Implementation of a domain specific notation language for the ModellingToolkitV2

Master Thesis

Master of Science in Business Informatics

Submitted by

Jonas Nydegger

(Matriculation Number: 15-255-896)

Under the supervision of

Prof. Dr. Hans-Georg Fill

Digitalization and Information Systems Group

Department of Informatics

University of Fribourg

Bern, 2022

Abstract

Keywords

Contents

Abbreviations

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem Statement

1.2 Research Question

Research Question 1 (RQ1):

Research Question 2 (RQ2):

Research Question 3 (RQ3):

1.3 Structure

Chapter 2

Methodology

Problem identification and motivation: Definition of the objectives for a solution: Design and development: Demonstration: Evaluation: Communication:

Chapter 3

Theoretical Foundations

3.1 Modelling methods

To gain a common understanding of the elements and terms of modelling methods for further reading, let us refer to a framework from Karagiannis and Kühn proposed in 2002 - see Fig. 3.1. According to this framework, a modelling method is divided into two components: The modelling technique, consisting of the modelling language and a modelling procedure, and a set of mechanisms and algorithms which are applied on the models instances derived from the modelling language. In figure 3.1 the different components and interrelationships of the aforementioned framework are illustrated. (Karagiannis & Kühn, 2002).

3.1.1 Modelling language

The modelling language is a textual or graphical language and describes all elements which can be used in a given model. A modelling language itself consists of syntax, semantic and notation. These terms are now explained in detail:

Syntax: The Syntax defines the grammar of a modelling language. The grammar describes the elements and rules which are implemented for creating models based on a certain language. In the context of modelling languages, two major approaches for defining their syntax: graph grammars (Rozenberg, 1997) and meta-models (Geisler, Klar, & Pons, 1998). For describing the syntax with a meta-model approach, UML class diagrams are often used (Karagiannis & Kühn, 2002).

Semantics: The Semantic applies a meaning to the syntax and notation. It consists of a semantic domain and a semantic mapping. The semantic domain is responsible for describing the meaning by using e.g. ontologies or mathematical expressions. The semantic mapping on the other hand connects the syntactical constructs with the defined semantic domain (Karagiannis & Kühn, 2002).

Notation: The notation defines the visual representation of the modelling language and its underlying syntax and semantics. There is a differentiation between static and dynamic notation approaches. static approaches define visualisations as described, but do not take the state of model construct instances into

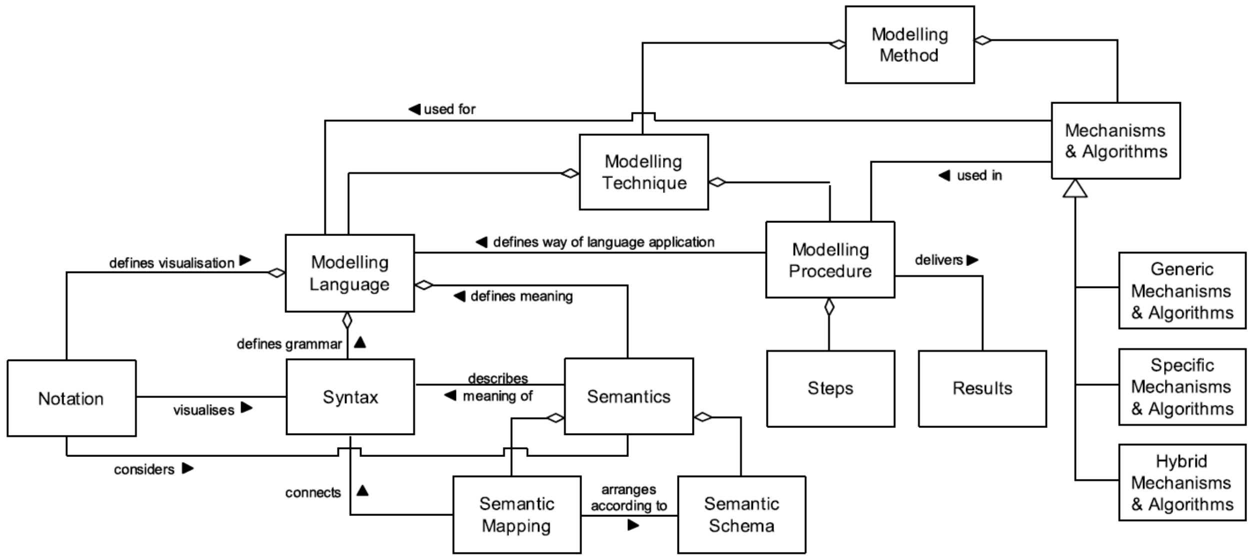


Figure 3.1: The components and interrelationships of the referred framework. (Karagiannis & Kühn, 2002, p. 185). Reprinted from *Proceedings of the Third International Conference on E-Commerce and Web Technologies* (p. 185), by D. Karagiannis and H. Kühn, 2002, Springer.

account. Dynamic approaches differ in that they additionally consider the model state by dividing the notation in a control part and a representation part. The representation part is congruent to the static notation approach. The control part consists of a set of rules, that can change the visualisation depending on the model construct instance state (Karagiannis & Kühn, 2002).

Alogrithms and mechanisms: Algorithms and mechanisms can be applied to the modelling language and in the modelling procedure. Algorithms can be used for processing model information directly in a machine-based approach. One example for the usage of algorithms is to run simulations on a certain model instance. Mechanisms on the other hand, can be used for interactions and the evaluation of a certain model instance derived from the model language. One example can be mechanisms to check to proper application of the defined modeling language. We differentiate between generic mechanisms and algorithms, specific mechanisms and algorithms and hybrid implementations. generic mechanisms and algorithms are defined at the meta-meta model level and are applicable to all meta-models derived from the meta-meta model. specific mechanisms and algorithms are defined on the meta-model and can therefore only be used on the aforementioned meta-model. Hybrid implementations are defined at the meta-meta model and can be adapted on a meta-model level (Karagiannis & Kühn, 2002; Kühn, Junginger, Karagiannis, & Petersen, 1999). In this thesis, we define a meta-model as a model of a modelling language.

Modelling hierarchy

A commonly used method to formalize modelling languages is the meta-modelling approach (Kern, Hummel, & Kühne, 2011). The creation of a meta-model is achieved by using a modelling language called meta-modelling language. The model which describes the meta-model is called $meta^2model$ or meta-meta-model. In other words, the meta-meta model can be seen as a model that defines underlying meta-model languages. A four layer modelling language architecture as described in figure 3.2 is often used in practice (e.g. the

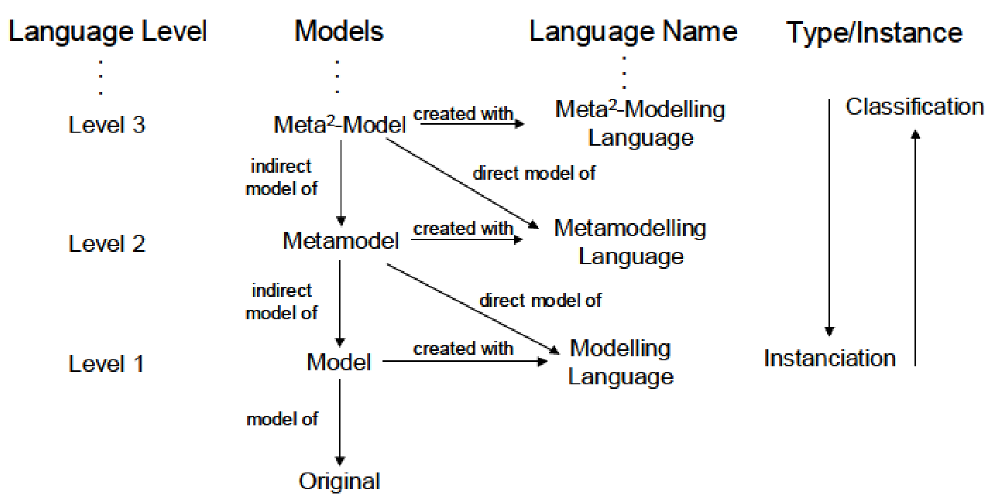


Figure 3.2: The four layer modelling language architecture. (Karagiannis & Kühn, 2002, p. 185). Reprinted from *Proceedings of the Third International Conference on E-Commerce and Web Technologies* (p. 185), by D. Karagiannis and H. Kühn, 2002, Springer.

