universität wien

DISSERTATION / DOCTORAL THESIS

Titel der Dissertation / Title of the Doctoral Thesis

„Language-Oriented Modeling Method Engineering“

verfasst von / submitted by

Nikša Višić

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Doktor der technischen Wissenschaften (Dr. techn.)

Wien, 2016 / Vienna 2016

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 786 880
Dissertationsgebiet lt. Studienblatt / field of study as it appears on the student record sheet:	Informatik
Betreut von / Supervisor:	o. Univ.-Prof. Dr. Dimitris Karagiannis

Abstract

This dissertation tackles the scientific issues concentrated on the realization of modeling methods and their industrial applications. It extends upon a well-established metamodeling approach and the technology that supports it – metamodeling platforms. The main focus of the work at hand is on a discipline called “Modeling Method Engineering” and its final product: modeling tools.

The technology utilized to produce a modeling tool has a big influence in forming a modeling and metamodeling community. Thus, there are several communities that have been formed around particular metamodeling software. However, on a closer inspection, it can be recognized that a majority of available metamodeling platforms are built upon very similar meta^{2}models. They differentiate in naming of the concepts which have the same underlying semantics, and in technical description of concepts, which means that the semantically same concepts have a different syntactical representation.

To leverage this similarity, a language (MM-DSL) that includes all the relevant metamodeling concepts has been designed. In the requirements engineering phase, two design approaches have been used: top-down and bottom-up. In top-down approach several modeling methods have been analyzed. The purpose was to determine the most commonly used concepts and to establish their appropriate abstractions. The bottom-up approach gave insight on how are metamodeling technologies applied for realization of modeling methods.

MM-DSL on its own would not be of much use in the real world. Therefore, an IDE in which one can code modeling methods and a connector (a translator) to the metamodeling platforms have been developed. MM-DSL together with its supporting technology enables language-oriented modeling method engineering, where one writes a program that describes a modeling method and translates it to a modeling tool by utilizing the already existing functionality of a metamodeling platform. A side effect of this approach is that the development of modeling tools does not depend on a particular technology, because MM-DSL programs can be executed on several different metamodeling platforms.

The language-oriented modeling method approach utilizing MM-DSL has been evaluated by the metamodeling community through two different evaluation studies. One study focused on the understandability and expressivity of MM-DSL itself, while the other was used to evaluate the MM-DSL IDE and its usability.

Zusammenfassung

Diese Dissertation behandelt die wissenschaftlichen Fragen, die sich mit der Realisierung von Modellierungsmethoden und ihren industriellen Anwendungen befassen. Sie stützt sich auf und erweitert gleichzeitig einen gut etablierten Metamodellierungsansatz und die Technologie, die diesen Ansatz unterstützt: Metamodellierungsplattformen. Der Schwerpunkt der vorliegenden Arbeit liegt auf einer Disziplin namens „Modellierungsmethoden-Engineering“ und deren Endprodukt: Modellierungswerkzeugen.

Die Technologie, die verwendet wird um ein Modellierungswerkzeug zu produzieren, hat großen Einfluss bei der Bildung einer Modellierungs- und Metamodellierungsgemeinschaft. Somit gibt es mehrere Gemeinschaften, die um bestimmte Metamodellierungssoftware gebildet wurden. Jedoch kann bei einer näheren Inspektion erkannt werden, dass die Mehrheit der verfügbaren Metamodellierungsplattformen auf sehr ähnliche Meta^{2}Modelle gebaut werden. Die Meta^{2}Modelle unterscheiden sich in der Benennung von Konzepten, die die gleiche zugrundeliegende Semantik haben, und in der technischen Beschreibung von Konzepten, was bedeutet, dass semantisch gleiche Konzepte eine andere syntaktische Repräsentation haben.

Um diese Ähnlichkeit auszunutzen, wurde eine Sprache (MM-DSL) entwickelt, die alle relevanten Metamodellierungskonzepte beinhaltet. In der Anforderungsanalyse wurden zwei Gestaltungsansätze verwendet: Top-Down und Bottom-Up. Im Top-Down Ansatz wurden verschiedene Modellierungsmethoden analysiert. Das Ziel war es, die am häufigsten verwendeten Begriffe zu ermitteln und deren entsprechende Abstraktionen zu etablieren. Der Bottom-Up Ansatz ermöglichte den Einblick in die Anwendung von Metamodellierungstechnologien für die Realisierung von Modellierungsmethoden.

MM-DSL allein würde nicht von großem Nutzen in der realen Welt sein. Daher sind eine integrierte Entwicklungsumgebung (IDE), in der man Modellierungsmethoden codiert, und ein Verbinder (Übersetzer) für die Metamodellierungsplattformen, entwickelt worden. MM-DSL, zusammen mit den Technologien die es unterstützen, ermöglicht spracheorientiertes Modellierungsmethoden-Engineering, wo man ein Programm schreibt, das eine Modellierungsmethode beschreibt, und es durch die Nutzung der bestehenden Funktionalität einer Metamodellierungsplattform in ein Modellierungswerkzeug übersetzt. Ein Nebeneffekt dieses Ansatzes ist die Entwicklung von Modellierungswerkzeugen, die nicht von einer bestimmten Technologie abhängen, weil ein MM-DSL Programm auf verschiedenen Metamodellierungsplattformen ausgeführt werden kann.

Der Ansatz des spracheorientierten Modellierungsmethoden-Engineering, der MM-DSL verwendet, ist von der Metamodellierungsgemeinschaft durch zwei verschiedene Evaluationsstudien bewertet. Eine Studie konzentrierte sich auf die Verständlichkeit und Expressivität von MM-DSL selbst, während in der anderen die MM-DSL IDE und ihre Verwendbarkeit bewertet wurde.

Acknowledgment

This research project would have not been possible without continuous inspiration, support and assistance from my supervisor. It has been my privilege to work closely with Univ.-Prof. Dr. Dimitris Karagiannis. I have enjoyed the opportunity to observe and learn from his knowledge and experience. His insights and patience with me will always be appreciated.

I would like to thank my colleagues and modeling and metamodeling community members that were a great help during my research. I am very thankful for all the conversations and discussions we had.

I wish to thank my parents who were a constant source of encouragement during my whole life, without whose love and understanding I would not have completed this work.

Copyright Notice

I have tried my best to find all copyright owners of images and to collect their agreements to use the content in this thesis. Should there be a copyright infringement, please inform me about it.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

DSL	Domain-specific language
DSPL	Domain-specific programming language
DSM	Domain-specific modeling
DSML	Domain-specific modeling language
GPL	General-purpose language
GPPL	General-purpose programming language
GPMaL	General-purpose markup language
GPMoL	General-purpose modeling language
IDE	Integrated development environment
MM-DSL	Modeling Method Domain-Specific Language
MME	Modeling method engineering
EBNF	Extended Backus-Naur Form
UML	Unified Modeling Language
SysML	Systems Modeling Language
XML	Extensible Markup Language
HTML	HyperText Markup Language
SGML	Standard Generalized Markup Language
RDF	Resource Description Framework
OMG	Object Management Group
MOF	Meta-Object Facility
EMF	Eclipse Modeling Framework
ISO	International Organization for Standardization
CASE	Computer-aided software engineering
SDK	Software development kit
SUS	Subject under study
W3C	World Wide Web Consortium
LHS	Left-hand side
RHS	Right-hand side
OMiLAB	Open Models Initiative Laboratory

KE	Knowledge engineering
SVG	Scalable Vector Graphics
API	Application programming interface
RDBMS	Relational database management system
JVM	Java Virtual Machine
AST	Abstract syntax tree

Foreword

As a computer scientist, there are fewer experiences that are more satisfying than developing new concepts and seeing those concepts help people realize their ideas. These concepts can take many forms. For example, a new software development methodology, different ways of defining system or software architecture, a computer language, or an innovative software application.

Concepts are developed to solve existing and future problems. In my case, the problem was in the particular notion present in modeling research communities: it is assumed that modeling tools are available. This assumption entails another assumption, which is that someone created those modeling tools. Therefore, someone used a tool for creating modeling tools. The alternative is far more difficult: from scratch implementation with a chosen programming language. So, my assumption in the time I started working on this dissertation project was that the majority of modeling tools is created using metamodeling technology. This has proven to be true, because without the use of metamodeling platforms, or frameworks, it would be very difficult to develop modeling tools within a feasible time frame.

Providing metamodeling tools that satisfy the needs of a modeling community is an important metamodeling research activity. I wanted to provide a solution that can be easily used by domain experts, but at the same time avoid creating another click-based metamodeling platform. I chose to provide a programming-based solution, because programming as a discipline has become incredibly pervasive nowadays and it will continue to be so in the future. Trends indicate that every future discipline may have something to do with writing code. For example, there are many domain experts outside computer science that use programming languages in their everyday work: chemists, biologists, physicists, economists, mathematicians, etc. These languages are specifically designed for solving one or several connected problems in a chosen domain.

This observation has motivated the development of a domain-specific programming language called MM-DSL (Modeling Method Domain-Specific Language). It is meant to be a straightforward, easy to learn language for coding modeling methods. The code written with MM-DSL is on an abstract level, therefore technology independent – it can be compiled and executed on different execution environments, typically metamodeling platforms.

In this work I present the research that has been done in order to create MM-DSL, which includes creation of the unified taxonomy used in metamodeling research, detailed background research in various metamodeling technologies, as well as language design concepts and best practices, specification details, usage scenarios, and several different evaluations.

I believe that MM-DSL provides several advantages to the modeling tool engineering. The most prominent ones are: technology (platform) independency, fast prototyping and reusability. Some of the evaluation results indicate these advantages as well. MM-DSL may not be perfect, but it is a step in the right direction. Hopefully, it will continue to be developed further as a joint effort of modeling and metamodeling communities, because it can only help improve metamodeling concepts and technology.

1 Introduction

1.1 Computer Languages

Computer languages are one of the core tools computer scientists have in their arsenal. They offer various means of communication between software (system) developers, and can be considered as one of the greatest achievements in the software engineering domain. Nowadays they are the most prevalent way for telling computers what to do. The word *"telling"* is intentionally used instead of the word *"programming"*. Applying the modern languages, especially the domain-specific ones, is more like a communication between a developer and a machine, and it doesn't resemble the old ways from a couple of decades ago. That is because, as everything on this Earth, languages evolve as well. It is a fact that can be tracked all the way from 1950's – the time of FORTRAN – until present day – time of C#, Java, and a huge palette of all kinds of domain-specific languages (DSLs) [1]. Evolution, in this context, does not only mean that one language has changed over a period of time. It also means that new languages have been created using some of the concepts from older, already existing, languages. Good examples are C++ and C#. Both of them have taken something from C and improved on it. C# has also, among improving C, as well as C++, taken some concepts from Java. PHP has a C-like syntax; JavaScript has some of the concepts used in Java, etc. New languages are designed together with emergence of new technology, with a purpose to control and manipulate it as easier as possible. This is especially true, if existing languages feel clumsy, and not as productive as they should be.

Designing a computer language has become much easier than it was in the time of FORTRAN. Nowadays, a considerable amount of new languages are designed by a single person or a small team [1]. This is one of the reasons DSLs are becoming increasingly popular. Among other reasons are: affordable investments, standardization is not as necessity as it used to be in the past, out-of-the-box tools for development of any kind of languages, either textual or graphical, and the Internet as a medium for distribution and user feedback and requests.

Although, it is much easier to design and develop DSLs than it was before, one should not discard an already existing language and start creating another language from scratch as soon as technology changes. Naturally, a lot of concepts from the previous language can be used, but one still needs to define the syntax and semantics, develop tool support (e.g., an editor or integrated development environment), write documentation, and prepare training materials. All of this can be avoided if the DSL has been designed to follow the progress of technology it depends upon, and predict the future changes in the domain it describes. For example, C++, Java and C# are very good at adapting to the technology changes. Since their creation (C++ in 1983, Java in 1995, and C# in 2000) these languages have evolved drastically. This has been possible be-

cause a lot of work has been invested in making these languages modifiable (addition of new features, replacement or removal of old features without much effort). They adapt and evolve as people use them. Only through usage some of the design flaws surface, or a need for a specific feature arises. One of the modern language design philosophies is explained in a paper describing the history of Lua [2], which is not as popular as Java or C++, but it has been around since 1993 and has grown into one of the most known scripting languages. The authors indicate that most successful languages are raised rather than designed, which means that they start as a small language, with modest goals, and evolve through their usage.

Concerning the DSL created during this dissertation project, one should consider the following: (1) the future development of metamodeling platforms, and (2) the possible changes in the application domain, which can come from various sources: new founding in the academia or industry, or insights during the use of a DSL.

To be able to cope with the upcoming issues, the DSL should be able to evolve through modifications, particularly through extension. However, one always needs to keep in mind that usability and expressiveness must not be lowered with introduction of new concepts. It is very important to keep the equilibrium between the following three key properties of a language: usability, expressiveness, and extensibility.

Usability means that a language must be fit for its purpose. Defining a new domain-specific language doesn't make much sense if it is not fit to describe a problem at hand much better than other available general-purpose languages (GPLs). It should support the developer in capturing features of the application without placing unnecessary barriers in the way of development [3].

DSLs should always be more expressive than any GPLs in the domain that they describe, because they always provide high-level features which make a language easier to use and contribute to the readability of the code. Even a domain expert without any programming experience can easily learn and apply a DSL.

Extensibility is very important, not just for DSLs, but for every computer language. If the language possesses the right means for extension, it is easier to introduce new concepts to all aspects of a language, from the syntax to the compiler. This entails that the complete architecture of a language needs to be available to the developer.

1.2 The Term "DSL"

Although, it is assumed that the reader of this work is familiar with the meaning of the term *domain-specific language* or *DSL*, it somehow always has a slightly different meaning to different individuals (we are talking only about DSLs in computer science). Therefore, DSL as a term nowadays has very blurred boundaries. Some things are clearly DSLs, but others can be argued one way or the other. The term has also been around for a while now and, like most things in software engineering, has never had a very firm

definition. This fuzziness of terms is an issue that is making confusion between various scientific communities.

Let us take a look at the following *"definitions"* of a DSL:

> *"… a computer programming language of limited expressiveness focused on a particular domain." [4]*
>
> *"A DSL is a language designed to be useful for a limited set of tasks, in contrast to general-purpose languages that are supposed to be useful for much more generic tasks, crossing multiple application domains… There are strong relations between DSLs and models … A DSL is a set of coordinated models." [5]*
>
> *"DSLs suggest an appealing escape route from the restriction of programming languages, enticing us with the idea of languages whose syntax and semantics can be customized to our purpose… they are customized languages which allow classes of related problems to be more quickly and precisely realized than with traditional techniques." [6]*
>
> *"DSLs abstract from the domain entities and operations to represent domain knowledge in the form of an executable language." [7]*
>
> *"… is a programming language or executable specification language that offers trough appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain." [8]*
>
> *"Domain-specific languages (DSLs) are languages tailored to a specific application domain. They offer substantial gain in expressiveness and ease of use compared with general-purpose programming languages in their domain of application." [9]*

As one can see in previous statements, there are general traits of a DSL that almost all acknowledge. Most important ones are domain specificity and higher abstraction level. Confusing are the traits that are opposed. Some of them are: expressiveness (which is characterized as *"limited"* by one author and *"enhanced"* by the other) and executability (Do DSLs need to be executable or not?). DSLs are also classified as programming languages by some authors, immediately throwing most of specification languages and non-executable languages out of this category.

Varieties of DSL definitions bring more confusion than clarity into this topic. Therefore, for the purpose of this work, the following definition is used:

A domain-specific language (DSL) is a computer language particularly targeting a specific application domain, and is very good at expressing solutions to problems in that specific domain. Its concrete syntax can be either textual, or graphical or both. It can be executable, but it doesn't have to! It can be Turing complete, but it doesn't have to (and typically it is not)! It can be independent or embedded into a host language. Therefore, if languages like markup language, modeling language, specification language, script-

ing language, or programming language contain the above mentioned properties, they are considered domain-specific.

1.3 The Term “Modeling Method”

As the title of this work states, the main goal of this research is to introduce domain-specific languages into the process of modeling method engineering. There are dozens of reasons why this is beneficial, and it will be explained in detail in the following chapters. To be able to perceive the purpose of this DSL, it is important to understand the term *modeling method*, and the difference between a modeling language and a modeling method.

A modeling method [10] consists of two components: (1) a modeling technique, which is divided in a modeling language and a modeling procedure, and (2) mechanisms & algorithms working on the models described by a modeling language (see Figure 1). The modeling language contains the elements with which a model can be described. The modeling procedure describes the steps applying the modeling language to create results, i.e., models. Algorithms and mechanisms provide *“functionality to use and evaluate”* models described by a modeling language. Combining these functionalities enables the structural analysis, as well as simulation of models.

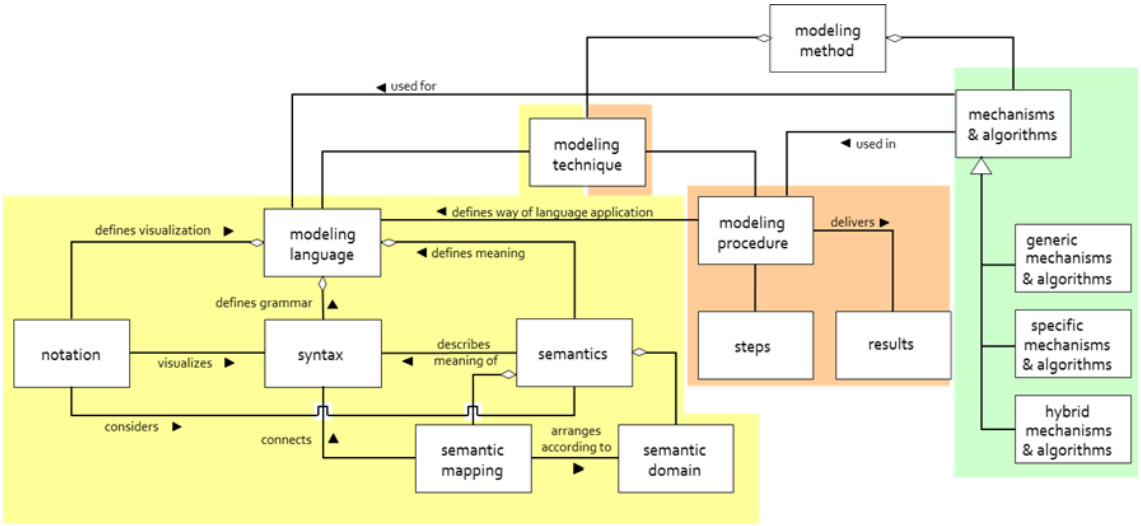


Figure 1: Modeling Method Framework (adapted from [10])

Usually, the graphical modeling language is the primary building block of a modeling method. It is divided in three parts: syntax (also known as abstract syntax), semantics, and notation (also known as concrete syntax). Syntax is the grammar of the language, with a set of predefined rules that need to be obeyed unconditionally. Semantics give meaning to the syntax of a language, defining how the set of constructs (language sentences) is used by the machine (and human). Notation defines the graphical representation of a modeling language. Take note that not all of the constructs introduced in the syntax of a modeling language need to have a graphical representation – these con-

structs are typically addressed as abstract and therefore cannot be instantiated in a modeling space.

Secondary building blocks are modeling algorithms, modeling mechanisms, and modeling procedures.

While a modeling language defines a structure of a modeling method, modeling algorithms define functionality that can be applied to models. Common tasks for which algorithms are constructed are analysis and simulation.

Mechanisms are typically intertwined with specific metamodeling platform functionality, therefore hard to separate and isolate. Simple example of a mechanism is automated model publishing which is essentially a part of the metamodeling platform functionality, but it can be configured to support any modeling method hosted by the platform.

Procedures are typically informal guidelines and steps that need to be taken when developing models with a specific modeling method. For example, in the domain of information security a modeling method can constrain the user by requiring that physical security (e.g., servers containing sensitive data locked in a secure room) is modeled before virtual security (e.g., users' authorization & authentication, firewall, etc.)

1.4 Motivation

While performing a comprehensive research on the existing metamodeling technologies, particularly on the tools used for implementing graphical modeling editors which support concepts like modeling languages and modeling methods, it has been noted that in some cases these tools have been performing insufficiently. In general, understanding how a particular metamodeling technology functions requires considerable time investments. There is no common ground regarding file formats or application programming interfaces (APIs). An exception to this statement, are the tools built upon the Eclipse IDE. However, their functionality is lacking advanced features and most of them require the use of pure Java to implement more demanding concepts.

Once you choose an appropriate tool and start implementing your solution on it, you are typically stuck with that tool until the end. There is no easy way to reuse what has already been implemented in one platform on the other platform. In other words, the development is locked-in to one platform only. This is a major issue, because even after reading through the manuals and looking at the examples, one cannot be certain if the tools under consideration will be sufficient to implement concepts exactly how they were specified, or how long the implementation will take.

However, the real issue is not in the difficulty to learn or in not being platform independent, but in the concepts the platform provides to the developer, which aren't domain-specific at all. The meaning of concepts differs among different metamodeling technologies. Something called *"class"* in one platform can have a different meaning in another platform. Sometimes the difference is not only syntactic (e.g., attribute versus property), but also semantic. For example, concept of a relation can be defined as a

special class containing the information about related objects, but it can also be defined as a part of a class that points to another class. These two concepts have entirely different meanings. First relation (class relation) is a stand-alone object. It can exist even without related classes. Second relation (reference relation) cannot exist without its containing class. The meaning of the underlying concepts influences the creation of metamodels on a particular platform. It also locks us into a way of thinking a metamodeling platform supports.

It is important to understand that by implementing a modeling tool one transfers the concepts of a modeling method from a conceptual space (e.g., design document) to the technical space (e.g., metamodeling platform). We want to make this process as seamless as possible by introducing a language that possesses the concepts familiar to the domain expert, and allows the transfer of the specification of a modeling method to a modeling tool which conforms to the specification.

Nowadays, most of the metamodeling technology, although packed with a lot of functionality, is limited to a set of features connected with the meta-metamodel (meta^{2}model) they provide. Every concept of a modeling language is inferred from that meta^{2}model. Many of these meta^{2}models are specific to the platform and come equipped with different concepts. Modeling method engineers are forced to work with concepts like: atom, paradigm, port, role, graph, attribute, property, stencil, element, set, object, class, etc. Most of them have similar semantics, like property and attribute, or atom and class, but one can never be sure without trying them out first.

Mentioned concepts lack domain-specificity. For example, by looking at the modeling method framework (Figure 1), one can see that every graphical modeling language has a concept called notation. Notation itself is composed of two concepts: graphical representation and user interaction. Graphical representation defines how a modeling element will be visualized in a model realized by a modeling tool. User interaction defines what properties of modeling elements will be exposed to the user, which ones can be modified, which ones are hidden, and how are they represented in the modeling tool. Both of these concepts cannot be directly found in typical metamodeling technologies, like metamodeling platforms. In most cases, graphical representation is described in a dedicated container somewhere in the tool, or in worst cases, there is only a possibility to associate an image with a modeling element. User interaction is, as well, described in a dedicated container, or it cannot be described at all. The way how notation is expressed in a metamodeling technology is not domain-specific. It is closer to the *"how computers see it"*, rather than *"how humans see it"*. With a DSL one can abstract from the implementation specific details, and express the concepts in a domain-specific way.

As mentioned in the previous paragraphs, platform dependency, difficulty to learn, lack of reusability and lack of domain-specificity are the four main motivating factors for this research.

1.5 Research Goals

The overall goal of this dissertation project was to fortify the realization of modeling methods. It emanates from the notion that models created by using modeling tools contain additional value which is greater than only representing knowledge in an acceptable form. They can, as well, help us understand the problem better, or even to analyze and simulate it. Models can be utilized in many different scenarios, from documentation purposes to direct execution.

However, to create good models, one needs great modeling tools, and to implement great modeling tools, one needs appropriate metamodeling technology. After analyzing the state of the art, it has become obvious that metamodeling tools are still not sufficiently efficient in terms of expressivity, understandability, applicability and usability. There is plenty of room for improvement.

The primary focus was on improving metamodeling platforms. They were the most advanced metamodeling technology at the time this research was conducted. The disclosure of fundamental inconveniences, particularly platform dependency, difficulty to learn, lack of reusability and lack of domain-specificity made us consider an alternate idea that has precisely the inverse: it is platform independent, simple to learn, inherently reusable and domain-specific. This is how a domain-specific language called MM-DSL came to existence.

MM-DSL was envisaged as a bridge between modeling method experts (domain experts) and metamodeling platforms. It augments the modeling tool development process with the domain-specific language concepts. Thus, one should have the entire feature packed functionality of a metamodeling platform combined with all the remarkable features of DSLs.

1.6 Main Results

The main contribution of this research is manifold: (1) a review of the different metamodeling technologies, (2) the MM-DSL specification including the grammar, semantics and exemplary utilization of language constructs, (3) a prototype of an integrated-development environment (IDE), (4) a framework for future MM-DSL extensions, and (5) the MM-DSL translator (compiler) targeting a particular metamodeling platform.

In the review of metamodeling technologies, the most known and utilized ones (according to the number of scientific publications and Internet search results) have been picked. Their meta^{2}models have been analyzed, as well as the provided features, with the purpose to discover similarities, differences, strengths and weaknesses. The secondary objective was to establish how easy was to realize a modeling tool by using a simple representative modeling method specifically designed for testing the expressivity and usability of metamodeling tools.

The MM-DSL specification followed after the metamodeling tools have been tested. The results gained by realizing modeling tools on various technologies have influenced

MM-DSL syntax and semantics. MM-DSL is a textual DSL which syntax is formally specified using the Extended Backus-Naur Form (EBNF) notation. Semantics are described in textual form.

IDE prototype has been implemented on top of the Eclipse IDE, which allows for reuse of common features, like auto complete, syntax coloring, code templates, etc. The implementation is done in such a way, so that it is easy to extend it with additional functionality. This is also true for the contained MM-DSL translator.

To be able to evaluate the usability of MM-DSL in real-world scenarios, the MM-DSL translator has been adapted for a particular metamodeling platform. The whole system, including the language, the IDE, and the translator, has been evaluated through several controlled evaluation scenarios with positive results.

Generally speaking, the fundamental results give an alternative method for developing modeling tools. If utilizing a DSL in modeling method engineering is a better way than directly using a metamodeling platform could not be objectively evaluated, because of reliability upon user's preference and level of expertise. With a DSL, developers write code – they program. With a metamodeling platform, developers use the provided functionality – they customize. A user with background in programming will find it simpler and much faster to use a DSL. A user without any programming knowledge (or aversion to programming) might find it difficult at first. Therefore, he might want to customize and use a metamodeling platform, instead of a DSL. The broader conclusion is that both of these approaches take time to get used to. However, evaluation results indicate that learning MM-DSL takes considerably less time than learning to correctly utilize a metamodeling platform, taking into account that the user is familiar with the basic metamodeling concepts.

1.7 Summary

This dissertation is structured in multiple connected parts. It is following a certain flow, starting with the basics, such as the introduction to the concepts that are important for understanding the state of the art, the taxonomy used throughout this document, and an overview of computer languages and language design principles. The rest of the document is dedicated to MM-DSL, including its relationship with metamodeling platforms, its formal specification, utilization and evaluation.

In this chapter (Chapter 1) some of the fundamental concepts have been introduced already, such as domain-specific language and modeling method. These concepts have also been briefly defined. More elaborated definitions can be found in the section dedicated to the taxonomy used in metamodeling research.

Chapter 2 is dedicated to the background relevant to this dissertation project. It includes the description of terminology used in metamodeling research, classification of computer languages (e.g., programming languages, specification languages, etc.) and

presents some of the language specification techniques such as the ones from formal language theory and the ones using a metamodeling approach.

Chapter 3 covers the state of the art and related research, including related concepts and technology. It additionally denotes the current challenges in metamodeling and modeling tool development.

Chapter 4 elaborates the research problem and goals by setting up the research environment and concisely defining the research questions.

Chapter 5 elaborates on the typical system development research methodology. It also describes in detail how is the chosen research methodology utilized.

Chapter 6 is dedicated to metamodeling platforms, and to the functional and non-functional requirements they should fulfill. These requirements played a significant role in the specification of MM-DSL.

Chapter 7 presents a case study where a metamodeling platform has been used for realization of a real-world modeling method. It later serves as a comparison to the modeling tool development approach where MM-DSL is used.

Chapter 8 is dedicated to MM-DSL. It describes the language creation process in detail, including the overall concept, domain analysis, syntax and semantics of a language. The description on how can MM-DSL augment metamodeling platforms is provided as well.

Chapter 9 gives a complete specification of the language. MM-DSL formal syntax, semantics, utilization examples, and programming concepts are discussed.

Chapter 10 describes the design, architecture and implementation of the MM-DSL IDE, its usability, as well as how can one generate platform-specific code from the code written in MM-DSL.

Chapter 11 is a follow-up on the work presented in Chapter 7. In Chapter 7 one can see how to use metamodeling platforms to develop modeling tools. Chapter 11 demonstrates on a representative example how one can use MM-DSL to develop modeling tools.

Chapter 12 presents several evaluation scenarios and detailed evaluation results. MM-DSL is evaluated as a language and as a development environment.

Chapter 13 is the closing chapter of this dissertation. It is composed of the summary containing the most important conclusions, as well as current limitation to this approach for the development of modeling tools. Some of the open issues and future improvements are discussed as well.

2 Background

2.1 Metamodeling Taxonomy

The terminology in the field of applied metamodeling, particularly modeling method engineering (MME), is currently in an unsatisfactory state. Use of homonyms and synonyms is causing confusion among researchers. For example, the term *modeling language* is very often used instead of the term *modeling method*. The term *modeling procedure* is often misunderstood. The difference between modeling algorithms and modeling mechanisms is not clearly stated. There are several synonyms for the technology utilized in MME, such as metamodeling platform, language workbench, metamodeling framework, or metamodeling toolkit).

This section explicitly defines the meaning of terms used in this dissertation, which also includes some of the terms used in the field of software engineering. It does not provide comprehensive explanation for every term. These will be provided in dedicated sections of this work.

2.1.1 Modeling Method

By including functionality in a form of modeling procedures, modeling algorithms and modeling mechanism to the modeling language, modeling methods are created. Because of this supplemental functionality, modeling methods need to be realized as modeling tools. Otherwise, their utilization would be limited.

2.1.2 Modeling Language

A modeling language is the essential part of each modeling method. It is defined by its notation, syntax and semantics. The language's notation (or concrete syntax) can be either textual, graphical or both. The (abstract) syntax is defined by a set of rules that need to be followed unconditionally. Semantics can be defined formally, semi-formally, or informally depending on the modeling language's application area. Formal semantics are helpful for execution languages, because program execution entails compilation, and writing a compiler is much easier if one understands unambiguously the meaning of language constructs. In case of modeling languages which purpose is limited to expressing information and sharing knowledge, semantics can be given in pure textual descriptive form.

2.1.3 Modeling Procedure

Modeling procedure is an important part of a modeling method. It provides steps required to be taken to achieve predefined results. For example, it recommends modeling

of *diagram A* before *diagram B*. By doings so, a modeling tool that realizes a modeling method can generate *diagram B* automatically, which would not be the case if we created *diagram B* first. Creating *diagram B* first is still a perfectly valid utilization of a modeling method. However, it is not a recommended way of doing things.

2.1.4 Modeling Algorithm

In mathematics, as well as in computer science, an algorithm is a set of rules for solving a problem in finite number of steps. It is a step-by-step procedure with results that need to be precise. A modeling algorithm conforms to the definition of an algorithm, with an addition – it is designed to solve problems in a specific domain. Most of the modeling algorithms can be grouped into two broad categories: analysis and simulation. Additionally, there are modeling algorithms responsible for the placement of objects on the modeling canvas, which are particularly important for graphical modeling languages. The rest of modeling algorithms can be placed into the custom algorithm category. Examples of custom algorithms include automatic object generation and on demand notation change.

2.1.5 Modeling Mechanism

A mechanism is a relaxed algorithm. It possesses most of the properties an algorithm has, except it does not have to solve a problem in finite number of steps and the result does not have to be precise. It can be viewed as an imperfect algorithm or even as a simplified algorithm template. The transition from mechanism to algorithm may happen if all exceptions a mechanism possesses are eliminated. A mechanism may be composed of several elements, typically other mechanisms or algorithms and its main purpose is to achieve an intended result. An example of a modeling mechanism is an export feature that is intended to produce bitmap images from models. The duration of the process is not specified and may vary. The same is true for the result, which will be an image, but it may be a slightly different image each time the export feature is utilized on the same model.

2.1.6 Software Engineering

For the purpose of this work the IEEE definition for software engineering is used [11]:

> *"[Software Engineering is] the systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software"*

2.1.7 Modeling Method Engineering

Modeling method engineering (MME) is an engineering discipline that applies software engineering and metamodeling techniques on the development of modeling methods and realization of tools supporting them. It can be roughly divided into two parts: (1)

conceptualization of modeling methods, and (2) realization of modeling tools. The first part can be considered as a design process, where one specifies a modeling method according to the requirements gathered from an application domain. The second part is concentrated on the tooling of a modeling method where engineers employ metamodeling technology to achieve appropriate results. The final product is software which supports the conceptualized modeling method.

2.1.8 Meta-

In the metamodeling research once can notice the rich usage of the prefix meta-. Sometimes even two or three times in a sequence: meta-meta-, meta-meta-meta-. Multiple occurrences of meta- can be abbreviated as follows: meta-meta- as meta^2, meta-meta-meta- as meta^3, etc. Using this rule, instead of writing meta-meta-model, one can write meta^{2}model. The satisfactory definition for meta- taken from [11] is:

> *"[meta- is] a prefix to a concept to imply definition information about the concept."*

2.1.9 Metalanguage

The concept metalanguage is very important for this work. It's meaning is [11]:

> *"a language used to specify some or all aspects of a language"*

In this work we use Extended Backus-Naur Form (EBNF) to specify a DSL. Later on that DSL is used to specify a modeling method. From the modeling method point of view, the DSL is a metalanguage, and EBNF is a meta^{2}language. From the DSL point of view, EBNF is a metalanguage.

2.1.10 Model and Metamodel

According to [5] models are the unifying concept in IT engineering and can come in various flavors: an UML model, a Java or C# program (practically any kind of program written in any kind of programming language), an XML or RDF document, etc. In [5] and [12] they are formally defined using the graph and set theory as a labeled directed multigraph with special properties. This work does not focus on issues of formal model and metamodel definition. Thus, it is enough to understand that [12]:

> *"A model is an abstraction of a (real or language based) system allowing prediction or inference to be made"*

A metamodel can be defined as a model of models, meaning that a model is an instance of a metamodel. OMG in their MOF specification define a metamodel as [13]:

> *"… a model used to model modeling itself"*

For example, Java specification (at least the Java grammar) is a metamodel of a Java program, XML schema is a metamodel of an XML document, and UML (the language)

is a metamodel of an UML model. EBNF is then a meta^{2}model of a Java program, and MOF is the meta^{2}model of an UML model.

2.1.11 Metamodeling

Metamodeling, particularly in software and system engineering, as well as in modeling method engineering, is the analysis, construction and development of the modeling languages and modeling methods we use to model with. It is a layered approach to the design process, where one has at least two layers. The maximum number of layers is unlimited, but in practice it is typically not bigger than four [13]. Two layers are mandatory, because we want to represent and navigate from meta-concept (class) to its concept (instance) and vice versa. The metamodeling approach is used to design modeling methods and realize modeling tools supporting them.

2.1.12 Metamodeling Platform

Metamodeling platform is a technology that supports a metamodeling approach. The central part of a metamodeling platform is an underlying meta^{2}model. The platform's functionality is built around it. Various modeling method development tools, like (abstract) syntax designer, notation (concrete syntax) designer, and additional API libraries are all a part of the platform's functionality. There exist many variations on the metamodeling platform concept: language workbench, metamodeling toolkit, metamodeling framework, etc. Typically, these are only synonyms for the same technology. In this work this kind of metamodeling technology is addressed simply as metamodeling platform.

2.1.13 Domain-Specific Language

A domain-specific language (DSL) is a computer language particularly targeting a specific application domain. It is very good at expressing solutions to problems in that specific domain. Its concrete syntax can be either textual, or graphical or both. It can be executable, but it doesn't have to! It can be Turing complete, but it doesn't have to (and typically it is not)! It can be independent (standalone language) or embedded into a host language. The size of a language does not define its domain specificity. For example, SQL is a huge DSL, with over 800 reserved words. Inside the scope of this dissertation, if languages like markup language, modeling language, specification language, scripting language, or programming language contain the above mentioned properties, we consider them to be domain-specific.

2.1.14 Translator and Compiler

A term translator is used to describe *"a computer program that transforms a sequence of statements expressed in one language into an equivalent sequence of statements expressed in another language"* [11]. It is not specified if the translation is made from a

higher-order language to a lower-order language (e.g., from C++ to assembler). Both source and destination language may be of the same order.

A compiler is a special instance of a translator. It translates programs (models) expressed in a high-order language into their machine language equivalents [11].

2.2 Types of Computer Languages

In computer science one distinguishes many classifications of computer languages. Generally, the term computer language and programming language are used as synonyms. Thus, it is common to classify computer languages into low level, high level and everything in between. Machine language, which is a pure binary code, and assembly language, which uses mnemonics to substitute binary code, are considered low level computer languages. Examples of high level languages are C++ and Prolog. In this classification a concept of generation is used. It indicates how close the computer language is to the natural language. Currently, there are four generations, machine language being first, assembly language second, C++ (and equivalent) being third, and SQL (and other domain-specific languages) being fourth.

Level- and generation-based classification can be applied on programming languages, but not on all kinds of existing computer languages. In this work the term programming language is not the same as the term computer language. Computer language is *"a language designed to enable humans to communicate with computers"* [11]. Programming language is then only one of computer language subcategories, as well as specification language, modeling language, etc. Therefore, a different classification is used – the one that distinguishes between language's notation, purpose and generality. A language can belong to multiple categories at the same time. For example, a domain-specific graphical modeling language, would be a DSL with graphical notation as well as a modeling language. See Figure 2 for other possible combinations.

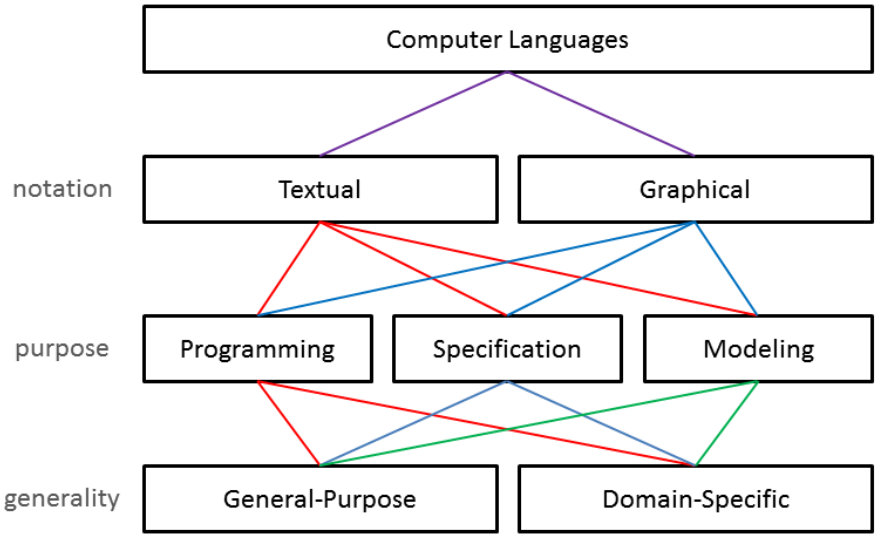


Figure 2: Classification of Computer Languages

The most interesting categories in the context of this work are: (1) textual, (2) graphical, (3) general-purpose, (4) domain-specific, (5) programing, (6) specification, and (7) modeling languages.

2.2.1 Textual vs. Graphical

According to the notation a computer language possesses, it can be classified either as textual, or as graphical. In case the language uses both textual and graphical notation, it can be classified as a language with a hybrid notation.

Computer languages with a textual notation, or shortly textual languages, are the most common class of computer languages. Describing programs (or models) is done by chaining characters into strings, and strings into statements. Keywords – reserved words with a special meaning – are the core part of textual language's notation. Every textual language has keywords.

Most of programming languages fall into this category: C, C++, C#, Java, Prolog, COBOL, Pascal, Python, etc. Textual specification languages are not known as widely as textual programming languages, but they do exist. For example, CASL, VDM, Z notation, Spec#, etc. As for textual modeling languages, there are many present in the Eclipse community. Typically, they are constructed around EMF and influenced by the OMG standards like MOF and UML.

Defining a textual computer language is relatively straight forward. One needs to define the grammar of a language. Grammar represents the language's abstract and concrete syntax. Nowadays, EBNF is the most common way of defining the language's context-free grammar. A parser can be automatically generated from an EBNF-like grammar definition using the tools such as the legendary LEX and YACC, or the modern Eclipse-based framework Xtext. Defining semantics of a textual language can be accomplished by using one of the mature formal methods, like operational semantics, denotational semantics, and axiomatic semantics, or by utilizing plain text (prose) to describe the meaning of language's constructs. Using one approach does not exclude the other. If a language is meant to be executable, its syntax and semantics should be formally expressed. Without formal syntax it is not possible to correctly parse programs. If semantics are not defined in detail, it may happen that the translators, particularly compilers developed for the language in question are not fully conform to the language's specification.

Graphical computer languages contain a graphical notation. Their concrete syntax is not build from keywords and special characters. It is represented by various graphical symbols. Programs in graphical languages are represented as graphs, typically as labeled attributed multigraphs.

Main representatives of graphical computer languages are graphical specification languages and graphical modeling languages. Visual programming languages are also a subcategory of graphical languages. Every programming language that lets users cre-

ate programs by manipulating program elements graphically, rather than by specifying them textually, is considered to be a visual programming language.

Distinction between graphical specification languages and graphical modeling languages is vague. In practice, they are treated as synonyms. The difference, however minor it is, does exist. It is in the formality of a language, as well as its application.

A specification language is a formal computer language. It is used during systems analysis, requirements analysis and systems design to express desired properties about the software to be built. The main focus on what the software should do, without stating how to do it. An important use of specification languages is enabling the creation of proofs of program correctness. A common fundamental assumption is that programs are modelled as algebraic or model-theoretic (using mathematical structures such as groups, fields, graphs, universes of set theory) structures. This level of abstraction is equal with the view that the correctness of the input/output behavior of a program takes precedence over all its other properties. Nowadays, among other application scenarios, they are used in model driven engineering (MDE), for example to specify model to model transformations [14].

A typical modeling language, like UML, is semi-formal, meaning that it has a well-defined (formal) syntax, but not unambiguous and precise semantics. The semantics of UML are provided in prose, which is inherently ambiguous. As a result, models created with a semi-formal modeling language can be read by a computer, but due to the lack of formalized semantics, a computer cannot compute them in a deterministic way. UML is only one example. In practice there are many domain-specific graphical modeling languages, which do not have formally defined semantics. Some of them do not have formally defined syntax as well. This can be seen in various implementations of the same modeling language. Graphical modeling languages are commonly used to express information, knowledge, or systems by using a diagraming technique with named symbols that represent concepts, and lines that connect the symbols and represent relationships among concepts. Graphical notation is used to express constraints as well.

2.2.2 General-Purpose vs. Domain-Specific

A general-purpose language (GPL) is a computer language that is broadly applicable across multiple application domains, and lacks specialized features for a particular domain. We can divide them into multiple categories, such as general purpose programming languages (GPPL), general purpose markup languages (GPMaL), and general purpose modeling languages (GPMoL).

The main property of a GPPL is that it is computationally universal or Turing complete. However, the languages that belong to the Turing tar-pit are not considered to be general purpose languages. Turing tar-pits are characterized by having a simple abstract machine which requires that the programmer deals with many completely unnecessary

details of the solution, because only the minimum functionality necessary to classify the language as Turing complete is provided by the language. There is a saying [15]:

> *"Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy."*

In other words, a language has to be useful as well.

General purpose markup language is a category of document annotation languages that can be used in various domains. Leading example for this category is the Standard Generalized Markup Language (SGML), which is an ISO standard based on the following postulates: (1) it should be declarative, and (2) it should be rigorous. Declarative means that the language should describe a document's structure, rather than specify the processing to be performed on it. Rigorous means that the techniques available for processing rigorously-defined objects such as programs and databases can be used for processing of documents as well. GPMaLs are also used as a base for implementing domain-specific markup languages like HTML. The second well-known example for GPMaL is XML, a markup language that defines a set of rules for encoding documents in a format that is readably both by a machine and a human (although, complex XML representations tend to become unreadable to humans). XML, just like HTML is extended from SGML, but its usage is applicable to multiple application domains.

General purpose modeling language is a category of modeling languages which are general enough to represent various facets of an object or a system. UML, with its 13 distinct diagrams is an example of a GPMoL. Nowadays, in the programming community, XML as a data modeling language for code modeling is also considered as GPMoL.

On the opposite side, we have domain-specific languages (DSLs), which are explicitly designed for the use in a one well-define application domain. They are typically not Turing complete, but they provide more expressivity in their dedicated domain. Describing and solving problems in a DSL for a particular domain is much simpler than using a GPL. Additionally, DSLs are supported by various tools that are in most cases inseparable from the DSL itself. SQL code without its relational database management system (RDBMS) is just a structured text. But with RDBMS it becomes a powerful language for database querying. DOT, which is a graph description language, is only useful together with Graphviz – a tool for graph visualization. MATLAB is considered to be a numerical computing environment and a DSL at the same time. These are only well-known examples. General rule is that if a DSL is meant to be interpreted, compiled or utilized to generate a different representation (e.g., image, sound, code, etc.) it needs to have software support.

Analogue to GPLs, DSLs can also be categorized into different categories, such as domain-specific programming language (DSPL), and domain-specific modeling language (DSML). DSPLs are executable and typically have textual notation. DSMLs are not executable, but one can generate code templates from their models. DSMLs are graphical modeling languages.

2.2.3 Programming vs. Specification Language

The major difference between programming and specification languages is in the way how they describe computer systems.

Specification languages express desired properties about a system. They abstract from the implementation details and describe a system at a much higher level than a programming language. Specification languages are not directly executable. However, there are ways to generate programming language code templates from the system's specification.

Programing languages express computer programs in a machine understandable way – they communicate instructions to the machine. Therefore, they implement a system. This involves a computer performing some kind of computation. The programming language statements can as well be used to control all or some of the hardware components of a computer system, such as printers, scanners, disk drives, etc.

2.2.4 Graphical Modeling Languages

Modeling languages are used to model information, knowledge or systems. They can be textual, graphical, or both. Their usage has grown significantly in the last couple of years, as well as their number.

Current trend of creation and utilization of modeling languages is connected with the emerging practices in the software development, where the inclusion of a customer in the development process has become a common practice. The customers, even the managers, are not required to be familiar with all the details of software engineering, like knowing the technologies and languages used in the development, or any other aspects of computer science. Semi-formal graphical modeling languages have proven to be very helpful to communicate ideas in a simplified form between computer engineers, managers, and customers.

At the same time, the systems are becoming more complex. Modeling languages are a mean to cope with this increasing complexity, by allowing the construction of conceptual models on a higher level of abstraction. An overview of a whole system can be shown without the unnecessary low level details. By using conceptual models to capture the fundamentals, overall understanding of a system can be increased.

Well-known languages like Entity Relationship Model (ER), Data Flow Diagram (DFD), Structured Analysis and Design Technique (SADT), State Diagram, and Unified Modeling Language (UML) are all examples of graphical modeling languages. UML is a special example, because it is a general-purpose modeling language which integrates various modeling languages as a part of its many diagrams. For example, a modified State Diagram is an integral part of UML.

2.2.4.1 UML

This section describes the fundamentals of UML according to the OMG's UML specification [16]. Modeling method engineering as a discipline is reusing the practices from software engineering in the process of modeling tool development. Therefore, UML is a valuable tool, not only in software engineering but in modeling method engineering as well. Class diagram and its variations are commonly used to represent meta²models and metamodels of modeling languages.

Because UML is a well-known and dominant general-purpose graphical modeling language for modeling business and similar processes, analysis, design, and implementation of software-based systems, it is presented here as an example to illustrate the differences between general-purpose and domain-specific graphical modeling languages.

UML is composed of multiple diagrams. Each of them graphically represents a different view of a system's model. In the current specification there are fourteen types of diagrams divided into two categories: structural diagrams and behavioral diagrams (see Figure 3). Structural diagrams point out the things that must be present in the system being modeled. Together they form the architecture of a system. Behavioral diagrams represent what must happen in the system being modeled. Together they form the functionality of a system.

Structural UML diagrams are: (1) class diagram, (2) component diagram, (3) composite structure diagram, (4) deployment diagram, (5) object diagram, (6) package diagram, and (7) profile diagram.

Behavioral UML diagrams are: (1) activity diagram, (2) communication diagram, (3) interaction overview diagram, (4) sequence diagram, (5) state diagram, (6) timing diagram, and (7) use case diagram.

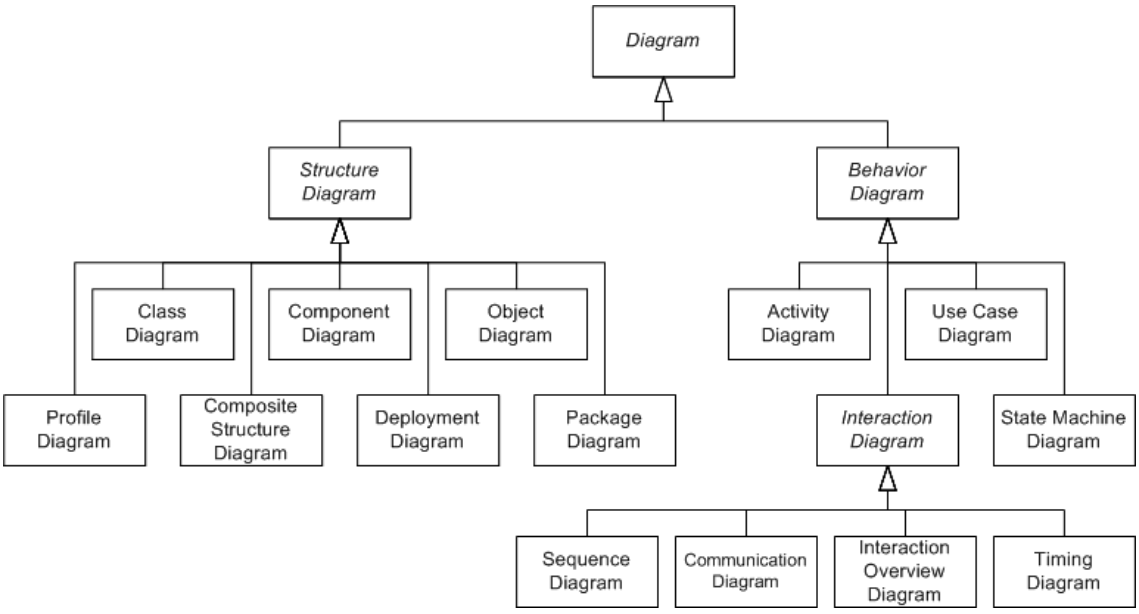


Figure 3: UML Diagram Hierarchy (adapted from [16] and [17])

According to the UML specification, communication diagram, interaction overview diagram, sequence diagram, and timing diagram belong to a subset of behavior diagrams called interaction diagrams. They emphasize the flow of control and data among the elements in the system being modeled.

