs for node and edge type diagrams. The languages can be used for the model driven development of graphical modeling tools in almost any environment. The presented approach can be used for any specific domain which uses diagrams for visualization. It is not necessary to be an IT expert to use the DSLs, because the DSLs are quite easy to learn, read and write. The necessary meta model and the DSLs can be tailored to the specific needs, thus the models can be very concise. In this paper we showed some elements of the BPMN, but the same approach could be used for various domain-specific modeling languages.

The described approach is by no means finished and still lacks a number of useful features. Examples of such features are the inclusion of shadows, the direct import of SVG images and the

definition of rapid buttons around an element in the editor. These are just a few ideas for some enhancements. More important is the question of the limitations of the approach. In terms of the languages itself, the style DSL could be extended to all useful features of CSS. This would offer more styling and layout options for the user. For some kinds of the usability of the element DSL it would be helpful to import or reference existing SVG images or to integrated a live preview of the described shapes.

The DSLs presented by us are targeted towards graphical languages based on the notion of nodes and edges. In UML, most diagram types fit into that category, however there are a few exceptions. The sequence and the timing diagram show time as one dimension and differ in that sense. They cannot be described with the presented DSLs without considerable extensions.

We see more challenges with the development of the appropriate generator, which generates the graphical editor. Other topics, like multi-user support in collaborative modeling environments, evolution of Metamodels or diffing and merging changes in versions of graphical models remain topics of research and are independent of the presented or a model driven approach.

REFERENCES

[1] A. Aziz and S. Mitchell. An introduction to javascript object notation (json) in javascript and .net. Microsoft Developer Network, 2007.

[2] M. Gerhart, J. Bayer, J. M. Hoefner, and M. Boger. Approach to define highly scalable metamodels based on json. BigMDE 2015, page 11, 2015.

[3] M. Gerhart and M. Boger. Evaluation plan. http://www.zeta-project.org/evaluation/ Evaluierungsplan-MoDiGen-DSLs.htm, 2016. Accessed: 2016-08-04.

[4] M. Gerhart and M. Boger. Evaluation report, http://www.zeta- pro ject.org/evaluation/evaluationsbericht-modigen- dsls.htm. http://www.zeta-project.org/evaluation/ Evaluationsbericht-MoDiGen-DSLs.htm, 2016. Accessed: 2016-08-04.

[5] J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley. The Java Language Specification, Java SE 8 Edition. Addison-Wesley Professional, 1st edition, 2014.

[6] H. W. Lie and B. Bos. Cascading style sheets. Addison Wesley Longman, 1997.

[7] O. L. Madsen and B. Moller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In Conference Proceedings on Object-oriented Programming Systems, Languages and Applications, OOPSLA '89, pages 397–406, New York, NY, USA, 1989. ACM.

[8] Object Management Group. Business Process Model and Notation, Specification V2.0. Needham, Massachusetts, Vereinigte Staaten, 2011. Accessed 2015-01-22.

[9] -. Ore. Theory of Graphs. Number Teil 1 in American Mathematical Society colloquium publications. American Mathematical Society, 1962.

[10] R. S. Scowen. Extended bnf-a generic base standard. Technical report, Technical report, ISO/IEC 14977. http://www. cl. cam. ac. uk/mgk25/iso-14977. pdf, 1998.