AdoXX meta-model architecture approach (Fill & Karagiannis, 2015)) for formalizing modelling languages with the meta-model approach. The highest language level forms the meta-meta model and is created based on the meta-meta modelling language. From this meta-meta model, arbitrary meta-models can be defined by using a meta-modelling language. a Model can be created with a dedicated modelling language. This modelling language is a model of the meta-model. On the lowest language level resides the original, from which an abstraction is mapped to a model. (Karagiannis & Kühn, 2002). As described in **??**, a modelling language consists of syntax, semantics and notation. The syntax can be further divided into abstract syntax and concrete syntax. The concrete syntax is an indirect model of the abstract syntax. For instance, the concrete syntax is represented by a model and the abstract syntax is represented by a meta-model.

3.1.2 Modelling Procedure

The modeling procedure defines the necessary steps for applying the modelling language to deliver results (Karagiannis & Kühn, 2002).

3.2 ModellinkgtoolkitV2

For a convenient way of working with modelling methods, the use of (meta-)modelling platforms can be seen as state-of-the-art. These modelling platforms support the definition of (graphical) modelling languages in the form of meta-models and the creation of model instances. Often, they provide further services like e.g. exchanging and storing meta-models, in order to enhance the users convenience (Fill & Karagiannis, 2015). Various solutions in the form of such modelling platforms have been implemented. In chapter 4, multiple modelling platforms will be introduced. However, those modelling platform's meta-meta model Architectures are limited to 2 Dimensional meta-modelling functionalities according to Muff and Fill(2022).

The ModellingtoolkitV2 differs in that it's meta-meta model is designed for 3 Dimensional models and notations with the idea of Supporting AR and VR in future modelling scenarios. It is the first modelling platform that implements this capability on the meta-metalevel in a generic way(Muff & Fill, 2021). It has been implemented in a prototypical manner and is currently under development at the University of Fribourg. So far, only parts of BPMN diagrams and ERD have been implemented for testing purposes. For elaborating the relevant concepts used in traditional 2 Dimensional meta-modeling, Muff and Fill(2021) identified existing meta-meta model solutions. In a next step, the necessary requirements to describe 3 Dimensional notations where derived. The findings and measures drawn from them resulted in the meta-metamodel shown in figure 3.3 for the ModellingtoolkitV2. It has the innovative aspect that is designed for 3 Dimensional notations but can cope with 2 Dimensional input too (Muff & Fill, 2021).

The functionality of the ModellingToolkitV2 is planned to be more extensive than the definition of meta-models and creation models from it. For this thesis, it is sufficient to limit ourselves to aforementioned functionalities. The ModellingToolkitV2 consists of a development toolkit and a modelling toolkit. With the development toolkit it is possible to create arbitrary meta-models. This meta-model creation is currently achieved by interacting with a dedicated API(Application Programming Interface). A web-based GUI for the development toolkit is planned but not yet implemented. The development toolkit can then be used to create models based on previously created meta-models. A web-based GUI is implemented for the development toolkit and is currently accessible on the internal network of the University of Fribourg.

3.2.1 Meta-metamodel

The meta-metamodel shown in Figure 3.3 is only an extrat of the actual constructs due to the limited space. It consists of a meta layer and an instance layer to illustrate the interrelationships between the definition of a modeling language and the actual instances of the specific objects.

The **meta layer** consists of the superclass *metaobject* and the sucessor classes from metaobject are *class*, *role*, *scene_type*, *attribute* and *attribute_type*. These core classes are all inheriting the features of metaobject (Muff & Fill, 2021). The meta layers core part is consisting of classes and relationclasses. The scene_type represents a closed 3 dimensional space of a model and consists of all instantiable classes and relationclasses. Classes, relationclasses and scene_types have own attributes. Those attributes are described with exactly one attribute_type. a class is a construct to define objects which share the same properties and behaviour. With the usage of relationclasses, classes, scene_types, and other relationclasses can be set in relation to each other. A relationclass has always to roles assigned: a *role_from* and a *role_to*. In order to describe to what a role can connect, each one must at least be referenced to one class, relationclass or scene_type. Furthermore, classes and scene_types can contain ports, which can be assigned to roles(Muff & Fill, 2021). An attribute with special importance with respect to this thesis is the *geometry*. This attribute stores the information for the visual representation of derived constructs. This representation is described with a domain specific language(DSL) called VizRep that defines the 3D representation and behaviour of an object. The DSL will be explained in detail in subsection 3.2.3. Moreover, each object with a visual representation contains relative and absolute 3D coordinates. The relative coordinates are used for positioning objects in AR and VR environments. The absolute coordinates can be used to position objects based on real world coordinates like GPS. An attribute to store 2D coordinates is also available in the metaobject.