UML element types are not restricted to a certain diagram type. It is common practice to reuse element types in multiple diagram types. Because of it, one can note similarities between various structural diagrams, as well as between various behavioral diagrams. For additional flexibility, UML provides a possibility to extend diagram types, or even to create new diagram types. These new additions to the UML core are addressed as UML profiles.

Class diagrams are probably the most known diagram type in UML. They show the classes of the system, their inter-relationships, their attributes and operations (also known as methods). Class diagrams are typically used to: (1) explore domain concepts in the form of a domain model, (2) analyze requirements in the form of a conceptual model, and (3) depict the detailed design of object-oriented or object-based software [18].

Component diagrams describe how a system is split up into components and shows dependencies among these components. They are used to illustrate the structure of a complex system on a higher level than class diagrams.

Composite structure diagrams show the internal structure of a class and the collaborations that this structure makes possible. Diagrams are composed of internal parts, ports through which the parts interact with each other or through which instances of the class interact with the parts and with outside world, and connectors between parts and ports.

Deployment diagrams models the physical deployment of artifacts on nodes. A single node in the diagram may conceptually represent multiple physical nodes, like clusters or database servers. In other words, deployment diagrams show the execution architecture of a system, including the hardware of a system, the software that is installed on that hardware, and middleware used to connect the disparate machines to one another [18].

Object diagrams depict objects and their relationships at a specific point in time. They represent instances of a class diagram. Therefore, they are often used to provide examples or act as test cases for class diagrams. Because an object diagram is a more concrete representation than a class diagram, it needs to include enough information so that it is a recognizable instance [18].

Package diagrams shows how model elements are organized into packages, as well as the dependencies between packages. Consequently, they are making UML diagrams simpler and easier to understand. Packages group semantically related elements and can be used on any of the UML diagrams. Typically, they are applied on use case and class diagrams (these have the tendency to grow quite large), as a mechanism to help keep system dependencies under control.

Profile diagrams work at a metamodel level and describe a lightweight extension mechanism to the UML by defining stereotypes (meta-classes), tagged values (meta-attributes), and constraints [19]. This mechanism is not a first-class extension mechanism. It does not allow to modify existing metamodels or to create new metamodels. Profile only allows adaption or customization of an existing metamodel with constructs that are specific to a particular domain, platform, or method.

Activity diagrams show flow of control or object flow with emphasis on the sequence and conditions of the flow [19]. Because they are a graphical representation of workflows of stepwise activities and actions, they are typically used for business process modeling, for modeling the logic captured by a single use case scenario, or for modeling the detailed logic of a business rule [18].

Communication diagrams (called collaboration diagrams in UML 1.x) shows interaction between objects and/or parts using sequenced messages [19]. They represent a combination of information taken from class, sequence and use case diagrams describing both the static structure and dynamic behavior of a system.

Interaction overview diagrams provide overview of the flow of control where nodes of the flow are interactions or interaction fragments. The current UML specification refers to these diagrams as interaction diagrams in some places, while in other places interaction overview diagrams are referred to as specialization of activity diagrams. The nodes within the diagrams are frames instead of the normal activities one can see on activity diagrams.

Sequence diagrams are probably the most popular interaction diagrams. They focus on the message interchange between lifelines. A lifeline represents an individual participant in the interaction. Sequence diagrams describe an interaction by focusing on the sequence of messages that are exchanges, along with their corresponding occurrence specifications in the lifelines [19].

State diagrams or state machine diagrams are an enchanted realization of the mathematical concept of finite automaton in computer science applications. They show discrete behavior of a part of a system through finite state transitions. This behavior is modeled as a traversal of a graph of state nodes connected with transitions. Transitions are triggered by the dispatching of series of events. During the traversal, the state machine can also execute some activities [19].

Timing diagrams are used to express the behaviors of one or more objects throughout a given period of time. They focus on conditions changing within and among lifelines along a linear time axis. Such diagrams have found their use in the design of embedded software systems, such as control software for fuel injection system in an automobile [18].

Use case diagrams present an overview of the usage requirements for a system [18]. They are behavioral diagrams that describe a set of actions (use cases) that a system should or can perform in collaboration with one or more external users of the system

(actors). Each use case should provide some observable and valuable result to the actors or other stakeholders of the system [19].

From this very brief summary of all currently available diagrams, it can be concluded that UML is a huge language, with a lot of interconnected elements and various constraints and rules. Consequently, even the specification documents are sometimes incomplete and/or incomprehensible (depending on the specific version, and there are many).

It may sound strange, especially for such a large language, that one of the biggest disadvantages of UML is that it doesn't cover all the aspects of system development. UML coverage in development of software systems is limited by its object-oriented modeling approach. A proof that UML alone cannot cover system engineering by itself is that OMG has specified another language just for that – the System Modeling Language (SysML). System modeling is only one example of a domain UML cannot model successfully. There are many other domains that are difficult to model with UML as well.

Some of other disadvantages of using UML are:

- Complexity. UML is a very complex language, which makes it a hard language to learn. Most of the developers tend to learn only the parts they need. Using every single element of UML is very rare in typical software system designs.

- Large diagrams. Diagrams increase in size during the development and can become overwhelming and hard to manage.

The mentioned limitations and disadvantages of UML can be solved with the introduction of domain-specific (graphical) modeling languages.

2.2.4.2 Domain-Specific Graphical Modeling Languages

Domain-specific (graphical) modeling languages, or shortly DSMLs, are an essential part of a certain software engineering practice called domain-specific modeling (DSM), where domain-specific languages are utilized as means to represent various facets of a (software) system. In this discipline one creates a language specifically targeting an aspect of a system, and afterwards uses this language to describe that aspect of a system. There are myriads of DSMLs today. Most of them do not have a dedicated name, making it very hard to list them as examples, even if we do know they exist. Probability that we do not know a particular DSML exist is considerably higher than vice versa. There are some known languages we can put into this category, like ER, which is used for describing a database in an abstract way, or state diagrams, which is used to model the behavior of a system. These languages have a historical value, and are very different than today's DSMLs. Nowadays it is common to create languages that describe refrigerators, biological neurons, web services, graphs, document layouts, user interfaces, and many other domains.

One of the important ideas in DSM is the generation of executable source code directly from the models. Consequently, DSMLs significantly increase developer productivity

[20]. Generation of source code is not possible from models modeled in a general-purpose modeling language like UML, because of its complexity and semi-formal nature. However, it is possible to create code templates which must be extended with additional code in order to become executable.

To enable the creation of DSMLs and afterwards the generation of output artifacts (it doesn't have to be executable source code; output can be in form of images, other language code, tool configuration templates, etc.) one needs to have some kind of a tool support. Building a development environment from scratch is almost never a feasible solution, because most of domain-specific (modeling) languages have a small, typically closed, community. The other reason is that during the system development, many DSLs are created. Having dedicated development environments for each and every of them is pointless and an additional work for the developers, which takes more time than the creation of a DSL itself.

This is why domain-specific modeling language environments are created. A DSML environment is a metamodeling tool, meaning that it is a modeling tool used to define a modeling tool. In this dissertation, these tools are addressed as metamodeling platforms. In other communities they are known as language workbenches [21] or meta-CASE (computer-aided software engineering) tools [22]. Metamodeling platforms are considered to be stand-alone metamodeling tools. The popularity of metamodeling approaches has led to the development and addition of DSML frameworks to very popular IDEs as well. Eclipse IDE has EMF and GMF, and many others in development. Visual Studio IDE has Visualization and Modeling SDK (formerly known as DSL Tools).

Today, the most know metamodeling platforms are: MetaEdit+, GME, and ADOxx. These tools, together with the most popular frameworks will be covered in a separate section. It is important to understand how this technology works, its advantages and disadvantages. Metamodeling platforms and frameworks are a foundation on which this dissertation builds upon.

2.3 Language Specification Techniques

This section is dedicated to the existing artificial language specification techniques, which have become quite relevant for the creation of artificial languages that had originated in computer science. Some of these established techniques are used in the specification of MM-DSL, which is a domain-specific language for describing modeling methods. This specification is provided and thoroughly discussed in a dedicated chapter.

There are two distinguishable approaches used for specification of languages. The first one is the formal language theory, particularly a set of formalisms called formal grammars, which are used to specify formal textual languages, for example textual programming languages, textual specification languages, or textual DSLs. The other technique for the specification of languages is addressed as metamodeling approach, and it is used for the specification of metamodels that describe a language. This approach is

typically utilized to specify an abstract syntax of graphical modeling languages. However, it can also be applied to specify textual languages.

2.3.1 Formal Language Theory

Formal language in this context means a set of strings of symbols which are constrained by rules. Therefore, one can only describe textual languages. Graphical languages have additional elements, which cannot be described only using the formal language theory. A formal language is often defined by means of a formal grammar, such as regular grammar or context-free grammar. Regular grammars describe regular languages, for example regular expressions. Context-free grammars describe the structure of sentences and words of a formal language. It is a collection of production rules, which are made of terminal and non-terminal symbols. Context-free grammars which are extended with parameters or attributes are called attributed grammars. Formal rules on how to extend the context-free grammar are described by attribute grammars. Attributes associated with the rules are usually used to express static semantics of a language.

This section will not go into detail on all possible type of grammars. It will present a brief summary and focus only on those important for the definition of textual languages. Mainly context-free grammars and attributed grammars. Additional information about artificial language linguistics, as well as on the construction of language grammars can be found in the following works of Noam Chomsky, the *"father of modern linguistics"*: [23], [24], and [25]. A brief summary can also be found in [26].

2.3.1.1 Formal Grammar

A formal grammar is a quadruple $G = (T, N, S, P)$, where:

- P is a finite set of production rules. Each production rule has the following form: $LHS \rightarrow RHS$, where LHS indicates left-hand side and RHS indicates right-hand side of a rule. Each side consists of a sequence of nonterminal and terminal symbols.

- N is a finite set of nonterminal symbols. A nonterminal symbol indicates that a production rule can be applied.

- T is a finite set of terminal symbols. A terminal symbol indicates that no production rule can be applied.

- S is a distinguished start symbol. It is a special nonterminal symbol, which can only be found on the LHS of one or more production rules.

A formal grammar defines a formal language, which is usually an infinite set of finite-length sequences of symbols. The language is constructed by applying production rules to another set of symbols which initially contains just the start symbol. A rule may be applied to a sequence of symbols by replacing the occurrence of the symbol on the

LHS of the production rule with those that appear on the RHS. A formal language defined by such a grammar consists only of terminal symbols which can be reached by a derivation from the start symbol. A derivation is a sequence of rule applications to the sequence of symbols.

The following is a toy example of a formal language described by its grammar.

Let $G = (\{0,1\},\{S,T\},S,P)$, where P contains the following productions:

1. $S \rightarrow 0T$
2. $S \rightarrow 01$
3. $T \rightarrow S1$

This grammar describes a language with which we can construct the strings of symbols such as (1) *01*, (2) *0011*, or (3) *000111*, because:

1. $S \rightarrow 01$
2. $S \rightarrow 0T \rightarrow 0S1 \rightarrow 0011$
3. $S \rightarrow 0T \rightarrow 0S1 \rightarrow 00T1 \rightarrow 00S11 \rightarrow 000111$

We can say that this grammar describes the set $\{0^n1^n | n \geq 1\}$.

Let us consider one more example. This time we want to be able to describe English sentences. The terminal symbols in this case are English words. The nonterminals correspond to the structural components in an English sentence, such as *<sentence>*, *<subject>*, *<predicate>*, *<noun>*, *<verb>*, *<article>*, and so on. The start symbol is *<sentence>*. Some of the production rules would look like this:

1. $< sentence > \rightarrow < subject >< predicate >$
2. $< subject > \rightarrow < noun >$
3. $< predicate > \rightarrow < verb >< article >< noun >$
4. $< noun > \rightarrow Niksa$
5. $< noun > \rightarrow language$
6. $< verb > \rightarrow created$
7. $< article > \rightarrow a$

The rule 1 expresses that a sentence can consist of a subject phrase and a predicate phrase. The rule 2 expresses that a subject consists of a noun. The rule 3 expresses that a predicate phase consists of a verb, article and a noun. The rules 4 and 5 mean that both *Niksa* and *language* are possible nouns (they are also terminals). This approach to grammar was introduced by Chomsky in [23].

This grammar can derive sentences like *Niksa created a language*, or *language created a language*, because:

i. $< sentence > \rightarrow < subject >< predicate >$
ii. $< sentence > \rightarrow < noun >< predicate >$
iii. $< sentence > \rightarrow < noun >< verb >< article >< noun >$
iv. $< sentence > \rightarrow Niksa\ < verb >< article >< noun >$

v. $< sentence > \rightarrow Niksa\ created\ < article >< noun >$
vi. $< sentence > \rightarrow Niksa\ created\ a\ < noun >$
vii. $< sentence > \rightarrow Niksa\ created\ a\ language$

In the same way we can derive many sentences which conform to the grammar rules, by introducing other nouns, verbs, and articles to the grammar.

2.3.1.2 Hierarchy of Grammars

According to Chomsky, grammars can be divided into four classes by gradually increasing the restrictions on the form of the productions [25][24]. This is why this hierarchy is also known as Chomsky hierarchy.

Let $G = (T, N, S, P)$ be a grammar, then [26]:

1. G is also called a *Type-0* grammar or an unrestricted grammar.

2. G is a *Type-1* or context-sensitive grammar if each production $A \rightarrow B$ in P satisfies $|A| \leq |B|$. We also allow a *Type-1* grammar to have the production $S \rightarrow \varepsilon$ (ε denotes an empty string), provided S does not appear on the right-hand side of any production.

3. G is a *Type-2* or context-free grammar if each production $A \rightarrow B$ in P satisfies $|A| = 1$, that is A is a single nonterminal. B is a string of terminals and/or nonterminals, and it can be empty (ε).

4. G is a *Type-3* or regular or right-linear grammar if each production has one of the following three forms: $A \rightarrow cB$, $A \rightarrow c$, $A \rightarrow \varepsilon$, where A, B are nonterminals (with $A = B$ allowed), and c is a terminal.

The language generated by a *Type-i* grammar is called a *Type-i* language, where $i = 0, 1, 2, 3$. A *Type-1* language is also called a context-sensitive language (CSL), and a *Type-2* language is also called context-free language (CFL). A *Type-3* language is also called a regular language.

The four classes of languages in the Chomsky hierarchy have also been completely characterized in terms of Turing machines and natural restrictions on them [26].

- *Type-0* languages equal the class of languages accepted by Turing machines.

- *Type-1* languages equal the class of languages accepted by linear bounded automata.

- *Type-2* languages equal the class of languages accepted by pushdown automata.

- *Type-3* languages equal the class of languages accepted by finite automata.

A linear bounded automata is a possibly-nondeterministic Turing machine that on any input x uses only the cells initially occupied by x, except for one visit to the blank cell immediately to the right of x (which is the initially-scanned cell if $x = \varepsilon$). Pushdown automata may also be nondeterministic.

2.3.1.3 Context-Free Grammar

Context-free grammars are an essential formalism for describing the structure of programs in a textual language. In principle the grammar of a language describes the syntactic structure only, but since the semantics of a language are defined in terms of the syntax, the grammar is also instrumental in the definition of the semantics [27]. Parsing of computer programs, where it is determined if a given string belongs to the language described by the grammar, usually brings out the meaning of the string [26]. The meaning is associated with how the program should be executed. In context-free grammars derivation is represented as a parse tree or derivation tree. The start symbol is the root of a parse tree. Every leaf corresponds to a terminal (or to ε), and every internal node corresponds to a nonterminal. For example, if A is an internal node with children B, C, and D, then $A \rightarrow BCD$ must be a production. The concatenation of all leaves from left to right yields the string being derived [26].

It has already been mentioned that a context-free grammar rules are always in the $A \rightarrow B$ form, where A is a single nonterminal, and B is a string of terminals and/or nonterminals which can also be empty. There is also a special context-free grammar form called *Chomsky normal form*, in which every rule has the following structure: $A \rightarrow BC$ or $A \rightarrow \alpha$, where A, B, C are nonterminals and α is a terminal. B and C may not be the start symbol. Additionally, if a language contains an empty string, we allow $S \rightarrow \varepsilon$ where S is the start symbol, which must never be on the right-hand side of a production. Every context-free grammar can be transformed in the Chomsky normal form.

The following example shows the differences between context-free grammar and Chomsky normal form. More information about the transformation between context-free grammar to Chomsky normal form can be found in [26] and [28].

Consider the following context-free grammar:

1. $S \rightarrow AbA$
2. $A \rightarrow Aa \mid \varepsilon$

Chomsky normal form from this grammar is:

1. $S \rightarrow TA \mid BA \mid AB \mid b$
2. $A \rightarrow AC \mid a$
3. $T \rightarrow AB$
4. $B \rightarrow b$
5. $C \rightarrow a$

Full transformation steps can be found in [29] and [30]. The “|” symbol indicates “or” relationships, meaning that we can write $A \rightarrow AC \mid a$ like $A \rightarrow AC$, $A \rightarrow a$.

Both forms of this grammar accept the same set of strings. For example, we can prove this on *"aabaa"* by deriving it from the context-free grammar:

i. $S \rightarrow AbA$
ii. $S \rightarrow AabA$
iii. $S \rightarrow AabAa$
iv. $S \rightarrow AaabAa$
v. $S \rightarrow AaabAaa$
vi. $S \rightarrow aabAaa$
vii. $S \rightarrow aabaa$

The same string can be derived from the Chomsky normal form:

i. $S \rightarrow TA$
ii. $S \rightarrow ABA$
iii. $S \rightarrow ACBA$
iv. $S \rightarrow ACBAC$
v. $S \rightarrow aCBAC$
vi. $S \rightarrow aaBAC$
vii. $S \rightarrow aabAC$
viii. $S \rightarrow aabaC$
ix. $S \rightarrow aabaa$

In the examples only capital letter were used to represent nonterminals, and small letters to represent terminals, but in general, all kind of letters, words, or constructs can be used to represent terminal and nonterminals. As long as they are assigned to the set of terminals T and to the set of nonterminals V.

2.3.1.4 Extended Backus-Naur Form

Extended Backus-Naur Form or EBNF is a syntactic metalanguage used to express context-free grammars. First application of BNF (Backus-Naur Form) was for the definition of the programming language Algol 60 by Peter Naur in 1960 [31]. Since then BNF has been extended or slightly altered many times. In 1996 it has become an ISO standard and is commonly used to formally define all kinds of textual languages. The ISO EBNF is based on the BNF and includes the most widely adopted extensions. Full specification can be found in [32]. Slightly different EBNF specification which is also very widely used belongs to the World Wide Web Consortium (W3C). The W3C EBNF specification can be found in [33].

The following is a grammar of a toy language in W3C ENBF:

1. $program ::= operation*$
2. $operation ::= operand\ operator\ operand$
3. $operator ::= '+' | '-' | '*' | '/'$
4. $operand ::= digit$
5. $digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'$

The terminals are indicated as symbols in single quotes, for example *'0'* is a terminal. Everything that is not in quotes is either a part of the EBNF metalanguage or a nonterminal. In this case, *program*, *operation*, *operand*, *operator* and *digit* are terminals, and symbols *::=* and *** are EBNF operators. Meaning of all the operators and their precedence can be found in [32] and [33].

A program that conforms to this grammar can look like this:

$$0+1 \quad 3-2 \quad 4/1 \quad 5*9 \quad 1-1 \quad 4*3$$

Similar notation is used in the specification of the modeling method domain-specific language grammar later on in this dissertation.

2.3.1.5 Attributed Grammar

The restrictions on the structure of valid strings that are hard or even impossible to express using formal grammars are called static semantics. For example, checking that every identifier is declared before it is used, or checking that every identifier is used in the appropriate context. One way to express these rules is using attributes and the corresponding attribute grammars. Important contributions to this approach have been provided by Knut in [34] and [35], and by Paakki in [36]. Examples how to use attribute grammars in the development of programming languages can be found in [37]. This section will only cover the relevant details that are important in the context of this dissertation.

First of all, note that there is a difference between *attributed grammars* and *attribute grammars*. An attribute grammar is a formal way to define attributes for the productions of a formal grammar. An attributed grammar is a formal grammar augmented with attributes defined using an attribute grammar. In the construction of language translators, attributes provide semantic value to the syntactic constructs.

Let $G = (T, N, S, P)$ be a context-free grammar, where, as before, T is a set of terminals, N is a set of nonterminals, S is a start symbol, and P is a set of production rules. Let V be the finite vocabulary of terminals and nonterminals, that is $V = T \cup N$. Elements in V are called grammar symbols. According to [34] and [36] semantic rules are added to G in the following manner:

- To each symbol $X \in V$ we associate a finite set $A(X)$ of attributes, where A is partitioned into two disjoint sets: the synthesized attributes $A_S(X)$ and the inherited attributes $A_I(X)$.

- The start symbols and terminals do not have inherited attributes.

- Each attribute α in $A(X)$ has a set of possible values W_a, from which one value will be selected for each appearance of X in a derivation tree.

- A production $p \in P$, $p: X_0 \rightarrow X_1 \ldots X_n$ $(n \geq 0)$, has an attribute occurrence $X_i.a$, if $a \in A(X_i)$, $0 \leq i \leq n$.

- A finite set of semantic rules R_p is associated with the production p with exactly one rule for each synthesized attribute occurrence $X_0.\alpha$ and exactly one rule for each inherited attribute occurrence $X_i.a$, $0 \leq i \leq n$.

- R_p is a collection of rules of the form $X_i.\alpha = f(y_1, \dots, y_k)$ $k \geq 0$, where

  1. either $i = 0$ and $\alpha \in A_S(X_i)$, or $1 \leq i \leq n$ and $\alpha \in A_I(X_i)$;
  2. each y_j, $1 \leq j \leq k$, is an attribute occurrence in p; and
  3. f is a function, called a semantic function, that maps the values of $y_1, \dots, y_k$ to the value of $X_i.\alpha$. In a rule $X_i.a = f(y_1, \dots, y_k)$ the occurrence $X_i.a$ depends on each occurrence y_j, $1 \leq j \leq k$.

- R is then a finite set of all the rules, that is $R = \cup R_p$.

Often, Attributes can be represented with an extended parse tree called *attributed tree*, where each node n, labeled by X, has an attribute instance attached that corresponds to the attributes of X. For each attribute $a \in A(X)$ the corresponding instance is denoted with $n.a$. Attribute instance values are computed by executing semantic rules. Synthesized attributes pass information from the RHS (right-hand side) to the LHS (left-hand side) symbols of productions, and inherited attributes pass information from the LHS to the RHS symbols of productions. In other words, synthesized attributes are those that have been computed for children nodes, and are being passed up the tree. Typically, values of synthesized attributes are combined to produce an attribute for the parent node. Inherited attributes are those passed from the parent down to the child.

For example, let $X.a \rightarrow Y_1.a \dots Y_n.a$ be an attributed production rule.

- $X.a$ is a synthesized attribute, because it is a function of $Y_i.a$
- $Y_k.a$ is an inherited attribute, because it is a function of X and $Y_i.a, i \neq k$

2.3.1.5.1 Attributed Extended Backus-Naur Form

To demonstrate how to attribute EBNF grammar, let us extend the previously shown example of a toy language in W3C EBNF, by allowing operations between numbers, which are defined as a string of digits. The EBNF grammar is attributed by an attribute called *value* which holds number and digit values (integers). Semantic rules are written inside curly brackets:

1. $digit ::= '0' \mid '1' \mid '2' \mid '3' \mid '4' \mid '5' \mid '6' \mid '7' \mid '8' \mid '9' \{digit.value = toInt()\}$
2. $num ::= digit \{num.value = digit.value\}$
3. $num_1 ::= num_2\ digit \{num_1.value = num_2.value * 10 + digit.value\}$

The following figure (Figure 4) illustrates the use of synthesized attributes. It can be seen that the values are propagated to the parent nodes.

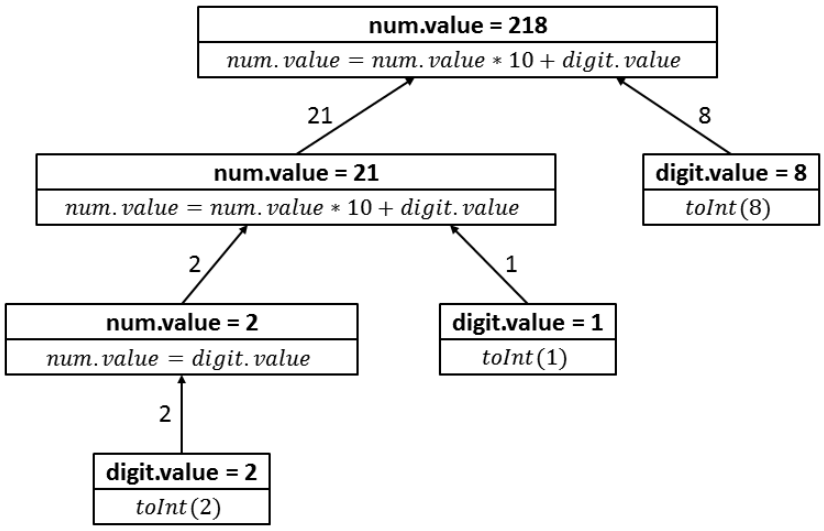


Figure 4: Propagation of Synthesized Attributes through the Parse Tree

Another way is to use inherited attributes, where the values of attributes are propagated from parents to children nodes. Here is a toy example:

1. $num_1 ::= digit\ num_2\ \{num_2.value = (num_1.value + digit.value) * 10\}$
2. $num_1 ::= digit\ \{num_1.value = num_1.value + digit.value\}$

Figure 5 illustrates the propagation of inherited attributes on a parse tree. Here we need to assume that initial value is $num.value = 0$.

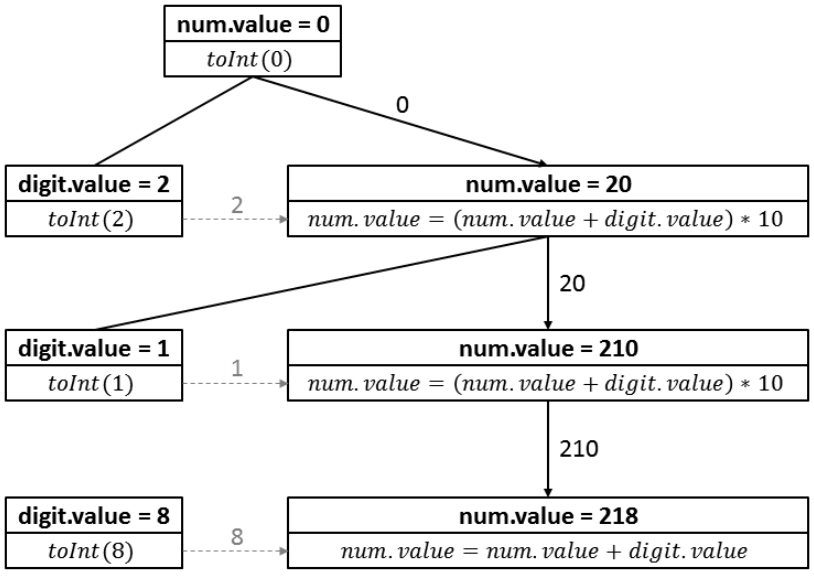


Figure 5: Propagation of Inherited Attributes through the Parse Tree

Depending which kind of attributes are used, one can separate attributed grammars into *L-attributed* grammars, or *S-attributed* grammars. There are other classifications as well, but they are all subcategories of these two attributed grammar types. In L-attributed grammars all inherited attributes in a semantic rule are a function only of

symbols to their left, meaning that inherited attributes can be evaluated in one left-to-right traversal of the abstract syntax tree (top-down parsing). S-attributed grammars contain only synthesized attributes. Any S-attributed grammar is also an L-attributed grammar. Attribute evaluation in S-attributed grammars can be either by top-down parsing or bottom-up parsing.

2.3.2 Metamodeling Approach

The metamodeling approaches have started to emerge as standard design tools for entity-relationship-based modeling languages. The basic idea is to define the abstract syntax of an entity-relationship-based modeling language with a model that is expressed either in the same language, or in a simpler, generalized, entity-relationship-based language. We can assert that every description of a language is a metamodel, including languages described by formal grammars or logic. However, the term metamodel is typically used in the context of entity-relationship-based languages, which is a category where most of the modeling languages belong, especially graphical modeling languages.

Nowadays, UML, particularly the UML class diagram, is used to describe metamodels. Entities are depicted as UML classes, relationships as UML relationships. There are many instances of relationships in UML. Most commonly used in the metamodeling approach are associations, aggregations and generalizations (or specializations, if looked from the opposite side). To support the graphical representation of metamodels, some researchers have gone one step further and envisaged the transformations from formal methods, such as EBNF, to UML class diagrams [38]. On the other hand, one can find research investigating approaches that use rigorous EBNF-based definition for specifying the syntax of graphical modeling languages [39]. In this case, authors try to solve the lack of expressiveness in UML-based metamodeling approaches. Alternatively, some approaches use textual constraint languages (e.g., OCL) in conjunction with UML class diagrams. Rigorous EBNF-based approaches try to avoid this.

Most known metamodeling approach is based on a multilayered metamodel architecture [40]. The smallest number of layers is two, one were we define the meta-concepts, and the other where these concepts are instantiated [13]. The most common layered architecture is the *four layered metamodel architecture* that is referred to in various OMG specifications. For example UML is specified using this kind of metamodel architecture [16]. In theory, there can be infinite number of layers, but four is typically enough.

Figure 6 illustrates the four layered metamodel architecture. It shows that in every layer, excluding the layer 0, there is a model, which is realized using a specific modeling language.

Layer 0 represents a system under study (SUS), which is a delimited part of the world considered as a set of elements and interactions. SUS is also referred to as real world, reality or original.

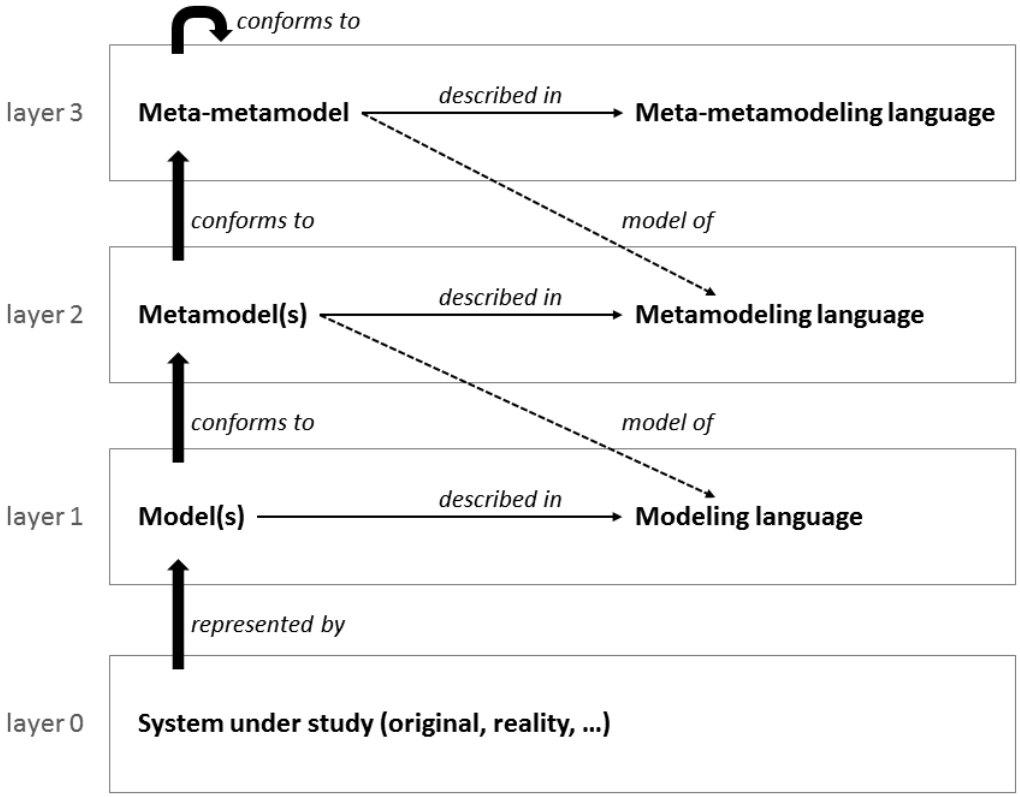


Figure 6: Four Layered Metamodel Architecture (adapted from [10])

Layer 1 is occupied by a model, a representation of a given SUS, which is defined as a directed multigraph that consists of set of nodes, a set of edges, and a mapping function between nodes and edges. Nodes may be connected with more than one edge. This model is such that its reference model is a metamodel.

Layer 2 is occupied by a metamodel, which is a model such that its reference model is a meta^{2}model. In its broadest sense, a metamodel is a model of a modeling language, and it must capture the essential features and properties of the language that is being modeled.

Layer 3 is occupied by a meta^{2}model, which is a model that is its own reference model, meaning it conforms to itself. Meta^{2}model is the key to metamodeling as it enables all modeling languages to be described in a unified way. That is, all metamodels are described by a single meta^{2}model. In case of any other n-layered metamodeling architecture, for example six layered architecture, the model that is found in layer five would be its own reference model.

Concepts mentioned in here represent different tiers of abstraction of the real world. System under study can be viewed as the lowest or tier zero abstraction, and meta^{2}model as highest or tier three abstraction.

It is also important to note the connection between a reference model and its instance. If a metamodel is a reference model for a model in layer 1, then it is also true that a model from layer 1 is an instance of a metamodel from layer 2. This is true for all lay-

ers, even for layer 0, where SUS is an instance of a model from layer 1. This relationship is transitive. Therefore, we can assert that a model from layer 1 is an instance of a meta^{2}model (layer 3) as well.

2.3.2.1 Designing a Modeling Language

In the modeling language design process, one does not tackle with many metamodel layers. The meta^{2}model is typically given. Therefore, one needs to design a metamodel upon the given meta^{2}model that describes a modeling language. We realize a modeling language by applying a metamodeling language. If it will be possible to capture, not only the syntax, but notation and semantics of a modeling language as well, depends on the expressivity of a metamodeling language which is described by its own model, a meta^{2}model.

In reality, design and realization of modeling languages heavily relies on the metamodeling platforms and frameworks which come equipped with predefined meta^{2}models and various tools for their instantiation.

Design and realization are two separate phases in modeling language development. We can draw a parallel with software engineering. The first phase, design, is very similar to the design of a software application. We construct a metamodel using a specific technique. This process is similar to the construction of a domain model in software engineering. The second phase, realization, is concerned with the implementation of a modeling tool, which is a piece of software. Therefore, it is not very different from the implementation of any other software application. The one significant difference is the use of metamodeling platforms and frameworks which provide the basic functionality out-of-the-box. One can as well implement a modeling tool from scratch. However, because of the complexity that comes with this kind of software, it would take a lot of effort and considerable time investment.

The metamodeling platforms will be discussed in a dedicated chapter. Let us first focus on the design phase of modeling languages.

Nowadays, the most prominent metamodeling technique is the OMG-UML approach and many of its variations. This technique is considered semiformal in comparison with the techniques coming from the formal language theory. The core OMG-UML approach allows the definition of: (1) abstract syntax, (2) static semantics, and (3) dynamic semantics. In case of a graphical modeling language, concrete syntax is typically given as a collection of images and its mapping to the abstract syntax.

The abstract syntax is given in a (very) simplified UML class diagram, which shows the meta-concepts of language's elements: entities and relationships. Because of a heavy use of class diagrams in metamodeling approaches, it is very common to use the word *class* instead of *entity*. The expressivity of class diagrams allows enforcing some of the well-formedness rules directly into diagrams, such as relationship multiplicity requirements.

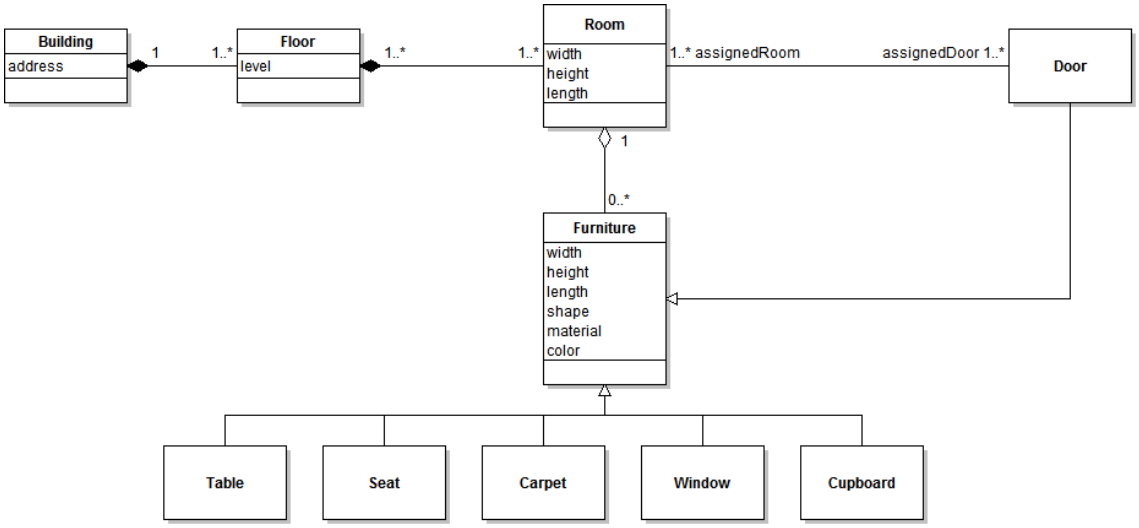


Figure 7: Metamodel of a Building

Figure 7 depicts an exemplary metamodel of a generic building. From it one can read all the important relations between various entities, and what kind of attributes describe these entities.

It is customary to provide a short informal description in prose for each element that is provided in the metamodel. This description typically contains syntactic and semantic explanations mixed together. For example:

> *A building consists of floors, which are ordered by levels. The floor with a higher level must be on top of the floor with lower level (e.g., floor 2 is on top of floor 1). There can be no skipping of floor levels. Each floor is composed of rooms, which may have different functions: a bathroom, office, corridor, or a lobby. Rooms are connected with other rooms by doors. Each room can contain furniture of various kinds, such as table, seat, carpet, window, cupboard, etc.*

The above example is a vague description of a building represented in Figure 7. In reality, these informal descriptions need to be more concise and cover all the details that are not explicitly visible on the metamodel.

Static semantics of a modeling language are defined using well-formedness rules, which define a set of invariants of an instance of a meta-concept. The rules specify constraints over attributes and relationships defined in the metamodel and need to be satisfied for the meta-concepts to be meaningful. In OMG-UML approach well-formedness rules are defined by an OCL expression together with an informal explanation of the expression. More generally, rules can be defined using first-order logic. Object Constraint Language (OCL) is nothing else but a variant of first-order logic.

Here are a couple of examples of well-formedness rules written in OCL:

- Employees must not work more than 5 hours.

```
invariant NoMoreThanFiveHours: self.numberOfHours > 5;
```

- Title of a movie has to be at least 3 characters long.

```
invariant AtLeastThreeCharacters: self.title.size() >= 3;
```

- Timetable needs to have at least one session.

```
invariant AtLeastOneSession: self.containedSession->isEmpty() = false;
```

Dynamic semantics of a modeling language are defined in prose accompanied by exemplary models produced with a modeling language in question. As with the abstract syntax, the provided description mixes syntactic and semantic explanation. This time, the syntax in question is the concrete syntax (notation). The explanation describes the meaning of elements in a modeling language and their relationships. Dynamic semantics definition in metamodeling is considered to be the least formal part of the metamodeling approach.

To explain these kinds of semantics one typically uses exemplary models produced by the modeling language in question. Figure 8 depicts elements and their relationships for the generic building modeling language. In this figure one can only see a subset of the elements, mainly a room, and its belonging furniture. The explanation in prose would look something like this:

> *This model shows a classroom layout plan. The detailed placement of furniture is shown, including a teacher's desk, chalkboard, student desks, windows and bookcases.*

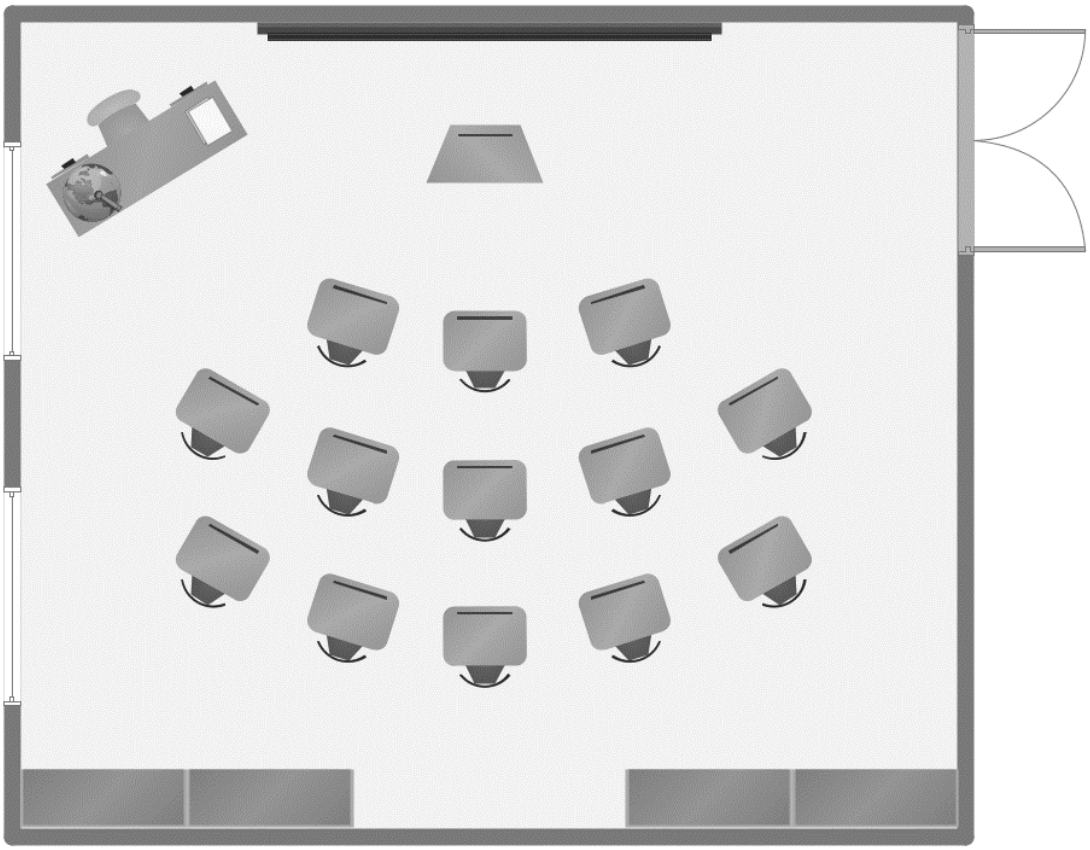


Figure 8: Model of a Room with Furniture (adapted from [41])

The last part that needs to be described during the design of a graphical modeling language is its concrete syntax (also known as notation). One does need to describe the notation before dynamic semantics in case the designed language does not use the standard UML symbols. Describing the notation is not a part of the OMG-UML approach. A metamodeling community has found various ways to represent graphical symbols and its mappings to the abstract syntax. Typically, a simple quasi-tabular view is used, that describes which symbol belongs to which concept.

Figure 9 shows how this would look for the part of the modeling language depicting a room with furniture in Figure 8. One can see that a graphical symbol on the left side is connected with a modeling element on the right side (which is nothing more than a text in this case).

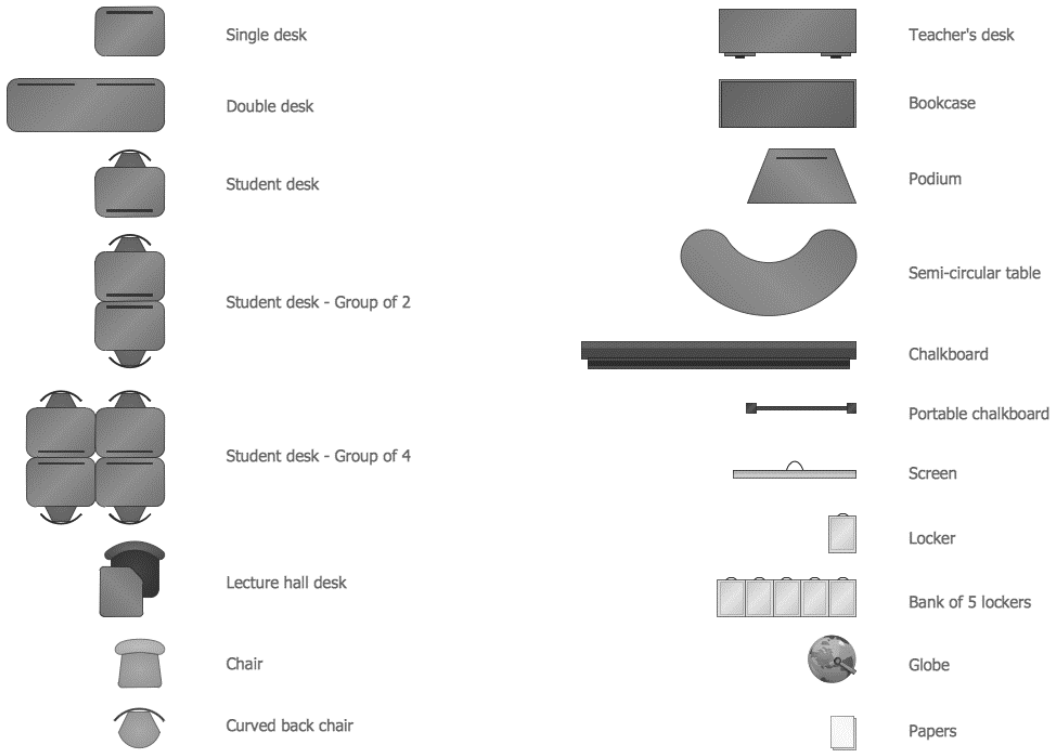


Figure 9: Notation of Modeling Elements (adapted from [41])

To complement the images of symbols from a tabular view, one can use technologies such as SVG. Because SVG is an XML-based format and an open standard developed by W3C, it provides us with means to define the concrete syntax in a formal way. This may provide very useful in the later realization of a modeling language on a metamodeling platform in case the platform supports SVG or similar graphical object descriptions.

3 State of the Art and Related Research

3.1 Language-Oriented Engineering

This chapter is dedicated to the currently most known and utilized metamodeling techniques and technologies. It gives an overview of the technical side of metamodeling research and serves as a basis on which the tools for language-oriented modeling method engineering have been built upon.

On a closer inspection one can become aware that various metamodeling techniques utilized for the construction of modeling tools belong to the *Language-Oriented Engineering* (also known as *Language-Oriented Programming*) paradigm. After all, a modeling tool is nothing else than a software developed for a specific purpose. Therefore, it is natural to apply software engineering paradigms in its construction.

The main idea is to create domain-specific languages for the problem at hand, rather than using existing general-purpose programming languages for solving the problem. In current domain, the problem one tries to solve can be described as efficient modeling tool development. And the construct created for its solving is a metamodeling technique, which comprises one or more domain-specific languages and metamodeling tools. These DSLs are exposed to the user (typically modeling method engineer) in many different ways. Some metamodeling tools use configuration dialogs on top of DSLs, while the others expose DSLs directly to the user. Most metamodeling tools mix both approaches together, which leads to unnecessary complications in development. For example, user is typically unaware that configuration dialogs do not expose all of the metamodeling tool's functionality. This issue is present in almost all metamodeling platforms.

In general, there are three main approaches to Language-Oriented Engineering: (1) the use of internal DSLs, (2) the use of external DSLs, and (3) the use of metamodeling platforms (or language workbenches) [42].

Internal DSL are implemented as application libraries in a given host language. Dynamic programming and scripting languages (for example Groovy) are particularly good hosts for various DSLs. This approach may be applied in the development of simple modeling tools with a textual notation. Internal DSLs are constrained by the possibilities of host languages. Therefore, it is not advised to use them for development of complex graphical modeling tools.

External DSLs are implemented as stand-alone languages and come equipped with translators (compilers, interpreters, and generators). These DSLs are more powerful than internal DSLs, because their syntax and semantics do not depend on the host language. This also makes them user friendlier and easier to learn, as one does not

need to be familiar with the host language, which is the case when using internal DSLs. However, developing external DSLs is a much longer process than developing internal DSLs, and it is very similar to developing general-purpose programming language. It is affordable to develop such a DSL only in case if it can be reused multiple times. For example, if the DSL can be utilized to create dozens of modeling tools than it is considered affordable to invest in its creation. Regarding external DSL expressivity, it is the most expressive from the three mentioned approaches. It has the ability to describe all the aspects of a modeling tool.

Metamodeling platforms are integrated development environments (IDEs) for defining and using modeling methods (graphical modeling languages as well). They combine graphical interfaces such as dialogs together with DSLs to provide configuration options to the user. DSLs included in the platform are limited by the platform's functionality (similar to the internal DSLs). The greatest advantage of using a metamodeling platform comes with its out-of-the box functionality, which includes an expressive meta^{2}model and algorithms that can be configured differently for each modeling method.

A fourth language-oriented engineering approach is proposed in this dissertation: augmenting metamodeling platforms with an external DSL. The DSL in question is called MM-DSL. Its design, requirements, specification, utilization in modeling method engineering, and evaluation is discussed in dedicated chapters. By combining the external DSL approach with metamodeling platforms one gets the best from both words, without introducing any noticeable disadvantages. It is also important to mention that this approach is not connected with any particular metamodeling platform. Thus, it is completely platform independent.

The technique for developing modeling tools for modeling methods proposed in this dissertation is as follows:

1. Use a domain-specific language to describe a modeling method;
2. Translate a modeling method to the metamodeling platform of choice;
3. Apply a metamodeling platform to generate a modeling tool.

Because this technique is mainly driven by MM-DSL, a development environment supporting it needs to be provided, as well as various translators that allow the translation of modeling method code written in MM-DSL to the modeling method code (or format) understandable by a metamodeling platform.

By augmenting modeling tool development with domain-specific concepts, we make it more structured and more explicit, thus also more understandable. Additional value of using a language in the development of modeling tools comes in form of the following benefits: self-documenting code, versioning and reuse. A language able to describe modeling method constructs is also easy to learn by modeling method engineers, because it contains the concepts already familiar to them, such as class, relation, attribute, model type, notation, etc.

3.2 Related Concepts

There are three technology-based ways one can pursue when developing modeling tools: (1) from scratch, (2) use metamodeling framework, or (3) use metamodeling platform. All of them come with advantages and drawbacks.

Developing a modeling tool from scratch might be considered a big effort, but in some cases it is the most feasible solution. For example, metamodeling platforms and frameworks would present an obstacle in a scenario where the developer needs to be in full control of every single implementation detail of a modeling tool. It can also happen that a modeling tool requires novel features that are not supported by any existing platforms or frameworks. In this case, using language-oriented engineering paradigm may prove to be beneficiary. Of course, the tool can also be directly programmed in a general-purpose programming language.