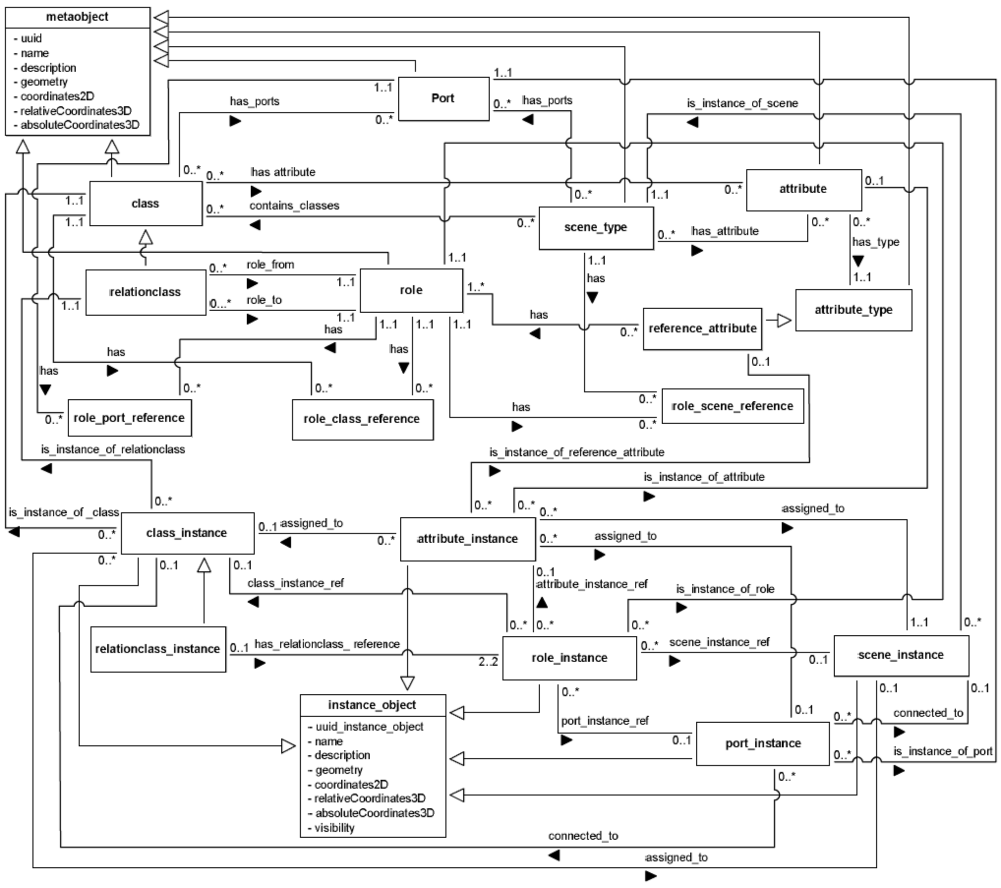


Figure 3.3: The UML class diagram of the Meta-Metamodel Enabling 3 dimensional and 2 dimensional notations (Muff & Fill, 2021, p. 51). Reprinted from *Proceedings of the ER Demos and Posters 2021 co-located with 40th International Conference on Conceptual Modeling* (p. 51), by F. Muff and H.-G. Fill, 2021, CEUR-WS.org.

The **instance layer** of the meta-metamodel is described in the lower half of Figure 3.3. It describes the constructs of how information of actual metamodel instances are kept(Muff & Fill, 2021). In other words, the instance layer classes can be seen as actual instances of the abstract meta layer classes. When an actual model based on a certain meta-model description is instantiated, the instance infromations are stored on the instance layer class instances.

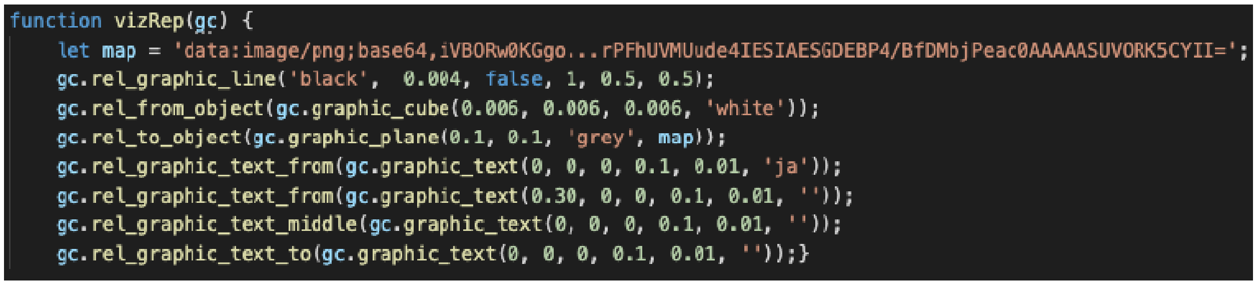
3.2.2 Technology stack

The evaluation of a state-of-the-art technology stack was largely influenced by the requirement that the modeling platform can be run on arbitrary AR an VR devices using a web-based approach(Muff & Fill, 2021). Due to that requirement, the chosen technologies for the ModellingtoolkitV2 are the following: The GUI part is implemented with the Typescript language and the Aurelia Framework. Typescript is a programming language that adds additional syntax to Javascript leading to a tighter integration with the developer. Typescript will be compiled to regular Javascript code (*Typescript Introduction*, n.d.). Aurelia is a fronted Javascript framework used to facilitate the development of Javascript applications (Hunter, 2018). For persisting meta-models, models, notations etc. (i.e. the described classes in Figure 3.3), PostgreSQL is used. PostgreSQL is an open source relational Database System (*PostgreSQL*, n.d.).

3.2.3 VizRep

One important ability for creating modelling languages is the ability to formally define (graphical) shapes of elements for the underlying modelling language via appropriate notations (Karagiannis & Kühn, 2002). The ModellingToolkitV2 has an implementation in the form of a domain-specific coding convention named VizRep for the visual description of shapes in place. With VizRep it is possible, to create programmatic descriptions for a variety of different three-dimensional shapes (Muff and Fill 2021).

VizRep serves as a wrapper to describe one or multiple shapes defined in a *GraphicContext* class. If one describes multiple shapes in VizRep, the shapes will be combined. The GraphicContext class supports the creation of models of type *class* and *relationclass.* In chapter 3.2.1, the terms class and relationclass have been introduced in detail. The GraphicContext class contains a predefined set of functions for defining shapes and an instance of it must be given as an input value to the VizRep wrapper function. See figure 3.5 for a sample VizRep description. With this VizRep description, a total of seven shapes are created. Six of them are of the model type class and one is of the model type relationclass. However, those classes serve the purpose of visualising a relation. This example will be visualised as a black line with text at the beginning, at the middle and at the end of the line. Furthermore, the black line will have a wite cube at its beginning and a plane with a specific *map* applied to it at the end. A map can be an arbitrary encoded PNG image defined beforehand. It is also possible to assign dynamic values with VizRep by defining variables that can access model instance data during run time. The shape definitions itself are implemented with *Three.js.* Three.js is a JavaScript library for facilitating the creation and visualisation of arbitrary 2D and 3D content on a Website (*Three.js Fundamentals*, n.d.). See figure 3.5 for a business process diagram modelling the functionality of describing shapes in VizRep. There are two different sub processes in place. One describes the definition of class model types and the other describes the definition of relationclasses. Possible class definitions are: cube, plane and sphere. To each of those, it is possible to add dimensions, a color and a



```
function vizRep(gc) {
    let map = 'data:image/png;base64,iVBORw0KGgo...rPFhUVMUude4IESIAESGDEBP4/BfDMbjPeac0AAAAASUVORK5CYII=';
    gc.rel_graphic_line('black',  0.004, false, 1, 0.5, 0.5);
    gc.rel_from_object(gc.graphic_cube(0.006, 0.006, 0.006, 'white'));
    gc.rel_to_object(gc.graphic_plane(0.1, 0.1, 'grey', map));
    gc.rel_graphic_text_from(gc.graphic_text(0, 0, 0, 0.1, 0.01, 'ja'));
    gc.rel_graphic_text_from(gc.graphic_text(0.30, 0, 0, 0.1, 0.01, ''));
    gc.rel_graphic_text_middle(gc.graphic_text(0, 0, 0, 0.1, 0.01, ''));
    gc.rel_graphic_text_to(gc.graphic_text(0, 0, 0, 0.1, 0.01, ''));}
```

Figure 3.4: An exemplary VizRep description. (?, ?)ow to cite this?

map. It is also possible to add a text element and a port element. Possible relationclass definitions are lines. To those lines, text can be added at the beginning, middle and a the end. Additionally it is possible to add a shape of model type class to the beginning and the end of a line. Those two sub processes can be called any numbers of time within VizRep. In order to eventually visualise the defined shapes on the ModellingToolkitV2, they first must be added to a Three.js *scene instance* and then the scene instance can be rendered by a Three.js *render instance*.