A metamodeling framework provides some of the basic functionality needed by a modeling tool. An implementation of a meta^{2}model is always provided, as well as means to instantiate it. The key difference between a framework and a platform is that a framework needs to be attached to something, while a platform is stand-alone software. Frameworks are typically additions to the popular Integrated Development Environments (IDE), such as Eclipse and Visual Studio. Therefore, they lack in the area of modeling tool interface development and customization. This functionality is limited by the host IDE. The usability of frameworks starts to decrease with the increased complexity of a modeling tool. More complex modeling tools have a higher chance of requiring functionality a framework does not provide. The only solution for this issue is to use a general-purpose programming language and develop additional functionality, which is actually reverting us to the *“from scratch”* approach.

Metamodeling platforms offer the most features compared to the other two approaches, including some of the advanced features, such as predefined model analysis and simulation algorithms, and model repositories. Same as frameworks, platforms are equipped with a predefined meta^{2}model. They are stand-alone and customizable. With metamodeling platforms, it is possible to realize some of the most complex modeling languages and modeling methods. The drawback, however, is that they require considerable time investments in order to get to know all the provided features, as well as learning how to apply them correctly. Similar to the metamodeling frameworks, metamodeling platforms also have boundaries on their functionality. However, it is difficult to extend them using a general-purpose programming language. In some cases, these kinds of extensions are not even possible, because of license restrictions (e.g., proprietary software or closed source). Because of it, metamodeling platforms are typically equipped with an internal DSL or scripting language through which platform APIs are exposed. In case that the required functionality cannot be achieved, one either needs to find another metamodeling platform, or revert to the *“from scratch”* approach as it is already suggested in case of metamodeling frameworks. Nevertheless, it has been noticed that the general behavior in this particular case is to continue developing the

modeling tool on the same metamodeling platform, but without that particular functionality. This happens, because it is almost impossible to transfer the work that has already been done on one platform to the other. Reimplementing everything on another metamodeling platform would be a considerable time investment. Thus, it is very costly.

3.3 Existing Tool Support

This section serves as an overview of different metamodeling tools. The focus is put primarily on well-known metamodeling platforms and frameworks. Some of the tools mentioned in here may be used to describe and implement any kind of textual and graphical languages, not only modeling languages or modeling methods.

There exist two types of tools one can use to realize languages: (1) textual language realization tools, and (2) graphical language realization tools. These types of tools implement different language specification techniques. The tools from the first category implement approaches from the formal language theory, such as formal grammars, and are used for the realization of textual languages, particularly textual domain-specific languages. The tools belonging to the second category implement a metamodeling approach. These tools are used to realize domain-specific graphical modeling languages and modeling methods.

3.3.1 Textual Language Realization Tools

Textual language realization tools are the product of the extensive research in programming languages and compiler design. Pioneers in this area are two famous programs: Lex (A Lexical Analyzer Generator), which was written by Michael Lesk and Eric Schmidt [43], and Yacc (Yet Another Compiler-Compiler), which was written by Stephen C. Johnson [44]. Both of them were developed in the early 1970s as part of the projects taking place at Bell Laboratories. Newer tools inherited the same concepts and extended upon Lex and Yacc. Nowadays, the most know representatives in this category of language development tools are ANTLR and Xtext. There exist several other similar tools, such as MPS and Irony. However, these are not very well known by the language development practitioners.

3.3.1.1 ANTLR

ANTLR [45] (Another Tool for Language Recognition) is a powerful language development tool that provides a framework for constructing recognizers (e.g., lexers or parsers), compilers, and other translators from grammatical descriptions in a variety of target languages including Ado, ActionScript, C++, C#, JavaScript, Python, Ruby and several others. ANTLR automates the construction of language recognizers. From an input similar to formal grammar, ANTLR generates a program that determines whether statements conform to that language. In other words, ANTLR is a program that writes other programs. By adding code snippets to the grammar, the recognizer becomes a compiler or interpreter. ANTLR provides excellent support for parse tree construction,

parse tree walking, and translation. It provides sophisticated automatic error recovery and reporting. Language grammars are defined in either ANTLR syntax (which is Yacc and EBNF like) or a special AST (Abstract Syntax Tree) syntax. Currently, ANTLR is the most recognized parser generator and it is widely utilized both in academia and industry to build all sorts of languages, tools and frameworks.

As most of the language realization tools, ANTLR also comes with several IDE implementations: ANTLRWorks (stand-alone), Eclipse IDE plugin and IntelliJ IDEA plugin [46]. Figure 10 shows Eclipse IDE with the ANTLR v4 plugin installed. The central part of the IDE is the grammar editor. One can see that the grammar utilized in ANTLR is very similar to the EBNF notation. At the bottom of the IDE is the syntax diagram (also known as the railroad diagram), which is automatically generated from the inputted grammar. The advantage of syntax diagrams is in the ease of navigation through the grammar. They present a better overview of the grammar as well. On the left side is the project overview, and on the right side is the outline currently showing the names of all the grammar rules.

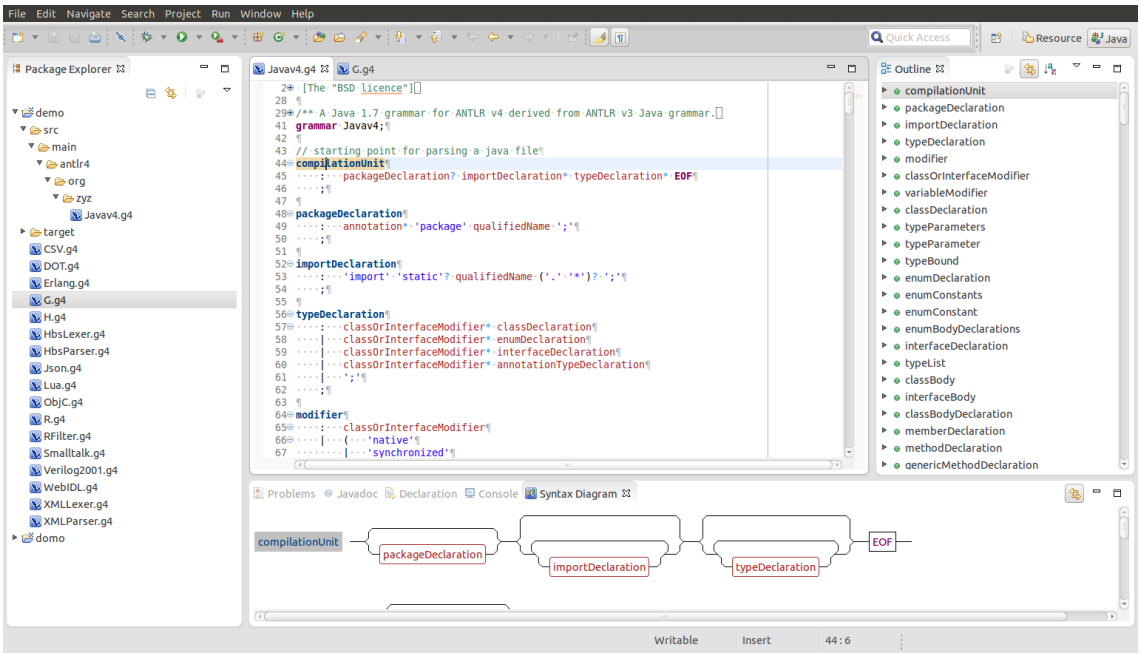


Figure 10: Eclipse Plugin for the ANTLR v4 (taken from [46])

3.3.1.2 Xtext

Xtext [47] is an open source framework for developing textual computer languages, particularly domain-specific languages, but it can also be used to develop general-purpose programming languages. Xtext is shipped as a set of plugins for the popular Eclipse IDE and it integrates seamlessly with other eclipse-based (modeling) technologies such as EMF, GMF, M2T, and parts of EMFT. It provides a specialized editor for describing a grammar of a language. From the grammar it generates APIs for programmatically manipulating instances, parsers, and formatters for reading and writing instances, and a rich user interface to support end-users, including a feature-rich editor

that is specific to the defined language. The generated editor includes features such as syntax coloring, code completion, and validation.

Additional semantic aspects of the language can be programmed in Java or Xtend. Xtend is a Java dialect with several useful features (such as support for code templates) for writing translators. Xtext uses code generation to create a parser from the language's grammar. Xtend uses the generated parser as a starting point to produce various translators (e.g., compilers, interpreters).

In comparison to ANTLR, Xtext's greatest advantage is the generation of IDE artifacts which support the parser and seamlessly integrate into the Eclipse IDE. Regarding actual parser generation, Xtext utilizes ANTLR, thus the quality of the generated parsers is identical.

Figure 11 shows the Eclipse environment running the Xtext framework. It looks very similar to the eclipse ANTLR plugin: center is occupied by the grammar editor, on the left side one can find the projects overview, and on the right side there is an outline showing all the defined grammar rules. The screenshot, in addition to the Xtext framework, shows a small excerpt of the MM-DSL grammar and accompanying MM-DSL projects.

Currently, Xtext framework is mostly use to developing DSLs that compile to Java or JVM languages. It is also very popular in the Eclipse community. On the dedicated Xtext community page it is mentioned that Xtext has been used more than 40 commercial and non-commercial projects [48]. However, ANTLR is a more reliable tool when it comes to the development of cross-platform general-purpose programming languages.

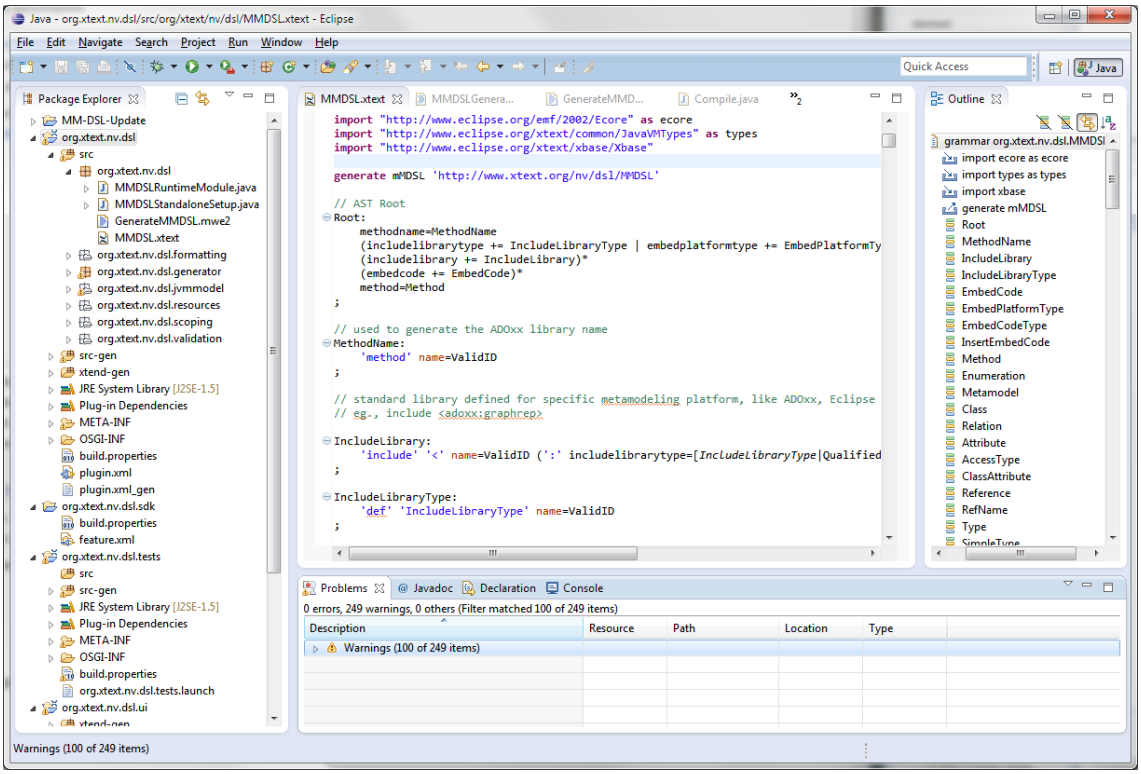


Figure 11: Xtext Framework for Eclipse

3.3.1.3 MPS

Meta Programming System (MPS) is an open source language realization tool based on the language-oriented programming. The creators address it as a language workbench. Thus, it is not a framework or a plugin, but a stand-alone IDE for creating and using the created languages. It uses a projectional editor that renders the abstract syntax tree in a notation that looks and feels textual while the user directly edits the tree. What this means is that writing a keyword (e.g., *while*), which is a part of the expression (e.g., *while <condition> do <something>*) will also add the rest of the expression to the program, with placeholders (in our example the words started with < and ending with >) which need to be filled with values (terminal symbols or other expressions). This also means that programs can include syntactic forms other than text, such as tables or mathematical symbols [49].

Defining a language begins with defining the abstract syntax. The editor (projection rules) is defined in a second step. The third step is defining a generator which provides semantics by mapping defined language constructs to one of the several existing languages. MPS currently supports mapping to C, Java, XML, or plain text.

The most successful application of MPS nowadays is the mbeddr language, which is based on the extensible version of C programming language used in the embedded software engineering. Figure 12 shows how the mbeddr projectional editor based on MPS represents text and tables in programs. What we actually see is the editing of abstract syntax tree, where it is only possible to insert expressions in a valid form. The editor warns us about wrongly imputed parameters (the value parts of the expressions).

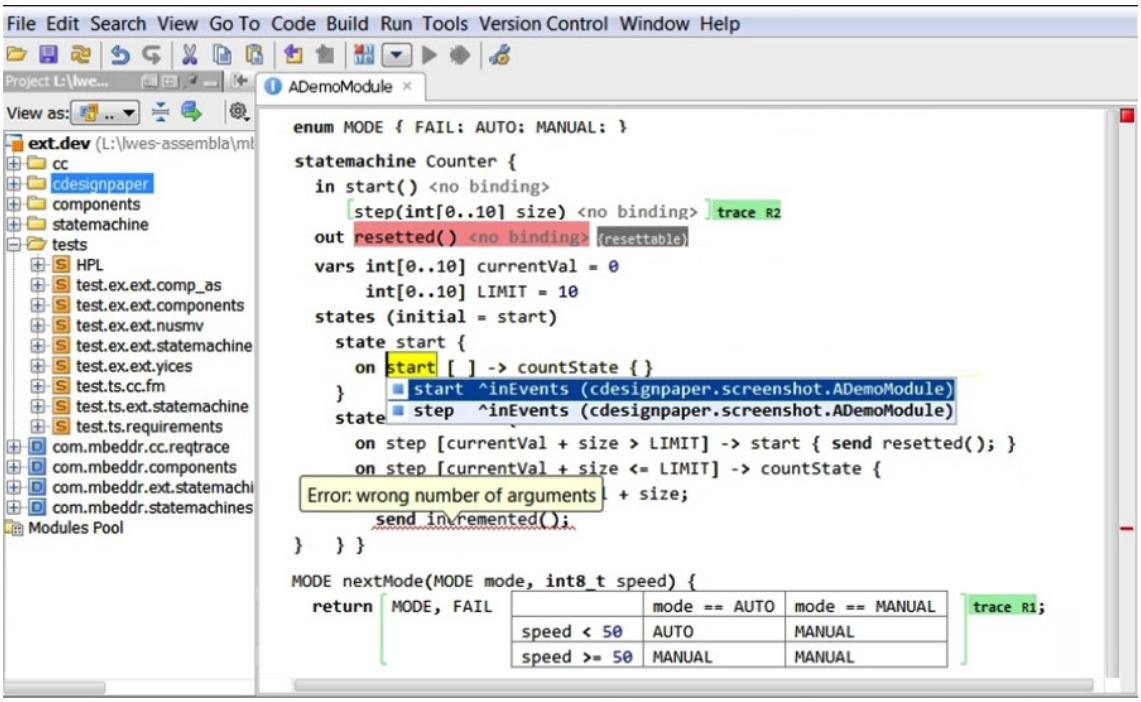


Figure 12: mbeddr IDE and Projectional Editor (taken from [50])

3.3.1.4 Irony

Irony is an open-source development kit for implementing languages on .NET platform. Unlike most existing Yacc/Lex-style solutions Irony does not utilize any scanner or parser code generation from grammar specifications written in a specialized meta-language. In Irony the target language grammar is coded directly in C# programming language using operator overloading to express grammar constructs. Irony's scanner and parser modules use the grammar encoded as C# class to control the parsing process [51].

The language grammar definition is very similar to EBNF notation. Lexical and parser specification are combined in one C# file. Since the file is already in C#, no code generation takes place. This creates a solution that is easier to debug and maintain. Irony comes equipped with an explorer tool for viewing and debugging the defined languages, and with multiple sample grammars for languages like GW Basic, Java, C#, Scheme, SQL, JSON and more.

Because Irony is not a standalone solution, Visual Studio IDE is a prerequisite, which can also be used as a rich-featured IDE by integrating the defined language via Visual Studio language service. A language service provides language-specific support including syntax coloring and highlighting, statement completion, validation, brace matching, parameter information tooltips, etc. It is only possible to integrate a new language service with commercial versions of Visual Studio, thus making Irony not affordable to the developers and organizations that do not have access to these versions of Visual Studio. Although Irony itself is free, commercial versions of Visual Studio are fairly expensive.

The first MM-DSL concept and its parser have been implemented in Irony. During that time the language has been known as MetaDSL and it was not as expressive as MM-DSL. The full code listing can be found in Appendix B.

3.3.2 Graphical Language Realization Tools

By applying the concepts of metamodeling approaches to the technical space metamodeling platforms are developed, such as ADOxx, MetaEdit+, Generic Modeling Environment (GME), and modeling frameworks like Eclipse Modeling Framework (EMF) and Graphical Modeling Framework (GMF).

3.3.2.1 ADOxx

ADOxx is a mature metamodeling platforms used for the realization of modeling languages and modeling methods. It is based upon an expressive meta^{2}model which offers basic concepts such as class, relation class, attribute, model type, mode, and others. ADOxx comes equipped with advanced tooling features as well: (1) a scripting language called *ADOscript* used to add additional functionality to modeling languages, thus

extending them to modeling methods, (2) a graphical notation language called *GraphRep* used for creating the concrete graphical syntax, and (3) a language for exposing functionality to the user through the user interface called *AttrRep*. Realized modeling languages or modeling methods and their accompanying metamodels, as well as models created with them are saved in a repository.

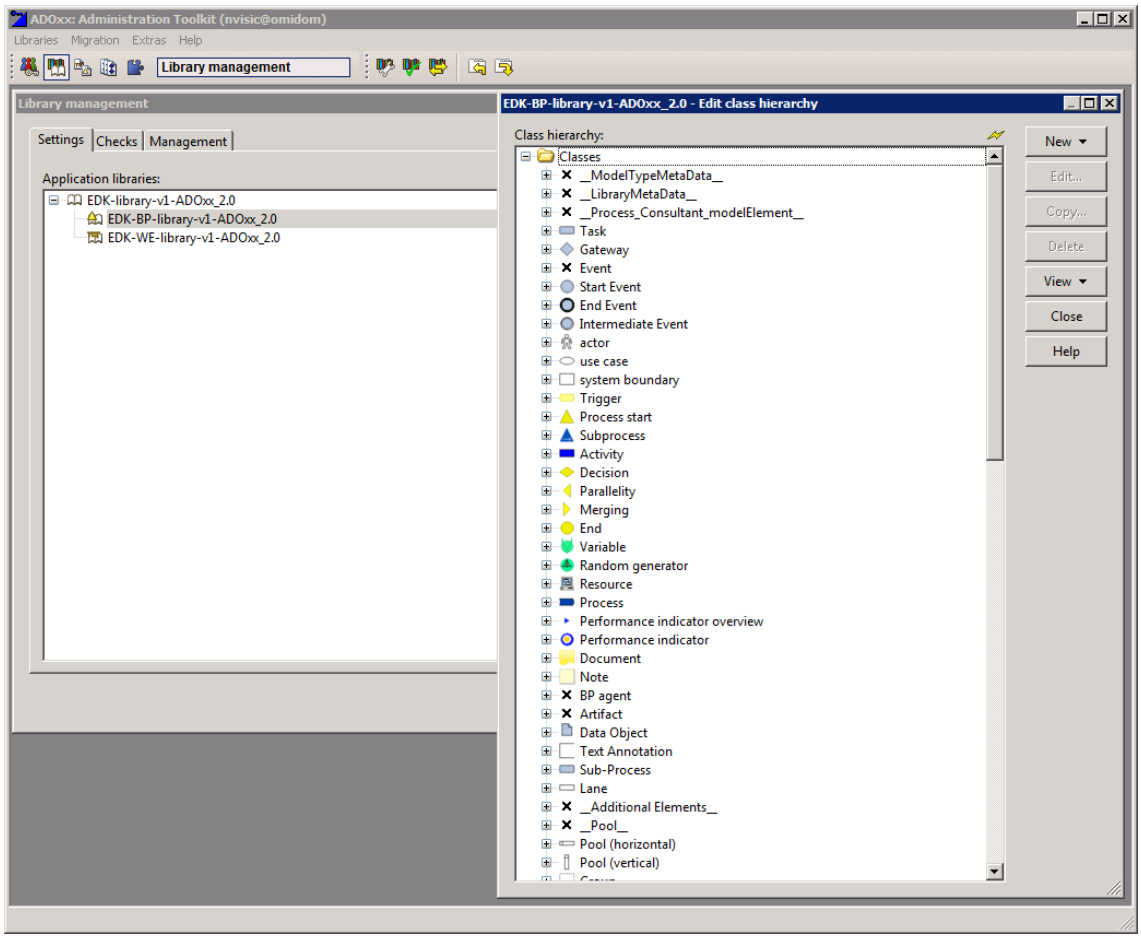


Figure 13: ADOxx - Creating a Metamodel

Figure 13 show a typical ADOxx user interface for interacting with its meta²model. On the left side of the screenshot one can see the ADOxx application libraries, which are containers for all the elements of a modeling method. On the right side of the screenshot is an editor called the class hierarchy tree. This editor is used to instantiate a metamodel from the provided meta²model by defining new classes, relation classes and its accompanying attributes.

Figure 14 shows three important ADOxx editors: (1) *GraphRep*, (2) *AttRep*, and (3) *ADOScript* editor. GraphRep editor is where graphical representation is created. AttRep editor is a place where one defines which attributes will be exposed to the user of a modeling tool. ADOScript editor provides basic functionality for writing modeling algorithms in the ADOxx scripting language. In each of the screenshots in Figure 14 one can also see a small implementation excerpt for the three key modeling method elements in the corresponding ADOxx languages.

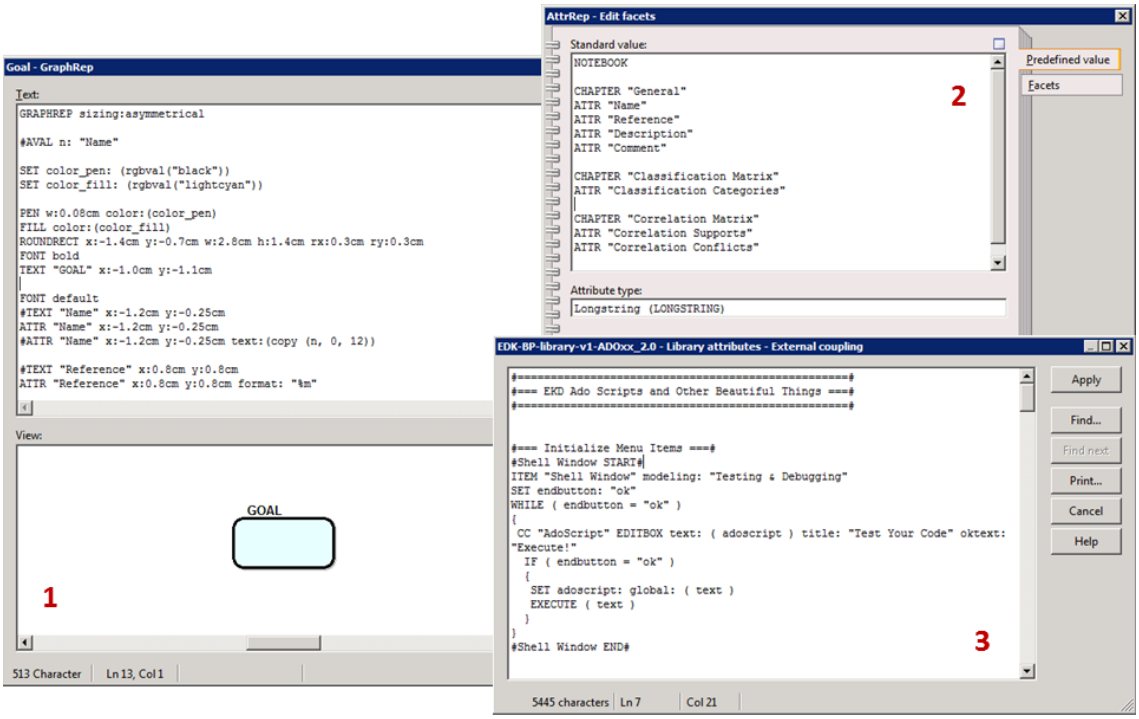


Figure 14: ADOxx Editors - 1. GraphRep, 2. AttRep and 3. ADOScript

3.3.2.2 MetaEdit+

MetaEdit+ is a completely integrated environment for building and using individual domain-specific modeling (DSM) solution applied in several industry projects. The meta^{2}model called GOPPRR offers the following basic concepts: graph, object, property, port, relationship and role. A diagram editor, object & graph browsers, and property dialogs support the definition of a new modeling language without manual coding. MetaEdit+ is very successful in providing supports for full code generation directly from the models.

Figure 15 is a screenshot of the MetaEdit+ user interface. Only the main modeling interface is shown. The majority of the screen is occupied by a modeling canvas. One can see the modeling toolbox in the top left corner. The instantiated objects can be seen in the tree list to the left, as well as the properties of a selected object. The basic GOPPRR concepts are also depicted. Only port is missing, because it is an optional concept. In the example depicted by the screenshot, role is directly connected to the object.

Similar to the ADOxx, MetaEdit+ also has several metamodeling editors and tools: (1) one tool for each GOPPRR concept (graph tool, object tool, property tool, port tool, relationship tool, and role tool), (2) graphical representation editors (symbol, icon and dialog editor, (3) code generation tools (generator editor, MERL language, generator debugger), and (4) modeling language management tools (metamodel browser, type manager, and info tool). For more information about these tools, which include full descriptions and screenshots see the MetaEdit+ manual [52]. An insight and description of the GOPPRR concepts can be found there as well.

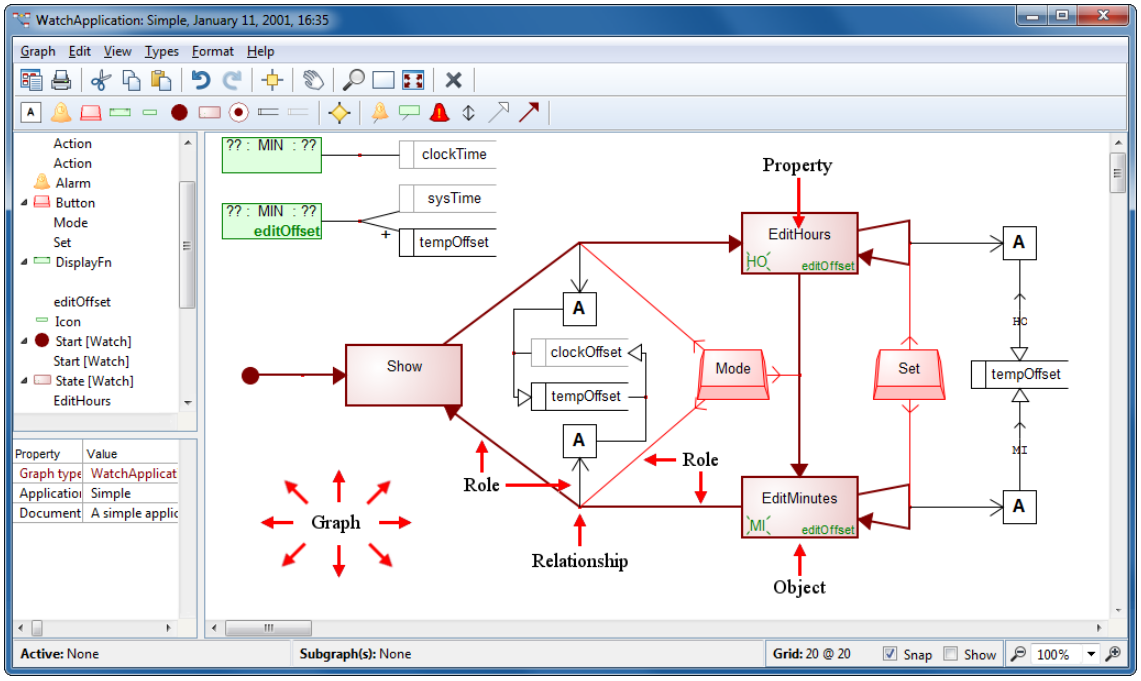


Figure 15: MetaEdit+ User Interface (taken from [52])

3.3.2.3 GME

The GME (Generic Modeling Environment) is a configurable toolkit for creating domain-specific modeling and program synthesis environments. The configuration is accomplished through metamodels specifying the modeling language of the application domain. The metamodeling language is based on the UML class diagram notation and OCL constraints. The metamodels specifying the modeling language are used to automatically generate the target domain-specific environment. The generated domain-specific environment is then used to build domain models that are stored in a model database or in XML format. GME has a modular, extensible architecture that uses MS COM for integration. GME is easily extensible; external components can be written in any language that supports COM (C++, Visual Basic, C#, Python etc.) [53].

GME has been designed to support the MIC (Model Integrated Computing) methodology which is utilized in building embedded software systems. MIC is a way to develop systems while addressing problems of system integration and evolution [54].

To describe metamodels GME uses its own meta^{2}model called MetaGME. The following concepts are defined in MetaGME: *atom*, *model*, *reference*, *set* and *connection*. These are also called *first class objects* (or *FCOs*). Additional concept used by GME is called the *aspect*, which is a collection (subset) of elements that are shown to the modeler at once. It is also addressed as the model's viewpoint [54].

Creating metamodels is a little different in GME than in the previously described metamodeling platforms. GME uses a drawing approach, where one instantiates MetaGME elements on the canvas in a way very similar to creating UML class diagrams. Figure 16 shows the typical GME metamodeling user interface. One can see that the defined

metamodel utilizes the UML stereotype notation to distinguish which elements are instantiated from which MetaGME concept. The newly created elements are shown inside the GME Browser on the right side of the screen. Left side of the screen is occupied by the GME Part Browser which contains all the elements that can be currently dragged and dropped on the drawing canvas in the middle of the screen.

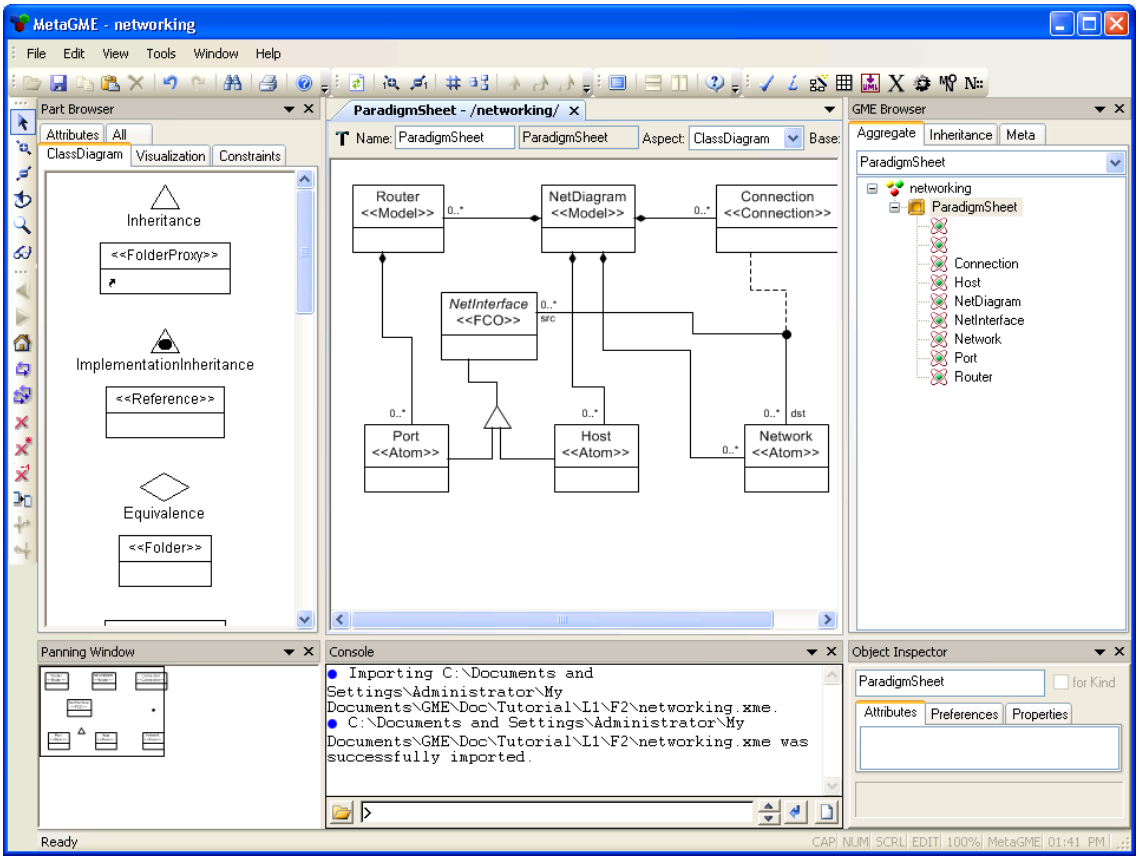


Figure 16: Metamodeling with GME (taken from [55])

More detail about the GME concepts, environment and its utilization, including the step by step explanations and screenshots can be found in [55].

3.3.2.4 Eclipse IDE with Modeling Frameworks

OMG's Meta-Object Facility (MOF), the open source Eclipse Modeling Framework (EMF), the Graphical Editing Framework (GEF) and the Graphical Modeling Framework (GMF) are no metamodeling platforms themselves. With MOF the OMG created a meta^{2}model standard, which provides a basis for defining modeling frameworks. UML is an example of instantiated metamodel of the MOF. The EMF which was influenced by MOF is an open source Java based modeling framework and code generation facility for building tools and other applications based on a structured data model [56]. The GEF provides technology for creating rich graphical editors and views [57]. The GMF provides generative components and a runtime infrastructure for developing graphical editors based on EMF and GEF [58]. It acts as a bridge between EMF and GEF. Together, these frameworks provide a possibility to create modeling tools.

3.4 Challenges

While researching the existing metamodeling environments (platforms and frameworks) some of the serious drawbacks have been revealed. These tools provide excellent support for development of modeling tools, but they are primarily designed to be used by an experienced developer. Only the basic features are sometimes usable and understandable by the application domain experts. The advanced features tend to get to programming intensive and require the knowledge most of the domain experts do not possess, thus need to acquire before they can implement their own modeling tools.

Here are some of the drawbacks that might prevent a domain expert from using these metamodeling tools for realization of complex modeling languages and modeling methods:

- ***Considerable time investments*** before choosing the right metamodeling tool for the job, because not all of them come with the same or even similar advanced features. For example, if analysis or simulation of models is per design an integral part of a modeling tool, a list of metamodeling platforms one can choose from shortens drastically. Even then, to pick the right one, it is necessary to learn how to utilize their functionality and invest time in testing them.
- ***Lack of domain specificity***, because some of these tools are built for programmers and software engineers, and not for domain experts. This can be noticed especially in application of metamodeling frameworks. Some of the more specialized tools, like ADOxx and MetaEdit+, have made considerable investments to be more domain expert friendly, by allowing a “no coding” approach. However, these are currently only an exception to the general rule.
- ***Platform lock-in is unavoidable***, meaning that when development has been started on one metamodeling platform, it is very hard to transfer what has been done to the new metamodeling platform. Complete reimplementation is necessary. OMG has made steps in the right direction by specifying the XML Metadata Interchange (XMI) format. However, only a part of a modeling language can be stored this way. Mainly, its abstract syntax.

The main challenge is to remove as many drawbacks from the currently used techniques for realization of modeling languages and modeling methods. This dissertation tries to do so by introducing a domain-specific language in the modeling tool development process – not as a replacement, but as an augmentation to the well-known metamodeling techniques.

4 Research Problem

The research on the modeling tool realization process has provided some insights into the issues of the current metamodeling techniques. However, to reach a more informed conclusion, the state of the art needed to be evaluated by an unbiased group of domain experts which are not familiar with the metamodeling techniques. For this purpose, a task has been designed and given to the group of Business Informatics Master's students to solve. The purpose was to search for additional challenges that could not be identified as part of the investigation of related work. In this case, students had the role of domain experts and were using various metamodeling technologies to implement their own modeling tools. Afterwards, students were given a second task, in which they had to evaluate various domain-specific languages and identify the key features which argue for and against the utilization of DSLs in a development process. The first study provided clear results and a good identification of problems present in current metamodeling tools. The other study helped to enforce the decision to introduce DSLs as a part of the modeling tools development process.

The research environment and its limitations will be described in the next section. The results from both of the studies, which ultimately helped in shaping the research problem definition and the research methodology, are presented as well.

4.1 Environment

The research at hand has taken place at the University of Vienna, Faculty of Computer Science, in the Knowledge Engineering (KE) research group. The aim of the group is, and has been for many years, to research, develop and use metamodeling techniques and tools. The group has initiated an initiative called Open Model Initiative (OMI) in 2008. A couple of years later, in 2012, a laboratory for realization of modeling methods has been established – the OMiLAB, which has a physical presence located at the Faculty of Computer Science in Vienna. Large portion of the groups research, theoretical and practical, is based on the ADOxx metamodeling platform, which has been used to realize around thirty modeling methods in the last couple of years. These tools have found their use in academia, as well as a part of industry projects. Most of the produced modeling tools are hosted on the OMiLAB website.

KE research group has a major role in organizing and holding courses that tackle with the issues of knowledge engineering and metamodeling. Thus, both of the previously mentioned studies have been conducted as a part of the Master's degree course titled *"Metamodeling"*. The studies have been mandatory and graded according to the provided results. Twenty-nine students participated in these studies. They were organized in eleven teams composed of two or three participants.

4.1.1 Study 1: Metamodeling Tools

This study was focused on discovering and comparing various features of metamodeling tools. The features have been extracted by using various metamodeling tools to implement fully working modeling tools which realize a representative modeling method (meaning the one that contains all the important artifacts).

The task for the study was formulated as following:

> *"Find one metamodeling tool, describe it shortly and present its features. Demonstrate how one can implement a simple modeling method or a modeling language utilizing the chosen tool. Based on your experience gained by implementing this simple example, what are the advantages and the disadvantages of using the tool you picked for the task?"*

The students were given a list of important features with a short description they could use for the evaluation of a metamodeling tool:

- **Metamodel definition**. Ways of defining a metamodel, for example a class diagram, table, matrix, tree, or code.
- **Constraint definition**. Ways of defining various constraints on the metamodel, such as constraint language like OCL, configuration dialogs, or code.
- **Notation definition**. What kind of notation does a tool support: graphical, textual, or both? What mechanisms can be used to define it: a DSL like SVG, integrated drawing editors, or importing of pictures in a specific format.
- **Attribute representation**. Ways of interacting with the values of the attributes and controlling their representation in the modeling editor.
- **Modeling editor creation**. Ways of creating a modeling editor, such as interpretation, or generation.
- **Reusability**. Support for reusing objects created by the tool, like notations, parts of a metamodel, or algorithms.
- **License**. Under what circumstances can a tool be used and what permissions are needed to use it. For example, open source tool, open use tool, or commercial tool.

Every team had to pick a different tool. Students had to search for the tools themselves. This helped to determine the availability of metamodeling tools, as well as what is considered a metamodeling tool by a novice. The key terms used in search were: modeling platform, metamodeling platform, modeling framework, metamodeling framework, language workbench, and meta-programming.

The following eleven tools were selected by the students:

- GME: Generic Modeling Environment
- EMF: Eclipse Modeling Framework
- VSVM SDK: Microsoft Visual Studio Visualization & Modeling SDK
- ConceptBase
- Poseidon for DSLs
- Xtext
- Spoofax
- MPS: Meta Programming System
- MetaEdit+
- Obeo Designer
- KM3: Kernel Meta Meta Model

The list is a mix of metamodeling platforms, metamodeling frameworks, and domain-specific language frameworks. It serves to demonstrate what is considered to be a metamodeling tool. Students could not choose the ADOxx metamodeling platform, because they were already familiar with it, as it was used to present and learn various metamodeling concepts during the *"Metamodeling"* course.

Using the research results provided by this study a matrix has been constructed (see Figure 17). In this matrix several important metamodeling tools' features are shown. Take note that this is not a full list of feature, but a compilation of the most important ones which are concerned with the realization of modeling methods.

Approach / Platform	Modeling Method Artifacts				Modeling Tool			
	Structure	Operations	Notation	Attributes	Modeling Editor Creation	Repository Support	IDE Support & Integration	Reusability
ADOxx	Tree-based [G]	ADOscript [T]	GraphRep [T]	Notebook [T]	Interpretation	Yes	ADOxx Web Services	Partial
MetaEdit+	Dialog-based [G]	Dialog-based [G]	Graphical Editor [G]	Properties Window [T]	Interpretation	Yes	Eclipse	Partial
GME	Class Diagram [G]	OCL [T]	Images (BMP) [I]	Property Window [T]	Interpretation	No	Visual Studio, COM	Yes
Obeo Designer	Class Diagram (Ecore) [G]	Java [T]	Images (BMP) [I]	Properties Window [T]	Generation	Yes	Eclipse	Partial
Visual Studio Visualization & Modeling SDK	Class Diagram [G]	C# [T]	Geometry Shape, Image Shape [G]	Properties Window [T]	Generation	No	Visual Studio and .NET	Yes
Poseidon for DSLs	Class Diagram [G]	Predefined Constraints [T]	Shapes [T]	Attributes Window [T]	Generation	No	Eclipse	Yes
ConceptBase	View [G/T]	FOL formulas [T]	Views [G/T]	Text [T]	-	Yes	Java, C, Prolog Plugins	Yes
Spoofax	Production rules [T]	Production Rules [T]	Presentational Services [T]	-	Generation	No	Eclipse	Yes
MPS	BaseLanguage, AST [T]	BaseLanguage [T]	BaseLanguage [T]	-	Generation	No	Eclipse, Java	Yes
Xtext	EBNF-like grammar [T]	Xtend [T]	EBNF-like grammar [T]	-	Generation	No	Eclipse	Yes
KM3	Text [T]	-	-	Text [T]	-	-	-	-

Figure 17: Metamodeling Tools' Feature Matrix

One can see what kind of different techniques are used to cover the specific parts of the modeling method definition, such as its structure, operations, notations and attributes. The symbols [T] or [G] in the matrix serve to express if an approach is textual, which is indicated by [T], or graphical, which is indicated by [G]. The dominant way to define the structure seems to be in form of a class diagram. For the notation, two techniques prevail: a graphical editor, and a domain specific language for graphical objects (typically similar to SVG). Attributes are generally defined in so called attribute or property windows which look similar as property windows of the well-known IDEs, such as Eclipse or Visual Studio. There is no common way of defining operations on a modeling method's metamodel. Every tool possesses different techniques to define operations. These techniques come in form of various domain specific languages, or scripting languages.

Additionally, these tools have different approaches when it comes to modeling editor creation. Most common ones are interpretation of configuration files and code generation. In case of interpretation, metamodeling tools serve as a host to the modeling tool, while code generation approach creates new standalone artifacts which can be installed as add-ons. The latter approach is generally used by metamodeling frameworks (such as Eclipse-based frameworks), because they do not contain all the necessary artifacts to create stand-alone modeling tools, but are dependent on the particular IDE.

There are three more properties of metamodeling tools that are worth mentioning: repository support, integration with other IDEs and reusability. Repository is used as a mean to structurally save modeling method definitions, as well as models created with them. IDE integration is a helpful feature that allows us to extend the functionality of a metamodeling tool or to utilize it from external sources (API calls). Reusability plays important role in the development of modeling methods, because the implementation effort decreases with time, as the number of already available modeling method artifacts increases. It is relevant to notice that metamodeling platforms (such as ADOxx and MetaEdit+) are lacking in the area of artifact reusability when compared to metamodeling frameworks.

The general conclusion that can be made from this brief study is that most of the metamodeling technologies implement the same metamodeling concepts, but the implementation itself varies from technology to technology. Therefore, the creation and manipulation of metamodeling concepts is done differently on almost every metamodeling tool. This also encourages forming of communities around a particular metamodeling technology, which is not a bad thing by itself. But in a long run, simple metamodeling concepts like *metamodel*, *class*, *attribute*, *model type*, or *view* gain a specific meaning in the specific community. This meaning is influenced by the metamodeling technology members of a community use to realize modeling languages and modeling methods. For example, the same concept is addressed as *class* in one community, and as *atom*, or *object* in another. The naming of concepts would not present an issue if the members of a larger metamodeling community, where all of the technology communities belong, are aware that the semantics are the same.

The present situation is making it harder to switch metamodeling tools and reimplement modeling tools on a different metamodeling technology, even if the new technology is providing exactly the functionality we need in our modeling tool. On the road to the solution to this problem, one needs to solve the challenge of platform dependency almost every metamodeling technology has. One way is to introduce a concept that is by definition platform independent – a computer language, particularly a domain-specific language. In an ideal situation, realization of a modeling language or a modeling method in a DSL can be translated to the metamodeling technology of our choice, which can then create a modeling tool. In case one wants to switch metamodeling tools, there is no need for reimplementation, just translation of DSL code to the new metamodeling tool.

4.1.2 Study 2: Domain-Specific Languages

For the purpose of testing the expressivity and usability of domain-specific languages as a concept, the following task has been given to the students:

> *"Describe a DSL of your choice, including its general syntax and semantics. Use this DSL to create a small example which illustrates DSLs main features and usability."*

Students have chosen to describe the following DSLs:

- SQL (Structured Query Language)
- DSD 2.0 (Document Structure Description)
- VHDL (VHSIC Hardware Description Language, where VHSIC stands for very high speed integrated circuits)
- Groovy
- HTML (HyperText Markup Language)
- MediaWiki
- POV-Ray (Persistence of Vision Raytracer)

Some of the students did not participate in this task, so the number of teams has been lowered from eleven that were present in the first study, to seven. Therefore, the seven selected DSLs.

The list of selected DSLs contains several very well-known DSLs, like SQL, VHDL and HTML. One can also note that the majority of the DSLs on this short list are some form of text markup languages, such as DSD 2.0 (based on XML), HTML, and MediaWiki, then we have a ray tracing program, POV-Ray, that includes a Turing complete scene description language (SDL), and Groovy, which is considered to be a dynamic scripting language for the Java platform. Groovy itself, because of its support for metaprogramming, is used to create internal or embedded DSLs.

It can be observed that a DSL always comes with some sort of an execution engine – a computer program capable of running programs described by a DSL. Because of it, a DSL is sometimes identified with its execution engine. POV-Ray, MATLAB, Mathematica, and MediaWiki are all examples of computer programs which are considered to be DSLs, or they include a DSL which is then named after them.

In conclusion, for a DSL to be useful, it is not sufficient to design it, which means only to specify its syntax and semantics. A DSL needs to be implemented as well. It needs to come with a development environment to be useful to the end user. Thus, an environment supporting the development of desired artifacts by using a DSL is also a part of our research problem definition. Secondary, it will provide the possibility to evaluate the solution through prototyping and test cases.

4.2 Problem Definition

The studies conducted with the students helped us in identifying currently problematic and to-be-improved areas in the metamodeling research. Therefore, the research problem tackled in this dissertation was stated as:

> The aim is to research, design, develop and evaluate tools that improve the usability of metamodeling techniques and the productivity of modeling tool engineers by providing the following advantages:
>
> - Domain-specific features
> - Platform independence
> - Reduced complexity
> - Shorter learning curve
> - Self-documenting code

Shortly, the problem can be defined as *“enable language-oriented modeling method engineering”* and its solution lies in the accomplishment of the following objectives:

1. Design a domain-specific language for describing modeling languages and modeling methods
2. Provide an integrated development environment supporting the designed language
3. Implement a translator (e.g., compiler) for a specific metamodeling technology (e.g., platform, framework, …)
4. Evaluate the domain-specific language, as well as its integrated development environment on test cases and controlled usability experiments

5 Research Methodology

Having identified the research problem and described the environment in which the research takes place, one can select a research methodology that will best direct and describe the way the defined research problems are addressed. There are many existing frameworks and guidelines on how to tackle with information systems research and development. Some of these frameworks have been evolving for more than 30 years, but are very similar to each other and contain variations on the following core processes:

1. ***Conceptualization***: create constructs which constitute the vocabulary which is used to describe problems and specify solutions within the domain.
2. ***Implementation***: build the system based on the conceptualized constructs.
3. ***Evaluation***: determine the system's performance and usability.

In this dissertation a variation of the methodology applied in the research work preceding the development of one of the most established metamodeling tools, MetaEdit+, was utilized [59]. In time when MetaEdit+ was in development, agile and iterative approaches were not mature enough. Therefore, the waterfall-based process was used, where one does not go back to the previous step once it has been completed. Thus, some modifications to the methodology were needed to make it agile and iterative, as this was required for the current research project.

5.1 Choice and Description of Methodology

The research in this dissertation belongs in the following two areas: information system development and computer (programming) language theory. The project as a whole can be viewed as an information system development project. However, the domain-specific language developed within the project applies the programming language theory best practices. Thus, one research area is encapsulated inside the other. Therefore, a specific research methodology is needed.

Wynekoop and Russo have done an extensive research on system development methodologies, with the purpose to provide information needed for evaluation, selection and development of methodologies in a changing environment [60]. The framework they developed is based on these two dimensions: research purpose and research methods. Research purpose is divided into categories: (1) use, (2) selection, development and adaptation, and (3) evaluation. Research methods are classified in nine categories: (1) normative writings, (2) laboratory research, (3) surveys, (4) field inquiries, (5) case studies, (6) action research, (7) interpretative research, (8) descriptive research, and (9) practice descriptions. All of the mentioned methods are a part of the scientific method, which cannot entirely be used in this research. It is lacking guidelines

on how to proceed with the construction of a prototype and its evaluation. The research method needs to possess means to guide us through the creativity-based engineering process. At some point the research findings need to be applied to extend, or to make a new generation of metamodeling tools. Wynekoop's and Russo's study provides us with significant amount of information about various scientific methods, but none of them align with the needs of this research project – to conceptualize, implement and evaluate an information system.

On the other hand, the framework developed by Nunamaker and Chen is particularly designed for research involving system development and construction [61]. Thus, its modified version has been applied throughout this research project.

This section is mostly taken from the paper Nunamaker and Chen have published about system development in information systems research [61]. The findings presented in this paper were later elaborated in [62], and [63]. According to their framework, a system building process consists of the following stages:

1. ***Construct a conceptual framework***. This process consists of four tasks that should be carried out: (1) state a meaningful research question, (2) investigate the systems functionalities and requirements, (3) understand the systems building blocks, and (4) study the relevant disciplines for new approaches and ideas.

   Every research project should start with a meaningful research question (or questions) that describe the research problem. When the proposed solution to the research problem cannot be fully validated mathematically or tested empirically, or if it proposes a new approach, it is mandatory to find different means to confirm its validity. One way of doing this is by constructing a conceptual framework based on the new methods, techniques, or design. During this process, the research problem will become more concise and coherent and system's functionalities and requirements will be revealed.

2. ***Develop the system architecture***. This consists of following tasks: (1) develop a unique architecture design, and (2) define functionalities of systems components and interrelationships among them.

   Based on the conceptual framework constructed in the previous phase, an acceptable system architecture, which provides a road map for the system's building process, needs to be established. It puts the system components into the correct perspective, specifies the system functionalities, and defines the structural relationships and dynamic interactions among them. The constraints given by the environment must be identified. The objectives of the development efforts must be clear. The functionalities of the resulting system need to achieve the stated objectives. The requirements revealed in the previous phase need to be clearly defined in a way that they are measurable and, thus, can be validated at the system's evaluation stage.

3. ***Analyze and design the system***. The following tasks should be carried out: (1) design the processes that execute systems functions, and (2) develop alternative solutions and choose one appropriate (currently the best) solution.

   Design is the arguably most important part of a system development process. Design involves the understanding of the studied utilization domain, the application of relevant scientific and technical knowledge, the creation of various alternatives, and the synthesis and evaluation of proposed alternative solutions. Design specifications will be used as a blueprint for the implementation of the system. For a software development project, design of data structures, databases, or knowledge bases, are determined in this phase. For a computer language development project, syntax and semantics of a language needs to be formally defined. The system modules and functions also should be specified at this time after alternatives have been proposed and explored and final design decisions have been made.

4. ***Build the (prototype) system***. By doing this one should (1) learn about the concepts, frameworks, and design through the systems building process, and (2) gain insights about the problems and complexity of the system.

   Implementation of a system is used to demonstrate the feasibility of the design and the usability of the functionalities of a system. Typically, the prototype's functionalities are limited to the most important ones, but the system has to run and has to be testable. The process of implementing a working system can provide insights into the advantages and disadvantages of the concepts, the frameworks, and the chosen design alternatives. The accumulated experience and knowledge will be helpful in case the system has to be redesigned. Building a prototype is a crucial stage in the information system development. Without a prototype, empirical studies of the functionalities and the usability could not be performed, because these are only executable on a running system.

5. ***Observe and evaluate the system***. The following step consists of: (1) observation of the utilization of the system by case study or field study, (2) evaluation of the system by laboratory or field experiment, (3) development of new theories based on the observation and evaluation of the system's usage, and (4) consolidation of the learned experiences.

   Once the system is build, researchers can test its performance and usability as stated in the requirements definition phase, as well as observe its impacts on individuals, groups, or organizations. The test results should be interpreted and evaluated based on the conceptual framework and the requirements of the system defined at the earlier stages. These results can indicate if the system is really solving the research problem it was built to solve, and at what efficiency. Development of a system is an iterative and evolutionary process. Experiences gained from development of the system usually will lead to the further develop-

ment of the system or even the discovery of a new theory to explain observed new phenomena.

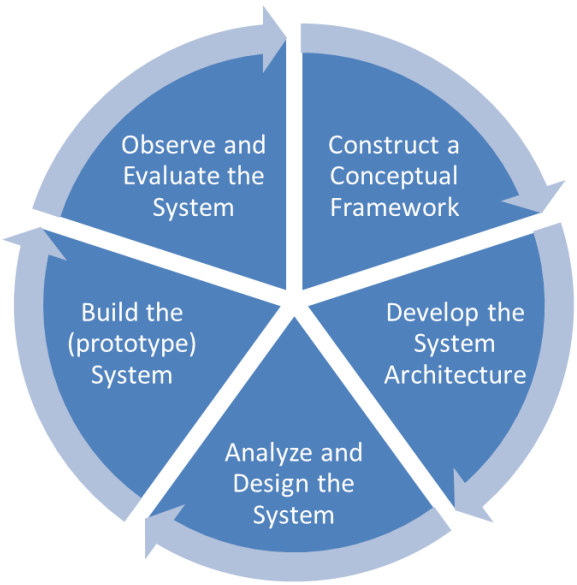


Figure 18: Information Systems Development Framework (adapted from [61])

Figure 18 shows the information systems development framework and all of its phases. The figure illustrates a closed circle, which indicates that all of the phases can be iterated through and the development of the system is not done when the fifth phase (observe and evaluate the system) is complete, but one can use the output of that phase to change the initial conceptual framework, the architecture, the design and the prototype as well, and then re-evaluate the system.

The use of system development as a research methodology in information systems should conform to the following five criteria [59]:

- The purpose is to study an important phenomenon in areas of information systems and provide solutions through system building.
- The results have significant contribution to the domain and can be utilized in academia, as well as in industry.
- The system is testable against all the stated objectives and requirements.
- The new system can provide better solutions to certain problems than existing systems.
- Experiences and design expertise learned from building the system can be generalized so that they can be used in other situations.

In every phase of the system development process, researchers gain insights about a domain that will lead to changing some decisions made in previous phases. When the developed system is a software system, software engineering methods and techniques

should be used to improve the quality of both the development process and the research results.

The Nunamaker and Chens information systems research framework is well suited for the research this dissertation is conducting, because:

- It conforms to the research problem and objectives.
- In its steps the three core system development processes are covered: conceptualization, implementation, and evaluation.
- The research fulfills the five criteria mentioned in the framework.
- The research aims to produce a software product.

Therefore, the described system development research methodology is a good choice for this research project.

5.2 Application of the Methodology in this Research

Before utilizing the system development methodology from Nunamaker and Chen, a general research problem has already been stated and an overview in related and competitive research has already been conducted.

To construct a conceptual framework, an analysis of existing metamodeling technologies has been conducted. This analysis particularly targeted metamodeling platforms, such as ADOxx, MetaEdit+, and GME, and metamodeling frameworks such as Visual Studio Visualization and Modeling SDK, and Eclipse EMF. Advantages and disadvantages of metamodeling platforms provided the details used in specifying the research problem in a concise and precise manner. Additionally, metamodeling platforms as software systems themselves gave an insight on how these systems are built, and more importantly, how they can be extended. The concepts extracted during the metamodeling platform analysis served as a backbone for the conceptual framework by providing key requirements metamodeling technology should satisfy.

The development of a system's architecture was guided by the idea about augmenting the existing metamodeling platforms with domain-specific concepts. Therefore, it was important to bring together a metamodeling platform and a domain-specific language. The key concern was the possibility of writing code which could be run on multiple platforms.

The first step taken in designing the system was the specification of a domain-specific language. The syntax and semantics of a language have been defined. Afterwards, a way of communication between the most important two system components – metamodeling platform and domain-specific language – has been specified.

The specifications produced in the design phase are implemented as a prototype system. There were a couple of iterations between this phase and the design phase, each

time when new insights have arisen. Additionally, an integrated development environment had to be developed to support the designed domain-specific language.

After the prototype system has been built, it has been observed and evaluated. Multiple test cases were used to test the system's behavior and usability. The conducted evaluation scenarios are described late on in a dedicated chapter. The iterations between this phase and the previous phase – the build phase – were common. The reasons behind it were various bug fixes and introduction of functionalities that were shown to be lacking while conducting some of the tests.

6 Metamodeling Platforms

6.1 Introduction

A metamodeling platform, as a pinnacle of metamodeling technology, is an important part of the research at hand. In this chapter the main focus is on a generic metamodeling platform. The existing platforms are mentioned only as an example. It is important, because the conceptual framework depends on the concepts metamodeling platforms implement, such as a meta^{2}model, means of defining abstract and concrete syntax of a modeling language, means of implementing modeling algorithms, and model repositories.

To be able to augment the development process with domain-specific concepts using a domain-specific language, one needs to understands how is a metamodeling platform built, and how is it utilized to develop modeling tools, as well as what are the advantages the platform provides. The disadvantages that come with the use of a metamodeling platform are crucial in proposing and providing a better metamodeling solution.

6.2 Requirements

After the research conducted on several metamodeling platforms, some of the most important requirements have been elicited. These requirements have been divided into functional and non-functional requirements. The ones mentioned here are only the most important ones which should be fulfilled by every metamodeling platform.

The requirements extraction was a multi-step process that was repeated for every metamodeling platform under test. The process started by reading the available documentation and user manuals, followed by learning the platform's functionality and trying to implement a simple pseudo modeling language (by following a provided tutorial if applicable).

6.2.1 Seven Key Functional Requirements

In the field of software engineering, a functional requirement defines a function of a software system or its component [64]. Functional requirements may be various calculations, technical details, data manipulation and processing, and other specific functionalities that define what a system must accomplish. A metamodeling platform is a complex software system; therefore, its functionality needs to be defined as detailed as possible.

To expose the most important metamodeling platform functional requirements related to the proposed DSL design, several metamodel-based implementations of modeling

methods were thoroughly analyzed. All of these implementations are hosted on the OMiLAB website [65]. Here is the full list: BEN, BIM, CIDOC, eduWeaver, EKD, Horus, IMP 2.0, *i**, OM*i**T, InSeMeMo, MeLCa, OKM, Secure Tropos, UML, PetriNets, MoSeS4eGov, PROMOTE, SeMFIS, and VLML. Figure 19 contains a graph showing the complexity of the modeling methods expressed through the number of their basic elements, mainly classes, relationships and model types. Figure 20 contains numerical values of the previously mentioned graph (for the orientation purpose). As an output of this analysis several key metamodeling platform requirements were extracted.

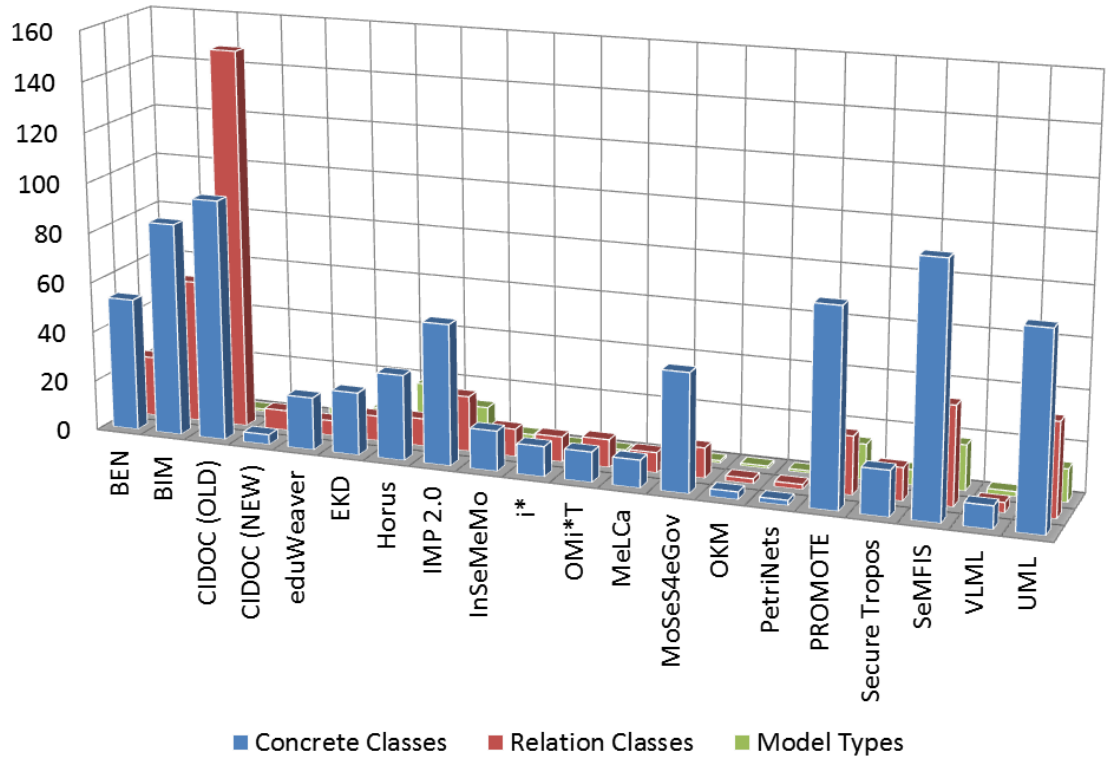


Figure 19: OMiLAB Modeling Method Statistics (Graph)

	BEN	BIM	CIDOC (OLD)	CIDOC (NEW)	eduWeaver	EKD	IMP 2.0	InSeMeMo	OMi*T	i*
Concrete Classes	53	85	96	4	21	25	56	16	12	12
Relation Classes	24	57	151	8	6	10	22	11	10	11
Model Types	19	4	1	1	5	7	12	3	1	1

	MeLCa	MoSeS4eGov	OKM	PetriNets	PROMOTE	Secure Tropos	SeMFIS	VLML	UML	Horus
Concrete Classes	11	47	3	2	78	18	99	9	77	34
Relation Classes	8	12	2	2	23	13	39	4	37	11
Model Types	1	1	1	1	14	6	18	2	13	19

Figure 20: OMiLAB Modeling Method Statistics (Table)

The seven key groups of functional requirements have been divided as follows: (1) meta^{2}model, (2) abstract syntax designer, (3) dynamic notation mapper, (4) dynamic notation designer, (5) modeling procedure designer, (6) open APIs and algorithm libraries, and (7) model repository requirements. These requirements can be implemented as metamodeling platform software modules.

6.2.1.1 Meta^{2}model

As the name indicates, metamodeling platforms utilize the metamodeling approach for development of graphical modeling languages and modeling methods. They are typically based on a complex meta^{2}model which contains the core concepts and functionality. By instantiating the meta^{2}model with the domain-concepts, a metamodel is defined, which specifies the modeling language's abstract syntax, and semantics. The first functional requirement for a metamodeling platform is to have a meta^{2}model that is generic enough to be able to instantiate concepts from a wide range of domains, but at the same time to be sufficiently rich with meta-concepts, enabling detailed modeling language specification. After more than dozen modeling method implementations, it has been noted that a simple meta^{2}model containing only the basic abstractions (class, relationship, and attribute) is not powerful enough for expressing some of the real world modeling languages and modeling methods.

6.2.1.2 Abstract Syntax Designer

A sophisticated control mechanism enabling structured abstract syntax and semantics definition and manipulation for the concepts instantiated from the meta^{2}model is the second functional requirement. Its importance comes to play together with the rising complexity of large metamodels. Smaller ones could be managed with a simpler control mechanism, but managing abstract syntax and semantics of large complex metamodels would be near to impossible.

6.2.1.3 Dynamic Notation Mapper

The development of graphical modeling languages is more complex than the development of textual languages. Textual languages (e.g., textual programming languages, textual specification languages) can have their syntax specified in a textual form (e.g., EBNF), where, most of the times, abstract and concrete syntax are joined together and defined at the same place. In case of graphical modeling languages, it is common to have multiple notations associated with one modeling element, meaning that abstract syntax can have multiple concrete syntax representations. The third functional requirement is to provide a mapping mechanism between the abstract syntax and their graphical representation. The importance of this requirement is even more elevated, if one takes under consideration the possibility of modeling elements having multiple graphical representations at the same time (dynamic notation).

6.2.1.4 Dynamic Notation Designer

In modern modeling methods and modeling languages, graphical modeling elements are not only nice figures on the modeling canvas. Together with the graphical representation and underlying syntax and semantics, they also provide a complex interface between the user and the model, enabling predefined functionality (e.g., jump to other parts of a model, input values into variables, change graphical representation, start analysis or simulation, etc.). Being able to design appropriate graphical representations including the user interface embedded into modeling elements is the fourth functional requirement.

6.2.1.5 Modeling Procedure Designer

Modeling methods, in addition to the concepts contained in a modeling language, include several functional extensions: modeling procedures, modeling algorithms and mechanisms. A metamodeling platform needs to provide generic functionality for supporting these extensions by enabling out-of-the-box use or adaption of existing features according to the user needs.

Modeling procedures enforce the order in which modeling elements need to be used. In most cases this is not necessary, because one wants to give more freedom to the modeler (human doing the modeling). However, there exist modeling methods which strictly define the order one can use the modeling elements. For example, if one wants to model information security, one should provide physical (e.g., server locked in a room which provides optimal working environment and protection from natural disasters) before virtual security (e.g., firewall, access control, etc.). The enforcement of modeling procedures is the fifth metamodeling platform functional requirement.

6.2.1.6 Open APIs and Algorithm Libraries

Algorithms are the means which are used to define and implement additional functionality of a modeling method. To be able to use and reuse already present platform functionality for defining various algorithms and mechanism, one needs an interface to this functionality, typically realized as a well-documented and interoperable set of APIs. Sixth functional requirement is that the platform provides open means of interfacing with its functionality, and allowing the utilization of these means for implementing additional, user defined functions.

6.2.1.7 Model Repository

All of the previous requirements were connected with implementation of modeling methods. However, there is still one not directly connected with implementation, but with storing of data. A metamodeling platform needs a dedicated repository for storing a modeling method definition, and a second one for storing models defined by a modeling method. Repositories provide the possibility to reuse already defined modeling elements, track changes for both development of modeling methods and models, and propagate changes done on the modeling method layer to the model layer. Reposito-

ries are an essential part of a powerful metamodeling platform, and therefore, the seventh functional requirement.

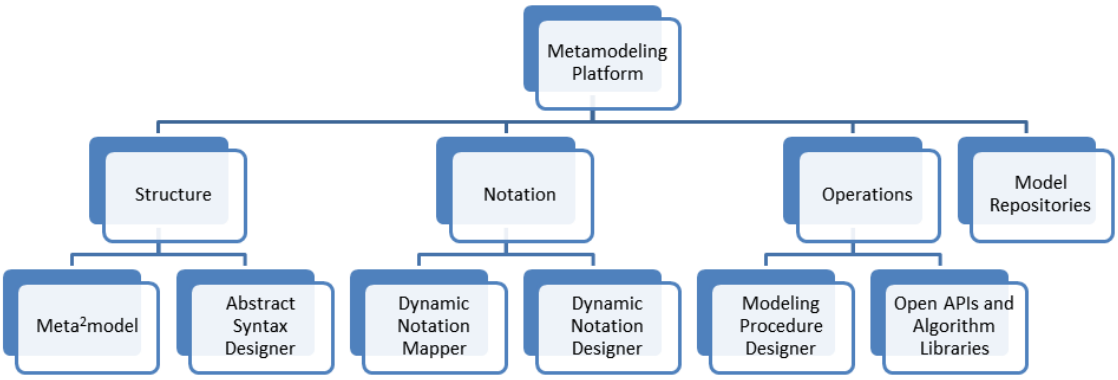


Figure 21: Metamodeling Platform Modules

Figure 21 shows a simplified implementation of a modeling platform according to the functional requirements. Each functional requirement is represented by a dedicated software module, seven in total (leafs in the diagram). Software modules are grouped corresponding to their functionality: structure, notation or operation. Model repositories do not have a dedicated group.

6.2.2 Important Non-functional Requirements

Along the several important functional requirements, there are still a couple of non-functional requirements every metamodeling platform should fulfill.

From many non-functional requirements [66], including accessibility, availability, compliance, reliability, security, usability, backup, documentation and others, there exist three very important for the metamodeling platforms: extensibility, interoperability, and scalability.

6.2.2.1 Extensibility

Extensibility [67] takes under consideration future growth. It is a measure of the ability to extend a system and the level of effort required to implement the extension. Extensions can be through addition of new functionality or through modification of existing functionality. It is of high importance for the current and future metamodeling platforms to provide mechanisms for change, while minimizing impact to existing functions. As a complex software system, metamodeling platforms should have a public application programming interface (API) that allows extension and modification of platform's behavior by developers who do not have access to the original source code.

6.2.2.2 Interoperability

Interoperability [68] is the ability of diverse systems to work together by exchanging information and using the information that has been exchanged. One of the means allowing metamodeling platforms to communicate between each other is in specifica-

tion of open standards. The products implementing the common protocols defined in the standard are thus interoperable by design. By providing users with a freedom to start their implementation of a modeling method on one platform, continue it on the second, and finish it on the third is a tangible benefit where one can choose a metamodeling platform, and have no fear that his implementation will not work on another platform. Currently, none of the existing metamodeling platforms fulfill this requirement.

6.2.2.3 Scalability

In today's world, demand for something can escalate very quickly from a very small to very large in a brief period of time. Scalability [69] allows handling a growing amount of work in a capable manner. This issue can be illustrated on an example where one provides an implemented modeling method as a service that runs in the cloud. At first, the metamodeling platform together with the underlying IT infrastructure needs to serve a dozens of concurrent users. However, in only a matter of days, number of users using a modeling method can rise exponentially. The platform needs to accommodate ad-hoc to support a larger number of users by providing higher throughput and be prepared to serve a higher number of concurrent users at the same time. Scalability is becoming increasingly important in today's delivery of software, as users require services that can be used immediately, without installing them locally on local machines.

7 Metamodeling Platform Applications

This chapter is dedicated to the established modeling method engineering practices where one utilizes metamodeling platforms to implement modeling tools. This approach has been used in most of the modeling tool projects that have been realized within the OMiLAB [65]. The study *"Very Lightweight Modeling Language (VLML): A Metamodel-based Implementation"* is a representative example on how is this approach applied in practice. Its focus is on the conceptualization of a modeling language from the given requirements and subsequent implementation of a modeling tool using the ADOxx metamodeling platform. The study was published in [70]. The following is an extended version.

7.1 Very Lightweight Modeling Language (VLML): A Metamodel-based Implementation

Very Lightweight Modeling Language (VLML) is designed as a semiformal and easily integrable with natural languages, by proposing modeling elements like: structure, relationship, influence, and flow. Its semiformal nature caused several issues during the metamodel design, making the implementation of the VLML-tool more challenging. In order to overcome these issues, new functional requirements have been defined for the ADOxx metamodeling platform. The VLML tool is an early prototype. Thus, it is fully functional and supports many of its envisioned concepts.

7.1.1 Introduction

Despite the efforts that went into development of requirements modeling languages, the vast majority of requirements specifications created today are still written in natural language, augmented with tables, pictures, and, increasingly, some model diagrams [71]. However, this is not an indication that currently commonly used modeling languages like UML [72] or ADORA [73] are unable to express complex requirements, but that they are very scarcely used in the early stages of requirements engineering [39]. This is the case, because requirements at an early stage are by their nature narrative and pictorial. Therefore, natural language augmented with pictures is the primary tool used to capture them [71]. Very Lightweight Modeling Language's (VLML) primary goal is to make early stage requirements specifications, not only analyzable by careful reading (current situation in the industrial early stage requirements engineering), but analyzable, editable, navigable, and traceable by providing a powerful tool support. Suffice to say, the modeling languages have a big advantage over textual languages, because of their ability to define models whose properties can be processed.

VLML is semiformal and easily integrable with natural languages. It contains modeling constructs for things that are hard to express textually: structure, relationship, influence

and flow. The most important VLML elements according to [74] are: objects, relationships, modifiers, context, missing/hidden information and enrichments. Objects are specialized into technical items, living items, connectors and pictures; relationships into static relationships, influences, flows and hierarchical structures; modifiers into external, fuzzy, multiple and boundary; enrichments into and-connector and or-connector.

The auxiliary goal of this study is to apply a modeling method conceptualization approach [75] in the presence of a language with vague, implicitly stated properties. VLML is an appropriate choice because it (1) contains relatively few modeling elements (see Figure 19), and (2) has a semiformal and partially unfinished specification [74].

The study continues with a brief introduction to the conceptualization, as a first phase in modeling method engineering. It is followed by a VLML metamodel design process, together with the challenges introduced in the conceptualization phase. In section *"Tool Implementation"* a short introduction to the ADOxx metamodeling platform (from an implementation point of view) is given (see [76] for more details), followed by a couple of examples produced by the implemented VLML tool. This study concludes with possible future work and reflections on the work that has been done so far.

7.1.2 Conceptualization

Before one can start with the actual modeling tool implementation, the modeling method, in this case VLML, is usually written in some form of a document (e.g., book, scientific publication, etc.), which demands further analysis. The concepts need to be extracted and transformed into a concrete, unambiguous, representation. This is done in the, so called, conceptualization phase, which precedes the implementation phase in the modeling method engineering process. Extraction of requirements and their shaping into a conceptual metamodel that contains the modeling method concepts is a crucial task method engineers need to accomplish. Insightful and careful mapping of concepts to a metamodel will, consequently, save us from a lot of trouble in the succeeding implementation phase, which is, thus, more predictable, easier and faster [77]. It is common that a metamodel goes through several iterations, until all the concepts are captured by it. After completion, conceptual model should be general enough for the implementation on a non-specific metamodeling platform. However, transforming it to an implementation-specific metamodel allows us to take advantage of the generic and advanced functionalities of a specific metamodeling platform. This also entails, that the implementation metamodel needs to be instantiated from the meta^{2}model of that particular platform.

The process covering the road from the modeling method general description to the tool implementation can be explained with a simple book-to-movie metaphor. At first we only have a book, collection of written thoughts meant to be consumed by the reader. The book alone is not enough to guide the movie production process. That is where four movie production stages come into play: (1) development, (2) pre-production, (3) production, and (4) post-production [9].

In development stage a story is selected, which may come from a book, another movie, real world event, etc. The first step is to organize a story into a structure of scenes and prepare a synopsis describing its mood and characters. Next, a screenwriter writes a screenplay, which broadens the synopsis with additional details, including the dialogue and stage direction [78]. It is common to rewrite the screenplay several times to improve dramatization, clarity, structure, characters, dialogue, and overall style. The screenplay can be viewed as equivalent to the conceptual metamodel, and the rewriting process is similar to the iteration process in our domain.

In pre-production, every step of actually creating the movie is carefully designed and planned [78]. Taking the screenplay as a starting point, concept artists and illustrators draw a storyboard. There are many responsibilities, e.g. the casting director finds actors; the location director finds and manages locations; the composer creates music; the costume designer creates clothing for the characters. Comparing the pre-production with the design of the implementation-specific metamodel, the storyboard itself would represent the metamodel. Music, costumes, scenery, actors, etc. would be the additional functionality supported by the specific metamodeling platform. Accommodating the conceptual metamodel for the specific platform enables the use of already present functionalities.

In the production phase the movie is shot and created. In post-production movie is assembled by the video editor. This is the final stage, after which the film is released to the public. Production is equivalent to the implementation phase, where modeling method is being implemented on a metamodeling platform, and post-production is equivalent to the deployment and distribution of a modeling tool.

The processes of realization of the VLML conceptual metamodel, as well as the implementation metamodel, are described in the following section.

7.1.3 Metamodels Design

Informal and semiformal modeling method specification has a general disposition to introduce new challenges into the conceptualization process. In these cases, one should take care to capture all that is possible and fill in the gaps with concepts that are only implicitly stated or even non-existent.

The following is a brief summary of issues that arose during the VLML conceptualization process together with the proposed solution given as an extracted conceptual and, later on, also as an implementation-specific metamodel.

The design requirements mentioned in [74] have been used as an input into the conceptualization process, and as an output a metamodel for the VLML has been designed. The main issues during this process were: (1) incompleteness of language specification, (2) no decomposition guidelines (possibility of models getting too big and cluttered with meaningless details), and (3) preparation of the metamodel for the implementation on the metamodeling platform.

One of the main challenges was to make the metamodel of the VLML as complete as needed for the upcoming implementation of the modeling tool. The issue of incompleteness comes from the fact that VLML is in an evolutionary stage, as is described in [74]:

> *"The specification in the appendix gives an impression of the look and feel of the language. This is not meant to be a complete and polished language design. Our intention is to make our vision of ULM more concrete, tangible and criticizable."*

Ultralightweight Modeling Language or ULM (not to be confused with UML; it is not a typo!) was the name VLML had in its early drafts that were produced during 2010. It was later renamed to Very Lightweight Modeling Language (VLML) in its subsequent publications [71].

The second issue was with VLML specification not defining any kind of model types – all modeling elements were grouped into one single representation. This kind of view on specification requirements modeled in VLML is very well suited for the ones that are viewing and analyzing the models. However, it is fairly difficult to work with for the ones producing the models. Taking the latter into consideration, refactoring of concepts was performed by dividing this single bunched up "model type" into two separate (but inter-referenced) model types: one containing the basic concepts, without all the details connected with them, and the other specifying the details lacking in the first one. That way a modeler can concentrate on a specific part of the requirements specification without all the clutter occupying his computer screen.

The third issue is not connected with VLML itself, but with the tooling process itself, meaning that the definition of a modeling language (design of a metamodel) had to be slightly modified to suit a specific meta^{2}model [10]. In our case, the VLML metamodel was modified for the implementation on the ADOxx meta^{2}model.

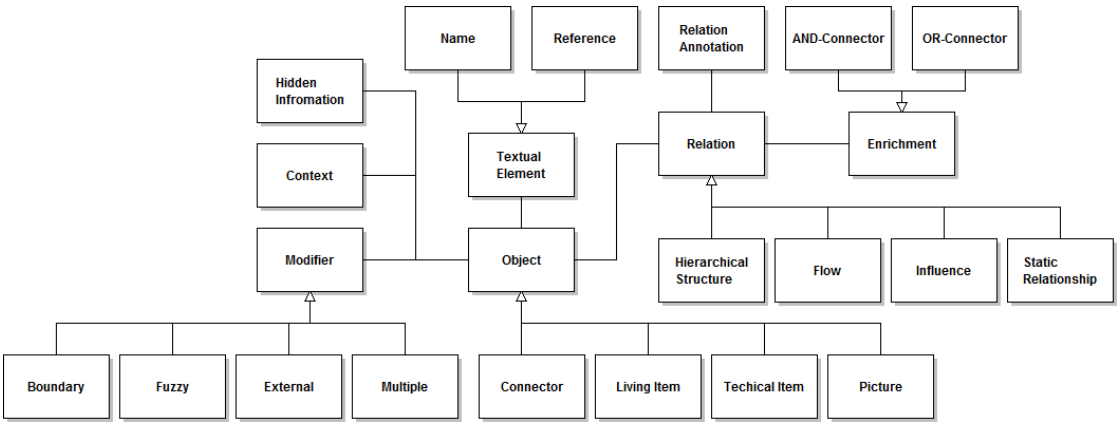


Figure 22: VLML Conceptual Metamodel

Figure 22 shows the VLML metamodel extracted from the preliminary language specification taken from [74]. Simplified UML class diagram notation is used to show various VLML elements and their relationships.

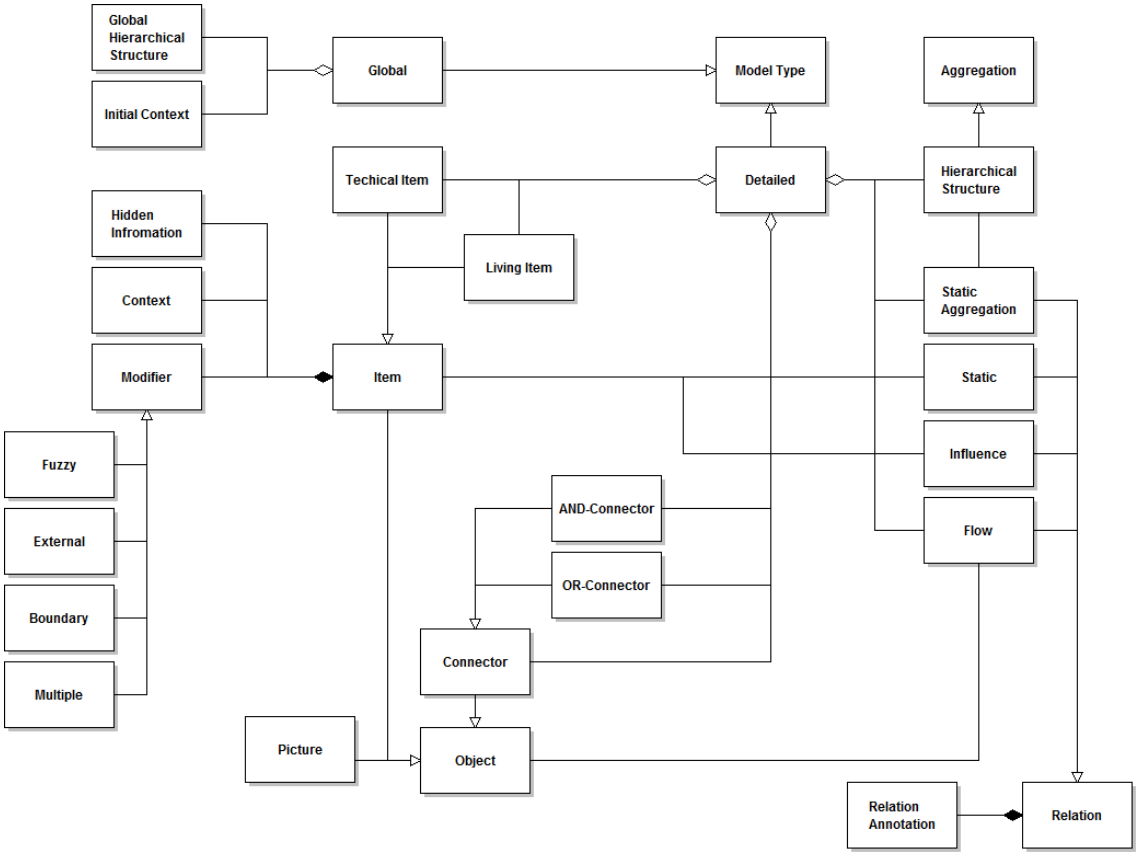


Figure 23: VLML Implementation Metamodel

Figure 23 shows the VLML metamodel produced in the conceptualization phase and modified for implementing a tool based on the ADOxx metamodeling platform. By comparing these metamodels, one can see the introduced changes and additions that were necessary for the viable tool implementation.

The most important changes to the original specification, when considering the modeling elements, are: (1) grouping into two model types (Global, and Detailed), (2) addition of elements (abstract elements: Item, Aggregation, and Model Type; concrete elements: Static Aggregation, Global Hierarchical Structure, and Initial Context), (3) removal of elements (Enrichment, Textual Element, Name – name is an integral part of every class, therefore it is not necessary to express it explicitly, and Reference), and (4) type change (AND-Connector, and OR-Connector are no longer a specialization of Enrichment, but specialization of Connector; Hierarchical Structure is no longer specialization of Relation, but specialization of Aggregation; Living Item and Technical Item are now specialization of Item, which is a specialization of Object).

New relationships between the modeling elements are introduced: (1) vague associations replaced with compositions (Hidden Information, Context and Modifier are now a part of Item; Relation Annotation is a part of Relation), (2) aggregations (used to aggregate modeling elements into model types), and (3) missing associations (between Item and Static, Item and Influence, Object and Flow, and Hierarchical Structure and Static Aggregation).

It can be noted both in Figure 22 and Figure 23 that relationships are depicted as relation classes (e.g., Static, Flow, etc.) that are connected with object classes (e.g., Item, Picture, etc.) only through associations. This representation is aligned to the ADOxx meta^{2}model, where both of these concepts are specialization of a class. The key difference between a class and a relation class in the ADOxx meta^{2}model is in two characteristics specific to a relation class: (1) it cannot be inherited, and (2) it contains two additional properties – from and to, which define the beginning (from) and the ending (to) part of the relation. There is also no multiplicity indicated on the associations, because none is specified in the language design. Therefore, it is taken for granted, for the purpose of the VLML prototype implementation, that every association has many-to-many relationship.

7.1.4 Tool Implementation

The VLML tool implementation is based on the ADOxx metamodeling platform and is one of the more than twenty modeling tool realizations which are utilizing the mentioned modeling method conceptualization process.

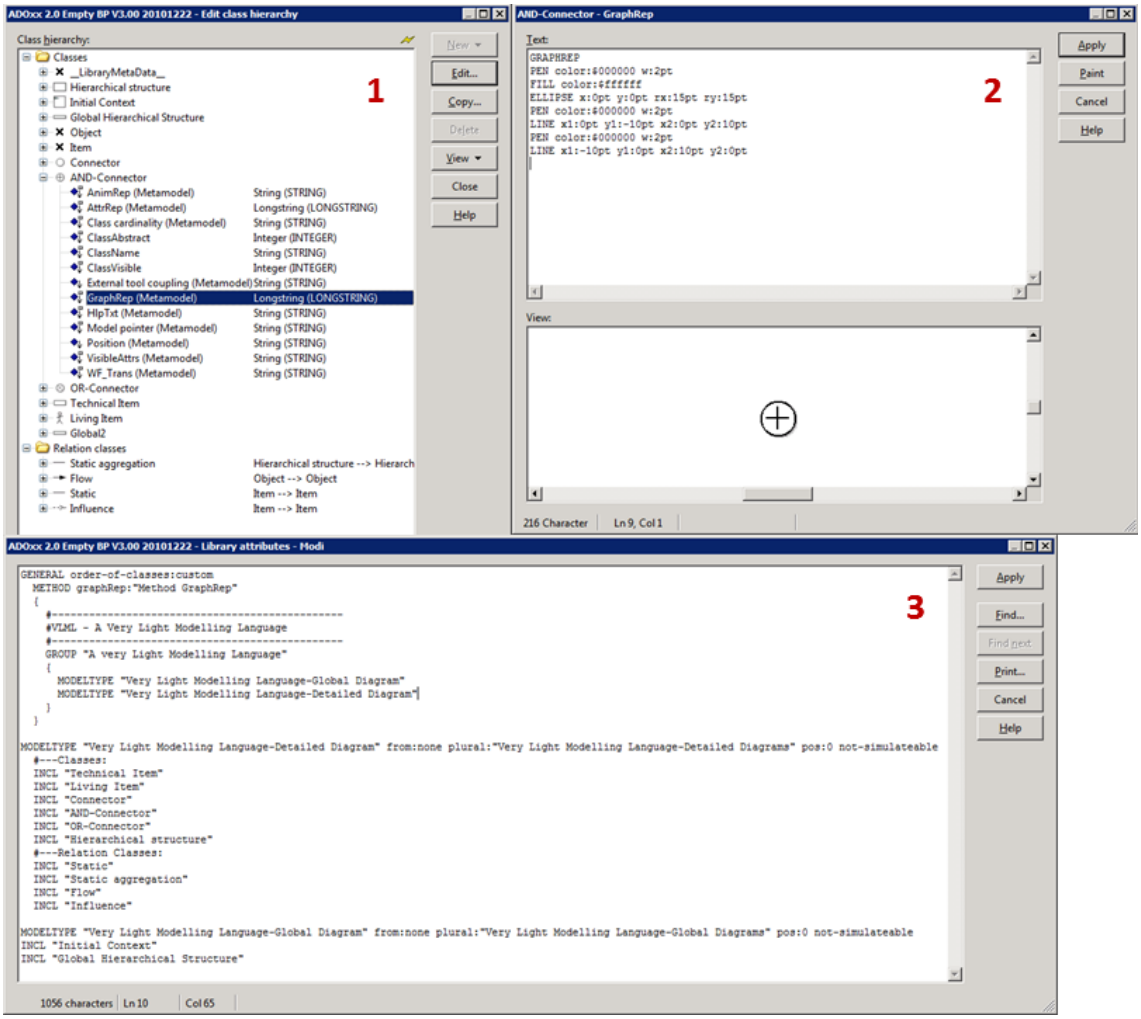


Figure 24: VLML Implementation View

ADOxx is a mature and extensible, repository-based metamodeling platform, which has been applied in various academic and industrial projects. It supports: (1) modeling languages by inheriting modeling concepts from a meta²model to define syntax, semantics and notation, (2) modeling mechanisms and algorithms by providing generic platform functionality that can be used or adapted by employing scripting possibilities, integration and interaction with third party add-ons, and (3) modeling procedures suggesting modeling steps related to the modeler's goals [76].

Figure 24 shows the implemented VLML metamodel inside the ADOxx (screenshot labeled with 1). One can see the hierarchical structure of metamodel elements, which have been instantiated from the ADOxx meta²model. FigureX also shows the GraphRep definition of one of the modeling elements (screenshot labeled with 2) and the definition of the two model types (screenshot labeled with 3). This is a brief overview of the ADOxx modeling tool implementation environment. There exist other editors and functionalities, but are not relevant to the example under consideration.

To preview the VLML modeling tool, parts of the short requirements engineering scenario taken from [76][7] (also used as a case study in [74]) are modeled (see Figure 25).

Figure 25 shows a couple of screenshots from the VLML modeling tool. In the top left corner one can see the use of Connectors, Technical Items and Hierarchical Structures, in the top right corner the use of the Global model type which aggregates all of the Detailed model types (the symbol […] denotes that the object can be expanded to show additional information, in this case the whole diagrams), in the bottom left corner the combined use of Technical Items and Living Items together with the appropriate Modifiers, and in the bottom right corner the use of Technical Items together with Modifiers and Influence relationships among them.

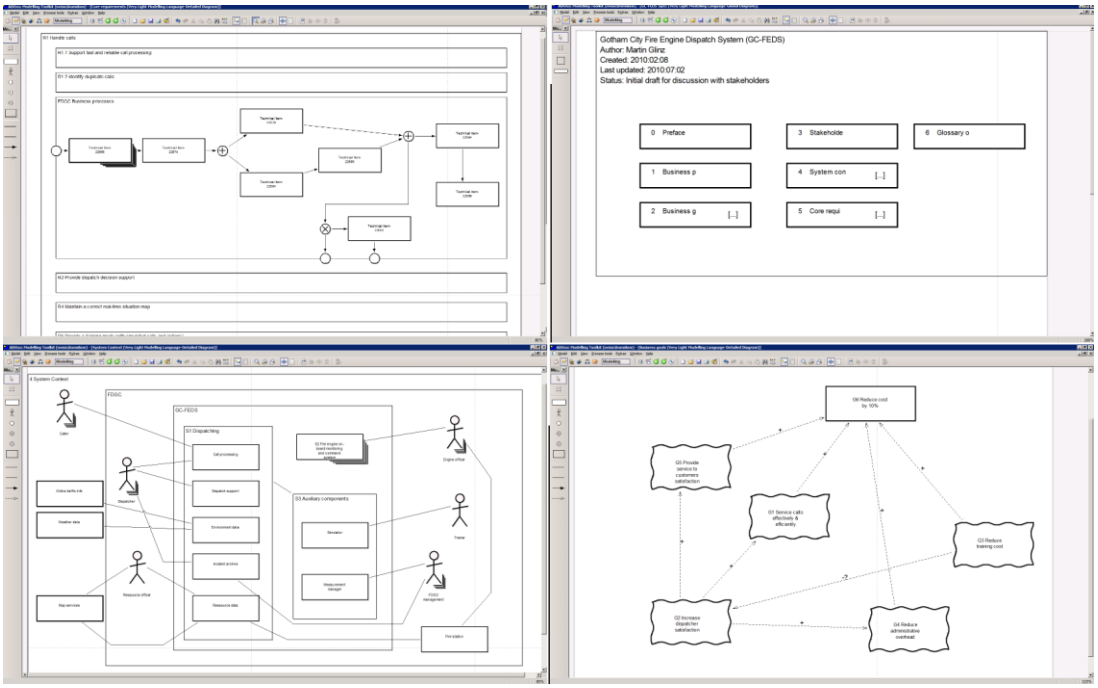


Figure 25: VLML Tool In Use

The tool also inherently supports all functionality provided by the ADOxx metamodeling platform, including: model explorer and navigator, different modeling views (e.g., diagram, table), a pane containing modeling components designed to make modeling easy and user friendly.

7.1.5 Conclusion

The first prototype of the VLML tool offers the full functionality of the described modeling language and is being developed throughout multiple iterations, each time updating the implementation metamodel. There are still open questions regarding the VLML language specification and some ideas under consideration regarding the extension and customization of the ADOxx platform to better suit the concept of VLML, not only as a language, but as a tool as well.

The most relevant open questions are directed toward the missing details in the language specification. Some of them have been partially answered for the purpose of making the implementation possible, but there are still some ambiguities that need additional explaining. For example, one cannot, using only the information contained in the language specification, entail: (1) what kind of relations (e.g., Static, Influence, Flow) does every object (e.g., Technical Item, Living Item, Picture) support, (2) possibility of objects to contain other objects (issue coming from the specification of the hierarchical structure in VLML), (3) do all objects support modifiers (e.g., Fuzzy, External, Boundary), and (4) which modifiers can be used in a combination. In future releases number of ambiguities will be lowered by providing a more explicit language specification. On the tool implementation side, algorithms for on-screen objects manipulation and positioning, annotation, and searching are needed to improve the overall user experience. One of the future considerations in this area is on how to make model type switching (from Global to Detailed, and vice versa) and inter-referencing as seamless as possible.

Implementing VLML on the ADOxx metamodeling platform has brought attention to the few insightful facts about the utilized modeling method conceptualization process. It is informal, time consuming, error prone and lacking documentation. There is also no tool support, other than paper & pencil, PowerPoint and Word. These are some of the reasons that gradually guided us into a decision to design a domain-specific language (DSL) which could solve some of the mentioned issues (informality and error proneness at least) and introduce new features like change tracking and version control. The proposed DSL will provide higher structured capabilities to the conceptualization process, and consequently, make the implementation of modeling methods more formal, powerful and easier to use.

Detailed information about the mentioned DSL and its development (including its conceptualization, implementation and evaluation) can be found in the following chapters.

8 MM-DSL

There are multiple ways one can pursue when developing modeling tools. The most common are the ones where modeling tool engineers implement by using multiple graphical editors and various programming languages to realize the requirements of a modeling method. Several approaches have already been discussed in previous chapters (see chapters 3, 6 and 7). In the state of the art, implementing artifacts such as abstract and concrete syntax or algorithms is linked to a specific technological platform. This motivated the development of a DSL, which entails its conceptualization (design and specification), implementation and evaluation. The proposed domain-specific language (MM-DSL) is based on a metamodeling approach, and it gives us the ability to be technology independent. With MM-DSL a specification for a modeling tool is programmed on an abstract level. The code can be compiled and executed on different metamodeling platforms.

This chapter holds a very detailed description of MM-DSL (Modeling Method Domain-Specific Language). The paper *"Developing Conceptual Modeling Tools Using a DSL"* gives a condensed overview of the work that will be presented in here. It has been published in [79].

8.1 Introduction

After thorough research in metamodeling technologies present in industry and academia, it was found out that many of them are built on the same concepts. One would think that this may allow seamless exchange of information between them. However, that is not the case. Competition, as well as an unsatisfactory state of terminology, has led to platforms that are typically stand-alone software which is locking-in the development. It is very hard, or even impossible, to migrate what has been done so far on one platform to the other. Because of the different terminology and approaches utilized by different metamodeling platform vendors, it is also a considerable time investment to learn, understand and use an alternative platform. The reasons for this kind of state of the art are not covered by this research work. If that were so, this would be a study in sociology and marketing. As computer engineers and computer scientists, it is not our job to change how people thing about the word. It is to propose and provide solutions to problems. The proposed solution to this particular problem is provided as a computer language – Modeling Method Domain-Specific Language or MM-DSL. The language that is designed to bridge the differences between various metamodeling technologies, and can be used as a common ground for implementation of modeling tools, regardless on which metamodeling platform will those modeling tools run on.

A brief scenario about how one utilizes MM-DSL in the development of modeling tools is the following:

1. Describe modeling method components using MM-DSL (this includes modeling language's abstract and concrete syntax, and, in case of modeling methods, it includes description of algorithms).
2. Translate MM-DSL code to the representation understandable by a metamodeling platform of choice.
3. Use the metamodeling platform to add functionality that is not possible to code with MM-DSL.
4. Generate a modeling tool using a metamodeling platform.

Using MM-DSL allows that everything that has been coded in it can be translated (e.g. compiled) to various metamodeling platforms. If additional functionality is added by using a metamodeling platform after the compilation has took place, it is not possible to convert it back to the MM-DSL code. However, it is possible to extend MM-DSL with additional features that will support the needed functionality.

As it can be seen from this brief MM-DSL utilization overview, MM-DSL is designed to augment metamodeling platforms, and not to replace them, in the process of modeling tool development. Later on in this chapter the use of MM-DSL will be illustrated on a representative example.

Additional benefit of using MM-DSL is in faster prototyping of modeling tools, because MM-DSL requires less information to describe a modeling method than a metamodeling platform. For example, if one wants to develop a modeling method and test its behavior as soon as possible, specifying the concrete syntax or graphical representation is not necessary. MM-DSL will use predefined graphical objects. This way, only a metamodel of a language needs to be specified in MM-DSL. This kind of development is typically not possible with metamodeling platforms – graphical representation needs to be specified for every modeling element.

The rest of the benefits come from a fact that MM-DSL is a domain-specific language. As such, it is supported by its own IDE (Integrated Development Environment) which comes with various helpful features that make coding easier, for example: syntax highlighting, auto complete, code templates, and compile time error checking. The architecture and utilization of the MM-DSL IDE is covered in its dedicated chapter.

8.2 Related Work

During the last couple of decades of applied research in metamodeling approaches and technologies many meta-metamodels (meta^{2}models) have emerged, some of them very complex, the others very simple and easy to use, but typically created for the same purpose: to capture the requirements imposed on them by the modeling domain. Initially, most of the meta^{2}models have been designed to be instantiated in a single

domain. For example, EMF Ecore is mostly used in the development of Eclipse applications, GME primarily in the area of electrical engineering, ConceptBase [80] for conceptual modeling and metamodeling in software engineering, and ADONIS was designed for describing and simulating business processes. However, over time it has been discovered that they are also applicable in similar domains. A good example is the extension of the ADONIS meta^{2}model, which later became the ADOxx meta^{2}model. The ADOxx meta^{2}model and the ADOxx metamodeling platform have until today been used for the realization of a myriad of domain-specific modeling methods. A couple of dozen of implemented modeling tools can be found on the OMiLAB web site [65]. Another very successful example is the GOPPRR meta^{2}model employed by the MetaEdit+ language workbench, which has been used in many industrial projects during the last decade [22].

Meta^{2}models are typically joined together with a metamodeling tool (Ecore comes with EMF and Eclipse IDE, ADOxx comes with its own metamodeling platform, MetaEdit+, ConceptBase, and GME as well), which makes them available for use out-of-the-box. A brief comparison of meta^{2}models and their accompanying tools is compiled in [81].

As we can see, metamodeling approaches have been applied in many real world scenarios, like functional and non-functional requirements definition in software engineering, social modeling for requirements engineering [82], as well as modeling tool development, where metamodeling platforms have become a very popular go-to software. The answer behind the metamodeling platform attractiveness lies within the fact that one gets the meta^{2}model and the tools that work with it for free. What is left for developers to do is: (1) instantiate the meta^{2}model and (2) apply the platform's functionality to bring a modeling method to life. This process is explained in more detail in [70] and [75].

A notable difference between MM-DSL and metamodeling platforms is in their way of tackling with the development of conceptual modeling tools. MM-DSL provides domain-specific functionality which is not linked to a specific technological platform. Developing by using only a metamodeling platform is very technology-specific. One has the entire platform's functionality out-of-the-box and the possibility to reuse and extend it. However, this comes with a drawback of locking in the future development only to one platform. There is, in most cases, no way of reusing any developed artifacts (e.g., code, files, etc.) on another metamodeling platform. This is one of the primary issues MM-DSL tries to address.