3.3 Visualisation

The visual representation is an integral part when working with graphical modelling languages. It has a profound effect on usability and effectiveness of modelling languages (Moody, 2009). Visual (modeling) languages are particularly useful in domain specific models since they are able to support visual metaphors tailored to their domain (Wolter, 2013). In this context, the term 'visualisation' is referred to the definition given in (Fill, 2009) as "the use of graphical representations to amplify human cognition". The symbols for visualising the syntactical construct can be defined by using e.g. pixel-based or vector based graphics (Karagiannis & Kühn, 2002). If not further restricted by the the meta modelling suite, the development and design of adequate visualisations is in the responsibility of the method engineer (Fill & Karagiannis, 2015). Since the design of notations has a significant impact on the usability of the described model, one should invest at least (if not more) the same amount of resources on notation design as he does on semantic issues (Moody, 2009).

3.4 Domain-specific languages

Domain-specific language(DSLs) are languages that are designed to fulfill the requirements for a certain application domain(Mernik, Heering, & Sloane, 2005). A well designed DSL is given when a profound understanding of the underlying application domain is available and it's design is implemented accordingly (Van Deursen & Klint, 2002). With the provision of notations and constructs tailored to a certain application domain in the form of a DSL, the dedicated DSL can lead to a reduced time to market(Van Deursen & Klint, 2002), higher productivity(Mernik et al., 2005) and reduced maintenance cost(Mernik et al., 2005) compared to general purpose Languages(GPL). Furthermore, it results in a reduction of needed domain and programming expertise which opens up an application domain to a larger group software developers

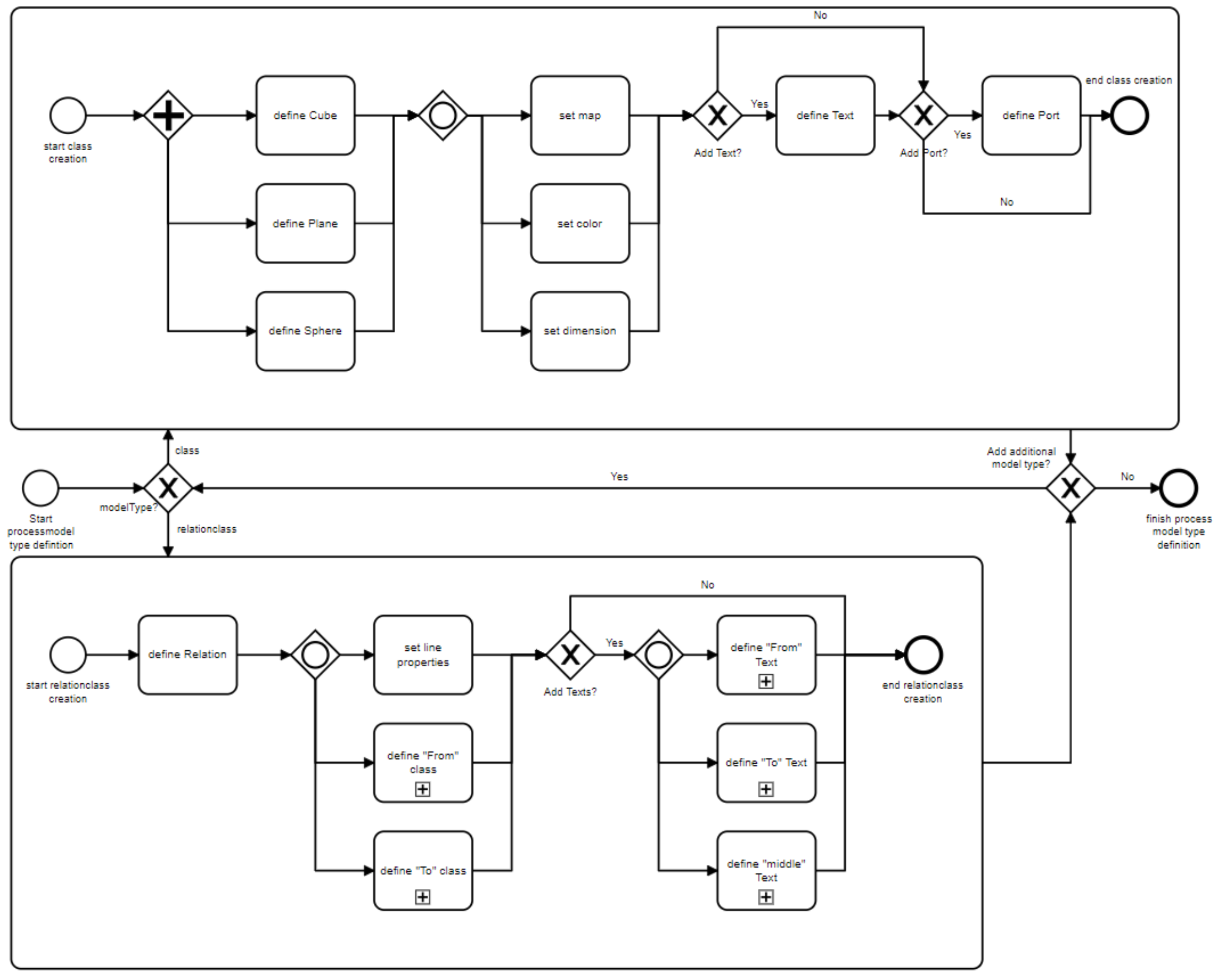


Figure 3.5: BPMN diagram for creating model types with vizRep (?, ?)ow to cite this?

compared to GLP's(Mernik et al., 2005). A well known example for a DSL is Excel from the Microsoft company. Java, which is a Programming language, on the other hand, is a popular GPL.

3.4.1 Feature model

The architecture of a DSL is based on an in-depth analysis of the structure of the involved application domain that leads to a profound understanding. Guides to aquire this understanding are provided by the research field of *domain analysis* (Van Deursen & Klint, 2002). A *feature model* is a domain analysis's most significant output(Czarnecki, Østerbye, & Völter, 2002). The feature model discusses the similarities and differences among members of the same software family as well as the inter dependencies between the variable features (Van Deursen & Klint, 2002). The *feature diagram*, a graphical representation for showing interdependence between (variable) features, is a key component of the feature model. It describes every configuration (called instances) possible for a software system with a focus on the features that possibly differ amongst multiple configurations. In Chapter 6, a feature diagram of the implemented DSL is provided. Only a cursory explanation of the feature diagram notation is provided, with examples serving as the bulk of the explanation. In this thesis, we will follow the notation definition of (Van Deursen & Klint, 2002) which is inspired by (Kang, Cohen, Hess, Novak, & Peterson, 1990).

Soll ich hier noch ein beispiel einbringen, um notation zu beschreiben oder reicht es, wenn ich das im design chapter direkt mache?

3.5 Lexers and Parsers

Parsing is the process of examining a series of terminal symbols that complies with the requirements of a formal grammar(Aho, Lam, Sethi, & Ullman, 2006). In this context, a grammar refers to a variety of things. It can describe a natural language like German, a programming language like Java, or just a data format like Json (Ortin, Quiroga, Rodriguez-Prieto, & García, 2022).

A parser is consisting of the parser itself and a lexer. A lexer recognizes terminal symbols in an input and creates tokens out of it. This process is also called lexical analysis (Louden, 1997). The parser uses the corresponding lexer to detect combinations of terminal symbols of a given context-free grammar and verifies its syntax. typically this context-free grammar is described with the Backus Naur Form(BNF) notation (Hutton, 1992) or with the Extended Backus Naur Form(EBNF) notation (Ortin, Quiroga, Rodriguez-Prieto, & García, 2022). The context-free grammar defined for this thesis in the BNF form can be found in chapter6. If the verification against a defined syntax is successful, a parse tree is created that transforms the input into a hierarchical data structure(Aho et al., 2006). The field of usage for parser implementations in computer science is vast. Compilers make usage of the provided parsers functionality for byte-code translations, software development tools for e.g. debugger functionality and the JSON format for formal data processing. This leads to the fact that parsing has a great importance in today's computer science and the need for automating the generation of parser arose (T. J. Parr & Quong, 1995).