There are dozens of DSLs designed to describe a metamodel of a modeling language, the most significant being KM3 [5], HUTN [83], and Emfatic [84]. KM3 is designed particularly for specifying the abstract syntax. HUTN (Human Usable Textual Notation) is an OMG standard for specifying a default textual notation for each metamodel, developed with a purpose of solving the problem of the XMI/XML (XML Metadata Interchange) format, which is intended to be processed by the machine, therefore it is neither succinct, nor easily readable or writable. Emfatic is a language used to represent EMF Ecore models in a textual form. There exist even more DSLs that specialize in

model processing, most of them connected with the Eclipse community and EMF. Epsilon [85], for example, is a family of languages and tools for code generation, model-to-model transformation, model validation, comparison, migration and refactoring that works out-of-the box with EMF and other types of models. FunnyQT [86] is a querying and transformation DSL embedded in JVM-based Lisp dialect Clojure. FunnyQT primarily targets the modeling frameworks JGraLab and EMF. GReTL [87] is an extensible, operational, graph-based transformation language, which allows specifications of transformations in plain Java using the GReTL API.

Among all of these different languages and frameworks, there are two similar to our approach: Graphiti [88] and XMF (Xmodeler) [89]. Graphiti is an Eclipse-based graphics framework that allows the development of graphical editors from domain models typically specified with EMF. It is a significant productivity improvement when compared to Eclipse GMF. XMF is a meta-programming language with an OCL-like syntax that allows construction of an arbitrary number of classification levels. Xmodeler is a metamodeling platform developed around XMF. Both of these technologies try to improve the productivity of modeling tool development, and both of them suffer from the same issues: platform independency and complexity. Graphiti currently only supports Eclipse. XMF only supports Xmodeler and it is relatively hard to learn for domain experts with limited programming knowledge. The mentioned technologies are specialized in realizing modeling languages as modeling tools, but none of them fully covers the realization of modeling methods (e.g., possibility to implement all the essential parts: abstract syntax, concrete syntax, and algorithms that can be executed on models).

8.3 Applied Concepts and Technologies

In addition to the related research, the work at hand is mostly influenced by the following concepts and technologies: Language Oriented Programming [90], Language Driven Design [91], Model Driven Architecture [92], Software Factories [93], Language Workbenches [4], and Metamodeling Platforms [10]. All of them are used to ease the development of various kinds of software (e.g., systems, applications, tools) by providing the means to define custom programming or modeling artifacts and code generation facilities.

Language Oriented Programming [90] is a novel way of organizing the development of a large software system. The approach starts by developing a formally specified, domain-oriented, high-level language which is designed to be well-suited to develop a software system under consideration. After the system has been implemented in the before developed language, it is translated using a compiler or an interpreter to the existing technology. Among claimed advantages for domain analysis, rapid prototyping, maintenance, portability, and reuse of development work, LOP provides higher development productivity and faster time to market.

Table I: Concepts and Technologies Influencing the Development of MM-DSL

Concept / Technology	**MM-DSL**
Language Oriented Programming	Out-of-the box DSL which can be extended to better suit the domain of a modeling tool under development
Language Driven Development	Integration of different DSLs into a single DSL (e.g., a DSL for defining the concrete syntax is integrated with the DSL for defining the abstract syntax of a modeling language)
Model Driven Architecture	Generation (compilation) of platform-specific code from the DSL code; multiple code generations may occur during that process
Software Factories	The DSL IDE is created upon the Eclipse IDE
Language Workbench	Xtext is used for the implementation of the DSL grammar and various compilers
Metamodeling Platform	Common execution environment for the code written in DSL (ADOxx is used as a proof of concept for the DSL approach and metamodeling platform synergy)

Language Driven Development [91] is a software development method which involves the use of multiple DSLs at various points in the development life-cycle. It is fundamentally based on the ability to rapidly design new languages and tools in a unified and interoperable manner. By allowing engineers and domain expert to express their designs in the language that they are most comfortable with and that will give them most expressive power, productivity can be increased. The LDD vision relies heavily on the language integration. It is claimed that languages should be weaved together to form a unified view of the software system.

Model Driven Architecture [92] is a term commonly used to mean the generation of program code from (semi-)formal models (e.g., UML, UML profiles, various DSLs). System functionality is defined using a platform-independent model (PIM) which is described in an appropriate DSL. The PIM is then translated to one or more platform-specific models (PSM) that computers can run. The transformation process is generally automated and performed by tools [94].

Software Factories [93] is a term used to describe a collection of software used to create specific types of software. They help structure the development process and are used for developing languages that support the construction of software components. A software factory may include processes, templates, integrated development environ-

ment (IDE) configurations and views. The type of software a factory may produce is defined when the factory is created.

Language Workbench is a term proposed by Martin Fowler to mean the IDE support for development of domain-specific languages [21], [4]. There are many examples of such IDEs, some of them specifically designed for development of textual languages (e.g., Xtext [95], Irony [51]), and some for development of graphical languages (e.g., VS Visualization & Modeling Tools [96], MetaEdit+ [22]).

Metamodeling Platform [10] is a term used to describe an environment specifically targeting the development of graphical modeling languages and modeling methods using a metamodeling approach – a layered approach (see OMG's MOF [40]) where one describes a modeling language, as well as a modeling method, by instantiating already existing meta^{2}model provided by the platform. Because of this approach, a platform can provide support to the modeling language being developed through already existing features and functionality (e.g., algorithms and mechanism for model analysis and simulation) [77].

How every of the mentioned concepts have influenced the MM-DSL development is aggregated in Table I. All the software being used is either open-source (e.g., Xtext, Eclipse IDE) or open-use (e.g., ADOxx) software.

8.4 Clarifying Design Decisions

The requirements for MM-DSL have been gathered using a top-down and bottom-up approach. In top-down approach several modeling methods have been analyzed. The inspected modeling methods are compiled in [97]. Structural complexity of the analyzed modeling methods can be seen in Figure 19. These are the same methods that have been used to define the key functional requirements of a metamodeling platform. This time, the purpose was to determine the most commonly used concepts and to establish their appropriate abstractions (e.g., *"class"* is an appropriate abstraction for concept *"actor"*, *"relation"* is an appropriate abstraction of the concept *"uses"*, etc.).

The bottom-up approach gave insight on how are metamodeling technologies applied for realization of modeling methods. Several meta^{2}models have been carefully examined to determine their building blocks. Abstractions of concepts extracted with top-down approach have been matched with meta^{2}model concepts. In this process the backbone for abstract and concrete syntax of the MM-DSL has been formed, as well as mechanisms that allow compilation to different metamodeling technologies. The MM-DSL abstract syntax is conforming to the already established modeling method generic framework [10]. The chosen concrete syntax (e.g., language keywords) was adopted from the modeling community our research group belongs to (see the ADOxx meta^{2}model depicted in Figure 26).

The similarity between meta^{2}models is what MM-DSL translators (e.g., compilers) take advantage of. By leveraging this property new compilers do not need to be implement-

ed from scratch, but configured from the same template. Figure 26 shows similarities between meta²models. The concept of an *"attribute"* is highlighted with a rectangle, the concept of a *"class"* with an ellipsis, and the concept of a *"model type"* with a round-rectangle. These highlighted concepts are all singletons (consist only of one element). However, there exist concepts that are represented with multiple elements, such as *"relation"*. Relation in ADOxx is represented through *Relation Class* and its two *End Points*. In GOPPRR it is represented with elements: *Relationship*, *Binding* and *Connection*. In Ecore and GME relation is a single element: *EReference*, respectively *Reference*. From this observation, it is clear that most of the concepts described with different meta²models are semantically identical (or at least very similar).

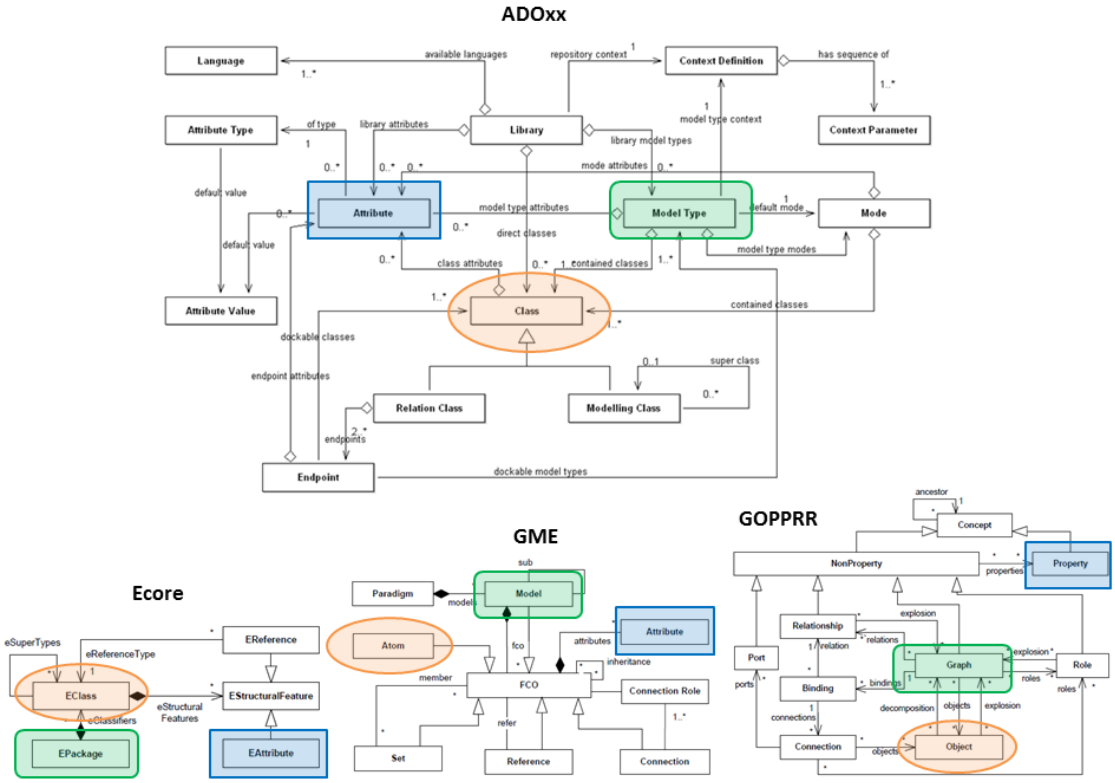


Figure 26: Meta²model Comparison (adapted from [81])

Additionally, the overall design of the language was driven by the idea of minimizing lines of code, in other words: to reduce the programming effort and raise productivity. This is the reason it has been decided to introduce concepts such as inheritance (transmission of characteristics from parent object to child object) and referencing (re-using previously defined objects by passing their identifier to other objects). How these concepts work in practice is demonstrated in the chapter dedicated to the utilization of MM-DSL in the modeling tool development. For the complete overview of the supported MM-DSL concepts, including the formal language's syntax, semantics and usage examples, see the chapter dedicated to the specification of the language.

8.5 Language Design Best Practices and Guidelines

Before the actual language design phase it helps to be familiar with the typical desirable and undesirable language features. By default, learning to program is difficult and takes a lot of time and effort. Designers need to avoid the common undesirable features that make the learning and using of the language harder than it should be. Substantial part of this difficulty arises from the structure, syntax and semantics of a language.

Language designers are highly intelligent experts in the field of programming, and are consequently far removed both temporally and cognitively from the difficulties experienced by the novice programmers. In our case these would be domain experts, which are typically not familiar with programming techniques. This mostly results in languages which are either too restrictive or too powerful (or sometimes, paradoxically, both) [98]. To avoid falling into a trap of designing a language only highly trained experts are able to use efficiently, a collection of desirable language features, as well as undesirable ones, which should be avoided if possible, has been established. The features compiled in here can be applied on any computer language.

8.5.1 Desirable Language Features

Taken from the experience of lecturers teaching various programming languages to students, there are generally good language features that make the language easier to learn. According to [98] most notable ones are:

- ***User expectation conformity***. Languages should be designed so that reasonable assumptions based on prior non-programing-based knowledge (e.g., domain expert knowledge) remain reasonable assumptions in the programming domain, meaning that the constructs of a language should not violate user expectations.

- ***Readable and consistent syntax***. By choosing the constructs with which the recipient is already familiar (e.g. *if* rather than *cond*, *head/tail* rather than *car/cdr*) syntactic noise can be minimized; on one hand, reducing syntactic noise might involve minimizing the overall syntax; alternatively, it may be better to increase the complexity of the syntax in order to reduce homonyms which blur the signal.

- ***Small and orthogonal set of features***. A small non-overlapping set of language features with distinct and mnemonic syntactic representations and with semantics which mirror as closely as possible the real-world concepts; features that are not necessary should not be included in the language.

- ***Error diagnosis***. Without good error detection and debugging support users can spent hours studying the code (which is fairly difficult for novice programmers) trying to decipher why isn't the program doing what it is intended; on the

other hand, error messages should be meaningful and without unnecessary technical jargon.

Generally, a language designer should follow commons sense and try to put himself in the skin of a novice language user – not an easy task to do for an expert. It also helps if the designer is knowledgeable in the domain for which the language is being developed, because of the greater awareness of the real-world concepts that need to be included into the language and the ways these concepts are expressed. A priori knowledge of the application domain can save the time needed to make a thorough domain analysis, which is a time consuming and complicated task.

8.5.2 Undesirables Language Features

Every language designer starts with good intentions, but during the design process one needs to take care and keep in mind which features make the language harder to learn and use. Some of the most undesirable features from the language user's perspective, according to [98], are:

- ***Paradigmatic purity***. Strict adherence to a single functional, logical or object oriented paradigm can make for a certain conceptual simplicity and elegance, but in practice it can also lead to extremely obscure and unreadable code; in some cases, relatively simple programs must be substantially restructured to achieve even basic effects such as input and output.

- ***Language bloat***. Extreme complexity and a huge palette of features might seem as a good idea at first, but they come together with a substantial cost: steeper learning curve, higher level of confusion, difficulties of adequate error detection, very complex syntax and semantics.

- ***Syntactic synonyms***. Two or more syntaxes are available to specify a single construct; common example is dynamic array access in C, where the second element of an array may be accessed by any of the following syntaxes, some of which are legal only in certain contexts: *array[1]*, **(array+1)*, *1[array]*, **++array*; in Prolog *[1, 2, 3]* is equivalent to *.(1, .(2, .(3, [])))*.

- ***Syntactic homonyms***. Constructs which are syntactically the same, but which have two or more different semantics depending on the context are perhaps a more serious flaw in a language then syntactic synonyms; an extreme example may be seen in Turing, in which the construct *A(B)* has five distinct meanings, but not as extreme as LISP and its variants, which can be viewed as one massive homonym.

- ***Hardware dependency***. There seems to be no convincing reason why the user, already struggling to master syntax and semantics of various constructs, should also be forced to deal with details of representational precision, varying

machine word sizes, or awkward memory models; the data types are particularly problematic in C as they are generally not portable, for example, the standard *int* type varies from 16-bit to 32-bit representations depending on the machine and the implementation; this can lead to strange and unexpected errors when overflow occurs.

- ***Backward compatibility***. This property is surely useful from the experienced programmer's point of view, as it promotes reuse of both code and programming skills, but one needs to be careful, because it constraints the design of a new language; even the father of C++ B. Stroustrup [99] acknowledged this problem [100]: *"Over the years, C++'s greatest strength and its greatest weakness has been its C compatibility. This came as no surprise."*

Avoiding these undesirable features will produce a language with a cleaner syntax and semantics, more readable code, and that is easier to maintain, as opposed to languages that incorporate some or all of the mentioned characteristics. In the long run some of these features are very hard to avoid, especially if there was no plan for language evolution in the design phase. Taking a systematic approach and considering future needs of language users plays a significant role in the further development of a language.

The importance of the continuous language development (language evolution) has already been discussed as a part of the introduction in this research project (see section 1.1.). The most important part that is directly connected with the design process of the MM-DSL is summed in the following couple of sentences.

While creating MM-DSL, one should consider the following: (1) the future development of metamodeling platforms, and (2) the possible changes in the application domain, which can come from various sources: new founding in the academia or industry, and insights during the language utilization.

To be able to cope with the upcoming issues, MM-DSL should be able to evolve through modifications, particularly through extension. However, one always needs to keep in mind that usability and expressiveness must not be lowered with introduction of new concepts into a language. It is very important to keep the equilibrium between the following three key properties of a language: usability, expressiveness, and extensibility.

By using the mentioned best practices, including the desired language features and considering future language evolution, the MM-DSL formal specification has been created.

9 MM-DSL Specification

This chapter describes the MM-DSL language in detail through its syntax, semantics and examples. MM-DSL is specified using EBNF, which is presented in text form, as well as in a form of syntax (railroad) diagrams.

The specification is separated in four parts. Each part is dedicated to a group of language statements. These are: (1) global statements, (2) structure statements, (3) visualization statements, and (4) operations statements.

Global statements are used to specify the name of a modeling method, embed and cross-reference embedded code throughout the entire program. Some of the statements belonging to this group are pervasive and can be used in other parts of MM-DSL as well (e.g., insert statement).

Structure statements are used to define typical structures of a modeling method, such as classes, relations, attributes and model types, which are also the names of most important statements that belong to this group of statements.

Visualization statements are used to define graphical notation of modeling elements predefined with structure statements, mainly classes and relations. The main statements, also called the symbol statements, may contain multiple SVG-like substatements. These statements define graphical objects such as rectangles, circle, polygons, and others.

Operations statements are used to define algorithms that can be applied on modeling method artifacts, as well as generic algorithms a modeling editor should poses. Operations statements group also includes the statements responsible for triggering various actions when an event occurs (event statements).

9.1 How to read the Grammar

Throughout the whole specification the MM-DSL grammar is presented utilizing the same formatting rules and guidelines. First the formal syntax is given using the EBNF and syntax diagrams. Afterwards, the semantics are described. Each grammar part finishes with a representative utilization example.

The grammar is given in form of production rules:

```
rule ::= expression
```

Within the expression on the right-hand side of a rule, the following meta-symbols are used: (), ?, |, + and *. See Table II for description. Table III illustrates the use of these symbols.

Table II: EBNF Meta-Symbols

Symbol	Name	Description
()	group	Used to group expressions which belong together.
?	optional	Represents an option, which means that expression may be taken or not.
|	or	Means that either the expression before the vertical bar or the expression after the vertical bar may be taken.
+	one or more	Means that the expression can appear one or more times in a row
*****	zero or more	Means that the expression may appear several times in a row, but it does not need to appear at all.

Table III: EBNF Meta-Symbols Application

Example	Produced the following words
x?	(empty), x
x | y	x, y
x+	x, xx, xxx, xxxx, …
x*	(empty), x, xx, xxx, xxxx, …
(x | y) z	xz, yz
(x | y)? z	z, xz, yz
(x | y)+ z	xz, yz, xxz, yyz, xyz, yxz, …
(x | y)* z	z, xz, yz, xxz, yyz, xyz, yxz, …

Every terminal (keyword which has to be taken literally) is surrounded by single quotes, e.g., *'method'*. MM-DSL is case sensitive, which means that the keywords (terminals) have to be written as it is specified in the rules.

MM-DSL uses cross-referencing by identifier. In the grammar rules an identifier can be recognized as a non-terminal symbol *name* (if it is not explicitly mentioned that something else is an identifier). Cross-references are represented in the following form:

```
name-rule
```

where *rule* is the name of a production rule. In the following example *name-x* is the reference of *x*:

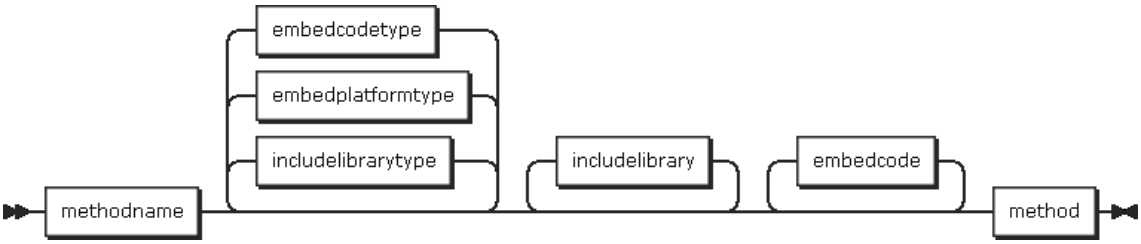
```
x ::= 'keyword' name
y ::= 'otherkeyword' name-x
```

9.2 Global Statements

9.2.1 Root Statement

Syntax

```
root
```




```
root ::=
  methodname
  (includelibrarytype | embedplatformtype | embedcodetype )*
  includelibrary* embedcode* method
```

Semantics

A MM-DSL program starts with a method name, followed by multiple occurrences of include or embed statements, which are also optional. This is called the program header. Program body is defined inside the method block, which is a mandatory part of every MM-DSL program.

Example

There are no concrete examples for this statement, because it shows the general structure of a MM-DSL program and does not contain any terminal elements.

9.2.2 Method Name

Syntax

```
methodname
```


```
methodname ::=
  'method' name
```

Semantics

Every modeling method specified by a MM-DSL program has a name which is also a valid identifier. There can only be one method name statement per MM-DSL program.

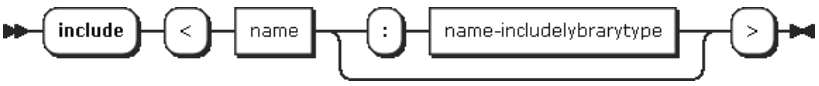
Example

```
// CarParkModeling is the method name (identifier)
method CarParkModeling
```

9.2.3 Include

Syntax

includelibrary



includelibrarytype


```
includelibrary ::=
  'include' '<' name (':' name-includelybrarytype)? '>'
includelibrarytype ::=
  'def' 'IncludeLibraryType' name
```

Semantics

Include statement includes foreign program code defined in an external library file. Before using the include statement, the library type needs to be defined using the include library type statements which starts with a keyword *def* followed by *IncludeLibraryType* and a name (identifier). Include statements starts with the keyword *include* followed by a name and a library type, which are both located inside the angled brackets (<, >).

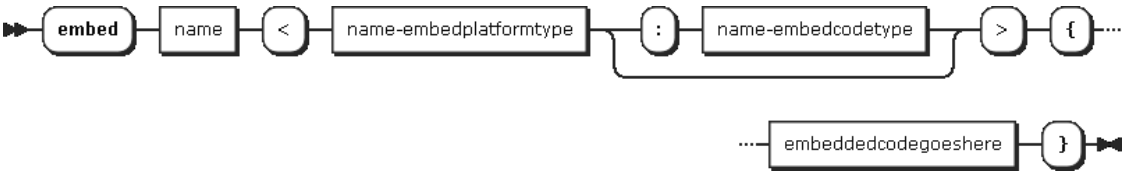
Example

```
// ADOxx is the included library type
// MyMetaModel is the name of the library
def IncludeLibraryType ADOxx
include <MyMetaModel:ADOxx>
```

9.2.4 Embed

Syntax

embedcode



embedplatformtype

embedcodetype


```
embedcode ::=
  'embed' name '<' name-embedplatformtype (':' name-embedcodetype)? '>'
  '{' embeddedcodegoeshere '}'
embedplatformtype ::=
  'def' 'EmbedPlatformType' name
embedcodetype ::=
  'def' 'EmbedCodeType' name
```

Semantics

Embed code statements always start with the keyword *embed* followed by an identifier called *name*. The identifier is important because it is a means to reference the code included between start and end tags (curly brackets: {, }) anywhere inside a program where insert statement may be used. Between the angled brackets (<, >) one can specify from which source does the code come from, and optionally, the type of code as an additional identifier. This structured information can be used by a translator to correctly include the embedded code into a compiled file.

Embedded code type and platform type must be defined prior they can be referenced in embed code statements. Same as embed code statements, these are also identified by their *name*.

Example

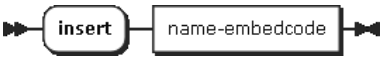
```
// def defines the platform type - ADOxx
// and code type - ADOscript
def EmbedPlatformType ADOxx
def EmbedCodeType ADOscript

// ADOitem holds the string between start and end tags
embed ADOitem <ADOxx:ADOscript>
```

```
{
   "ITEM \"MyAlg\" modeling:\"MyMenu\""
}

// AdoScript command for creating an menu item
embed URIImportItem <ADOxx:AdoScript> {
"ITEM \\\"URI import\\\" modeling:\\\"~AdoScripts\\\" pos2:1"
}

// optional extension for displaying object names
embed ShowNameGraph <ADOxx:Notebook> {
"ATTR \\\"Name\\\" x:0pt y:9pt w:c"
}
```

9.2.5 Insert

Syntax

```
insertembedcode
```




```
insertembedcode ::=
  'insert' name-embedcode
```

Semantics

Insert statement is used as a sub-statement in several other statements. It references the embed code statement by its defined *name* and during the translation of a program inserts the code found between *start* and *end* tags (or between curly brackets {, }) in the exact place where insert statement was used.

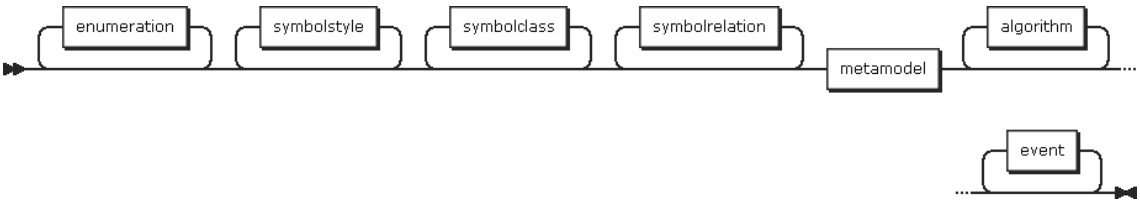
Example

```
def EmbedPlatformType ADOxx
def EmbedCodeType ADOscript

// code to be embedded
embed ADOitem <ADOxx:ADOscript>
{
   "ITEM \"MyAlg\" modeling:\"MyMenu\""
}

algorithm MyAlgorithm {
   // inserting ADOscript code inside the definition of an algorithm
   insert ADOitem

   // alternatively we can use the native MM-DSL code
   ui.item.menu.insert MyAlg to MyMenu
}
```

9.2.6 Method

Syntax

```
method
```




```
method ::=
  enumeration* symbolstyle* symbolclass* symbolrelation* metamodel
  algorithm* event*
```

Semantics

Method statement is composed of code blocks that define enumerations, visualization (e.g., graphical symbols), structure (metamodel), and operations (algorithms and events). Only the metamodel part is the mandatory part of every method statement.

The order of the code blocks needs to be respected. Enumerations need to be defined first, followed by symbol styles, then graphical symbols for classes and relations, then metamodel, and at the end algorithms and events. The order is important, because enumerations need to be defined before they can be used as data types in metamodel statements. Styles need to be defined if we want to reference them in symbol statements. Graphical symbols need to be defined if we want to reference them in metamodel statements. Algorithms need objects defined within metamodel statements, and events reference algorithms in order to trigger them.

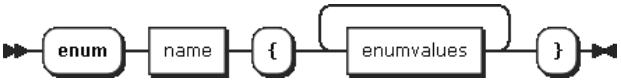
Example

There are no concrete examples for this statement. It shows the general structure of a part of the MM-DSL program that holds the definition of a modeling method.

9.2.7 Enumeration

Syntax

```
enumeration
```



```
enumeration ::=
  'enum' name '{' enumvalues+ '}'
```

Semantics

Enumeration is an ordered collection of strings, which can be of arbitrary length. Even numeric values need to be defined as strings (invalid: {1, 2, 3}, valid: {“1”, “2,”, “3,”}). Valid enumeration holds at least one element. Every enumeration is identified through its *name*, which is used to reference them by other statements. The default value is always set on the first element of an enumeration. For example, if we query for the default value of an enumeration with the following elements {“blue”, “red”, “yellow”}, “blue” will be the answer.

Enumerations are currently the only supported user defined data type in MM-DSL.

Example

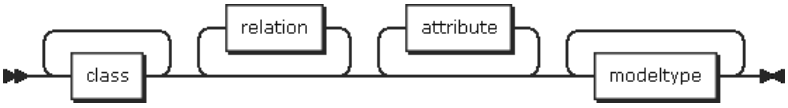
```
// simple enumerations
enum EnumParkType { "car" "truck" "motorcycle" "bicycle" }
enum EnumPayment { "ticket machine" "mobile phone" "cash" }
enum EnumNumbers { "1" "2" "3" }
enum EnumCO2Emmision {"low" "medium" "high"}

// using enumeration in a class
class Car extends Vehicle symbol CarGraph {
   attribute co2emmision:enum EnumCO2Emmision
}
```

9.3 Structure Statements

9.3.1 Metamodel

Syntax

```
metamodel
```




```
metamodel ::=
  class+ relation* attribute* modeltype+
```

Semantics

The metamodel statement is composed of several code blocks: class, relation, attribute, and model type. Each of these code blocks is described by the appropriate statements. For example, class code block can only contain class statements and its substatements. The same is true for relation, attribute and model type blocks. However, insert statement from the global statements group can be used as well.

The only mandatory parts of a metamodel statement are the class statements and the model type statements. There can only be one metamodel statement per modeling method.

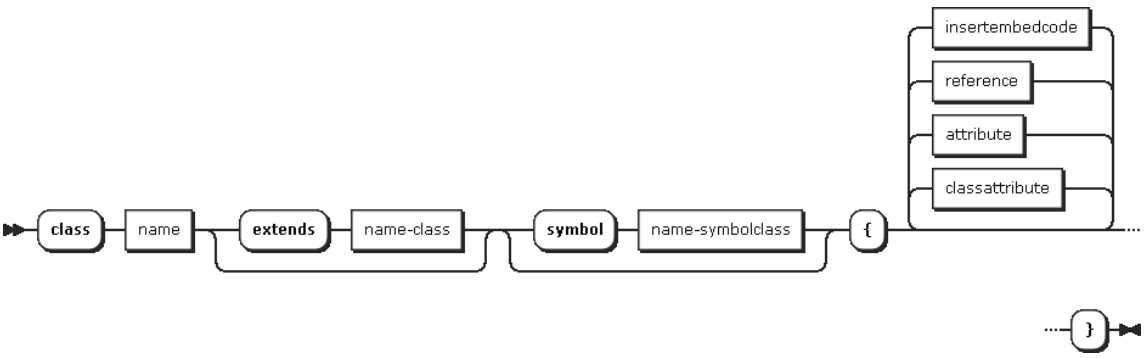
Example

There are no concrete examples for this statement. It shows the general structure of a part of a MM-DSL program that holds the definition of a modeling method structure (metamodel).

9.3.2 Class

Syntax

```
class
```




```
class ::=
  'class' name ('extends' name-class)? ('symbol' symbolclass)?
  '{' (classattribute | attribute | reference | insertembedcode)* '}'
```

Semantics

Class statement describes the class concept, which is a container that can hold attributes, can be related to other classes or self-related, can be inherited, and can be represented graphically through related symbol statements. It starts with the keyword *class*, followed by an identifier (*name*). Optionally, a class can be extended from another class by using the keyword *extends* followed by a reference to another class (name). Every

class can be associated with a symbol by using the keyword *symbol* and a reference to a class symbol statement. Extended class only inherits properties found between curly brackets ({, }). Symbol is not inherited. Class statement allows the following sub-statements: (class) attribute statements, reference statements and insert statements.

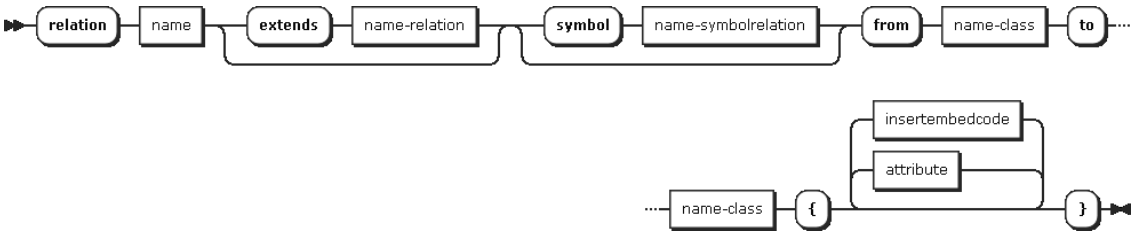
Example

```
class Vehicle {
   attribute color:string access:write
   attribute lenght:int access:write
   attribute width:int access:write
   attribute height:int access:write
   attribute weight:int access:write
}

// extends Vehicle class and has a symbol CarGraph
class Car extends Vehicle symbol CarGraph {
   attribute doors:int access:write
   attribute co2emmision:enum EnumCO2Emmision
}
```

9.3.3 Relation

Syntax

relation




```
relation ::=
  'relation' name ('extends' name-relation)? ('symbol' name-symbolrelation)?
  'from' name-class 'to' name-class '{' (attribute | insertembedcode)* '}'
```

Semantics

Relation statement describes a concept of a relationship. Relation can be observed as a special class that can contain attributes only. A relation statement starts with a keyword *relation* followed by an identifier (*name*). Optionally, it can extend another relation, and can be associated with one relation symbol. Relation statement must define two end points: *from* and *to*. From, respectively, to are followed by a class identifier (*name*). Insert sub-statement may also be used. Extended relation inherits only properties between curly brackets ({, }). Symbol and end points are not inherited.

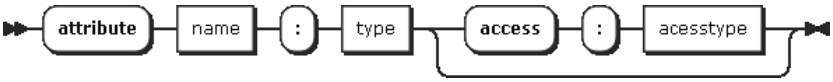
Example

```
// this relations has a symbol IsParkedGraph
// it is going from class Vehicle to class Park
relation isParked symbol IsParkedGraph from Vehicle to Park {}

// this relation has a default symbol (plain black line)
// it contains attributes
relation belongsTo from Park to City {
   attribute description:string
   attribute notes:string
}

// extending belongsTo relation with additional attributes
// embedding code by using insert statement
relation belongsToDetailed extends belongsTo from Park to City {
   attribute details:string
   attribute usage:string
   insert SomeEmbeddedCode
}
```

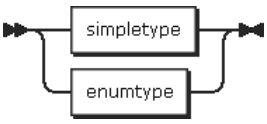
9.3.4 Attribute

Syntax

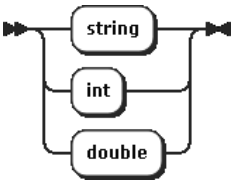
attribute



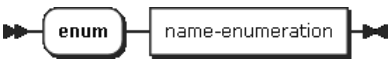
type



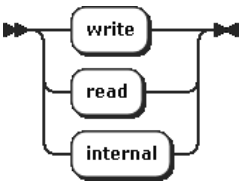
simpletype



enumtype



accesstype



```
attribute ::=
  'attribute' name ':' type ('access' ':' acesstype)?
type ::=
  simpletype | enumtype
simpletype ::=
  'string' | 'int' | 'double'
enumtype ::=
  'enum' name-enumeration
accesstype ::=
  'write' | 'read' | 'internal'
```

Semantics

Attribute statement describes a concept of an attribute, which is a property associated with an object. Attribute is identified by its *name*, and it must have a data type (indicated by type sub-statement). A data type can be either a simple type or an enumeration type. Supported simple types are string, integer and double. Enumeration type begins with a keyword *enum* followed by an enumeration identifier. Only attributes can have data types assigned.

Optionally, an access type can be assigned to the attribute definition: *write*, *read*, or *internal*. Write indicates that a user of a modeling tool can edit the value of an attribute. Read indicates that the value is read-only. Internal indicates that the attribute will not be visible and directly modifiable by the user; only attributes and events can modify its value. Internal attributes are typically used to store intermediate results of algorithms. Default access type is set to write.

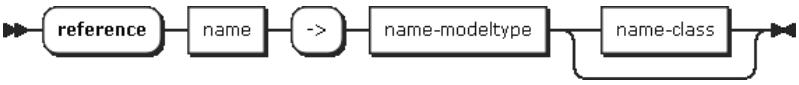
Example

```
// we use a class as a container for attributes
class ShowAttrDef {
   // basic attribute definitions
   // if access is not specified, it defaults to write
   attribute attr1:int
   attribute attr2:string
   attribute attr3:double
   attribute attr4: enum Colors

   // extended attribute definitions
   attribute attr5:int access:internal
   attribute attr6:string access:read
   attribute attr7:string access:write
}
```

9.3.5 Reference

Syntax

```
reference
```




```
reference ::=
  'reference' name '->' name-modeltype name-class?
```

Semantics

Reference statement defines the internal association between two types of object: classes and model types. Semantically, it means that an object containing the reference is associated with the object being referred to.

Currently only classes may contain the reference sub-statement. Statement starts with the keyword *reference* followed by an identifier (*name*). The symbol -> means “refers to”. After it a name (identifier) of a model type needs to be given, and optionally a class name contained inside the given model type. If only a model type name is given, the class will reference that model type. In case a class name that belongs to the referenced model type is given as well, the class containing the reference will directly reference that class.

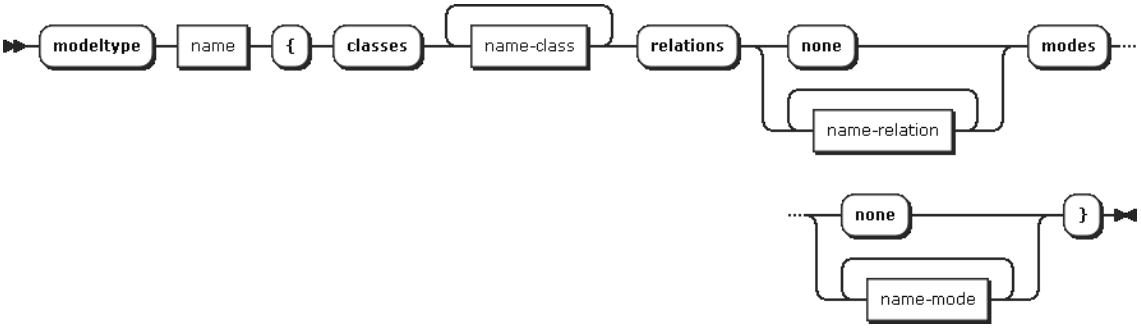
Example

```
class City {
   // the city can be associated with multiple car parks
   // the reference identifier is carParksInCity
   reference carParksInCity -> modeltype CarPark
   attribute latitude:int
   attribute longitude:int
}

// referenced model type containing several classes, relation and 2 modes
modeltype CarPark {
   classes Car Truck Motorcycle Bicycle ParkingLot ParkingGarage
   relations isParked
   modes
   // show multiple vehicles that can be parked
   mode ShowMultiple include
      classes Car Truck Motorcycle Bicycle ParkingLot ParkingGarage
      relations isParked
   // show only cars (and car parks)
   mode OnlyCars include
      classes Car ParkingLot ParkingGarage
      relations isParked
}
```

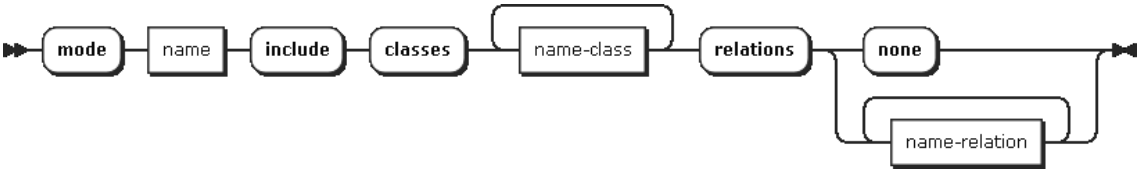
9.3.6 Model Type

Syntax

modeltype



mode




```
modeltype ::=
  'modeltype' name '{' 'classes' name-class+
  'relations' ('none' | name-relation+) 'modes' ('none' | name-mode+) '}'
mode ::=
  'mode' name 'include' 'classes' name-class+
  'relations' ('none' | name-relation+)
```

Semantics

Model type statement defines an aggregator of predefined objects (classes and relations). Model type as a concept is a blueprint of a modeling diagram. The statement starts with a keyword *modeltype* followed by an identifier (*name*).

Between the curly brackets ({, }) there are three sections. First one (starts with a keyword *classes*) references classes, the second one (starts with a keyword *relations*) references relations, and the third one (starts with a keyword *modes*) defines modes.

Modes are different views of a model type and can contain a subset of already referenced classes and relations. A model type contains a default mode if no modes are defined (if a *mode* keyword is followed by keyword *none*).

Example

```
// the simplest model type contains only one class
modeltype ModelTypeA
{
   classes Abc
   relations none
```

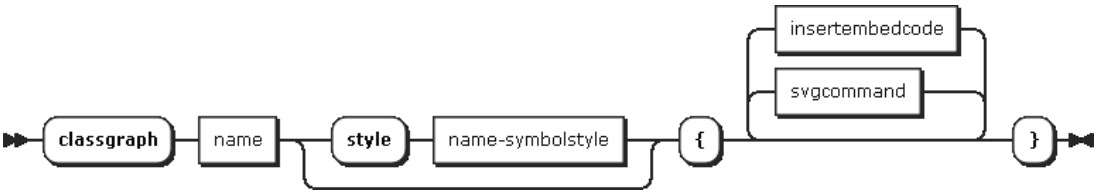
```
    modes none
}

// simple model type without modes
modeltype ParkingMap {
    classes City ParkingArea Car Truck Motorcycle Bicycle
    relations acceptsVehiclesOfType contains
    modes none
}

// a model type containing several classes, relation and 2 modes
modeltype CarPark {
    classes Car Truck Motorcycle Bicycle ParkingLot ParkingGarage
    relations isParked
    modes
    // show multiple vehicles that can be parked
    mode ShowMultiple include
        classes Car Truck Motorcycle Bicycle ParkingLot ParkingGarage
        relations isParked
    // show only cars (and car parks)
    mode OnlyCars include
        classes Car ParkingLot ParkingGarage
        relations isParked
}
```

9.4 Visualization Statements

9.4.1 Class Symbol

Syntax

symbolclass




```
symbolclass ::=
  'classgraph' name ('style' name-symbolstyle)?
  '{' (svgcommand | insertembedcode)* '}'
```

Semantics

Class symbol statement defines class-specific graphical symbols. It starts with the keyword *classgraph* followed by an identifier (*name*). Optionally, the class symbol statement can reference a style. Between the curly brackets ({, }) one can use SVG command sub-statements, as well as insert statements.

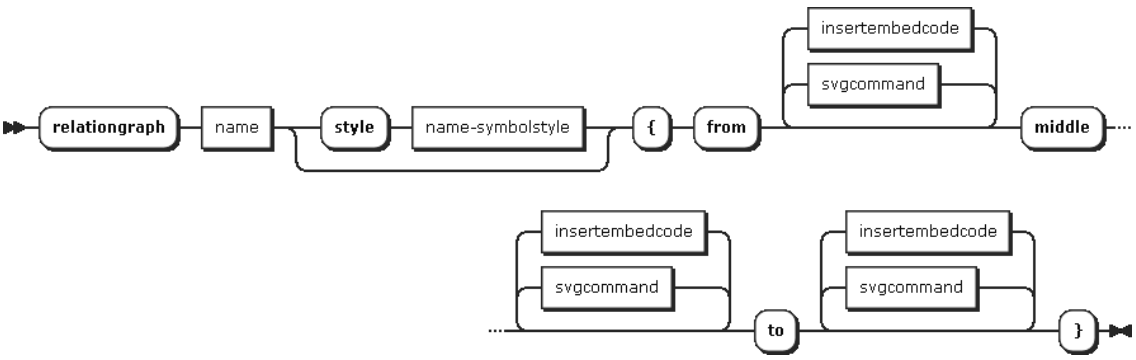
Example

```
// optional extension for displaying object names
// only use if you defining graphs yourself
embed ShowNameGraph <ADOxx:Notebook> {
"ATTR \\\"Name\\\" x:0pt y:9pt w:c"
}

// simple class symbol with a global style Green and an insert statement
classgraph BicylceGraph style Green {
   circle cx=-5 cy=0 r=10
   circle cx=5 cy=0 r=10
   insert ShowNameGraph
}

// more complex class symbol
// each SVG command has a style assigned
classgraph ParkingLotGraph {
   rectangle x=-20 y=-20 w=40 h=40
      style Orange {fill:orange stroke:black stroke-width:1}
   polygon points=-20,20 0,20 -20,0
      style Red {fill:red stroke:black stroke-width:1}
   polygon points=0,-20 20,-20 20,0
      style Red {fill:red stroke:black stroke-width:1}
   // text uses default style
   text "L" x=-2 y=-2
}
```

9.4.2 Relation Symbol

Syntax

symbolrelation




```
symbolrelation ::=
  'relationgraph' name ('style' name-symbolstyle)?
  '{' 'from' (svgcommand | insertembedcode)* 'middle'
  (svgcommand | insertembedcode)* 'to' (svgcommand | insertembedcode)* '}'
```

Semantics

Relation symbol statement defines relation-specific graphical symbols. It starts with the keyword *relationgraph* followed by an identifier (*name*). Optionally, the relation symbol

statement can reference a predefined style. Between the curly brackets ({, }) one can use SVG command sub-statements, as well as insert statements. However, there is a difference between relation symbol and class symbol statements. Relation symbol statement contains three sets of SVG commands. Every set starts with a keyword: *from*, *middle*, or *to*. From part defines the graphical representation connected with the starting point of a relation. Middle part defines, as the name indicates, the graphical representation of a middle part (the line). To part defines the graphical representation connected with the end point of a relation. Every part can use all the available SVG command statements.

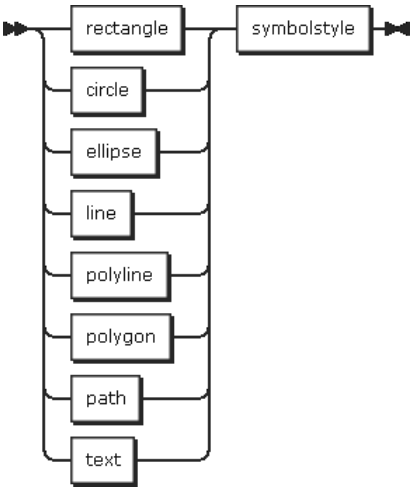
Example

```
// simple relation symbol which uses the global style Black
relationgraph IsParkedGraph style Black {
    // from part of a symbol
    from
    rectangle x=-2 y=-2 w=4 h=4
    // middle or line part of a symbol
    middle
    text "is parked in" x=0 y=0
    // to part of a symbol
    to
    polygon points=-2,2 2,0 -2,-2
}
```

9.4.3 SVG Command

Syntax

svgcommand




```
svgcommand ::=
  (rectangle | circle | ellipse | line | polyline | polygon | path | text)
  symbolstyle
```

Semantics

SVG command statement is a collection of all the available graphical object statements, such as rectangle, circle, line, polygon, text, etc. Every SVG command starts with one of the graphical object statements and, optionally, is followed by a style substatement. All graphical object statements use points (1 pt ≈ 0.3527 mm) as a measurement value.

Example

There is no concrete example for this statement. It shows the general structure of every SVG command statement and contains no terminals. SVG command statements can only be used inside the class and relation symbol statement.

9.4.4 Rectangle

Syntax

```
rectangle
```


```
rectangle ::=
  'rectangle' 'x' '=' REALNUMBER 'y' '=' REALNUMBER
  'w' '=' NUMBER 'h' '=' NUMBER
```

Semantics

Rectangle statement defines a rectangle with the given parameters. It starts with the keyword *rectangle* followed by parameters: *x*, *y*, *w*, and *h*. X and y stand for coordinates of the top-left corner of a rectangle. W stands for width, and h stands for height.

Example

```
classgraph RectangleGraph style Blue {

   // simple rectangle statement using the global style Blue
   rectangle x=-10 y=-10 w=20 h=20

   // simple rectangle statement using the local style Orange
   rectangle x=-20 y=-20 w=40 h=40
      style Orange {fill:orange stroke:black stroke-width:1}
}
```

9.4.5 Circle

Syntax

```
circle
```


```
circle ::=
  'circle' 'cx' '=' REALNUMBER 'cy' '=' REALNUMBER 'r' '=' NUMBER
```

Semantics

Circle statement defines a circle with the given parameters. It starts with the keyword *circle* followed by parameters: *cx*, *cy*, and *r*. Cx and cy are the coordinates of a center point, and r is the radius of a circle.

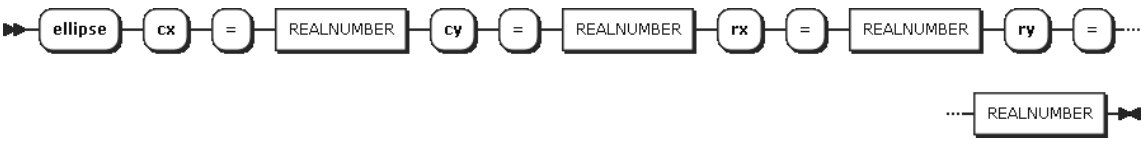
Example

```
classgraph CircleGraph style Blue {

   // simple circle statements using the global style Blue
   circle cx=0 cy=0 r=20
   circle cx=-5 cy=0 r=10
   circle cx=5 cy=0 r=10
}
```

9.4.6 Ellipse

Syntax

```
ellipse
```




```
ellipse ::=
  'ellipse' 'cx' '=' REALNUMBER 'cy' '=' REALNUMBER
  'rx' '=' REALNUMBER 'ry' '=' REALNUMBER
```

Semantics

Ellipse statement defines an ellipse with the given parameters. It starts with the keyword *ellipse* followed by parameters: *cx*, *cy*, *rx*, and *ry*. Cx and cy are the coordinate of a center point, rx and ry are distances from the two fixed points. Sum of rx and ry is equal to the major ellipse axis's length.

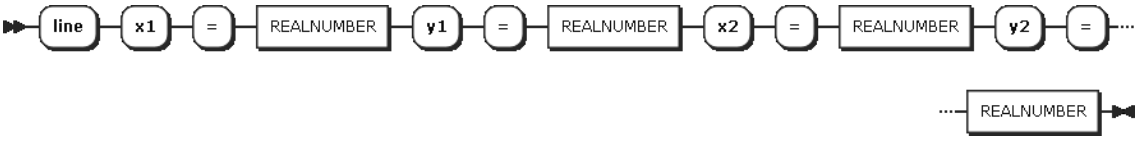
Example

```
classgraph EllipseGraph style Blue {

   // simple ellipse statement using the global style Blue
   ellipse cx=0 cy=0 rx=20 ry=30
}
```

9.4.7 Line

Syntax

line




```
line ::=
  'line' 'x1' '=' REALNUMBER 'y1' '=' REALNUMBER
  'x2' '=' REALNUMBER 'y2' '=' REALNUMBER
```

Semantics

Line statement defines a straight line. It starts with the keyword *line* followed by parameters: *x1*, *y1*, *x2*, *y2*. X1 and y1 are the coordinated of a starting point. X2 and y2 are the coordinates of an end point.

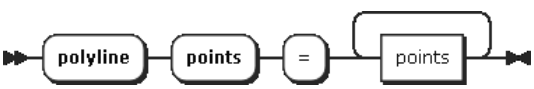
Example

```
classgraph LineGraph style Blue {

   // simple line statement using the global style Blue
   line x1=0 y1=0 x2=20 y2=10
}
```

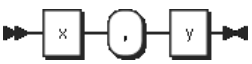
9.4.8 Polyline

Syntax

polyline



points



```
polyline ::=
  'polyline' 'points' '=' points+
points ::=
  x ',' y
```

Semantics

Polyline statement defines a set of connected straight lines. It starts with the keyword *polyline* followed by the keyword *points* and collections of parameter sets. Each parameter set is in the following form: “x, y”. Parameter sets are divided by an empty space (it can be more than one empty space in between, even a new line), e.g., 12, 3 4, 12 15, 20.

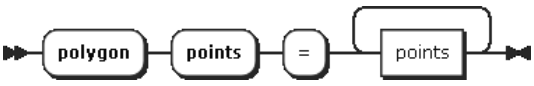
Example

```
classgraph PolylineGraph style Blue {

   // simple polyline statement using the global style Blue
   polyline points=0,0 30,30 20,30 30,20

   // simple polyline statement, parameters separated by a new line
   polyline points=
      0,0
      30,30
      20,30
      30,20
}
```

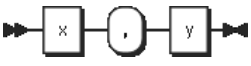
9.4.9 Polygon

Syntax

polygon



points




```
polygon ::=
  'polygon' 'points' '=' points+
points ::=
  x ',' y
```

Semantics

Polygon statement defines a graphical object that is bounded by a finite chain of straight line segments. It starts with the keyword *polygon*, followed by the keyword *points* and a collection of parameter sets. The format of parameter sets is the same as for polyline statement: x1, y1 x2, y2 x3, y3 …

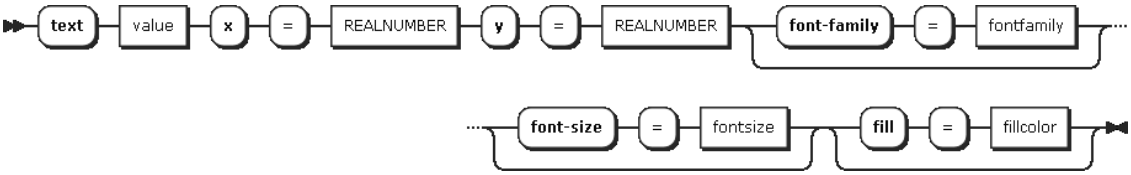
Example

```
classgraph PolygonGraph style Blue {

   // simple polygon statement using the global style Blue
   polygon points=0,-20 20,-20 20,0

   // simple polygon statement, parameters separated by a new line
   polygon points=
      0,-20
      20,-20
      20,0
}

// simple relation symbol which uses the global style Black
relationgraph IsParkedGraph style Black {
   // from part of a symbol
   from
   rectangle x=-2 y=-2 w=4 h=4
   // middle or line part of a symbol
   middle
   text "is parked in" x=0 y=0
   // to part of a symbol
   to
   // polygon definition inside the to part of a relation symbol statement
   polygon points=-2,2 2,0 -2,-2
}
```

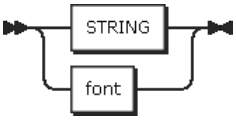
9.4.10 Text

Syntax

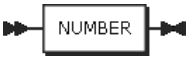
text



fontfamily



```
fontsize
```




```
text ::=
  'text' value 'x' '=' REALNUMBER 'y' '=' REALNUMBER
  ('font-family' '=' fontfamily)? ('font-size' '=' fontsize)?
  ('fill' '=' fillcolor)?
fontfamily ::=
  STRING | font
fontsize ::=
  NUMBER
```

Semantics

Text statement defines the chain of characters that can be represented as a graphical object. It starts with the keyword *text*, followed by a string of characters inside quotation marks (“this is a text”). X and y parameters indicate the position of the first character. Optionally, text statement can include the definition of a font family (keyword *font-family*), font size (keyword *font-size*) and character color (keyword *fill*). MM-DSL contains a predefined collection of font families (e.g., Georgia, Arial, etc.) and standard set of HTML colors (e.g., aliceblue in the example).

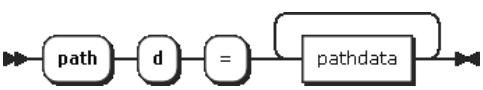
Example

```
classgraph TextGraph style Blue {

   // simple text statement using the global style Blue
   // it also uses optional arguments for font-family, font-size and fill
   text "This is a text" x=0 y=0
      font-family=Georgia font-size=12 fill=aliceblue
}
```

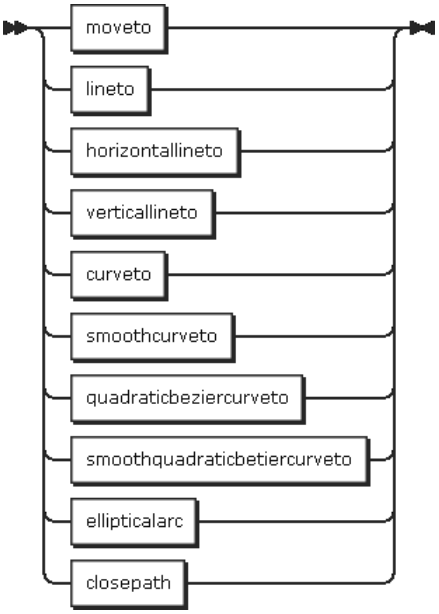
9.4.11 Path

Syntax

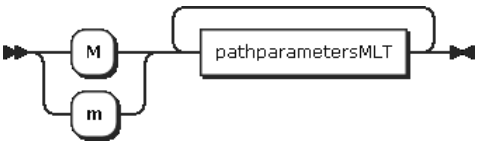
```
path
```



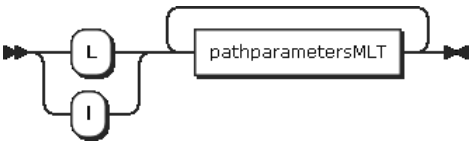
pathdata



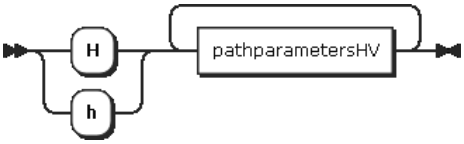
moveto



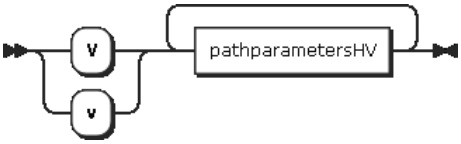
lineto



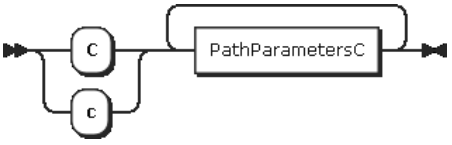
horizontallineto



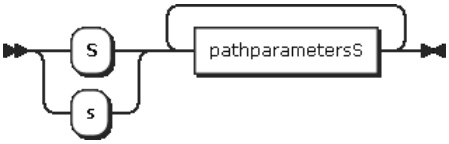
verticallineto



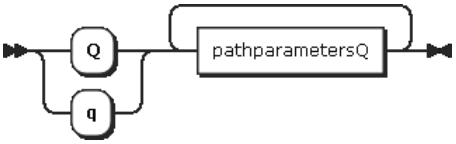
curveto



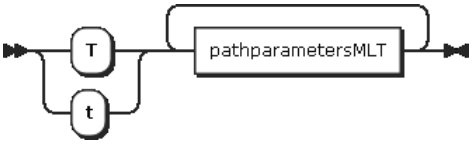
smoothcurveto



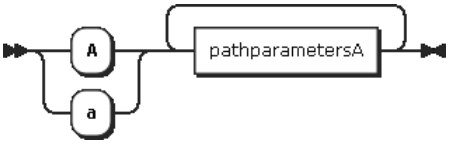
quadraticbeziercurveto



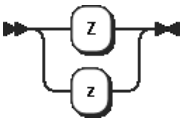
smoothquadraticbeziercurveto



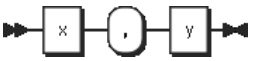
ellipticalarc



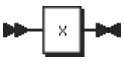
closepath



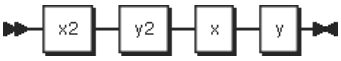
pathparametersMLT



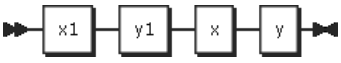
pathparametersHV



pathparametersS



pathparametersQ



pathparametersC

pathparametersA


```
path ::=
  'path' 'd' '=' pathdata+
pathdata ::=
  moveto | lineto | horizontallineto | verticallineto | curveto |
  smoothcurveto | quadraticbeziercurve | smoothquadraticbetiercurveto |
  ellipticalarc | closepath
moveto ::=
  ('M' | 'm') pathparametersMLT+
lineto ::=
  ('L' | 'l') pathparametersMLT+
horizontallineto ::=
  ('H' | 'h') pathparametersHV+
verticallineto ::=
  ('V'| 'v') pathparametersHV+
curveto ::=
  ('C' | 'c') PathParametersC+
smoothcurveto ::=
  ('S' | 's') pathparametersS+
quadraticbeziercurveto ::=
  ('Q' | 'q') pathparametersQ+
smoothquadraticbeziercurveto ::=
  ('T' | 't') pathparametersMLT+
ellipticalarc ::=
  ('A' | 'a') pathparametersA+
closepath ::=
  ('Z' | 'z')
pathparametersHV ::=
 x
pathparametersMLT ::=
  x ',' y
pathparametersS ::=
  x2 y2 x y
pathparametersQ ::=
  x1 y1 x y
pathparametersC ::=
  x1 y1 x2 y2 x y
pathparametersA ::=
  rx ',' ry xaxisrot largearcflag sweepflag x y
```

Semantics

Path statement is used to define a path, and it is one of the most complex SVG command statements. It starts with the keyword *path*, followed by the letter *d* (which stands for data or path data). There are several commands available for path data. Each of them has its own set of parameters. The commands are expressed as letters followed by various parameter sets. Capital letters mean absolutely positioned, and lower case letters mean relatively positioned. See Table IV for all the available command and their description.

Table IV: SVG Path Statement Commands (adapted from [101])

Com.	Param.	Name	Description
M	x,y	moveto	Move pen to specified point x,y without drawing.
m	x,y	moveto	Move pen to specified point x,y relative to current pen location, without drawing.
L	x,y	lineto	Draws a line from current pen location to specified point x,y .
l	x,y	lineto	Draws a line from current pen location to specified point x,y relative to current pen location.
H	x	horizontal lineto	Draws a horizontal line to the point defined by (specified x, pens current y).
h	x	horizontal lineto	Draws a horizontal line to the point defined by (pens current x + specified x, pens current y). The x is relative to the current pens x position.
V	y	vertical lineto	Draws a vertical line to the point defined by (pens current x, specified y).
v	y	vertical lineto	Draws a vertical line to the point defined by (pens current x, pens current y + specified y). The y is relative to the pens current y-position.
C	x1,y1 x2,y2 x,y	curveto	Draws a cubic Bezier curve from current pen point to x,y. x1,y1 and x2,y2 are start and end control points of the curve, control-

			ling how it bends.
c	x1,y1 x2,y2, x,y	curveto	Same as C, but interprets coordinates relative to current pen point.
S	x2,y2 x,y	smooth curveto	Draws a cubic Bezier curve from current pen point to x,y. x2,y2 is the end control point. The start control point is is assumed to be the same as the end control point of the previous curve.
s	x2,y2 x,y	smooth curveto	Same as S, but interprets coordinates relative to current pen point.
Q	x1,y1 x,y	quadratic Bezier curveto	Draws a quadratic Bezier curve from current pen point to x,y. x1,y1 is the control point controlling how the curve bends.
q	x1,y1 x,y	quadratic Bezier curveto	Same as Q, but interprets coordinates relative to current pen point.
T	x,y	smooth quadratic Bezier curveto	Draws a quadratic Bezier curve from current pen point to x,y. The control point is assumed to be the same as the last control point used.
t	x,y	smooth quadratic Bezier curveto	Same as T, but interprets coordinates relative to current pen point.
A	rx,ry xaxisrot largearcflag sweepflag x,y	elliptical arc	Draws an elliptical arc from the current point to the point x,y. rx and ry are the elliptical radius in x and y direction. The x-rotation determines how much the arc is to be rotated around the x-axis. It only seems to have an effect when rx and ry have different values. The large-arc-flag doesn't seem to be used (can be either 0 or 1). Neither value (0 or 1) changes the arc. The sweep-flag determines the direction to draw the arc in.
a	rx,ry xaxisrot largearcflag sweepflag x,y	elliptical arc	Same as A, but interprets coordinates relative to current pen point.
Z	-	closepath	Closes the path by drawing a line from current point to first point.

z	-	closepath	Closes the path by drawing a line from current point to first point.

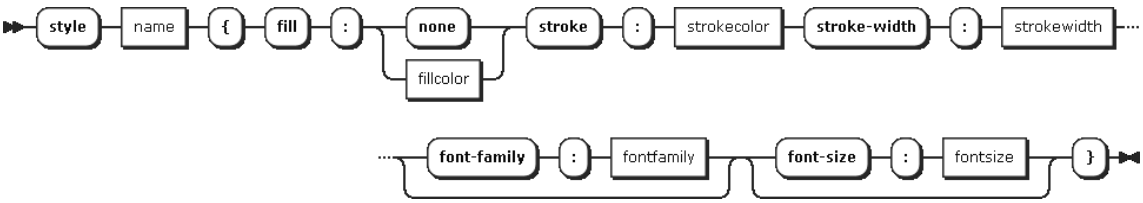
Example

```
classgraph PathGraph style Blue {

   // simple path statement using the global style Blue
   // it uses all the commands with absolute coordinates (capital letter)
   path d=
      M 10,10               // moveto
      L 10,20 20,30         // lineto
      H 40                  // horizontal lineto
      V 0                   // vertical lineto
      C 10 10 30 30 15 15   // curveto
      Q 10 10 0 0           // quadratic Bezier curveto
      T 20,0 0,20           // smooth qzadratic Bezier curveto
      A 0,0 45 10 10 50 50  // elliptic arc
      Z                     // closepath
}
```

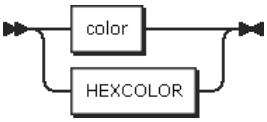
9.4.12 Symbol Style

Syntax

symbolstyle



fillcolor




```
symbolstyle ::=
  'style' name '{' 'fill' ':' ('none' | fillcolor) 'stroke' ':' strokecolor
  'stroke-width' ':' strokewidth ('font-family' ':' fontfamily)?
  ('font-size' ':' fontsize)? '}'
fillcolor ::=
  color | HEXCOLOR
```

Semantics

Symbol style statement defines a graphical style the class and relation symbols can use. The statement starts with the keyword *style* followed by an identifier (*name*). Be-

tween curly brackets ({, }) one can define parameters: *fill*, *stroke*, *stroke-width*, and optionally *font-family* and *font-size*. Color used for fill parameter can be given as name (e.g., blue, yellow, red, etc.) from a predefined set of colors, or as a hexadecimal code (e.g., #ffff00, #7F7E00, …). If no style is assigned to any of the SVG commands, all black style will be used as a default.

Example

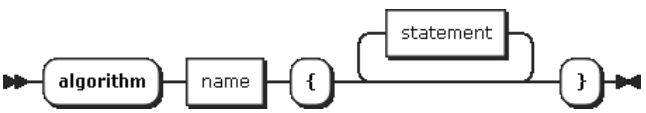
```
// some of style definitions
style Blue {fill:blue stroke:black stroke-width:1}
style Green {fill:green stroke:black stroke-width:1}
style Black {fill:black stroke:black stroke-width:1}
style Orange {fill:orange stroke:black stroke-width:1}
style Red {fill:red stroke:black stroke-width:1}

// using style Blue in a class symbol statement
classgraph CarGraph style Blue {
   rectangle x=-10 y=-10 w=20 h=20
}

// each SVG command has a style assigned (local style)
classgraph ParkingLotGraph {

   rectangle x=-20 y=-20 w=40 h=40
      style Orange {fill:orange stroke:black stroke-width:1}

   polygon points=-20,20 0,20 -20,0
      style Red {fill:red stroke:black stroke-width:1}

   polygon points=0,-20 20,-20 20,0
      style Red {fill:red stroke:black stroke-width:1}

   // text uses default style (black)
   text "L" x=-2 y=-2
}
```

9.5 Operations Statements

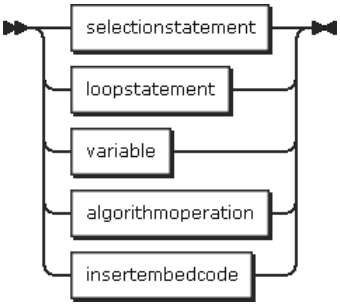
9.5.1 Algorithm

Syntax

algorithm



```
statement
```




```
algorithm ::=
  'algorithm' name '{' (statement)* '}'
```

Semantics

Algorithm statement defines an algorithm that can be executed inside a modeling tool. It starts with a keyword *algorithm*, followed by an identifier (*name*). Algorithms consist of algorithm statements: selection, loop, variable, and operation statements. All of the statements, except operation statements are the typical program flow statements.

Selection statements define which of the code blocks will be picked for execution. Loop statements loop through code until a condition has been fulfilled. Variable statements are used to declare and initialize variables.

Operation statements are specific to the development of modeling tools and its structure is different that the rest of the algorithm sub-statements.

By using the *insert* keyword one can also include foreign code found inside the embed statements to the algorithm statement.

Example

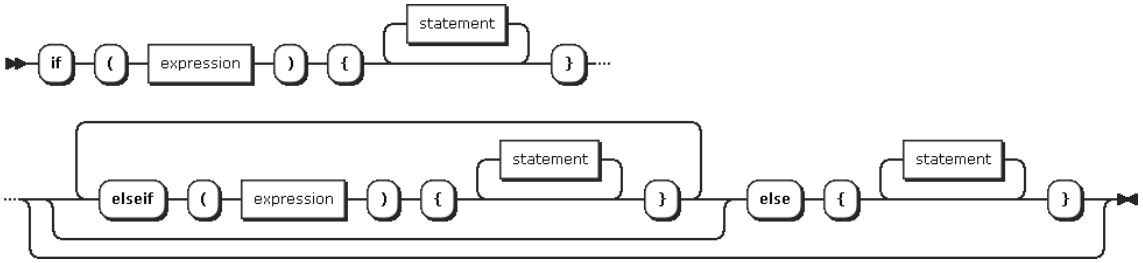
```
// inserting ADOscript code inside the definition of an algorithm
algorithm MyAlgorithm {
    // ADOitem is defined with embed statement before this code
    insert ADOitem

    // creating a simple infobox
    ui.infobox title "MM-DSL Info-Box"  text "Embedded from ADOxx"
}

// same algorithm with a native MM-DSL code
algorithm MyOtherAlgorithm {

    // creating a simple menu item
    ui.item.menu.insert MyOtherAlg to MyOtherMenu
}

algorithm MyAlgorithmTwo {

    // creating a simple warningbox
```

```
    ui.warningbox title "MM-DSL Warning-Box"
       text "Generated Warning with MM-DSL" button def-ok
}

// algorithm with inserted foreigh code that is defined in embed statements
algorithm URIImport {

    // embed the AdoScript code
    insert URIImportItem
    insert URIImportAlgorithm
}
```

9.5.2 Selection Statement

Syntax

selectionstatement




```
selectionstatement ::=
  ('if' '(' expression ')' '{' statement* '}') (('elseif' '(' expression ')'
  '{' statement* '}')* 'else' '{' statement* '}')?
```

Semantics

Selection statement is used to select one of multiple code blocks and execute it if a condition is satisfied. Each selection statement is divided into three parts: if part (starts with a keyword *if* and it is mandatory), elseif part (starts with a keyword *elseif*) and else part (starts with a keyword *else*). If elseif part is used, it is also mandatory to use else part. However, else part can be used without the presence of elseif part. This behavior is well known from the typical programming languages such as C, C++ or Java.

If part and else part need to contain an expression defining a condition. This expression is put in the brackets following the keyword (see the example below).

Example

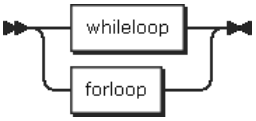
```
// simple selection statement inside an algorithm
algorithm IfElseAlgorithm {
    var someVariable = 1

    if(someVariable == 1)
    {
```

```
        ui.infobox title "If-Else Test" text "Variable is equal 1"
    }
    elseif(someVariable != 1)
    {
        ui.infobox title "If-Else Test" text "Variable is not equal 1"
    }
    else
    {
        ui.infobox title "If-Else Test" text "Variable value is unknown"
    }
}
```

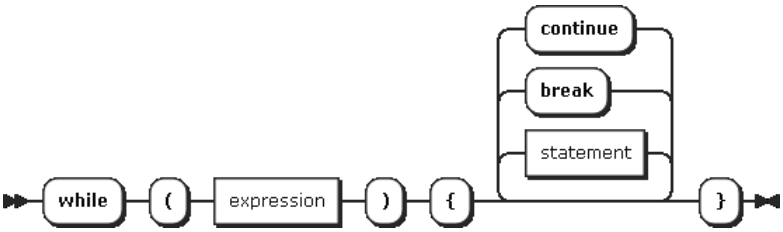
9.5.3 Loop Statement

Syntax

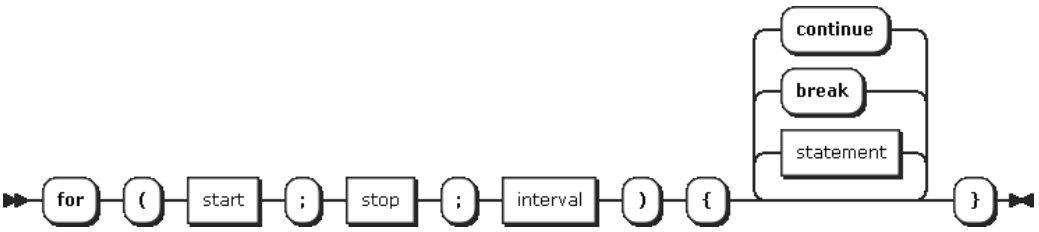
loopstatement



whileloop



forloop




```
loopstatement ::=
  whileloop | forloop
whileloop ::=
  'while' '(' expression ')' '{' (statement | ('break' | 'continue'))* '}'
forloop ::=
  'for' '(' start ';' stop ';' interval ')'
  '{' (statement | ('break' | 'continue'))* '}'
```

Semantics

Loop statement contains two statements that loop through the corresponding code block: while and for statement.

While statement starts with the keyword *while*, followed by an expression defining the condition. If the condition is fulfilled the code inside the curly brackets ({. }) will be executed. After the code is executed, the condition is re-evaluated. The execution of the code block will continue until the condition is no longer true.

For statement starts with the keyword *for*, followed by an expression defining the start and stop conditions, as well as the interval that will be applied on the start condition after each iteration. Loop stops when the stop condition is reached.

Both of these statements are very similar to the typical for and while statements found in the programming languages such as C, C++ and Java. For the simple utilization scenarios see the examples below.

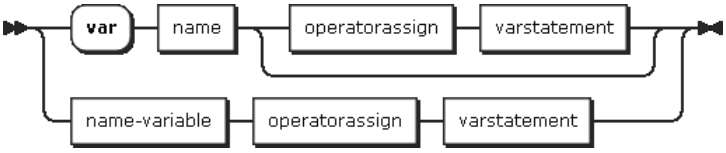
Example

```
// simple while and for statements
algorithm WhileAndForAlgorithm
{
   var someVariable = 1

   // repeat while someVariable is equal to 1
   while(someVariable == 1)
   {
      ui.item.context.insert DoSomething to Modeling
   }

   // repeat always
   while (true)
   {
      ui.errorbox title "Error" text "Infinite loop!" button def-ok
   }

   // repeat 10 times
   for(0;10;1)
   {
      ui.infobox title "Info" text "This is an info"
   }
}
```

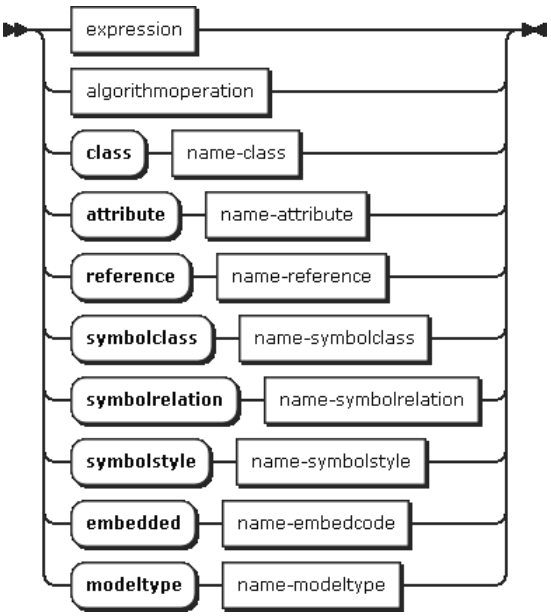
9.5.4 Variable Statement

Syntax

variable



varstatement




```
variable ::=
  ('var' name (operatorassign varstatement)?) |
  (name-variable operatorassign varstatement)
varstatement ::=
  expression | algorithmoperation | ('class' name-class) |
  ('attribute' name-attribute) | ('reference' name-reference) |
  ('symbolclass' name-symbolclass) | ('symbolrelation' name-symbolrelation) |
  ('symbolstyle' name-symbolstyle) | ('embedded' name-embedcode) |
  ('modeltype' name-modeltype)
```

Semantics

Variable statement is used to declare and assign variables. Variable declaration starts with the keyword *var* following a unique identifier (*name*).

A value that a variable hold is determined through the variable assignment. The value can be a simple data type, value of an attribute, class, relation, reference, model type, even algorithm operation or a complex expression.

Variables declared and initialized in one algorithm can be used, by invoking them through the fully qualified name, in another algorithm. See the examples for more detail.

Example

```
// simple variable statements
algorithm VariableStatements
{
   // simple data type assignment
   var simpleVariable1 = 1 + 2
   var simpleVariable2 = simpleVariable2 - 5

   // using variable from another algorithm as assignment
   var otherAlgVariable = WhileAndForAlgorithm.someVariable

   // assigning attribute value
   var attributeVariable1 = attribute Vehicle.Color

   // assigning class value
   var classVaraible = class Vehicle

   // assigning reference value
   var referenceVaraible = reference City.carParksInCity

   // assigning model type value
   var modeltypeVaraible = modeltype ParkingMap

   // assigning embedded code
   var embeddedVaraible = embedded ADOitem

   // assigning algorithm operations
   var operationVaraible = ui.infobox title "Variable Info" text "Assigned"
}
```

9.5.5 Expressions

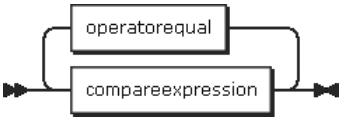
Syntax

expression

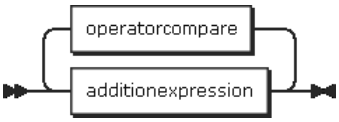
orexpression

andexpression

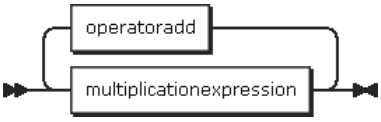
equalexpression



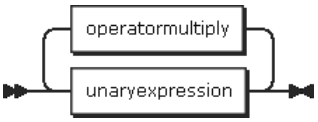
compareexpression



additionexpression



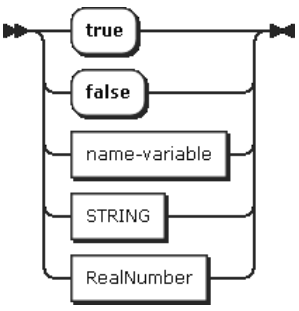
multiplicationexpression



unaryexpression

primaryexpression

atomicexpression




```
expression ::=
  orexpression
orexpression ::=
  andexpression (operatoror andexpression)*
andexpression ::=
  equalexpression (operatorand equalexpression)*
```

```
equalexpression ::=
  compareexpression (operatorequal compareexpression)*
compareexpression ::=
  additionexpression (operatorcompare additionexpression)*
additionexpression ::=
  multiplicationexpression (operatoradd multiplicationexpression)*
multiplicationexpression ::=
  unaryexpression (operatormultiply unaryexpression)*
unaryexpression ::=
  operatorunary? primaryexpression
primaryexpression ::=
  atomicexpression | ( '(' orexpression ')' )
atomicexpression ::=
  'true' | 'false' | name-variable | STRING | RealNumber
```

Semantics

Expression statements are used in combination with other algorithm statements to express various conditions and conduct specified operations. How and in which order will these operations be executed depends on the operators' precedence. See Table V for the detailed overview.

The definition of expressions and operators is divided in this language specification because of easier readability. In reality, these two grammar parts are bound with each other. Without expressions, operators would not make much sense.

MM-DSL defines the following expressions: atomic, primary, unary, multiplication, addition, comparison, equality, and, and or expression. Only expression that contains the terminal symbols is the atomic expression. Every other expression is a complex expression which is utilizing the atomic expression directly or transitively.

Example

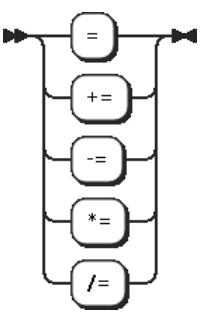
```
// simple expression statements
algorithm ExpressionAlgorithm
{
   // addition and multiplication expressions
   var expr1 = (1+2) * 3

   // and expression
   var expr2 = expr1 || true
   var expr3 = expr2 && false

   // comparison expression
   if(expr1 > 0)
   {
      // do something
   }
```

```
    // equality expression
    while(expr2 != true)
    {
        // do something
    }

    // assignment expressions
    var expr4 = 2
    expr4 += expr1

    // unary (negation) expression
    if(!(expr2 != expr3))
    {
        // do something
    }

    // using attribute values in expressions
    var vehicleColor = attribute Vehicle.Color

    if(vehicleColor == "blue")
    {
        ui.infobox title "Vehicle Color" text "Vehicle is blue"
    }
}

// more expressions
algorithm MyAlgorithmCounter {

    ui.infobox title "Info" text "Hello"

    var counter = 0

    while(counter < 0){
        ui.infobox title "InWhile" text "While"
        counter -= 1
    }

    var abcd = 4 * 3 + 1 / 53
    abcd += (3 + 3 + 3) / 3
}
```

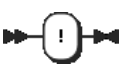
9.5.6 Operators

Syntax

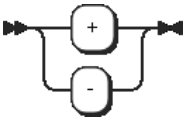
operatorassign



operatorunary

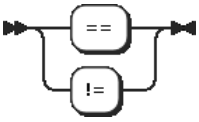


operatoradd

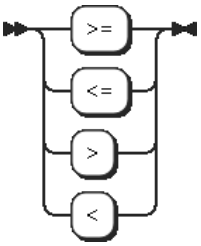


operatorand

operatorequal

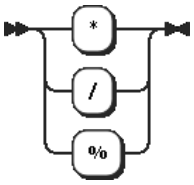


operatorcompare



operatoror

operatormultiply




```
operatorassign ::=
  '=' | '+=' | '-=' | '*=' | '/='
operatorunary ::=
  '!'
operatormultiply ::=
  '*' | '/' | '%'
operatoradd ::=
  '+' | '-'
operatorcompare ::=
  '>=' | '<=' | '>' | '<'
operatorequal ::=
  '==' | '!='
operatorand ::=
  '&&'
operatoror ::=
  '||'
```

Semantics

Operators defined in here are used inside expressions. The ones that are grouped behind the same name (e.g., operatorassign) have the same precedence and will be evaluated from left to right. Take note that the terminal symbols assigned to each operator can be seamlessly changed. The ones that are used right now are conform to the symbols used in well-known programming languages.

Meaning of the operators and their precedence can be found in Table V. The operators are ordered from higher to lower precedence. The operator group in the first row has the highest precedence.

Table V: Operators - Precedence and Meaning

Operator Group	Operators	Meaning
Unary	!	Negating the expression on the right, if expression is true, returns false, and vice versa.
Multiplication	*, /, %	Multiplies, divides or modulo left expression with the right expression, returns the operation result.
Addition	+, -	Adds or subtracts left expression with the right expression, returns the operation result.
Comparison	>=, <=, >, <	Compares the objects on the left side to the one on the right side, returns true or false.
Equality	==, !=	Checks if the expression on the left is equal to the expression on the right.
And	&&	logical and between expression on the left and right side, returns true or false
Or	||	Logical or between the expressions on the left and right side, returns true or false.
Assignment	=, +=, -=, *=, /=	Assigns the right expression to the left expression; in case of multi assign, returns the operation result and assigns it to the left expression.

Example

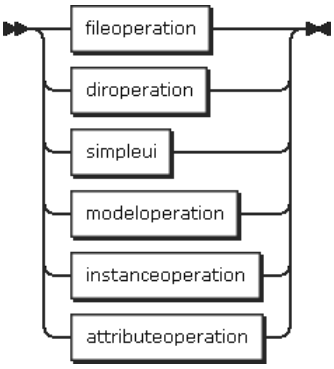
The examples on how to use the operators can be found in the part where expressions are explained. Their usage is very similar to the nowadays general programming practices.

9.5.7 Algorithm Operation

The following functionality is included to cover the most basic model manipulation functions of a metamodeling platform. It is best to view this group of statements as an extension on the basic MM-DSL. Thus, the following statements are most prone to change when aligning the MM-DSL translator (e.g., compiler) to a particular platform. The ones described in here have been designed to work seamlessly with the API available in the ADOxx metamodeling platform.

Syntax

```
algorithmoperation
```




```
algorithmoperation ::=
  fileoperation | diroperation | simpleui | modeloperation |
  instanceoperation | attributeoperation
```

Semantics

Algorithm operation statement defines operations that can be carried out inside a modeling tool. These are operations on: files and directories, models, object instances, and attributes. Basic user interface functions are also included.

The algorithm operation statements can be utilized together with other operations statements to describe the dynamic logic of a modeling tool.

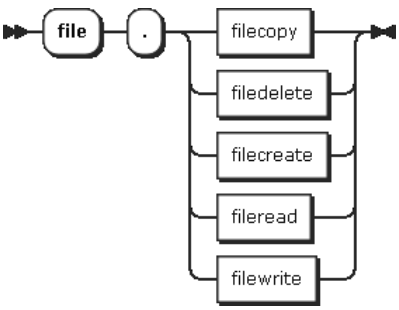
Example

It is not possible to give an example for this statement because it only shows all the possible algorithm operation statements and contains no terminals. The examples will be given as a part of the algorithm operation sub-statements descriptions.

9.5.8 File Operation

Syntax

```
fileoperation
```



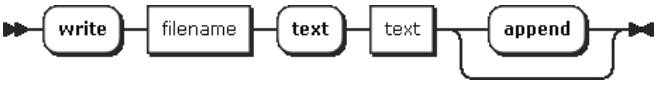
```
filecopy
```


```
filedelete
```


```
filecreate
```


```
fileread
```


```
filewrite
```




```
fileoperation ::=
  'file' '.' (filecopy | filedelete | filecreate | fileread | filewrite)
filecopy ::=
  'copy' 'source' src 'destination' dest
filedelete ::=
  'delete' filename
filecreate ::=
  'create' filename
fileread ::=
  'read' filename
filewrite ::=
  'write' filename 'text' text ('append')?
```

Semantics

File operation statement provides a set of functions that work with files. These functions are: copy, delete, create, read and write. This statements starts with a keyword *file* followed by a dot (.) and the name (which is also a keyword) of a function we want to utilize. Parameters that follow the function depend on the function in use. Copy requires the source and destination locations (file paths). Delete, create, read and write require the file name (file path). Additionally, write has an option to append data at the end of a file. See Table VI for a detailed description of all available functions.

Table VI: File Operation Statement Functions

Function	Parameters	Description
copy	src, dest	Copies the file which location is given in *src* to the location given in *dest*. If the destination file exists, it will be overwritten.
delete	filename	Deletes the file which location is given in *filename*.
create	filename	Creates a new file at the location given in *filename*.
read	filename	Reads the file which location is given with *filename*. The function returns the content of file.
write	filename, text, (append)	Write the content found in *text* to the file which is specified in *filename*. Optionally, *append* indicates that the content in *text* will be appended at the end of a file. Without append option, the content of a file will be deleted before the new content is written.

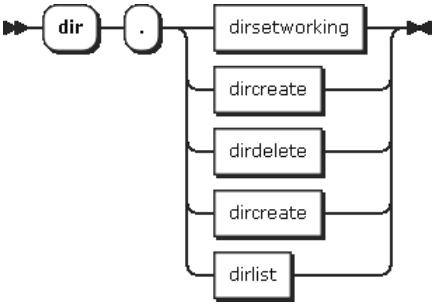
Example

```
// examples for file operation statement
algorithm FileOperations
{
   // copy file
   file.copy source "C:\\Temp\\SourceFile.mml"
      destination "C:\\Temp\\DestFile.mml"

   // delete file
   file.delete "C:\\Temp\\SourceFile.mml"

   // create file
   file.create "C:\\Temp\\SourceFile.mml"

   // read file
   var fileContent = file.read "C:\\Temp\\SourceFile.mml"

   // write file
   file.write "C:\\Temp\\SourceFile.mml"
      text "This is the content" append
}
```

9.5.9 Directory Operation

Syntax

diroperation



dirsetworking

dircreate

dirdelete

dirlist


```
diroperation ::=
  'dir' '.' (dirsetworking | dircreate | dirdelete | dircreate | dirlist)
dirsetworking ::=
  'set' dirname
dircreate ::=
  'create' dirname
dirdelete ::=
  'delete' dirname
dirlist ::=
  'list' dirname
```

Semantics

Directory operation statement provides a set of functions that work with directories. These functions are: set, create, delete and list. Every directory operation statement starts with the keyword *dir* followed by a dot (.) and the name of a function (which is also a keyword) we want to use. All the functions require the directory name parameter (directory path). See Table VII for a detailed description of all available functions.

Table VII: Directory Operation Statement Functions

Function	Parameters	Description
set	dirname	Set the directory given in *dirname* as a working directory.
create	dirname	Crete a directory at the location given with *dirname*.
delete	dirname	Delete the directory at the location given with *dirname*.
list	dirname	List all the sub-directories of a working directory. Default working directory is set to the current MM-DSL file directory.

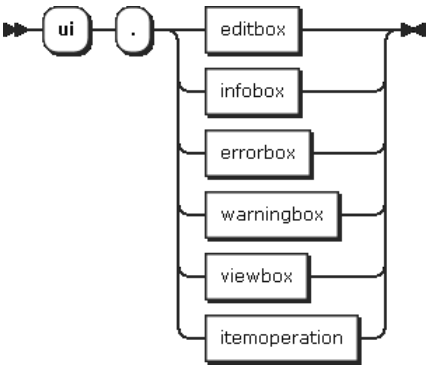
Example

```
// examples for directory operation statement
algorithm DirOperations
{
   // set working directory
   dir.set "C:\\Temp"

   // create directory
   dir.create "C:\\Temp"

   // delete directory
   dir.delete "C:\\Temp"

   // list files and direcotries in a directory
   // directory list is saved in a variable
   var dirContent = dir.list "C:\\Temp"
}
```

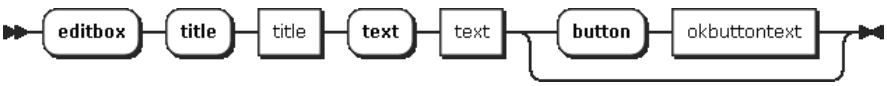
9.5.10 Simple User Interface

Syntax

simpleui



editbox



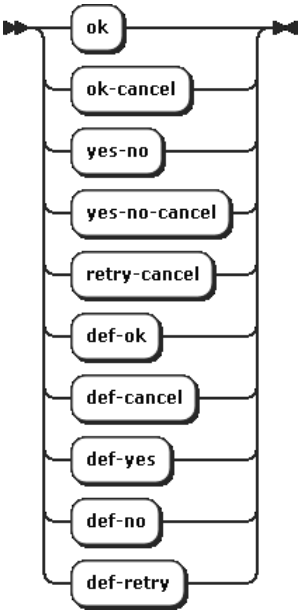
infobox

errorbox

warningbox

viewbox

buttontype




```
simpleui ::=
  'ui' '.' (editbox | infobox | errorbox | warningbox |
  viewbox | itemoperation)
editbox ::=
  'editbox' 'title' title 'text' text ('button' okbuttontext)?
infobox ::=
  'infobox' 'title' title 'text' text
errorbox ::=
  'errorbox' 'title' title 'text' text 'button' buttontype
warningbox ::=
  'warningbox' 'title' title 'text' text 'button' buttontype
viewbox ::=
  'viewbox' 'title' title 'text' text
```

```
buttontype ::=
  'ok' | 'ok-cancel' | 'yes-no' | 'yes-no-cancel' | 'retry-cancel' |
  'def-ok' | 'def-cancel' | 'def-yes' | 'def-no' | 'def-retry'
```

Semantics

Simple user interface (or simple UI) statement provides functions for construction of basic user interface objects. Currently, the following functions are supported: editbox, infobox, errorbox, warningbox, and viewbox. Every simple UI statement begins with the keyword *ui*, followed by a dot (.) and a function name (which is also a keyword). Each function has a set of parameters. See Table VIII for details. The realization of UI objects strongly depends on the execution platform possibilities. For example, infobox function may produce different results depending on the destination platform.

Table VIII: Simple User Interface Statement Functions

Function	Parameters	Description
editbox	title, text, (okbuttontext)	Creates an edit box with a title and text indicated in the parameters; optionally, text on the “ok” button can be specified.
infobox	title, text	Creates an info box with a title and text indicated in the parameters.
errorbox	title, text, buttontype	Creates an error box with a title and text indicated in the parameters; optionally, different button constellations can be specified.
warningbox	title, text, buttontype	Creates a warning box with a title and text indicated in the parameters; optionally, different button constellations can be specified.
viewbox	title, text	Creates a view box with a title and text indicated in the parameters.

Example

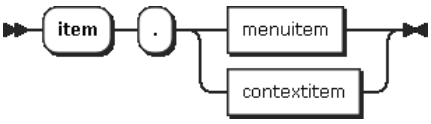
```
// examples for simple UI statement
algorithm SimpleUI
{
   // edit box with a custom button
   ui.editbox title "Edit" text "Cool Message" button "Press Me"

   // info box
   ui.infobox title "Info" text "Info Message"

   // error box with ok and cancel buttons
   ui.errorbox title "Error" text "This is an error" button ok-cancel
```

```
    // warning box with ok button
    ui.warningbox title "Warning" text "This is a warning" button ok
}
```

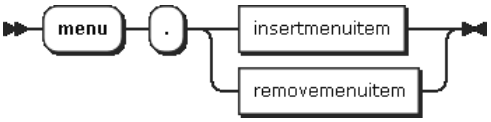
9.5.11 Item Operation

Syntax

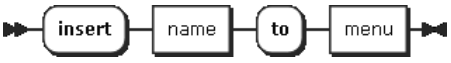
itemoperation



menuitem

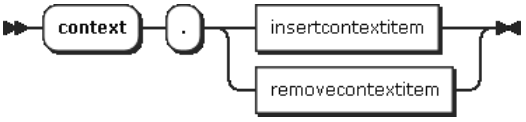


insertmenuitem

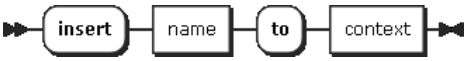


removemenuitem

contextitem



insertcontextitem



removecontextitem


```
itemoperation ::=
  'item' '.' (menuitem | contextitem)
menuitem ::=
  'menu' '.' (insertmenuitem | removemenuitem)
insertmenuitem ::=
  'insert' name 'to' menu
removemenuitem ::=
  'remove' name-menuitem
contextitem ::=
  'context' '.' (insertcontextitem | removecontextitem)
```

```
insertcontextitem ::=
  'insert' name 'to' context
removecontextitem ::=
  'remove' name-contextitem
```

Semantics

Item operation statement is a simple UI sub-statement. It creates the user interface menu items: normal menu items or context menu items. Both types of items have the following two functions: insert and remove. See Table IX for the detailed description.

Item operation statement begins with the keyword *item* followed by a dot (.) and the type of an item: *menu* or *context*. A menu item is a graphical object that can be typically found at the top-left corner of an application window. A context item is a graphical element that is located inside a menu which is accessed by right clicking inside an application window. Name of an item, and the menu parameter are unique identifiers.

Table IX: Item Operation Statement Functions

Function	Parameters	Description
menu.insert	name, menu	Inserts a menu item in the menu indicated with the menu parameter. If a menu does not exist, it will be created.
menu.remove	name-menuitem	Delete the menu item with a given name.
context.insert	name, context	Inserts a context item in the context menu indicated with the menu parameter. If a context menu does not exist, it will be created.
context.remove	name-contextitem	Delete the context item with a given name.

Example

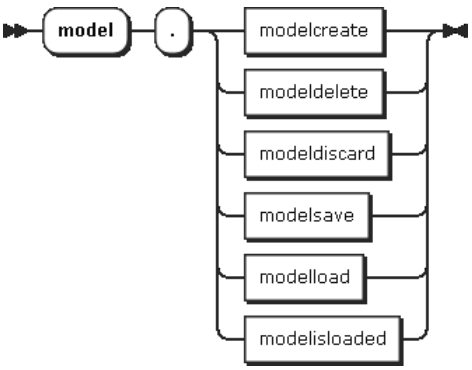
```
// examples for simple UI menu and context item operation statement
algorithm ItemOperations
{
    var menuItemName = "Cool Menu Item"
    var menuName = "Modeling"
    var contextMenuName = "Context Modeling"

    // insert menu item inside a menu
    ui.item.menu.insert menuItemName to menuName

    // remove menu item
    ui.item.menu.remove menuItemName

    // insert context item
    ui.item.context.insert menuItemName to contextMenuName
```

```
    // remove context item
    ui.item.context.remove menuItemName
}
```

9.5.12 Model Operation

Syntax

modeloperation



modelcreate

modeldelete

modeldiscard

modelsave

modelload

modelisloaded


```
modeloperation ::=
  'model' '.' (modelcreate | modeldelete | modeldiscard | modelsave |
  modelload | modelisloaded)
modelcreate ::=
  'create' name name-modeltype
```

```
modeldelete ::=
  'delete' name-model
modeldiscard ::=
  'discard' name-model
modelsave ::=
  'save' name-model
modelload ::=
  'load' name-model
modelisloaded ::=
  'isloaded' name-model
```

Semantics

Model operation statement provides a set of functions that work with models. These functions are: create, delete, discard, save, load and isloaded. Each of these functions requires some parameters. For a detailed description of functions see Table X.

Table X: Model Operation Statement Functions

Function	Parameters	Description
create	name, name-modeltype	Create a model with the given name from the indicated model type. (Instantiates a model type.)
delete	name-model	Delete a model with the given name. Model will be deleted from the repository.
discard	name-model	Discard a model with the given name. Model will be removed from the memory. It will still be present in the repository.
save	name-model	Save a model with the given name to the repository.
load	name-model	Load a model with the given name to the memory.
isloaded	name-model	Check if the indicated model is currently loaded in the memory.

Example

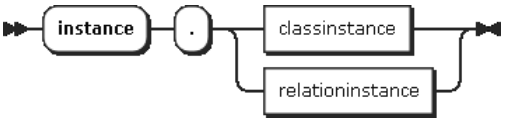
```
// examples for the model operations
algorithm ModelOperations
{
   // create a model from a model type ParkingMap
   model.create MyModel ParkingMap

   // alternatively we can use variable
```

```
    var myModel = "Super Parking Garage"
    model.create myModel ParkingMap

    // delete model
    model.delete MyModel

    // delete model inside another algorithm
    model.delete MyAlgorithm.MyModel

    // discard model
    model.discard MyModel

    // save model
    model.save MyModel

    // load model
    model.load MyModel

    // check if model is loaded
    var isLodaed = model.isloaded MyModel
}
```

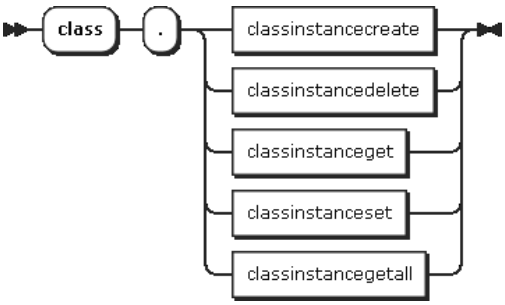
9.5.13 Instance Operation

Syntax

instanceoperation

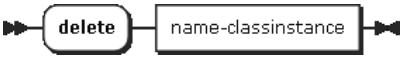


classinstance

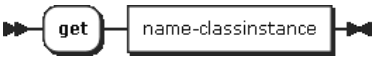


classinstancecreate

classinstancedelete



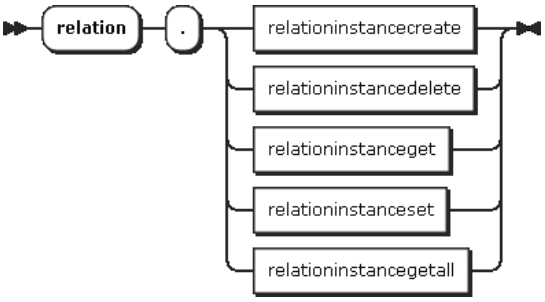
classinstanceget



classinstanceset

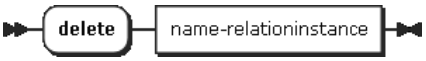
classinstancegetall

relationinstance

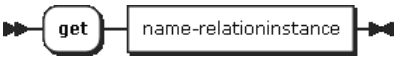


relationinstancecreate

relationinstancedelete



relationinstanceget



relationinstanceset

relationinstancegetall


```
instanceoperation ::=
  'instance' '.' (classinstance | relationinstance)
classinstance ::=
  'class' '.' (classinstancecreate | classinstancedelete |
  classinstanceget | classinstanceset | classinstancegetall)
classinstancecreate ::=
  'create' name name-class
classinstancedelete ::=
  'delete' name-classinstance
classinstanceget ::=
  'get' name-classinstance
classinstanceset ::=
  'set' name-classinstance
```

```
classinstancegetall ::=
  'getall' name-class
relationinstance ::=
  'relation' '.' (relationinstancecreate | relationinstancedelete |
  relationinstanceget | relationinstanceset | relationinstancegetall)
relationinstancecreate ::=
  'create' name 'from' name-classinstancefrom 'to' name-classinstanceto
relationinstancedelete ::=
  'delete' name-relationinstance
relationinstanceget ::=
  'get' name-relationinstance
relationinstanceset ::=
  'set' name-relationinstance
relationinstancegetall ::=
  'getall' name-relation
```

Semantics

Instance operation statement provides a set of functions that work with instances. Instance is a specific realization of an object. An object can be a class or a relation. Instance operation statement starts with the keyword *instance* followed by a dot (.), type of the instance (either *class* or *relation*) and a function name. This statement is used to create and delete instances. It can also be used to get instance-specific information. For a detailed overview of all the supported functions see Table XI. Note that, although class and relation use the same functions, the parameters are different.

Table XI: Instance Operation Statement Functions

Function	Parameters	Description
class.create	name, name-class	Create a class instance from the class given through parameter *name-class*.
class.delete	name-classinstance	Delete the class instance given as the parameter.
class.get	name-classinstance	Get the class instance given as the parameter.
class.set	name-classinstance	Set the class instance given as the parameter.
class.getall	name-class	Get all instances of a class given by the parameter.
relation.create	name, name-relation, name-classinstancefrom, name-classinstanceto	Create a relation instance from the relation given through parameter parameter *name-relation*; associated from and and

		class instances are also given.
relation.delete	name-relationinstance	Delete the relation instance given as the parameter.
relation.get	name-relationinstance	Get the relation instance given as the parameter.
relation.set	name-relationinstance	Set the relation instance given as the parameter.
relation.getall	name-relation	Get all instances of a given relation given by the parameter.