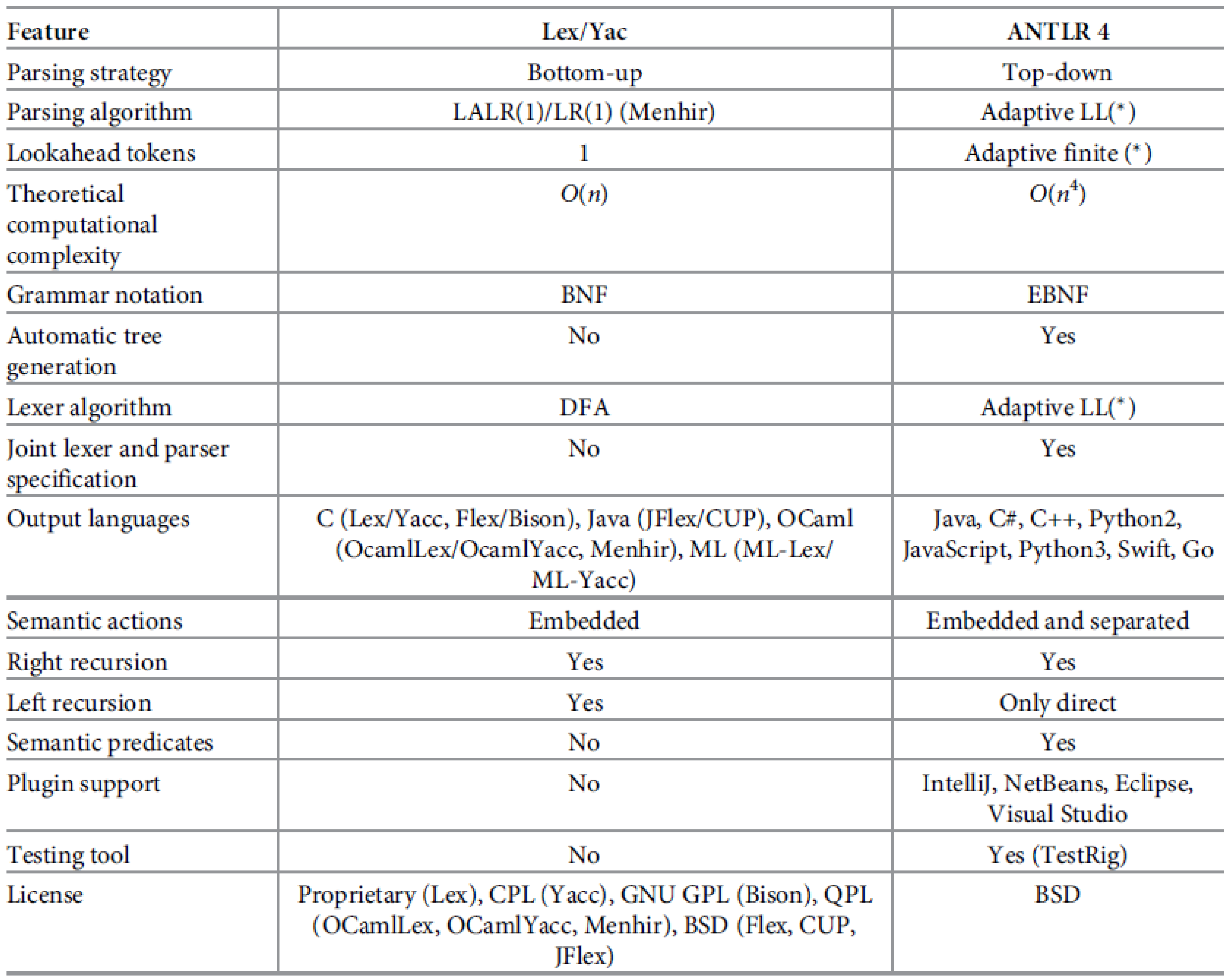
3.5.1 Parser generators

Today, Numerous parser generators are available. Yacc/Bison, ANTLR and JavaCC are some of the best known, openly available parser generators(Ortin, Quiroga, Rodriguez-Prieto, & Garcia, 2022). With the usage of these tools, it is sufficient to define the syntax of a certain context-free grammar and the parser generator will generate the parser for the defined language(T. J. Parr & Quong, 1995). Despite the great choice of parser generators, most developers aim either the Yacc/Lex approach or ANTRL. (Ortin, Quiroga, Rodriguez-Prieto, & García, 2022).

Comparison Lex/Yacc and ANTLR4

Figure 3.6 shows a comparison of the main characteristics between The Lex/Yacc and ANTLR4 parser generators. Yacc has a the bottom-up approach as parsing strategy in place. The bottom-up strategy only generates a node on the parse tree when all is child nodes have been generated beforehand. This strategy is implemented by a Look-Ahead LR (LALR(1)) parser algorithm (DeRemer & Pennello, 1982). ANTLR4 on the other hand uses the top-down approach as parsing strategy. This strategy generates the parent nodes before the child nodes within the parse tree. It is implemented by the adaptive LL(*) parser(T. Parr & Fisher, 2011). The LL(*) parser inherits a (theoretical) computation complexity of $\mathcal{O}(n^4)$ where n is the number of nodes, whereas LALR(1) has a (theoretical) computation complexity of $\mathcal{O}(n)$. However, in real programming language realizations, the computational complexity turned out to be linear(T. Parr, Harwell, & Fisher, 2014). Lex/Yacc uses the BNF to formalize the grammars of a certain language(Levine et al., 1992). ANTRL4 on the other hand has EBNF in place. The EBNF contains additional operations compared to BNF like operations provided in regular-expressions(Regex)(T. Parr, 2013). ANTLR4 automatically generate the parse tree, whereas in Yacc this step needs to be done by hand(Ortin, Quiroga, Rodriguez-Prieto, & Garcia, 2022). Moreover, ATNLR4 has EBNF and LL(*) in place for both, lexical and syntactic language descriptions. Yacc supports just syntax descriptions. The lexical description is realized by a dedicated lexer, commonly by using Lex, which uses Deterministic Finite Automata(DFA) as the lexer algorithm(Levine et al., 1992). In ANTLR4, it is possible to generate parsers in multiple programming languages by providing one grammar description. With the provision of so called listeners, ANTLR4 enables a decoupling of semantic actions (i.e. application code) and grammars by separating grammars from the actual behaviour. Yacc does not inherit this capability. This means semantic actions have to be written directly into the grammar description(Ortin, Quiroga, Rodriguez-Prieto, & Garcia, 2022). ANTLR4 supports semantic predicated to a certain degree in order to mitigate issues with grammar ambiguities. Eventually ANTLR 4 has plugins in place for different integrated development environments (IDE's) and its own testing tool.

In the research article of Wie soll ich das zitieren? an evaluation of empirical features regarding productivity, tool simplicity, intuitiveness and maintanability comparing ANTLR4 and Lex/Yacc has been conducted. The results show that ANTLR4 has better results in all three(productivity, tool simplicity, intuitiveness) measured features. The plugin support and the testing tool of ANTLR4 was not part of the quantitative evaluation. However, study participants and lecturers detected a qualitative difference by using these tools.