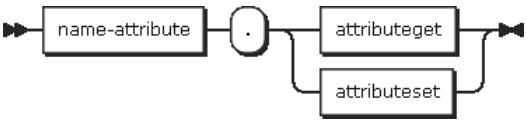
Example

```
// examples for the instance operations
algorithm InstanceOperations
{
   var classInstanceName = "Turbo 56"
   var secondClassInstanceName = "Mazda 626"
   var relationInstanceName = "Associated with Turbo 56"

   // create a class instance
   instance.class.create classInstanceName Vehicle
   instance.class.create secondClassInstanceName Vehicle


   // delete class instance
   instance.class.delete classInstanceName

   // get class instance
   var classInstance = instance.class.get secondClassInstanceName

   // get all instances of a class
   var allInstancesOfVehicles = instance.class.getall Vehicle

   // create a relation instance
   instance.relation.create relationInstanceName IsParked
      from classInstanceName to secondClassInstanceName

   // delete relation instance
   instance.relation.delete relationInstanceName

   // get relation instance
   var relationInstance = instance.relation.get relationInstanceName

   // get all instances of a relation
   var allInstancesOfIsParked = instance.relation.getall IsParked
}
```

9.5.14 Attribute Operation

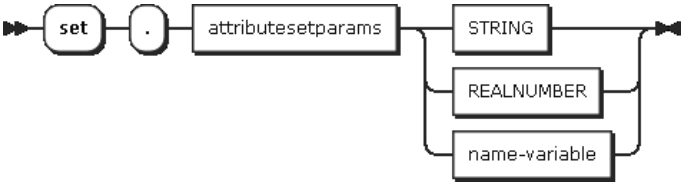
Syntax

attributeoperation

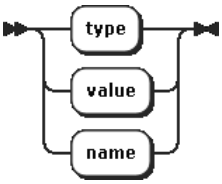


attributeget

attributeset



attributegetparams



attributesetparams


```
attributeoperation ::=
  name-attribute '.' (attributeget | attributeset)
attributeget ::=
  'get' '.' attributegetparams
attributegetparams ::=
  ('type' | 'value' | 'name')
attributeset ::=
  'set' '.' attributesetparams (STRING | REALNUMBER | name-variable)
attributesetparams ::=
  'value'
```

Semantics

Attribute operation statement manipulates with the properties of an attribute, particularly its values. It starts with a fully qualified name of an already defined attribute, followed by a dot (.) and a function name. Currently there are only two functions: get and set. See Table XII for the function description. The set function initializes the attribute val-

ues which can be used during the instantiation of an object (e.g., class or relation). Set and get currently do not work on instances.

Table XII: Attribute Operation Statement Functions

Function	Parameters	Description
set	value	Sets the value of an attribute.
get	type, value, or name	Gets either a type, value or a name of an attribute.

Example

```
// examples for the attribute operations
algorithm AttributeOperations
{
   // set attribute value through a variable
   var colorBlue = "blue"
   Vehicle.Color.set.value colorBlue

   // set attribute value directly
   Vehicle.Color.set.value "violet"

   // get attribute value
   var attributeValue = Vehicle.Color.get.value

   // get attribute name
   var attributeName = Vehicle.Color.get.name
}
```

9.5.15 Event

Syntax

event


```
event ::=
  'event' eventname '.' 'execute' '.' name-algorithm
```

Semantics

Event statement defines events that trigger algorithms. An event is an occurrence inside a modeling tool. When a change is detected, certain events are triggered. For example, opening a new model or instantiating a certain object may trigger an event. The statement starts with the keyword *event*, followed by a predefined event name, an ac-

tion that will be taken (currently only execute), and with a name of an algorithm that will be triggered.

Some of the supported and already recognized events in MM-DSL are the following: CreateInstance, CreateModel, CreateRelationInstance, DeleteInstance, DeleteModel, DeleteRelationInstance, OpenModel, SetAttributeValue, ToolInitialized, and so on. If an even is supported depends on the provided metamodeling platform functionality.

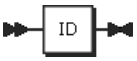
Example

```
algorithm MyAlgorithmTwo {
   ui.warningbox title "MM-DSL Warning-Box"
   text "Generated Warning with MM-DSL" button def-ok
}
// execute this event when any instance is deleted (e.g., class, relation)
event.DeleteInstance.execute.MyAlgorithmTwo
```

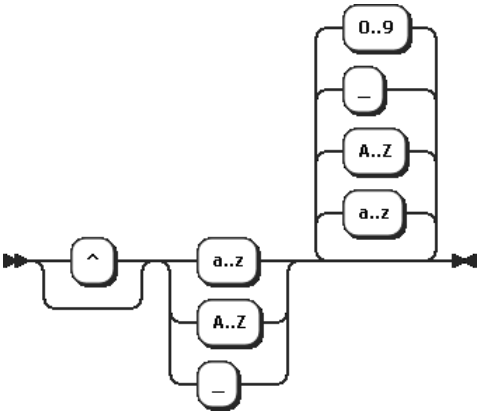
9.5.16 Terminals

Syntax

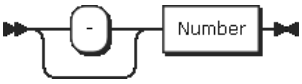
name



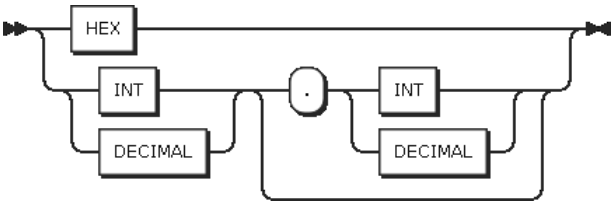
ID



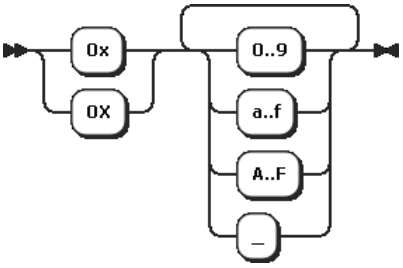
REALNUMBER



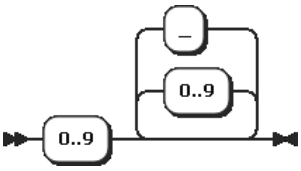
NUMBER



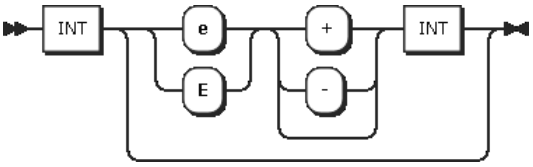
HEX



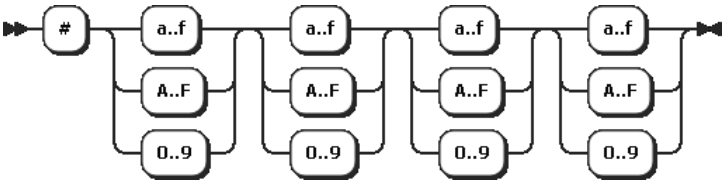
INT



DECIMAL



HEXCOLOR




```
name ::=
  ID
ID ::=
  '^'?('a..z'|'A..Z'|'_') ('a..z'|'A..Z'|'_'|'0..9')*
REALNUMBER ::=
  ('-')? Number
NUMBER ::=
  (HEX | (INT | DECIMAL) ('.' (INT | DECIMAL))?)
HEX ::=
  ('0x'|'0X') ('0..9'|'a..f'|'A..F'|'_')+
INT ::=
  '0..9' ('0..9'|'_')*
DECIMAL ::=
  INT (('e'|'E') ('+'|'-')? INT)?
HEXCOLOR ::=
  '#' ('a..f'|'A..F'|'0..9') ('a..f'|'A..F'|'0..9')
  ('a..f'|'A..F'|'0..9') ('a..f'|'A..F'|'0..9')
```

Semantics

Terminals are the symbols that, together with the language's keywords, build the actual program code. Some of the terminal symbols have already been presented through many of the grammar rules. The ones mentioned here are the ones constructing the basic data types of a MM-DSL program.

Example

Examples for the terminals can be found throughout this specification, particularly in the part that describes operations statements (see expression examples).

9.6 Programming Concepts

Upon closer inspection of the MM-DSL specification it can be seen that the MM-DSL program has a specific flow, where some concepts need to be defined before the others. General rule is that a concept needs to be defined before it can be used. To make translation (e.g., compilation) faster, the concepts that are used by other concepts need to be present in the code before they can be referred to. Figure 27 shows the abstract structure of a MM-DSL program, including all of its concepts.

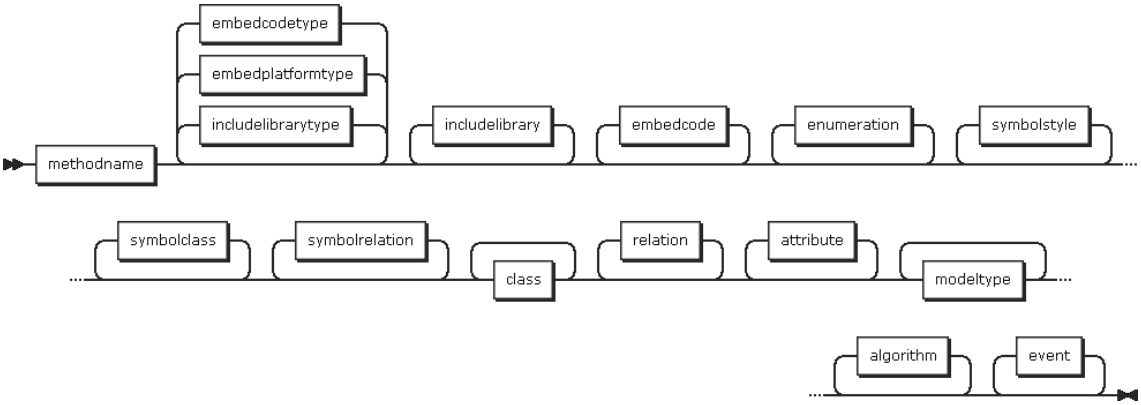


Figure 27: Abstract Structure of a MM-DSL Program

MM-DSL program is divided into several code blocks. Some of the code blocks are optional, while the others are mandatory. It begins with a statement that defines the modeling method name, continues through the entire relevant statements, and ends with the event statements. All of these statements and their utilization have been described in detail in the previous sections. This section will give focus to the most important programming concepts used when writing a MM-DSL modeling method description.

9.6.1 Smallest Working Program

As every other computer language, MM-DSL also has the smallest working program. It is a program that is fully functional, valid, and can be translated into a modeling tool. It consists of a modeling method name, a single class and a model type.

```
// the smallest describable method possible
method SmallestInstantiable

// empty class definition
class Abc {}

// the simplest model type contains only one class
modeltype ModelTypeA
{
   classes Abc
   relations none
   modes none
}
```

9.6.2 Inheritance

The concept of inheritance is often used while defining the structure of a modeling method. Thus, structure statements, such as class and relation statement are utilizing it.

```
// small method with inheritance and modes
method Minimalistic

// ClassB inherits from ClassA
class ClassA {}
class ClassB extends ClassA {}

// RelationB inherits from RelationA
relation RelationA from ClassA to ClassB {}
relation RelationB extends RelationA from ClassA to ClassB {}

modeltype NiceModel
{
   classes ClassA ClassB
   relations RelationA RelationB

   modes
   mode ModeA include
      classes ClassA
      relations RelationA
   mode ModeB include
      classes ClassB
      relations RelationB
}
```

9.6.3 Referencing

Referencing by an identifier is a concept used throughout the whole MM-DSL program. Almost each defined concept that has an identifier associated with it can be referenced.

```
// using referencing by identifier to reduce lines of code
method AlgMethod
def IncludeLibraryType ADOxxMetamodel
def EmbedPlatformType ADOxx
def EmbedCodeType GraphRep
def EmbedCodeType ADOscript

include <MyAwesomeLibrary:ADOxxMetamodel>

embed ADODrawRoundRectangle <ADOxx:GraphRep>
{
"embedded code goes here"
}

enum OneZero { "One" "Zero" }

// referencing a style
classgraph RoundRect style IsParkedGraph.Black
{
   circle cx=0 cy=0 r=20
   circle cx=0 cy=0 r=30

   // referencing the embedded code
   insert ADODrawRoundRectangle
}

class ClassF
{
   attribute alive : string
   attribute color : int

   // referencing an enumeration
   attribute available : enum OneZero
}

modeltype MyModelTyp
{
   // referencing a class
   classes ClassF
   relations none
   modes none
}

algorithm MyAlgorithm
{
   // create and discard a model from a referenced model type MyModelType
   model.create MyModel MyModelTyp

   //referencing a model
   model.discard MyModel
}

// referencing an algorithm inside an event
event.AfterCreateModelingConnector.execute.MyAlgorithm
```

9.6.4 Embedding

Code embedding is a powerful concept that allows for code segments that have been written in another language. The primary use is to include the code that has already been written in a format that a metamodeling platform understands. During the translation of a MM-DSL program the embedded code will be copied as verbatim in the specified places.

```
// an algorithm written in AdoScript
embed URIImportItem <ADOxx:AdoScript> {
"ITEM \\\"URI import\\\" modeling:\\\"~AdoScripts\\\" pos2:1"
}
embed URIImportAlgorithm <ADOxx:AdoScript> {
// always put \\\ before special characters and quotations (e.g. \\\\n, \\\")
"
SETL cls_name:(\\\"ParkingArea\\\")
SETL mod_type_name:(\\\"ParkingMap\\\")
SETL attr_uri_name:(\\\"URI\\\")
SETL obj_cnt:(0)

CC \\\"Modeling\\\" GET_ACT_MODEL
SETL pm_id:(modelid)
IF (pm_id = -1) {
  CC \\\"AdoScript\\\" ERRORBOX (\\\"The selected model could not be deter-
mined.\\\\nMake sure a model of type \\\" + mod_type_name + \\\" is opened
and selected.\\\")
  EXIT
}

CC \\\"Core\\\" GET_MODEL_MODELTYPE modelid:(pm_id)
IF (modeltype != mod_type_name) {
  CC \\\"AdoScript\\\" ERRORBOX (\\\"The selected model is of the wrong
type.\\\\nMake sure a model of type \\\" + mod_type_name + \\\" is opened and
selected.\\\")
  EXIT
}

CC \\\"AdoScript\\\" EDITBOX text:(\\\"\\\") title:(\\\"Enter URIs\\\") ok-
text:(\\\"Create\\\")
IF (endbutton != \\\"ok\\\") {
  EXIT
}
SETL uris:(text)

CC \\\"Core\\\" GET_CLASS_ID classname:(cls_name)
SETL cls_id:(classid)

CC \\\"Core\\\" GET_ATTR_ID classid:(cls_id) attrname:(attr_uri_name)
SETL attr_uri_id:(attrid)

FOR uri in:(uris) sep:(\\\"\\\\n\\\") {
  IF (LEN uri > 1) {
    CC \\\"Core\\\" CREATE_OBJ modelid:(pm_id) classid:(cls_id) ob-
jname:(cls_name + \\\"-\\\" + STR obj_cnt)
    SETL obj_id:(objid)
    CC \\\"Core\\\" SET_ATTR_VAL objid:(obj_id) attrid:(attr_uri_id)
val:(uri)
```

```
    CC \\\"Modeling\\\" SET_OBJ_POS objid:(obj_id) x:(2cm) y:(1cm+CM (1.5 *
obj_cnt))
    SETL obj_cnt:(obj_cnt + 1)
  }
}
"
}

// AdoScript algorithm embedded inside the MM-DSL code
algorithm URIImport {
   // embed the AdoScript code
   insert URIImportItem
   insert URIImportAlgorithm
}
```

9.6.5 Auto-generation

MM-DSL does not require the full specification of every modeling method element. Thus, the ones that are not explicitly defined in the program will be automatically generated. This concept is important for fast prototyping and testing of modeling method implementations on a particular metamodeling platform. Most of the platforms require that all the elements are present before modeling tool can be created.

The graphical elements that have not been explicitly defined will be generated during the translation of the program. The following code segment shows an implementation of a modeling method, where only the structure has been defined. This program will be fully runnable on a metamodeling platform that requires graphical definitions of elements, because of the automatic generation of graphical symbols.

```
// small method with inheritance and modes
method Minimalistic
// if a class symbols is not specified it will be generated
class ClassA {}
class ClassB extends ClassA {}

// if a relation symbols is not specified it will be generated
relation RelationA from ClassA to ClassB {}
relation RelationB extends RelationA from ClassA to ClassB {}

modeltype NiceModel
{
   classes ClassA ClassB
   relations RelationA RelationB

   modes
   mode ModeA include
      classes ClassA
      relations RelationA
   mode ModeB include
      classes ClassB
      relations RelationB
}
```

10 MM-DSL IDE

This chapter is dedicated to the MM-DSL integrated development environment (IDE) and its role in the process of modeling method engineering. In here reader will find important information about MM-DSL IDE architecture, as well as how the IDE communicates with the rest of the system, mainly with the language – MM-DSL, various code translators and metamodeling platforms. The chapter concludes with a short guide on how to use the IDE to create a new project and start implementing a modeling method in MM-DSL.

10.1 Introduction

The MM-DSL integrated development environment allows developers to utilize MM-DSL for describing modeling methods and creating modeling tools. Thus, the IDE is also responsible for generating metamodeling platform-specific format from the code written in MM-DSL. From a practical perspective it makes MM-DSL more users friendly by providing typical features of well-known and established IDEs (e.g., Eclipse, Visual Studio), such as code highlighting, suggestions and auto completion, compile time error checking, and code templates which can be extended.

10.2 Architecture

The three major components of MM-DSL metamodeling system are: (1) MM-DSL (the language), (2) MM-DSL IDE (the development environment), and (3) MM-DSL execution environment (the metamodeling platform). The language has been described in detail in previous chapters (see chapters MM-DSL and MM-DSL Specification). This section focuses on the MM-DSL IDE and its architecture.

MM-DSL IDE (referred to as just IDE in the following text) is separated into multiple components: translator framework, translator, user interface, language framework, and language(s). Multiple translators are based and inherited from one translator framework. The same is true for the language framework, which can support multiple different languages. In our case there is only one language, the MM-DSL.

The translator framework is utilized for defining and integrating various translators within the IDE. Its main duty is to link the abstract syntax of a language to the metamodeling platform's meta^{2}model, and, additionally, to the platform's APIs. Thus, the translators can use the code written in MM-DSL as an input, and give platform-specific code (or format) as an output. The translator framework has a well-defined association with the syntax and semantics of a language, which exposes the MM-DSL artifacts to the translator under development. However, it is up to the developer to link these artifacts

with particular metamodeling platform artifacts. This mapping decides what kind of platform-specific artifacts will be generated from the MM-DSL code.

The language framework is utilized for customizing MM-DSL by adding, as well as removing (deactivating) provided language artifacts. It is a tool mostly used for adapting MM-DSL's concrete and abstract syntax (the grammar of the language), or adding features that are not a part of the base language. Using the framework allows customization of MM-DSL in many different ways. One can, for example, change the whole concrete syntax by modifying the terminal symbols. This also includes changing the keywords (e.g., changing *'method'* into *'modeling method'*, *'algorithm'* into *'operation'*, etc.). If abstract syntax needs to be changed, caution should be exercises, as these kinds of changes may break the association between the language framework and translator framework. Thus, the translator framework would need to be adapted as well.

User interface (UI) is a central point of interaction between the IDE and the MM-DSL users. It exposes MM-DSL to the modeling method developers, similarly as almost any other modern IDE, giving them an environment which hides unnecessary complexities and helps with the focus on defining solutions. IDE is taking care of all the small details regarding the written code, and suggests quick fixes in the following situations: misspelling of keywords, using wrong program statements, helping with code visibility, providing code templates, etc.

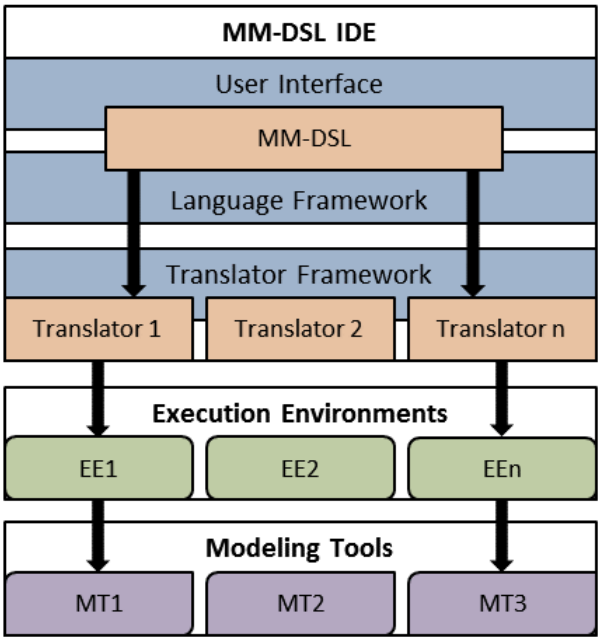


Figure 28: The MM-DSL IDE Architecute

Figure 28 shows an abstract view on the MM-DSL IDE architecture. One can see the parts already discussed in this section, such as user interface, language framework and translator framework, and their relationships. This kind of modular architecture, where systems are composed of separate components that can be connected together, allows the addition, replacement, or removal of a component without affecting the rest of the system.

10.3 Implementation

Two distinguished and competitive technologies have been considered for the implementation of an IDE prototype. On one side there is Visual Studio IDE and .NET framework together with all the accompanying programming languages (mostly C#) and formats. On the other side we have Eclipse IDE and Java-based technologies. After summarizing advantages and disadvantages of both technologies, it has been decided to use Eclipse.

The most important criterion that leaned the decision toward Eclipse is that Eclipse-based technologies are open source software. Secondary, Xtext framework, which is currently only available for Eclipse does not have a mature competition based on Visual Studio and .NET framework. There is a similar framework called Irony, but it is still in alpha phase, therefore not as stable as we wanted it to be.

Xtext was essential to the development of the prototype, because it provided several functionalities out-of-the-box. This allowed focusing the efforts on the implementation of the MM-DSL (the language) according to the specification, rather than investing time in reinventing new compiler-compilers and code generators. The grammar language of Xtext was used to implement the language, and Xtend was used to implement the translator.

The following is the definition of a class concept in Xtext. For the definition of all the MM-DSL concepts see the Appendix B: MM-DSL – Xtext Language Description.

```
Class:
   'class' name=ValidID ('extends' parentclassname=[Class|QualifiedName])?
   ('symbol' symbolclass=[SymbolClass|QualifiedName])?
   '{' (classattribute += ClassAttribute | attribute += Attribute |
   insertembedcode += InsertEmbedCode | reference += Reference)* '}'
;
```

The following code excerpt written in Xtend shows how one can map MM-DSL class defined above to the metamodeling platform meta²model. In this example, metamodeling platform is ADOxx, and the output format is ADOxx-specific.

```
//--- Class <__LibraryMetaData__> - default values--------------------------

   ATTRIBUTE <Position>
   VALUE ""

   ATTRIBUTE <External tool coupling>
   VALUE ""

«FOR Class c: root.method.metamodel.class_»
//=========================================================================
CLASS <«c.name»> : <«IF c.parentclassname !=
null»«c.parentclassname.name»«ELSE»__D-construct__«ENDIF»>
//=========================================================================
```

```
//--- Class <«c.name»> - Class attributes-----------------------------------
    «FOR ClassAttribute ca: c.classattribute»
    CLASSATTRIBUTE <«ca.name»>
    «IF ca.type.simpletype == SimpleType::INT &&
       ca.type.enumtype == null»
    «toTypeInt»
    «ELSEIF ca.type.simpletype == SimpleType::STRING &&
       ca.type.enumtype == null»
    «toTypeString»
    «ELSEIF ca.type.simpletype == SimpleType::DOUBLE &&
       ca.type.enumtype == null»
    «toTypeDouble»
    «ELSE»
    «toTypeEnum(ca)»
    «ENDIF»

    «ENDFOR»

    CLASSATTRIBUTE <ClassAbstract>
    VALUE 0

    CLASSATTRIBUTE <ClassVisible>
    VALUE 1

    CLASSATTRIBUTE <GraphRep>
    VALUE "GRAPHREP
    «IF c.symbolclass == null»
    «GenerateRandomClassSymbol»
    «ELSE»
    «c.generateCSymbol»
    «ENDIF»
    "

    CLASSATTRIBUTE <VisibleAttrs>
    VALUE ""

    CLASSATTRIBUTE <AttrRep>
    VALUE "NOTEBOOK
    CHAPTER \"Attributes\"
    ATTR \"Name\"
    «c.toNotebook»
    «IF c.parentclassname != null»«c.parentclassname.toNotebook»«ENDIF»
    "

    CLASSATTRIBUTE <WF_Trans>
    VALUE ""


    CLASSATTRIBUTE <AnimRep>
    VALUE ""


    CLASSATTRIBUTE <HlpTxt>
    VALUE ""


    CLASSATTRIBUTE <Model pointer>
    VALUE ""
```

```
   CLASSATTRIBUTE <Class cardinality>
   VALUE ""


//--- Class <«c.name»> - Instance attributes--------------------------------

   «FOR Attribute a: c.attribute»
   ATTRIBUTE <«a.name»>
   «IF a.type.simpletype == SimpleType::INT &&
      a.type.enumtype == null»
   «toTypeInt»
   «ELSEIF a.type.simpletype == SimpleType::STRING &&
      a.type.enumtype == null»
   «toTypeString»
   «ELSEIF a.type.simpletype == SimpleType::DOUBLE &&
      a.type.enumtype == null»
   «toTypeDouble»
   «ELSE»
   «toTypeEnum(a)»
   «ENDIF»
   «ENDFOR»
   «FOR Reference ref: c.reference»
   «ref.toReference»

   «ENDFOR»
   //--- Class <«c.name»> - default values----------------------------------

   ATTRIBUTE <Position>
   VALUE ""


   ATTRIBUTE <External tool coupling>
   VALUE ""

«ENDFOR»
```

One can note that the code is creating an output from the given template. Xtend is used to describe the dynamic parts of the template, which are filled by values provided by the MM-DSL program. The blue and all capital text is the original ADOxx format (e.g., *"ATTRIBUTE <Position>"*). The mapping we see here is only the mapping to the meta²model. However, the class concept also has a corresponding symbol and style. These are also mapped to the ADOxx class (this is not shown in the code excerpt above). Thus, MM-DSL artifacts are typically mapped to more metamodeling platform concepts (1 to n mapping), and vice versa, more MM-DSL artifacts can be mapped to one metamodeling platform concept (n to 1 mapping). In general, several MM-DSL artifacts may be mapped to several metamodeling platform concepts (n to m mapping).

As it can be seen from this small code excerpt, writing a translator is something one needs to get used to. It requires the knowledge of the MM-DSL language framework, but as well the knowledge about the platform-specific format. The translator code, as well as the other implemented MM-DSL artifacts can be found in [102].

10.4 User Guide

This section will briefly explain how to use MM-DSL IDE for development of modeling tools. All the steps will be covered, starting from setting up the environment, writing code, compiling to platform-specific format, and finishing with the modeling tool generation.

10.4.1 Development Environment

Users that are familiar with typical integrated development environments, such as Eclipse or Visual Studio, will have no difficulties with MM-DSL IDE. After all, it is built around Eclipse. It can also be considered as an Eclipse plug-in.

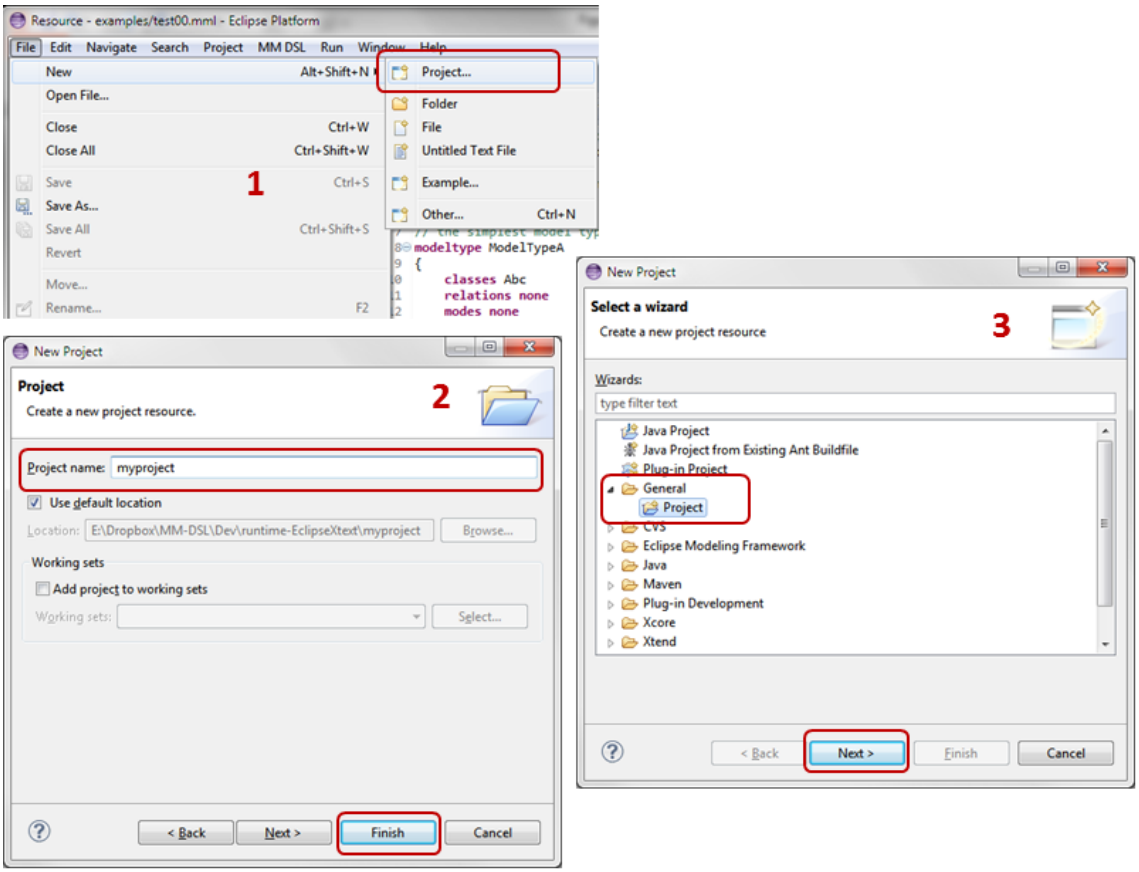


Figure 29: Creating a new MM-DSL Project

The development starts with the creation of a new project (see Figure 29). It is a folder where all of MM-DSL files will be placed, such as files containing MM-DSL source code, intermediate files, and files containing the code in a platform-specific format. All of these files have different file extensions. MM-DSL source code files are recognized by *"mml"* at the end. The files containing the platform-specific code may have an arbitrary extension. This depends on a translator used to create those files. Because ADOxx has been utilized as an execution platform for the MM-DSL programs, the compiled files have an extension *"abl"*. Everything necessary to generate a modeling tool with ADOxx is contained inside the application library files (ABL).

Figure 30 shows how to create and add a new MM-DSL (MML) file to the project. One project can have multiple MML files. The concepts created in one MML file can be referenced inside another MML files.

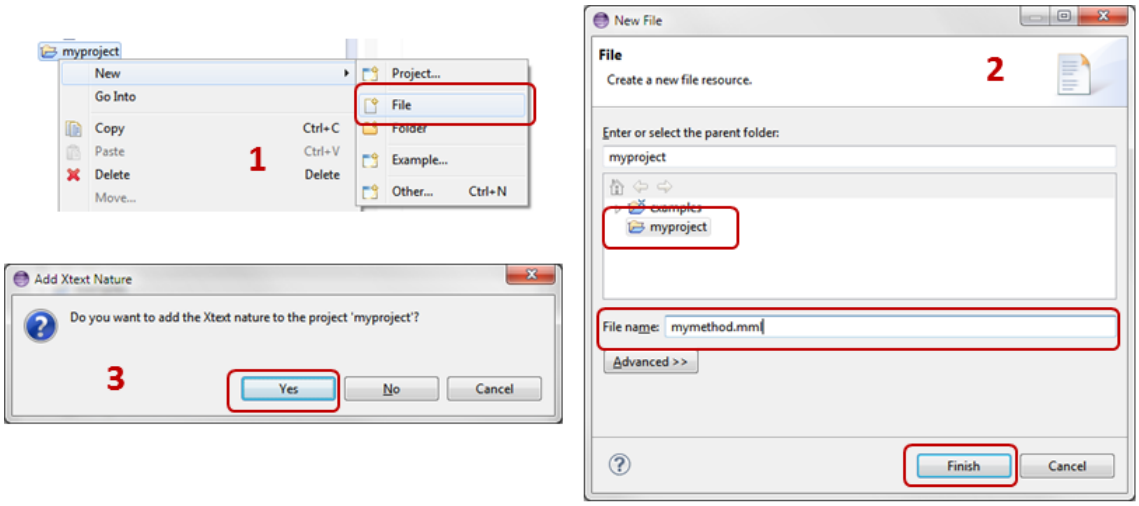


Figure 30: Creating a new MML File

MM-DSL IDE supports the same features as Eclipse IDE, including code highlighting, autocomplete, and templates. The available language keywords and statements can be accessed by holding *CTRL* and *SPACE* at the same time. The list that is shown is populated with currently available constructs that respect the overall syntax of the language. This is why the list changes according to the current cursor position in the code. At the bottom of the environment one can see the current problems in the MM-DSL program, and fix them correspondingly to the provided description. See Figure 31 for details.

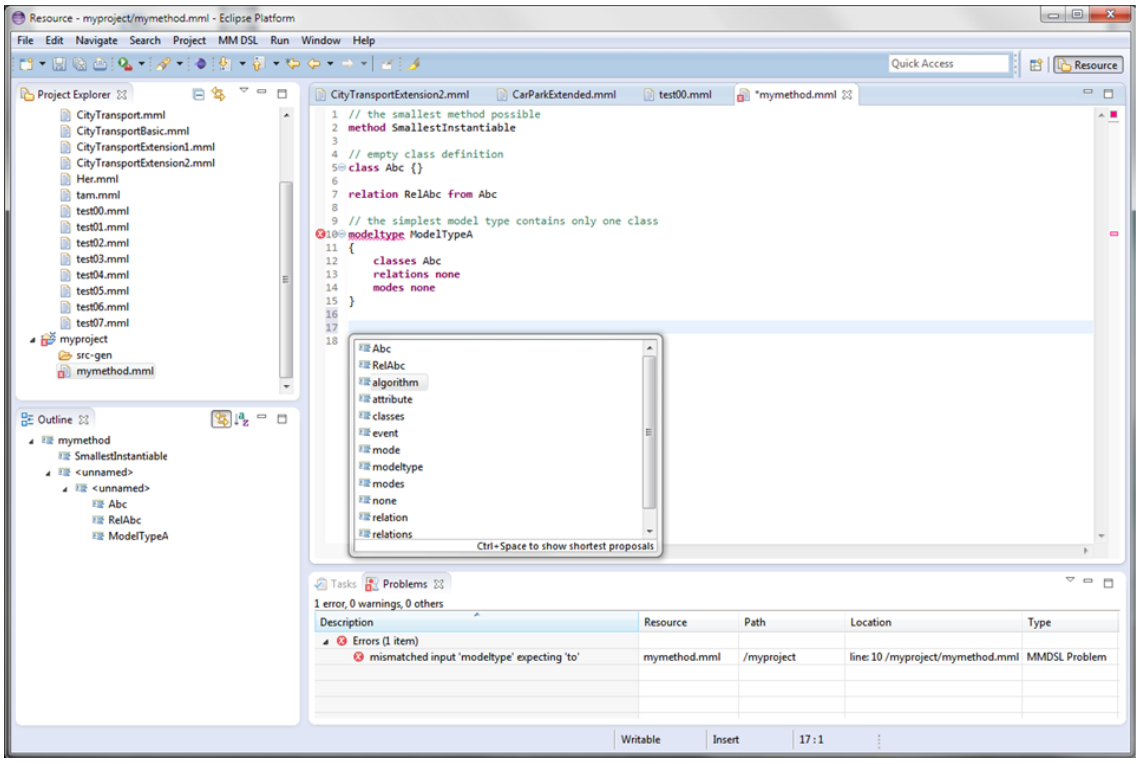


Figure 31: MM-DSL IDE Overview

10.4.2 Program Translation

Current prototype translates the MM-DSL program to the platform-specific code when the file is saved. The translation is automatic. During the development of the ADOxx translator, we wanted to include an option that allows users to edit platform-specific code (the files ending with *"all"*). Thus, the translation has two parts. First, the MML file is translated to ALL file, which is still a textual representation of a platform-specific code. Therefore, it can still be edited inside the MM-DSL IDE. The changes made to the ALL file will not be propagated to the MML file, and if MML file is translated again, all the changes made to the ALL file will be lost. Once the ALL file is translated to ABL file, which is a binary representation that can be imported into the platform, there is no way to edit it with MM-DSL IDE. However, it can still be modified using the destination platform. Translation from ALL to ABL is not automatic. User needs to click on the specified button to trigger it (see Figure 32).

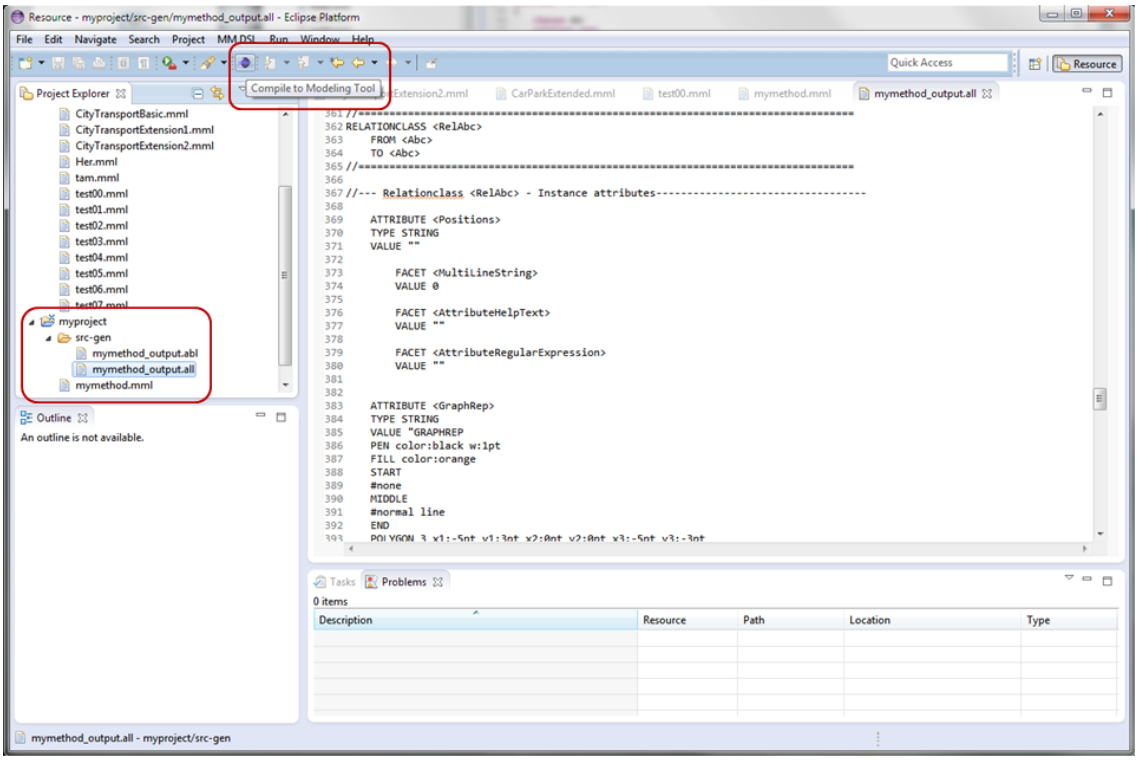


Figure 32: MM-DSL IDE ALL to ABL Translation

In general, to be able to translate the MM-DSL code to the representation a platform understands, one needs to construct a translator exactly for that purpose. As discussed before, the difficulty of translator construction depends on the destination platform, because the translator maps MM-DSL artifacts to the destination platform concepts. The translator framework provides a template for construction of various translators. In this template the MM-DSL artifacts are already exposed (as can be seen in the code excerpts in the previous section). What needs to be done is to specify what kind of destination platform concepts will be generated for every MM-DSL artifact.

10.4.3 Modeling Tool Generation

The final step in the language-oriented modeling tool development process is the generation of a modeling tool. This responsibility is passed on to the destination platform. There are two possibilities: (1) seamless integration of MM-DSL IDE with the platform, or (2) manual import of a file describing a modeling tool into the platform. With seamless integration the modeling tool is generated without any additional inputs from the user. If manual import is used, user still has the possibility to continue implementing the modeling tool using the destination platform. This is sometimes addressed as implementing (or including) additional functionality (the one that was not covered by MM-DSL program). Regardless of which of the possibilities has been chosen, the final product is always a fully functional modeling tool. Figure 33 shows the whole process of modeling tool development by utilizing MM-DSL.

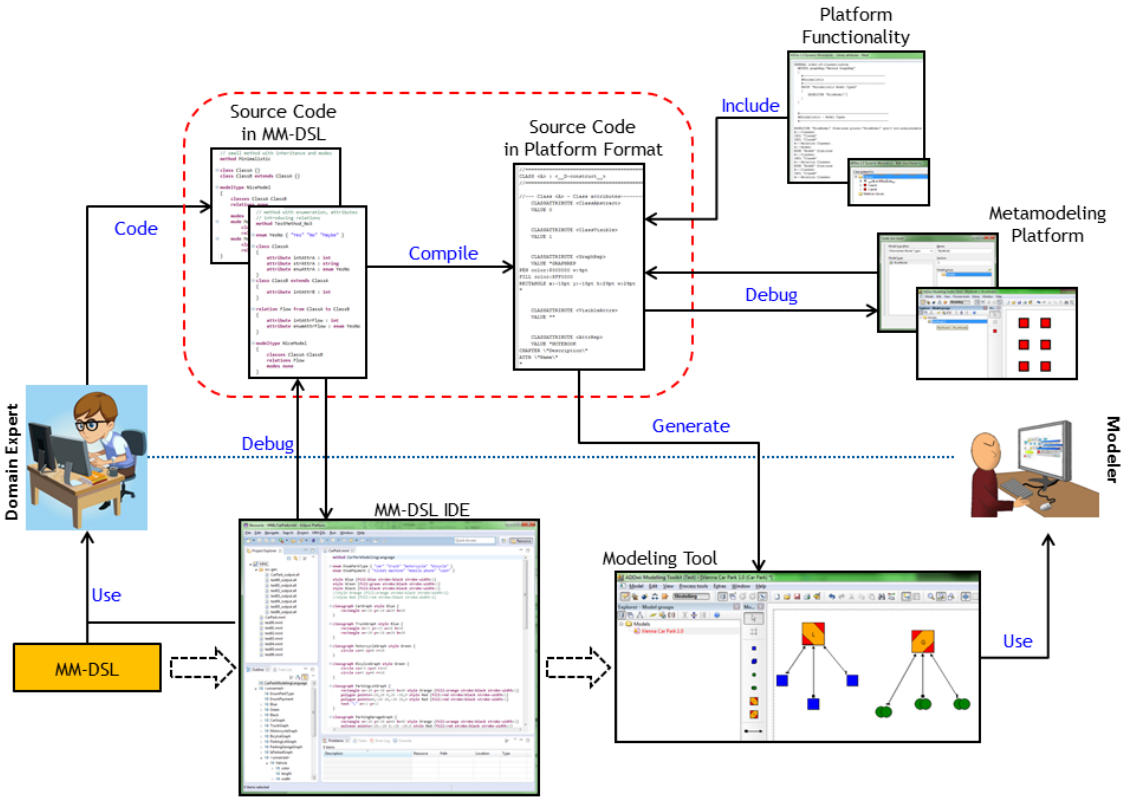


Figure 33: Using MM-DSL to Develop Modeling Tools

It is important to note that there is currently no reverse way, where we can generate MM-DSL code from a modeling tool. This kind of functionality may come in handy for extracting the implementation details from one platform and importing them into the other platform.

More about the underlying issues and possibilities of MM-DSL, as well as MM-DSL IDE can be found in the last chapter of this dissertation that gives focus on the future work that extends MM-DSL and accompanying tools with a purpose to increase the usability and include additional features.

11 MM-DSL Applications

This chapter illustrates a typical MM-DSL modeling tool development scenario. The same representative example is used to present how the modeling tool development process looks like when using only a metamodeling platform, and how it looks like when MM-DSL is included in it. Advantages and disadvantages of utilizing both of these techniques are clarified as well. A short version of this research has been published in [79].

11.1 Running Example: A Pseudo Modeling Method

To better grasp the differences and synergies between the development scenarios, a simple representative modeling method has been designed, and realized as a modeling tool using a state of the art metamodeling approach – the one employing only a metamodeling platform, as well as the one proposed in this research project – the one augmenting metamodeling platforms with MM-DSL. The pseudo modeling method, named *"Car Park"*, is used to explain both approaches. It is a simple, yet representative example, which comprises of the most important modeling method building blocks.

The Car Park modeling method models car parks in a specific city and provides insights about: (1) the number of car parks in a city, (2) the type (parking lot or parking garage), (3) the size, and (4) used spaces in a car park, (5) different vehicle types which are allowed to park in the current car park (car, truck, motorcycle, or bicycle), and (6) the available payment options in a car park. The details about the requirements of this modeling method are depicted in Figure 34.

The elements with the graphical representation are: *Car*, *Truck*, *Motorcycle*, *Bicycle*, *Parking Lot*, *Parking Garage*, and *is parked*. These are the ones we can model with. The rest of the elements are either abstract (e.g., *Vehicle* and *Park*) or do not have any kind of graphical representation (e.g., *City*, and *belongs to*). Together they form the metamodel of the Car Park modeling method. The UML Class Diagram notation has been used to specify the metamodel. The metamodel on the right side of Figure 34 shows the relations between various modeling method elements: *Vehicle* is associated with *Park* with a relationship *is parked*; *Park* is associated with *City* with a relationship *belongs to*; *Parking Lot* and *Parking Garage* are specializations of *Park*; *Car*, *Truck*, *Motorcycle* and *Bicycle* are specializations of *Vehicle*.

In the following sections we will see how successful can each of the development scenarios translate this pseudo modeling method into a modeling tool. Both of the development processes produce the same result at the end. Thus, the priority is set on the development process, and the steps that need to be taken to implement a modeling tool.

	Class	Super Class	Attributes
-	Vehicle	-	name, color, length, width, height, weight
	Car	Vehicle	-
	Truck	Vehicle	-
	Motorcycle	Vehicle	-
	Bicycle	Vehicle	-
-	City	-	name, latitude, longitude
-	Park	-	name, address, city, size, type, payment
	ParkingLot	Park	-
	ParkingGarage	Park	-

	Relationship	From / To	Attributes
	is parked	Vehicle / Park	-
-	belongsto	Park / City	-

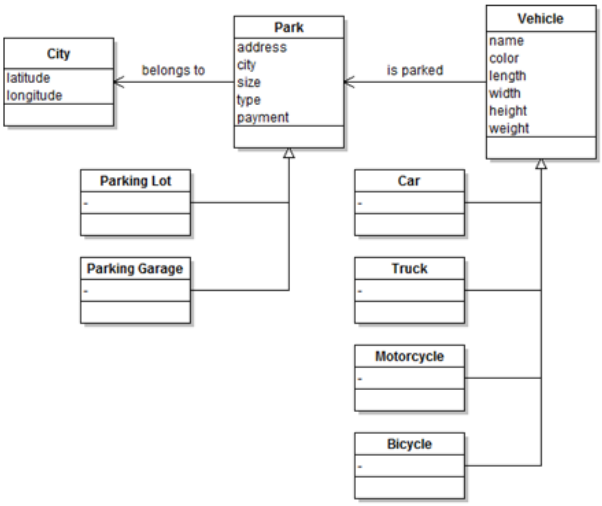


Figure 34: The Car Park Modeling Method Requirements

11.1.1 Developing with the ADOxx Metamodeling Platform

Before one can start to realize a modeling tool on a metamodeling platform, a metamodel of a modeling method needs to be designed, as well as specified in detail, and if necessary accommodated in a way so that it can be instantiated from the provided platform's meta^{2}model (see chapter 7).

The graphical representation of the modeling method elements (concrete syntax) depends on the functionality provided by the platform. Some of the platforms can only work with images (e.g. bitmaps), while the others have the mechanisms for drawing vector objects.

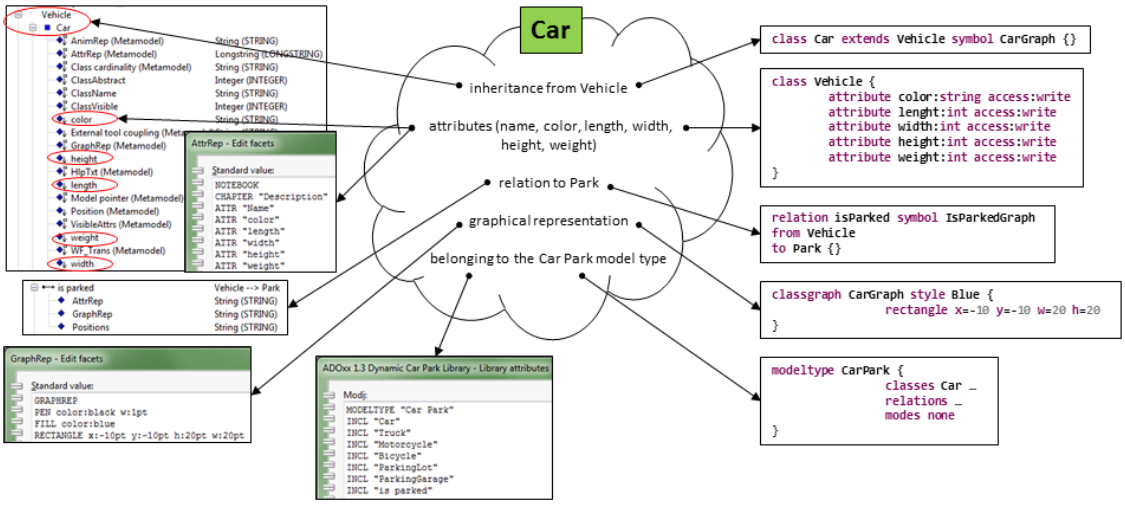


Figure 35: Different Implementations of the Concept Car

It is important to understand that by implementing a modeling tool one transfers the concepts of a modeling method from a conceptual space (e.g., design document) to the technical space (e.g., metamodeling platform). For example, the concept *Car* (see Figure 35) is described by its many facets: inherits from the concept *Vehicle*, has attributes (name, color, length, width, height, and weight), relates to the concept *Park*, has graphical representation, and belongs to the *Car Park* model type.

The implementation will be more successful if the technology applied provides similar concepts to the ones defined in the design documentation. Upon taking a closer look at Figure 35, one can see, in the middle, the illustration of a concept Car. On the left side are screenshots taken during the implementation of a modeling tool on the ADOxx platform. Note that the tree view at the upper left corner contains many attributes that have nothing to do with the attributes belonging to the Car. Nevertheless, a developer needs to be familiar with most of them, especially *AttrRep* and *GraphRep*. AttrRep is used to describe which attributes will be visible to the user of a modeling tool, and GraphRep is used to define the graphical representation of a modeling element. Model types are defined in a completely different part of the platform, which can sometime be an inconvenience during the prototyping phase, because of constant closing and opening of different windows, which may lead to inconsistencies and errors that cannot be quickly detected.