Feature	Lex/Yac	ANTLR 4
Parsing strategy	Bottom-up	Top-down
Parsing algorithm	LALR(1)/LR(1) (Menhir)	Adaptive LL(*)
Lookahead tokens	1	Adaptive finite (*)
Theoretical computational complexity	$O(n)$	$O(n^4)$
Grammar notation	BNF	EBNF
Automatic tree generation	No	Yes
Lexer algorithm	DFA	Adaptive LL(*)
Joint lexer and parser specification	No	Yes
Output languages	C (Lex/Yacc, Flex/Bison), Java (JFlex/CUP), OCaml (OcamlLex/OcamlYacc, Menhir), ML (ML-Lex/ ML-Yacc)	Java, C#, C++, Python2, JavaScript, Python3, Swift, Go
Semantic actions	Embedded	Embedded and separated
Right recursion	Yes	Yes
Left recursion	Yes	Only direct
Semantic predicates	No	Yes
Plugin support	No	IntelliJ, NetBeans, Eclipse, Visual Studio
Testing tool	No	Yes (TestRig)
License	Proprietary (Lex), CPL (Yacc), GNU GPL (Bison), QPL (OCamlLex, OCamlYacc, Menhir), BSD (Flex, CUP, JFlex)	BSD

Figure 3.6: A qualitative comparison between Lex/Yacc and ANTLR4 (?, ?)ow to cite this?

Chapter 4

Related Work

In this chapter it is shown what kind of methods and implementations for notations in the 3 and 2-Dimensional environment already exist. Since the MetamodellingtoolkitV2 is the first modeling platform that supports 3 Dimensional models and notations on a meta-meta model level (Muff & Fill, 2021), we also investigated in 2 Dimensional modelling platforms approaches regarding their capabilites for creating and visualising graphical notations and how to apply existing design patterns to a 3 Dimensional environment. Eventuell auch noch CAD query reinnehmen, da 3D dsl vorhanden. Ist halt nicht in Modellierungssprachen aber trotzdem hilfreich

4.1 Graphical notation modelling in 3D space

Several approaches for modelling graphical 3D notations have been conducted. For instance in the work of Betz et al. (2008), the goal was to represent business processes more compactly and increase the amount of provided information by adding a third dimension to them. The visualization capabilities of existing process modeling languages (e.g. BPMN model) are limited due to the fact that the amount of information contained in a process model is often larger than what can be visualized with 2D graphical notations. A current solution of 2D modeling languages is the provision of multiple models. By adding an additional dimension, the visual representation of data can be more compact and the new dimension can be used for new notation elements. Thanks to that, it is possible to overcome certain use cases where switching between models was necessary (Betz et al., 2008). One use case provided by Betz et al. (2008) was the visualization of the relationship between a process model and the organizational model with the usage of a third dimension. In this use case, the third dimension was used for visualizing the relationship between roles from the organizational model and the process steps from the process model. For the implementation, the petri net modeling language was chosen. Petri nets use the petri net markup Language (PNML) for defining petri net model instances in a standardized way. PNML is a XML-based standard format (Jüngel, Kindler, & Weber, 2000). Current versions of PNML do not support the definition of 3D notations for Petri nets (Betz et al., 2008). In the work of Betz et al. (2008) the PNLM grammar was extended to meet the criterions for 3D environments. Concretely they added properties like *zCoordinate* for specifying the position of an element on the Z-axis or *depth*, *height* and *width* to define the 3D size of elements. These properties are added to a dedicated, isolated XML schema embedded into the PNLM. Thanks to that, it is possible to provide notation capabilities for 2D and 3D concurrently (Betz et al., 2008). Even tough the generated visualizations of the model instances are in 3D and can possibly be used for VR and AR, it is restricted to the petri net meta-model instance. An adaption to a generic, meta-model instance is not possible due to it's implementation based on the extension of the PNML. In the work of Brown (2010), a 3D BPMN editor has been implemented in a prototypical manner that visualizes model instances in 3D. However, there is no information about how the BMPN standard has been implemented or enhanced to a third dimension (Brown, 2010). BPMN3D is another approach for visualizing graphical notations in the 3D environment. Hipp et al. (2014) empirically evaluated various technologies for visualizing business process models. BPMN3D is one of them and provides the capability to enhance certain(i.e. process tasks notations) BPMN elements into a 3D environment. This is achieved by adding a *pole* which is represented in the 3D space. However, this 3D pole is mapped afterwards back to the 2D space (Hipp, Strauss, Michelberger, Mutschler, & Reichert, 2014). See figure 4.2 for an example of an BPMN3D model instance. This leads to the fact that any interaction with the model in the 3D environment is impossible. Interactions in 3D are crucial for AR and VR applications. The UBBA tool proposed in the work of Abdul et al. (2019) is able to visualize and execute customized 3D BPMN instances. UBBA takes a standart BMPN definition (Model, 2011) in the form of an XML and a set of custom 3D graphics(must have the .fbx format) as input. Each of those graphics represent a notation element of the BMPN file. UBBA then maps the custom 3D element to the notation elements of the 2D process definition and visualizes it accordingly(Abdul, Corradini, Re, Rossi, & Tiezzi, 2019). The concept of separating the process model definition in 2D and the corresponding visualisations in 3D can be extended to arbitrary modelling languages. The main benefit is that no changes at the modelling language itself has to be done like in the work of Betz et al. (2008). however, the mapping process has to be implemented for each modelling language individually.

Another Graphical notation solution in the 3D environment is provided by *CadQuery*. CadQuery is a parametric model generator written in the Python language for defining and building 3D CAD models. How to cite github repos?? https://github.com/CadQuery/cadquery. Shapes are described by concatenate an arbitrary (must be conform with the syntax) chain of predefined parametric commands. See figure **??** for an example definition and figure **??** for a preview of the generated visualization. Although this technology is used in the description of model language notations, their language design for defining 3D elements is applicable to other fields of study.

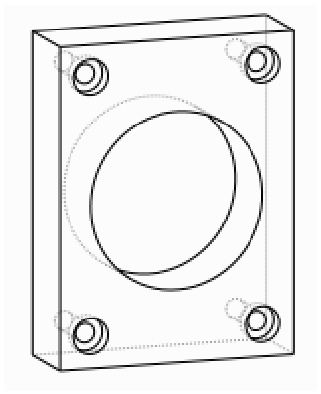
4.2 GRAPHREP in ADOxx

The ADOxx meta modelling platform is in use and under continuous development since over twenty years. It has been used and tested by a great number of research and industrial projects (Fill & Karagiannis, 2015). Similar to the VizRep coding convention of ModellingToolkitV2, it comes with an own notation DSL for describing visual shapes of elements called GRAPHREP. For the instantiation of meta models, the *meta*^{2}*model* of ADOxx provides two model types: classes and relationclasses (Fill & Karagiannis, 2015). A similar architecture can be found in the meta-meta model of ModellingtoolkitV2 (see figure 3.3.) Both model types are supported to be defined and visualised with the GRAPHREP DSL. The possibilities with GRAPHREP regarding graphical notation creation is more extensive compared to VizRep from Modelling-ToolkitV2. See the table 4.1 for a comparison of supported graphic elements. *Planes, spheres, cubes, texts*

```
#Defining and assigning variables
(length,height,diameter,depth,padding) = ( 13.0,40.0,22.0,10.0,8.0)

result = Workplane("XY")
        .box(length,height,depth)
        .faces(">Z")
        .workplane()
        .hole(diameter)
        .faces(">Z")
        .workplane()
        .rect(length-padding,height-padding,forConstruction=True)
        .vertices().cboreHole(2.4,4.4,2.1)
```

(a) Example shape definition in CadQuery



(b) Example shape visualisation in CadQuery cow to cite? https://cadquery.readthedocs.io/en/latest/intro.html

Figure 4.1: Example CadQuery application

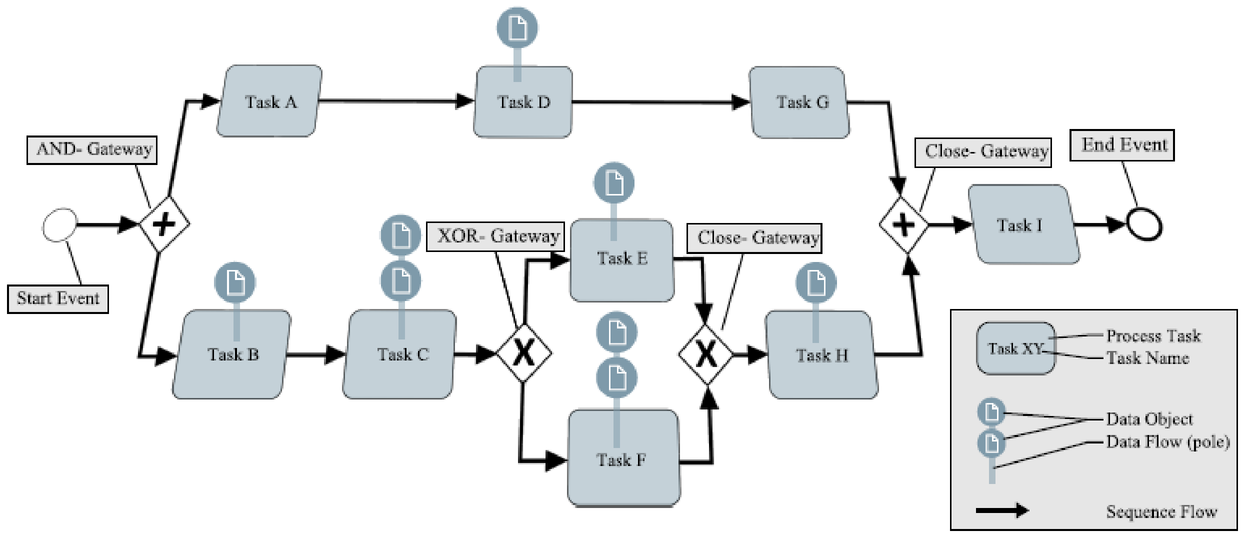
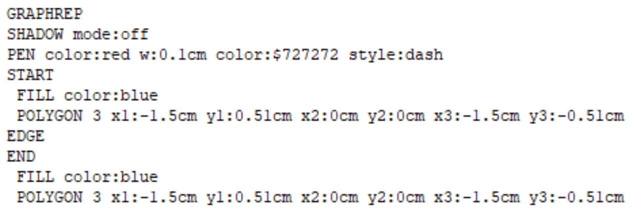
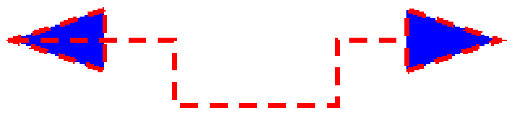


Figure 4.2: example BMPN3D model instance showing visualization concept (Hipp et al., 2014, p. 6)



```
GRAPHREP
SHADOW mode:off
PEN color:red w:0.1cm color:$727272 style:dash
START
 FILL color:blue
 POLYGON 3 x1:-1.5cm y1:0.51cm x2:0cm y2:0cm x3:-1.5cm y3:-0.51cm
EDGE
END
 FILL color:blue
 POLYGON 3 x1:-1.5cm y1:0.51cm x2:0cm y2:0cm x3:-1.5cm y3:-0.51cm
```

(a) Sample relation definition in GRAPHREP



(b) notation of sample relation definition

Figure 4.3: A figure with two subfigures

and *lines* are supported from both meta-modelling suites. With *curves* and *bezier curves*, lines of arbitrary characteristics can be defined. With *polygons* any two dimensional shape, consisting of straight lines can be defined. With *compound*, a combination of any polygons and curves is possible. All graphics elements are defined and visualized in the 2D space. Soll ich hier noch ein beispiel einfügen?

Graphic element	VizRep	GRAPHREP
Plane	Yes	Yes
Sphere	Yes	Yes
Cube	Yes	Yes
Line	Yes	Yes
Text	Yes	Yes
Curve	No	Yes
Bezier curve	No	Yes
Polygon	No	Yes
Point	No	Yes
Compound	No	Yes

Table 4.1: Comparison of supported graphical Element definitions.

Besides graphic elements GRAPHREP also provides style elements. They define the look of graphic elements(*ADOxx Documentation*, n.d.). The following style elements are provided by GRAPHREP: *PEN* determines how lines and curves are drawn (e.g. red, bold, dashed line), *FILL* defines how (closed) graphic elements have to be filled (e.g. blue, solid), SHADOW decides whether a graphic element applies a shadow or not, *STRETCH* determines defines if geometric stretching shall be applied to a graphic element or not and *FONT* defines the font for displayed texts. Furthermore, GRAHPREP provides classyfing elements. They are used exclusively for relations. The following classifying elements are supported: *START*, *MIDDLE*, *END*. They define which which point of the relation is defined. And additional classifying element is *EDGE*. It triggers the drawing of an edge withing the relation. See Figure 4.3a for a sample relation definition in the GRAPHREP DSL and Figure 4.3b for the corresponding notation.

The last element structure provided from GRAPHREP are *control elements*. The instantiation of graphical notations is processed sequentially. with control elements, it is possible to skip certain process step based on input variables. It is also possible to assign attribute values of object instances to a variable. A graphical representation can therefore be received depending the status of object instances. This leads to the fact that ADOxx follows a dynamic notation approach as described in subsection 3.1.1. The meta-meta model of ADOxx and its DSL GRAPHREP are deigned and implemented for a 2 Dimensional environment only. However, the constructs of the DSL can be evaluated for a possible extension to a 3 Dimensional space.

Classyfing elements and control elements seem to be independent of the Dimensionality of the provided environment. Graphic elements and (certain) style elements need corresponding extensions.

4.3 Eclipse modeling framework and Meta Edit+

In the Eclipse modeling framework(EMF), meta model can be described based on a provided meta-meta model. The framework is implemented in Java. Due to that, a developer must have in-depth knowledge of the language in order to implement a meta-model(Fill & Karagiannis, 2015). For the creation of graphical editors, EMF provides an additional, Java based framework called graphical editor framework(GEF). GEF again is used from a framework called graphical modelling framework(GMF). One of the provided functions from GMF is to map the in the EMF described metamodel elements to graphical elements on the editor provided from GEF(Bernd Kolb, n.d.). GMF knows certain base notation elements (e.g. rectangle). those can be used in the GEF without furhter specifications. GEF uses the graphic framework *draw2D* for the creation of graphical elements. If the provided graphical elements from EMF are not sufficient, it is possible to define custom shapes with draw2D(Bernd Kolb, n.d.). As one can see, the usage of EMF requires in-depth knowledge about java and a lot of different frameworks. For the definition oh notations, a set of predefined shapes is in place. If those are not sufficient, custom shapes have to be defined with another framework written in Java.

Meta Edit+ shares similar features with AdoXX and the ModellingToolkitV2. For instance, it offers a straight forward way to define visualisations for meta modelling languages comparable to the previously presented meta modelling suites(Fill & Karagiannis, 2015). The main difference in the context of visualisations is, that it is possible to describe arbitrary shapes without the need of a programmatic approach (Tolvanen & Rossi, 2003). Meta Edit+ comes with a *symbol editor* functionality, where one can define shapes in a visual manner. For instance, shapes can be selected via a drop-down, colors can be defined by selecting them on a color picker scheme and so on (*MetaEdit+ Documentation*, n.d.). Identical to the approach of ADOxx and EMF, Meta Edit+ provides only the shape definition in a 2D environment.

Chapter 5

Design Objectives and Requirements

5.1 Requirements of the Solution

Chapter 6

Design

Ich will: EBNF ==> schaue nochmal was Fill für visualiserung geschickt hat. feature model. Parse tree visualisierung.

6.1 Backus naur form

6.2 Parser rules

6.3 Lexer rules

Chapter 7

Implementation

Chapter 8

Demonstration and Evaluation

Chapter 9

Discussion and Outlook

References

Abdul, B. M., Corradini, F., Re, B., Rossi, L., & Tiezzi, F. (2019). Ubba: unity based bpmn animator. In *International conference on advanced information systems engineering* (pp. 1–9).

Adoxx documentation. (n.d.). Retrieved 2022-05-06, from `https://www.adoxx.org/live/graphrep`

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, techniques, and tools (2nd edition)..

Bernd Kolb, M. V. A. H., Sven Effinge. (n.d.). *Graphical modeling framework.* Retrieved 2022-12-25, from `https://voelter.de/data/articles/ix-gmf2.pdf`

Betz, S., Eichhorn, D., Hickl, S., Klink, S., Koschmider, A., Li, Y., ... Trunko, R. (2008). 3d representation of business process models. *Modellierung betrieblicher Informationssysteme (MobIS 2008).*

Brown, R. (2010). Conceptual modelling in 3d virtual worlds for process communication. In *Proceedings of the 7th asia-pacific conferences on conceptual modelling (apccm 2010)* (pp. 1–8).

Czarnecki, K., Østerbye, K., & Völter, M. (2002). Generative programming. In *European conference on object-oriented programming* (pp. 15–29).

DeRemer, F., & Pennello, T. (1982). Efficient computation of lalr (1) look-ahead sets. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *4*(4), 615–649.

Fill, H.-G. (2009). Survey of existing visualisation approaches. In *Visualisation for semantic information systems* (pp. 39–159). Springer.

Fill, H.-G., & Karagiannis, D. (2015, December). On the Conceptualisation of Modelling Methods Using the ADOxx Meta Modelling Platform. *Enterprise Modelling and Information Systems Architectures*, *Vol 8*, 4–25 Pages. Retrieved 2022-05-06, from `https://emisa-journal.org/emisa/article/view/99` (Artwork Size: 4-25 Pages Publisher: Gesellschaft für Informatik e.V. (The German Informatics Society)) doi: 10.18417/EMISA.8.1.1

Geisler, R., Klar, M., & Pons, C. (1998). Dimensions and dichotomy in metamodeling: Technical report № 98-5. *Berlin: TU Berlin*, 1–31.

Hipp, M., Strauss, A., Michelberger, B., Mutschler, B., & Reichert, M. (2014). Enabling a user-friendly visualization of business process models. In *International conference on business process management* (pp. 395–407).

Hunter, S. (2018). *Aurelia in action.* Shelter Island, NY: Manning Publications Co. (OCLC: on1002833529)

Hutton, G. (1992). Higher-order functions for parsing. *Journal of functional programming*, *2*(3), 323–343.

Jüngel, M., Kindler, E., & Weber, M. (2000). Towards a generic interchange format for petri nets–position paper. In *Meeting on xml/sgml based interchange formats for petri nets* (pp. 1–5).

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., & Peterson, A. S. (1990). *Feature-oriented domain analysis (foda) feasibility study* (Tech. Rep.). Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.

Karagiannis, D., & Kühn, H. (2002). Metamodelling platforms. In *Ec-web* (Vol. 2455, p. 182).

Kern, H., Hummel, A., & Kühne, S. (2011). Towards a comparative analysis of meta-metamodels. In *Proceedings of the compilation of the co-located workshops on dsm'11, tmc'11, agere! 2011, aoopes'11, neat'11, & vmil'11* (pp. 7–12).

Kühn, H., Junginger, S., Karagiannis, D., & Petersen, C. (1999). Metamodellierung im geschäftsprozeßmanagement: Konzepte, erfahrungen und potentiale. In *Modellierung'99* (pp. 75–90). Springer.

Levine, J. R., Mason, J., Levine, J. R., Mason, T., Brown, D., & Levine, P. (1992). *Lex & yacc.* " O'Reilly Media, Inc.".

Louden, K. C. (1997). Compiler construction: Principles and practice..

Mernik, M., Heering, J., & Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, *37*(4), 316–344.

Metaedit+ documentation. (n.d.). Retrieved 2022-12-25, from `https://www.metacase.com/de/mwb/draw_symbols.html`

Model, B. P. (2011). Notation (bpmn) version 2.0. *OMG Specification, Object Management Group*, 22–31.

Moody, D. (2009). The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on Software Engineering*, *35*(6), 756-779. doi: 10.1109/TSE.2009.67

Muff, F., & Fill, H.-G. (2021). Initial concepts for augmented and virtual reality-based enterprise modeling. In *Er demos/posters* (pp. 49–54).

Ortin, F., Quiroga, J., Rodriguez-Prieto, O., & García, M. (2022). An empirical evaluation of lex/yacc and antlr parser generation tools. *PLoS ONE*, *17*.

Ortin, F., Quiroga, J., Rodriguez-Prieto, O., & Garcia, M. (2022). Evaluation of the use of different parser generators in a compiler construction course. In *World conference on information systems and technologies* (pp. 338–346).

Parr, T. (2013). The definitive antlr 4 reference. *The Definitive ANTLR 4 Reference*, 1–326.

Parr, T., & Fisher, K. (2011). Ll (*) the foundation of the antlr parser generator. *ACM Sigplan Notices*, *46*(6), 425–436.

Parr, T., Harwell, S., & Fisher, K. (2014). Adaptive ll (*) parsing: the power of dynamic analysis. *ACM SIGPLAN Notices*, *49*(10), 579–598.

Parr, T. J., & Quong, R. W. (1995). Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, *25*(7), 789–810.

Postgresql. (n.d.). Retrieved 2022-30-10, from `https://www.postgresql.org/`

Rozenberg, G. (1997). *Handbook of graph grammars and computing by graph transformation* (Vol. 1). World scientific.

Three.js fundamentals. (n.d.). Retrieved 2022-05-07, from `https://threejs.org/manual/#en/fundamentals`

Tolvanen, J.-P., & Rossi, M. (2003). Metaedit+ defining and using domain-specific modeling languages and code generators. In *Companion of the 18th annual acm sigplan conference on object-oriented programming, systems, languages, and applications* (pp. 92–93).

Typescript introduction. (n.d.). Retrieved 2022-05-09, from `https://www.typescriptlang.org/`

Van Deursen, A., & Klint, P. (2002). Domain-specific language design requires feature descriptions. *Journal of computing and information technology*, *10*(1), 1–17.

Wolter, J. (2013). Specifying generic depictions of language constructs for 3d visual languages. In *2013 ieee symposium on visual languages and human centric computing* (pp. 139–142).