Every technology has its disadvantages, and ways how to deal with them. However, the real issue at hand is the lack of domain-specificity, which many metamodeling platforms share. In case of ADOxx, there are no concepts like *"attribute visibility"* or *"graphical representation"*, or any difference between a class and a relationship when defining a model type. For the platform everything is either a class, or a relation class, or an attribute. This is an issue that has been addressed with MM-DSL. On the right side of Figure 35, one can see the same concepts implemented in MM-DSL. Everything is in one place, in one file. Each artifact has its own syntax and semantics. Thus, the difference between them is immediately noticeable. How to use MM-DSL to augment the development of modeling tools is discussed in more detail later in this chapter.

The development scenario which uses only a metamodeling platform is structured in the following important steps: (1) adapt the provided metamodel if necessary, (2) instantiate the meta^{2}model with the (adapted) metamodel of a modeling method, (3) realize the graphical representation using the tools provided by the platform, (4) implement additional functionality (algorithms and mechanisms) described by the modeling method specification document by reusing or extending the platform's functionality, (5) compile a modeling tool, (6) debug the tool and fix bugs as soon they are found, (7) deploy and publish the modeling tool. This typical use of a metamodeling platform is illustrated in Figure 4 where ADOxx is employed to develop a modeling tool for the Car Park modeling method.

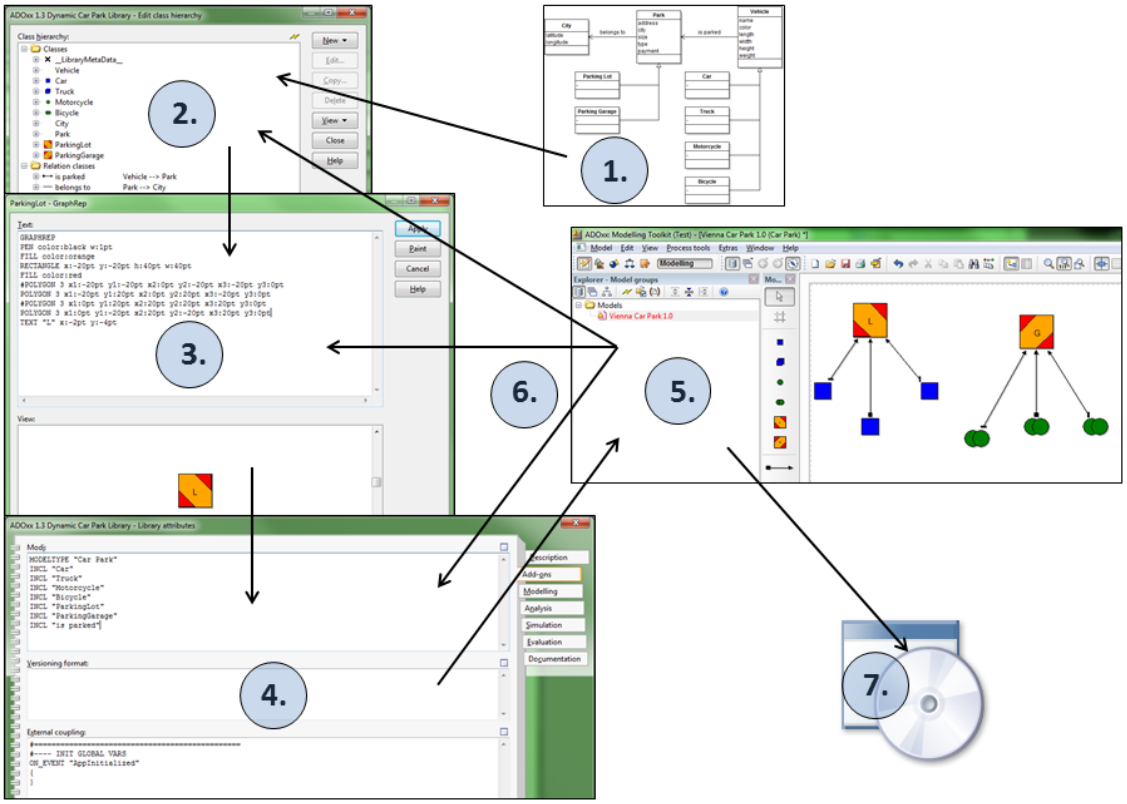


Figure 36: Developing a Modeling Tool Using ADOxx

The process starts with the creation of an empty modeling method library which will hold the modeling method elements (classes, relationships and their attributes), as well as their graphical representations, and additional functionality (algorithms, mechanism, and procedures). After the modeling elements have been created, they are assigned to one or more model types. Model types in ADOxx are essentially containers that contain parts of a metamodel, and can be instantiated as a diagram in a modeling tool. The implementation is continued by assigning platform functionality to the modeling elements, which is done using the *ADOscript* language. For the Car Par modeling method, one can define queries that return number of empty car park spaces for every car park, or dynamically change the graphical representation of modeling elements according to predefined conditions (e.g., car park full, price range, etc.).

As an interesting fact, it is worth to mention that it has taken around an hour to realize the Car Park modeling method on the ADOxx metamodeling platform and instantiate the modeling tool. This time excludes the implementation of any additional functionality (algorithms and mechanism) upon the basic modeling method. It also needs to be mentioned that an experienced ADOxx user has been conducting the mentioned implementation. Thus, the time investment for a novice user would be considerably higher.

The key advantages of using a metamodeling platform to transfer a modeling method specification into a modeling tool are the following: (1) provided meta^{2}model and additional functionality out-of-the-box, (2) click-based development with (almost) no pro-

gramming involved, and (3) most platforms typically come equipped with a model repository for storing and versioning of models and their artifacts.

11.1.2 Developing with the MM-DSL

In this section we will see how successfully MM-DSL can be applied in the development of modeling tools. The same pseudo modeling method used to illustrate the development with ADOxx is used to illustrate the development process with MM-DSL.

This approach extends on the language oriented programming (LOP) paradigm, in which, rather than solving problems in general purpose programming languages, the programmer creates one or more DSLs for the problem at hand, and then solves the problem in those languages. The details about LOP can be found in [90]. However, it is not necessary to begin designing a DSL for the domain one intends to model from the scratch. Therefore, MM-DSL has been created with that idea in mind. Most of the basic notions of the metamodeling approaches which have been established during the last couple of decades are already included in the language and can be used out-of-the box. In case that the already provided concepts are not adequate for realizing the envisaged modeling method, one can always introduce new concepts to the MM-DSL. Thus, allowing greater flexibility than the one that can be achieved with a fixed meta^{2}model. By using and extending MM-DSL in a way discussed in here, the modeling tool will have a greater conformity to the initial modeling method specification.

The necessary knowledge requirements have been taken under consideration and it has been made sure that MM-DSL in its basic state comes equipped with everything necessary for the development of modeling tools: (1) its own integrated development environment, (2) a framework for the development of translators for typical execution environments (e.g., metamodeling platforms, frameworks), and (3) with the implementation of a translator for the ADOxx metamodeling platform, which serves as a template for future translator implementations, and as proof of the concept.

Augmenting a metamodeling platform with MM-DSL allows one to deploy a prototype of a modeling tool very quickly. Because, the MM-DSL code is self-documenting, no previous modeling method specification is necessary. One codes and specifies the modeling method at the same time. The code is directly compiled into the platform-specific format and can be immediately executed, producing the modeling tool.

MM-DSL is also very useful in case one wants to implement a modeling method only once and deploy it to multiple execution environments. Under assumption that MM-DSL comes equipped with multiple translators, one for each metamodeling platform, it is very much possible to write code once, and execute it on multiple platforms at the same time.

It is not assumed that all the functionality one metamodeling platform provides can be coded with MM-DSL, as well as that not every platform possesses the same functionality, therefore the resulting modeling tools might be more complete (when compared to the modeling method specification) when using one, and less complete when using

another metamodeling platform as an execution platform for the MM-DSL programs. Nevertheless, it is possible to make finishing touches to the modeling tool utilizing the metamodeling platform itself.

The modeling tool development using MM-DSL (see Figure 33) is structured in the following steps: (1) code a modeling method, (2) compile the code to one or more execution environments (typically metamodeling platforms), (3) continue adding finishing touches using the functionality provided by execution environments, and (4) generate a modeling tool. Step (3) is optional. Continuing the development on the execution platform is only necessary if one requires features that are not covered with MM-DSL. In step (1) one works with MM-DSL IDE. The IDE discovers bugs and code errors before compilation. After the code has been compiled to the specific execution environment, future modifications inside the execution environment can no longer be debugged with the support of MM-DSL IDE. However, most of the metamodeling platforms provide the debugging support by themselves, which can be used in step (3).

There is no particular order in coding the modeling method with MM-DSL. One can start by defining a metamodel (abstract syntax), then structuring it in model types. Afterwards, graphical representation (concrete syntax) can be defined. Or vice versa, start with graphical representation, and then code the metamodel part. Nonetheless, some of the elements of a modeling method (e.g., classes, relations, attributes) need to be already present in the code before algorithms can be realized, as most of the algorithms require some sort of input. Inputs to algorithms can be any modeling elements and their values (attribute values, class or relation instances).

Looking at the code shown in Figure 37, it can be noted that enumerations have been defined first, followed by definition of styles. These artifacts need to be defined before they can be referenced. For example, *IsParkedGraph* is referencing *Black* style, and class *Park* contains attributes that are of type *EnumParkType* and *EnumPayment*. We can define styles inside class symbol and relation symbol statements as well. For example, style *Orange* is defined inside class style *ParkingLotGraph*. Symbols also need to be defined before they can be referenced by classes or relations. Concept of inheritance is used in definition of classes *Car*, *Truck*, *Motorcycle* and *Bicycle*. In the current version of MM-DSL inheritance only works for the structure. For example, attributes defined in the class *Vehicle* will be inherited by the class *Car*. However, if *Vehicle* had a symbol associated with it, it would not be inherited by the class *Car*. This was a design decision which helps in differentiating abstract and concrete classes (concrete classes always have a symbol assigned to them). There is one more relevant concept that needs addressing – model types. They are used to aggregate modeling elements into a diagram used inside a modeling tool. These elements are also visible inside a modeling tool tool-box (see Figure 38). For example, model type *CarPark* contains several classes, one relation, and only a default sub-view (indicated by *modes none* command). Modes or subviews allow selecting which from the existing modeling elements in a diagram we want to show. There can be multiple modes in one model type.

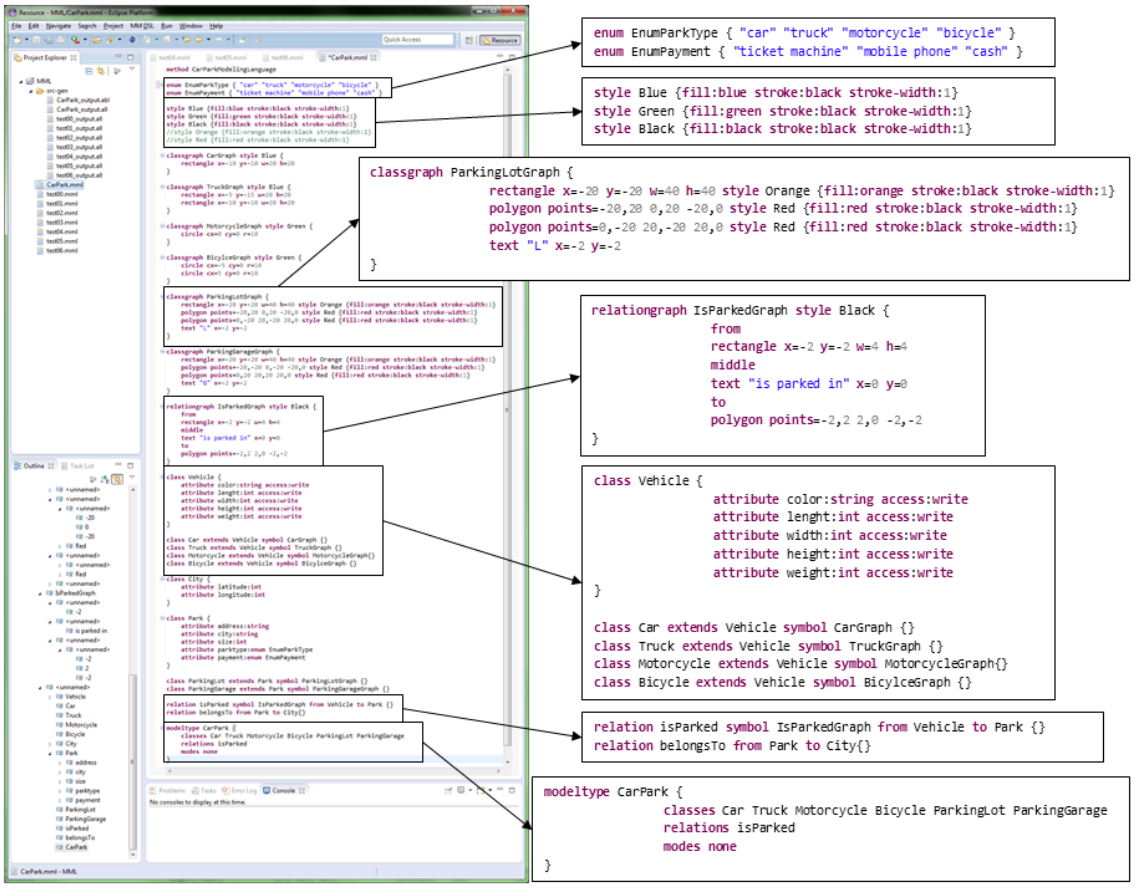


Figure 37: Developing a Modeling Tool Using MM-DSL

The complete implementation of the Car Park modeling method has only 89 lines of code. The same implementation on the ADOxx platform has 1349 line of code, 15 lines for each line in MM-DSL program. The reduction of programming effort is considerable. As an interesting fact, it is worth to mention that the implementation took less than 30 minutes – almost half the time of the ADOxx implementation. However, the implementation has been done by the MM-DSL creator, so this should also be factored in the calculation. Very similar example has been given to evaluators to implement. The time effort was around an hour.

The key advantages of using MM-DSL to code a modeling method and compile it into a modeling tool are the following: (1) MM-DSL is independent from the execution platform, (2) modeling method concepts coded in MM-DSL can always be reused with or without modifications, and (4) a short learning curve, because all the concepts are documented and exposed through the MM-DSL IDE.

Both of the development approaches have produced a modeling tool that looks and behaves very similarly. A screenshot of the Car Park modeling tool can be seen in Figure 38. At the top of the figure is a sketch of an expected model produced during the requirements specification for the Car Park modeling method. One can see that the modeling tool produces models conform to the given specification.

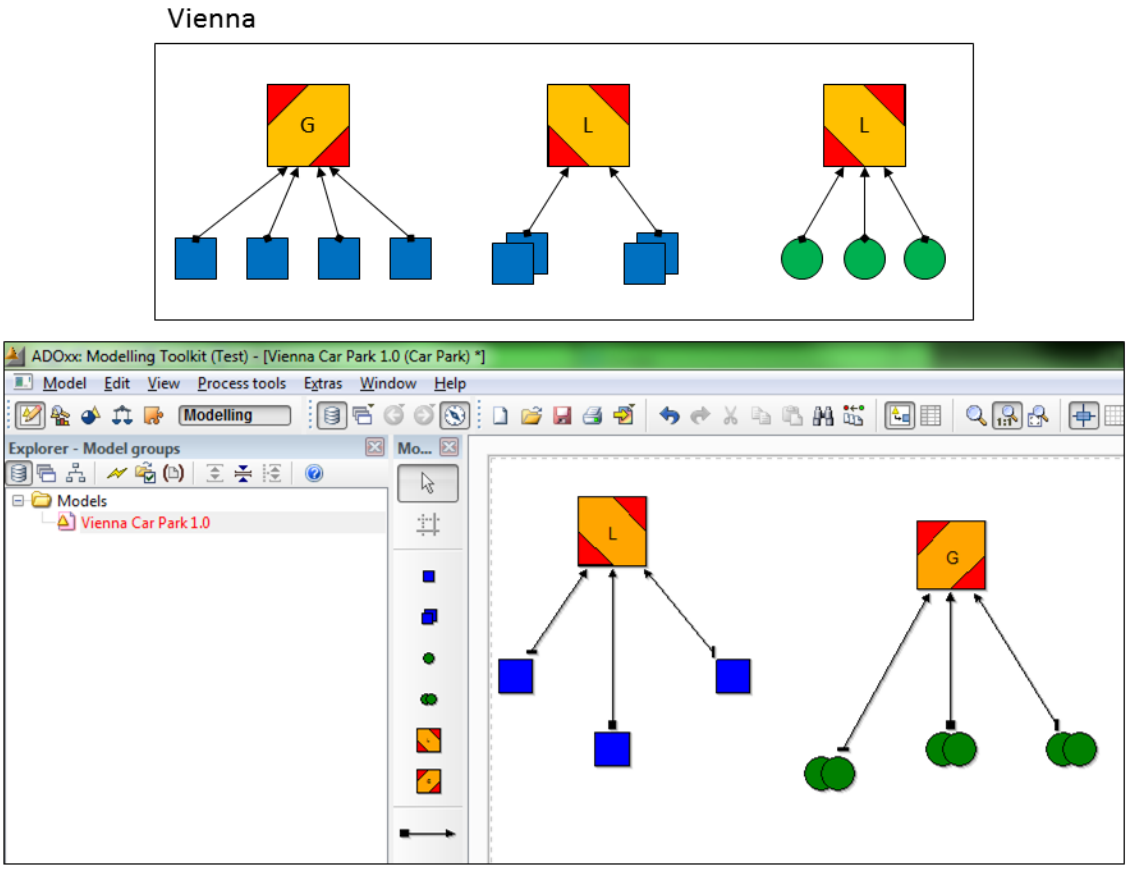


Figure 38: Resulting Modeling Tool

11.2 Conclusion

It is hard to argue which of the presented development scenarios is generally more viable. Both of them have advantages and disadvantages. At the end it all comes down to the user's preferences (although the author of this text firmly believes that MM-DSL increases effectiveness of modeling tool development in many areas). A user with the previous programming knowledge and experience will find working with MM-DSL familiar, intuitive and fast. The approach that directly uses a metamodeling platform for modeling tool realization is more attractive to users that are not familiar with programming (or do not want to program).

The biggest difference between MM-DSL and the metamodeling platform is in their level of abstraction. MM-DSL abstracts from the execution environments (e.g., metamodeling platforms) by providing the domain-specific functionality which is not connected with the technical space of an execution environment, making it a platform independent approach. The realized modeling tool quality does not only rely on the code written in the MM-DSL, but as well on the underlying translator which maps the language concepts to the execution environment concepts. The development scenario using only a metamodeling platform is typically very specific to the platform used. In this scenario one has the entire platform's functionality out-of-the-box and the possibility to reuse and extend it. However, this comes with a drawback of locking the future

development only to one platform. There is, in most cases, no way of reusing any developed artifacts (e.g., code, files, etc.) on a different metamodeling platform.

There are three main concerns one needs to tackle with when implementing modeling tools for modeling methods: abstract syntax, concrete syntax, and algorithms. Here is a brief overview on how a metamodeling platform handles these issues, and how are those issues handled with MM-DSL:

- Before one can start with the development of a modeling tool, a metamodel of a modeling method needs to be designed. This is what represents the abstract syntax of a modeling method. How will this abstract syntax be implemented on a metamodeling platform depends on the underlying meta^{2}model. Because MM-DSL is a language and can be freely extended, there is no dependency on a particular meta^{2}model.
- Secondly, graphical representation or concrete syntax needs to be specified. The realization of graphical representation depends on the functionality provided by the platform. As already mentioned, some of the platforms can only work with images (e.g. bitmaps), while the others have mechanisms for drawing vector objects. MM-DSL has SVG-like commands, which is a familiar (SVG is an open standard developed by W3C) and expressive approach for creating graphical objects.
- Thirdly, algorithms working on modeling elements need to be specified. Most of the metamodeling platforms use general purpose programming languages (e.g., Java, C++) to tackle with this issue. The minority has dedicated DSLs. MM-DSL includes concepts specialized for development of modeling algorithms, as well as commands for model manipulation.

Because this is a research work, and one does not want to base the research results solely on the problems envisaged and solved by the language's creator, MM-DSL usability and expressivity testing has also been conducted in the two different studies. In the first study participants had the opportunity to describe a modeling method with MM-DSL by using the provided language specification and modeling method requirements. No IDE has been provided, because the goal was to test understandability and learnability of the language. The IDE's usability has been tested in an exercise-like environment, where testers had to extend the given MM-DSL code and compile it to a modeling tool. Further details about these studies and their results are discussed in the next chapter.

12 Evaluation

This chapter is dedicated to the evaluation of two interconnected artifacts. The first one is the language – MM-DSL, and the second one is the environment – MM-DSL IDE. These two artifacts are a part of an approach this dissertation is focused on – the language-oriented modeling method engineering. The system as a whole is an environment where MM-DSL programs can be executed. Thus, it also includes a metamodeling platform for which a translator has been implemented.

12.1 The Language: MM-DSL

To evaluate a computer language external and internal evaluation criteria are used [103]. External evaluation criteria aggregate various aspects that answer the question whether a given language meets the needs of a given user community. These are: rapid development, easy maintenance, reliability and safety, portability, efficiency, learnability, reusability, and pedagogical value. Internal evaluation criteria are independent of the demands of its users and represent the good qualities of a language. These are: readability, writability, simplicity, orthogonality, consistency, expressiveness, and abstraction.

12.1.1 External Evaluation Criteria

The external evaluation criteria and how to evaluate them is summed up in Table XIII. Detail definition of each criterion can be found in dedicated sections afterwards.

Table XIII: External Evaluation Criteria Overview

Criterion	How to Evaluate
Rapid development	Compare duration of the implementaion to the representative technology.
Easy maintenance	Effort required to maintain the written code.
Reliablity and safety	The language includes fault tolerance and fault avoidance concepts.
Portability	Runnability of a written code on several different platforms.
Efficiency	Time required to translate (e.g., compile) the code.
Learnability	Effort required to invest in learning the language.
Reusability	Compare reusability of written code with the representative technology.
Pedagogical value	Ability to use the language to teach concepts it enforces.

12.1.1.1 Rapid Development

This requirement expresses the productivity of a programmer. In order for the language to be successful in real life scenarios, a programmer should be able to develop artifacts with it faster than with previously used technology (e.g., metamodeling platform or framework, or even another programming language). It is also not possible to evaluate the productivity of a programmer without considering both the language and its environment (IDE).

12.1.1.2 Easy Maintenance

Ability to maintain the code written with a computer language is very important in a long run. Therefore, code maintenance is always a hot topic, and it is also very subjective. What one programmer considers a maintainable code, another may consider very confusing and unreadable. Thus, bigger programming languages that support multiple ways of defining the same artifacts need to have their code formatted according to rules enforced by a group of programmers that are working together on a project or by an organization. Smaller languages, like DSLs, which typically provide only one way of defining a particular artifact, are easier to maintain.

12.1.1.3 Reliability and Safety

A programming language can greatly influence the reliability and safety of a software system. This is why proven languages, such as C/C++ are used for writing military grade software. For the most safety critical and long lived systems, the assembly languages are used. To achieve reliability and safety in a software system we need to write code in such a way that it doesn't contain faults (fault avoidance), which is a difficult task to do. Fault detection mechanisms in the development process helps in preventing the delivery of the faulty software to the customer. In case both fault avoidance and fault detection fail, software needs to be designed so that faults in it do not result in complete system failure.

12.1.1.4 Portability

Portability is the ability to run a program on many different platforms, with minimal or no code rewriting. This property of a language is connected with the availability of different translators (e.g., compilers, interpreters, code generators). A C++ program can run on a Windows, as well as on a Linux machine, because there are compilers for both of those platforms. In case of managed languages, such as Java or C#, a virtual machine is responsible for interpreting the intermediate binary code format to a native platform understandable format. The same principle applies to any executable programming language.

12.1.1.5 Efficiency

Efficiency is a measure that indicates how fast can a translator (e.g., compiler, interpreter) build the program from the given source code. The language design has a big

influence on translator's complexity and its speed. Simpler languages typically have faster translators.

12.1.1.6 Learnability

Short training time is one of the factors that make a computer language more attractive. It also makes it more affordable, because the costs of learning a computer language are typically high.

12.1.1.7 Reusability

This is one of the properties that increase productivity of software engineers manifold. It is always beneficial to write a part of the code once and reuse it many times. Therefore, a language should possess concepts that reduce writing of the same code to the absolute minimum.

12.1.1.8 Pedagogical Value

Computer languages should support and enforce concepts that are being thought. In case of C++ some of those concepts would be pointers, arrays, constructors or destructors. MM-DSL, for example, enforces metamodeling concepts, such as class, relation, attribute or model type. Pedagogical value of a language increases if it can be seamlessly used as an education tool. A typical example is PROLOG, which is very often used to demonstrate the applications of first-order logic in real world scenarios.

12.1.2 Internal Evaluation Criteria

The internal evaluation criteria and how to evaluate them is summed up in Table XIV. Detail definition of each criterion can be found in dedicated sections afterwards.

Table XIV: Internal Evaluation Criteria Overview

Criterion	How to Evaluate
Readability	Effort required to understand the meaning of the written code.
Writability	Express the meaning of code as concise and unambiguous as possible.
Simplicity	Measures the number of primitive concepts a language has.
Orthogonality	Expresses how primitive concepts are combined together.
Consistency	Measures consistency of languages syntax and semantics.
Expressiveness	Compare expressiveness of the language with the representative technology.
Abstraction	Measures the ability of a language to hide unessential properties and concentrate on the essential ones.

12.1.2.1 Readability

Readability is a feature that measures the understandability of computer programs. When developing a language, one of the aims is to make it easily interpretable by its users. Following the general trends is one way to accomplish this. For example, if a language needs to have selection statements, much better approach would be to reuse the syntax and semantics of already established statements such as if-else or switch statements than to introduce different selection statements without a good reason. Another way is to syntactically and semantically adapt the language's statements so that they are identical or very similar to the vocabulary already used by the domain experts (e.g., a class as a concept has a universal meaning in the metamodeling community). That way, the language constructs will be familiar to the users that do not have a lot of experience writing programs in it.

12.1.2.2 Writability

The ability to say what you mean without excessive verbosity is another feature one needs to consider when developing new computer languages. Writability of a language manifests itself through the ability to write concise statements that are rid of any ambiguities. It also manifests itself through the ease to modify the already written code, either by adding, deleting or changing something in it.

12.1.2.3 Simplicity

The feature of simplicity is one that measures the number of features a computer language possesses and compares it to the number of features the language should have, which is always the minimal number of features. For example, the language such as MM-DSL does not need a concept of a memory pointer, because there is no use for it in the metamodeling domain. However, C++ that is applied in the development of embedded systems does need pointers, because programmers need a way to tackle with memory issues.

12.1.2.4 Orthogonality

In computer programming, orthogonality is an important concept that addresses how a relatively small number of concepts can be combined in a relative small number of ways to get the desired results. It is strongly connected with simplicity – the more orthogonal the design, the fewer exceptions. This makes it easier to learn, read and write programs as well.

12.1.2.5 Consistency

A computer language should be consistent in its syntax and semantics. There shouldn't be any inconsistencies without a valid reason. For example, an assignment operator should be consistent throughout the language. The same goes for any kind of language concept.

12.1.2.6 Expressiveness

The programmer will be more productive if he is able to express the concepts naturally. This can be general concepts, such as algorithms, or very domain-specific concepts, such as model type. Typically, DSLs do a better job expressing a limited amount of domain-specific concepts, but fail to express anything unrelated with the particular domain they are developed for. GPLs are better at expressing generic concepts, which spread across multiple domains, but require a lot of boilerplate code to be able to express domain-specific concepts.

12.1.2.7 Abstraction

The ability of a computer language to remove (or hide) characteristics from something in order to reduce it to a set of essential characteristics is called abstraction, and it is one of the central principles in computer science. Using a language that supports abstraction, a programmer is able to hide all but the relevant data about an object in order to reduce complexity and increase efficiency. This is done by separation of concerns. In a typical programming language, most general example could be the separation of user interfaces, business logic and data models. MM-DSL also separates concerns, for example visualization is separated from abstract syntax definitions.

12.1.3 Evaluation Scenario

To evaluate the mentioned language properties, an exercise has been developed. In this exercise, the participants had to write code for a simple modeling method. The only help they were provided was the language specification document. No software support was given, except a plain text editor. The participants had the opportunity to study the specification document for one day, so they get familiar with the language's syntax and semantics. The actual implementation using MM-DSL was limited to 60 minutes. Full description of the exercise is given in Appendix D: Exercise Used to Evaluate MM-DSL.

12.1.4 Evaluation Results

Exercise results were graded between 0 and 5, where 0 represents that no MM-DSL code has been written, and 5 represents that the MM-DSL code describes the given modeling method perfectly, and that it can be compiled without any modification. See Table XV for the description of all the grades.

Table XV: Language Evaluation Grading and Description

Grade	Description
0	No code was submitted
1	Code describes the modeling method very poorly and it cannot be com-

	piled without severe rewrite
2	Code describes the modeling method sufficiently, but major modifications are needed to be able to compile the code
3	Code describes the modeling method sufficiently; minor modifications are needed to be able to compile the code
4	Code describes the modeling method very well; minor modifications are needed to be able to compile the code
5	Code describes the modeling method perfectly and can be compiled without any modifications

The number of participants was twenty-two. From these twenty-two participants, eight have scored the best grade (5), twelve have scored 4, and two have scored 3. No participant has scored lower than 3. The average grade was 4.27. These results indicate that most of the participants have understood the language concepts and were able to describe the given modeling method very well. However, some of the MM-DSL code had to be slightly modified before it could be compiled.

Table XVI and Table XVII show in what extend does MM-DSL satisfy the given language evaluation criteria. These results were extracted from the feedback that was collected from the participants after the exercise finished.

Table XVI: Fulfilment of External Evaluation Criteria

Criterion	Fulfillment	Comment
Rapid Development	high	Even without the IDE support, it was possible to describe a simple modeling method within one hour.
Easy Maintenance	medium	The code is easy to maintain in case there is no external (embedded) code included. Embedded code increased maintenance efforts.
Reliability and Safety	low	Because MM-DSL is not developed with safety critical systems in mind, it does not support such features.
Portability	high	It is possible to run MM-DSL programs on multiple metamodeling platforms.
Efficiency	high	There is no preprocessing and translation of code is relatively fast.
Learnability	high	It is possible to learn and understand the language in a less than a day time just by studying the specification.
Reusability	high	Most of the concepts defined in MM-DSL can be reused.

Pedagogical Value	high	MM-DSL contains all the essential metamodeling concepts and it is a good language for novices in this domain.

Table XVII: Fulfilment of Internal Evaluation Criteria

Criterion	Fulfillment	Comment
Readability	high	The concepts included in MM-DSL are identical or very similar to the concepts used by domain experts.
Writability	high	It is possible to write concise statements. The code is easily modifiable.
Simplicity	high	Only the essential features are included in the language. If more are required, the language is easily extendable.
Orthogonality	medium	It is easy to combine concepts to define other concepts. How one can combine these concepts is limited to avoid unnecessary complexity.
Consistency	medium	In the evaluated version, there were some inconsistencies between various commands.
Expressiveness	high	MM-DSL is very expressive in a metamodeling domain.
Abstraction	medium	MM-DSL separates concerns by providing specialized sections for e.g. visualization or abstract syntax. However, all concerns are typically located in one file (which is actually not required).

12.2 The Environment: MM-DSL IDE

MM-DSL IDE is a software system. Therefore, a different approach is used to evaluate it. The key criterion here is the usability. For this purpose, evaluators had to work with the system to accomplish tasks. For this purpose, a prototype implementation has been developed. This prototype has been tested by participants of the *"Next Generation Enterprise Modeling"* summer school that took place in Klagenfurt from July 6th to July 19th, 2014.

12.2.1 Evaluation Scenario

The task at hand was to extend the given MM-DSL program with additional code segments and compile it. During this process MM-DSL IDE has been used. The testers had to repeat this extension and compilation process several times. Afterwards, the

participants had to fill out the standard System Usability Scale (SUS) questionnaire. The SUS [104], as a tool for assessing the usability of MM-DSL IDE has been choses for its simplicity, effectiveness and large database of SUS scores (it is in use since 1986). It is also free.

Full description of the given exercise can be found in Appendix E: Exercise Used to Evaluate MM-DSL IDE.

12.2.2 System Usability Scale

The SUS is composed of ten statements, each having a five-point scale that ranges from *Strongly Disagree* to *Strongly Agree*. See Appendix F: Standard SUS Questionnaire for more detailed insight into the SUS questionnaire structure. Among these ten statements, five are positive statements and five are negative statements. The negative and positive statements alternate: 1, 3, 5, 7, and 9 are positive statements, and 2, 4, 6, 8, and 10 are negative statements. Each statement contributes to the SUS score in the following way [104]:

- Each statement's score contributes with range from 0 to 4;
- For items 1, 3, 5, 7, and 9 the score contribution is the scale position (1 – 5) minus 1;
- For item 2, 4, 6, 8, and 10 the score contribution is 5 minus the scale position (1 – 5);
- Multiply the sum of the scores by 2.5 to obtain the overall value of SUS score.

SUS as a usability evaluation tool provides an easy-to-understand score from 0 (negative) to 100 (positive). In this evaluation an adjective rate suggested in [105] is used to interpret the meaning of SUS scores. While a 100-point scale allows for relative judgments between systems, an adjective rate scale translates numeric scores into an absolute judgment. Table XVIII shows the mapping between the numeric score and the adjective rate scale. The empirical studies suggest that the SUS score below 50 indicated that the system needs usability improvements, and a score of 70 is generally a passing grade or system with above average usability.

Table XVIII: Descriptive Statistics of SUS Scores for Adjective Ratings (adapted from [105])

Adjective	Average SUS Score	Standard Deviation
Worst Imaginable	12,5	13,1
Awful	20,3	11,3
Poor	35,7	12,6

OK (Fair)	50,9	13,8
Good	71,4	11,6
Excellent	85,5	10,4
Best Imaginable	90,9	13,4

The following questions are typically found on the standard SUS questionnaire:

1. I think that I would like to use this system frequently.
2. I found the system unnecessary complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

These questions, without modification, have been used in the MM-DSL IDE evaluation. In the following section the results and meaning of scores each question has received are discussed.

12.2.3 Evaluation Results

Thirty-one participants have contributed to the evaluation results that will be presented and discussed here. Figure 39 shows the average results for each question. For question 1, 3, 5, 7 and 9 higher result is the better result. For question 2, 4, 6, 8 and 10, lower result is the better result. What the results on this graph show is that the group of testers had approximately the same amount of participants that have found the MM-DSL IDE very usable, and the ones that found it borderline usable. This may be noticed in average score of opposing questions. In SUS questionnaires there are two questions that represent the same facet of the system, but from opposite viewpoints. The questions are grouped in pairs: 1 and 2, 3 and 4, 5 and 6, 7 and 8, 9 and 10, where first question is in a positive form and second one in a negative form. For example, if we look the average score of questions 9 and 10, which is 3.226, respectively 2.968, we can see that the results do not differ significantly. High score for question 9 indicates that the tester has found MM-DSL very comfortable to work with. We can assume that

the tester has a good background in metamodeling and didn't have to learn a lot of new things before he could efficiently use the system. High score for question 10 indicates the lack of previous metamodeling knowledge. The group of people that has participated in testing the MM-DSL IDE has divergent backgrounds. Some of them are computer science students, while the others are business students. Some of them have worked with metamodeling techniques before, while the others have never used such techniques in the past. This is why some of the opposing questions have similar scores.

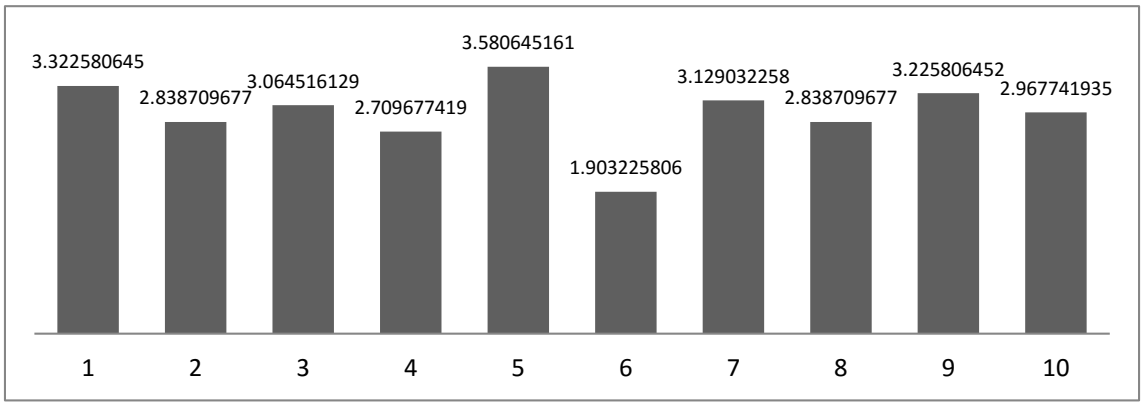


Figure 39: Average SUS Score for Each Question

Figure 40 shows the calculated SUS scores for each of the thirty-one participants. The highest SUS score is 95, and the lowest one is 20. The average SUS score is 58, which indicates average or fair usability. However, as mentioned before, not all participants had the same knowledge about metamodeling techniques. This also needs to be taken under consideration when interpreting the SUS score. It is also important to note that twenty from thirty-one (64.5 %) participant have a score higher than 50, three exactly 50, and the rest below 50 (25.8 %). Test results show the correlation between a high score on question 10 and the overall SUS score on the questionnaire. The participants that have scored lower than 50 all had very high scores on question 10.

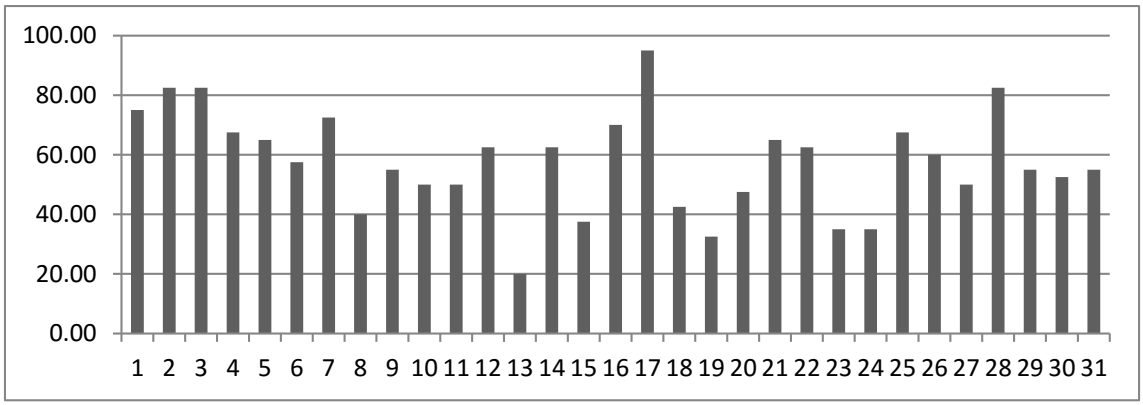


Figure 40: SUS Scores for Each Participant

Figure 41, 42, 43, 44, 45, 46, 47, 48, 49, and 50 show scores from a single question from every participant. That is, Figure 41 shows all the scores for question 1, Figure 42 shows all the scores for question 2, etc.

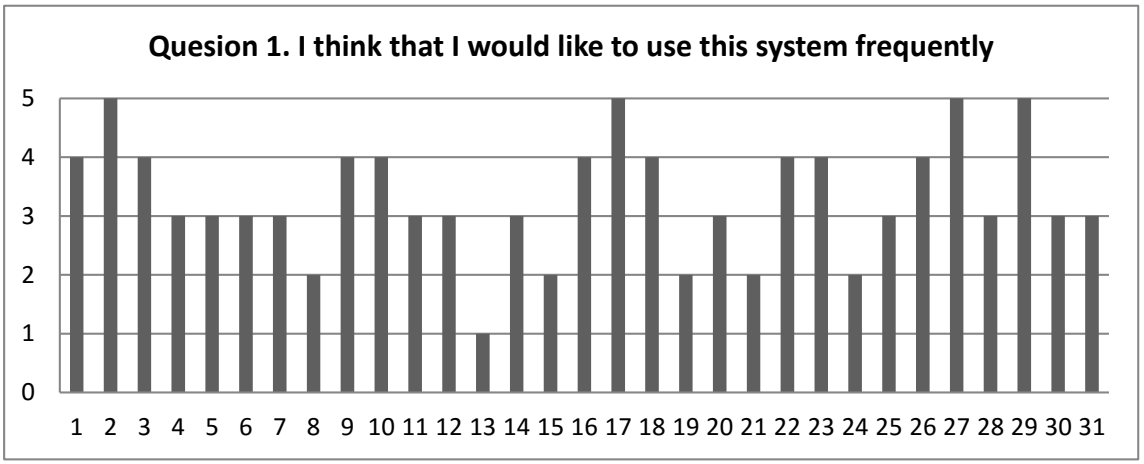


Figure 41: SUS Question 1

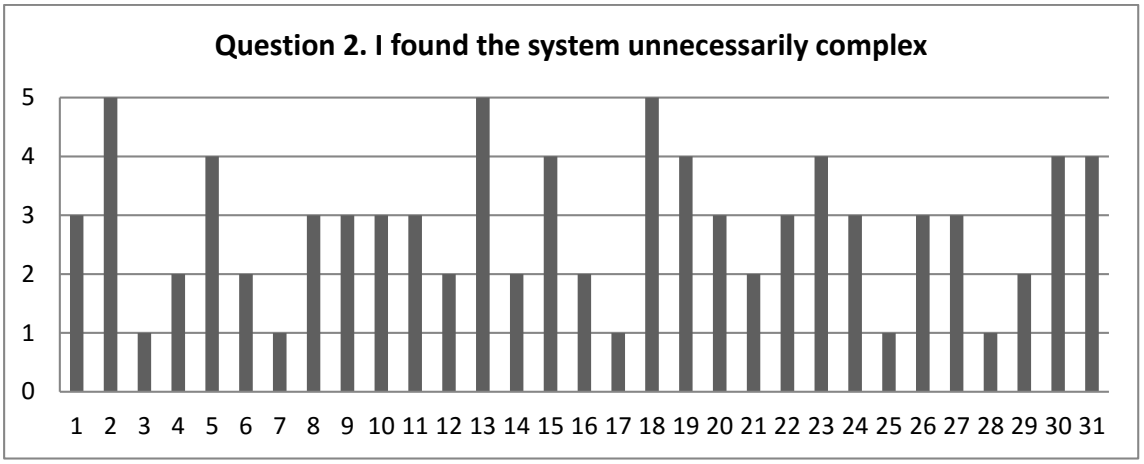


Figure 42: SUS Question 2

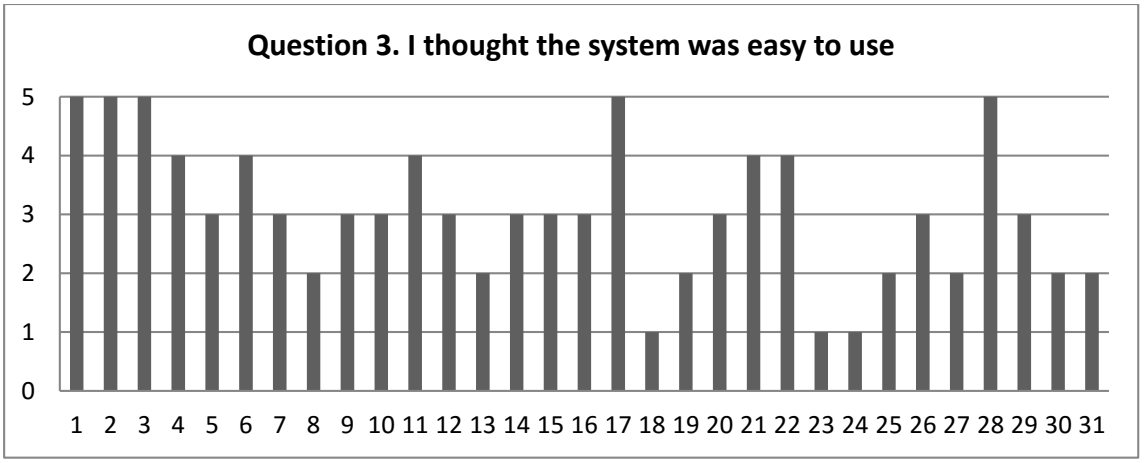


Figure 43: SUS Question 3

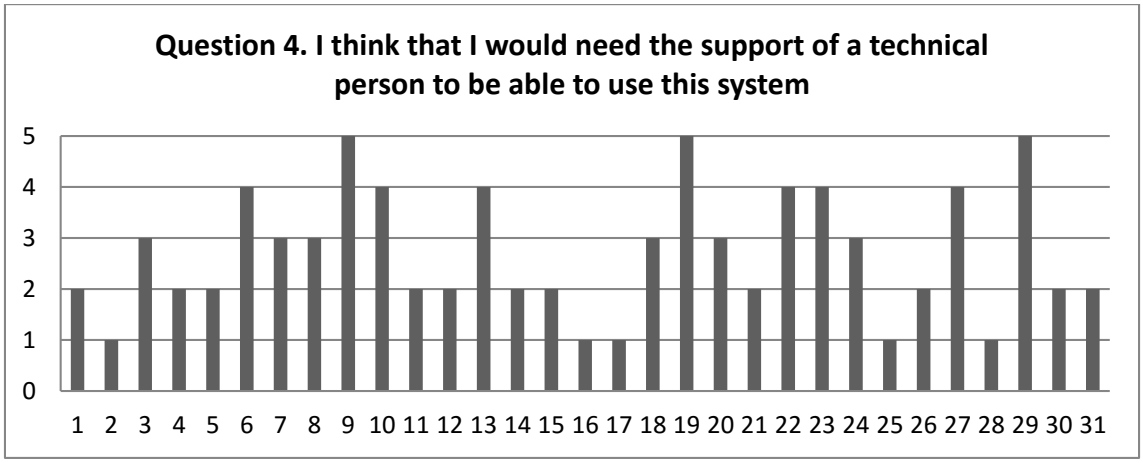


Figure 44: SUS Question 4

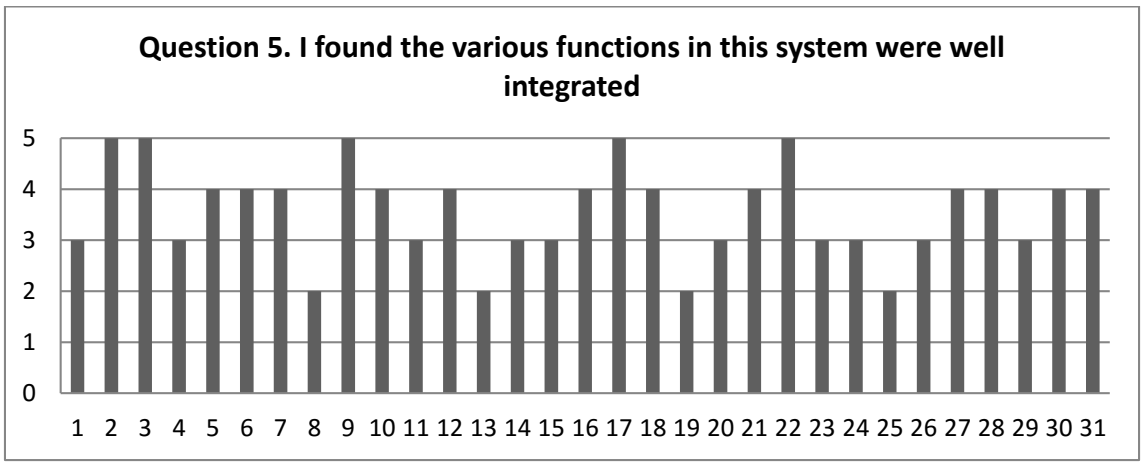


Figure 45: SUS Question 5

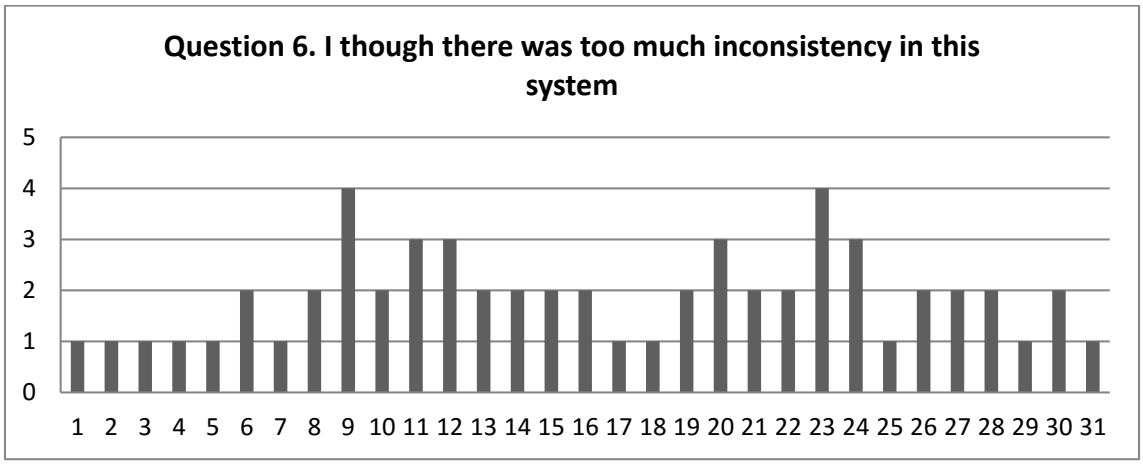


Figure 46: SUS Question 6

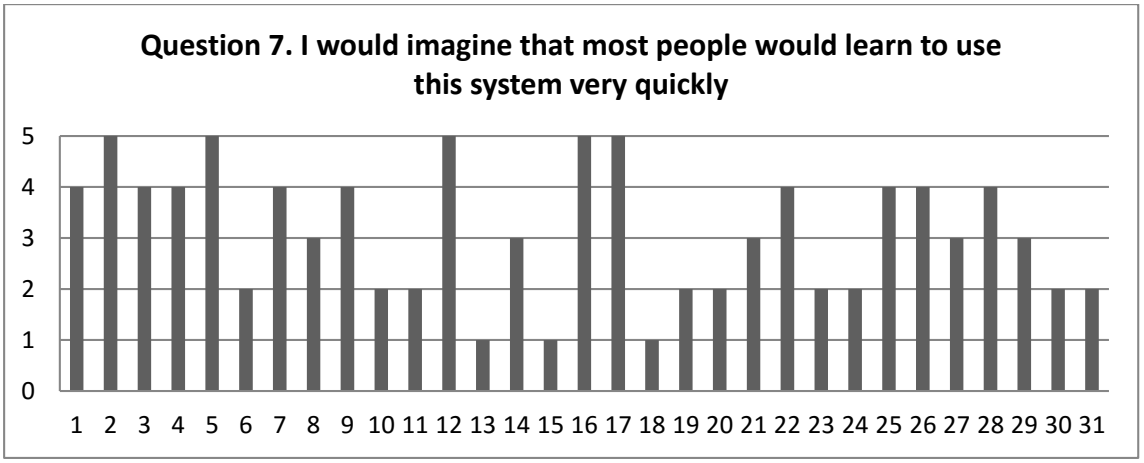


Figure 47: SUS Question 7

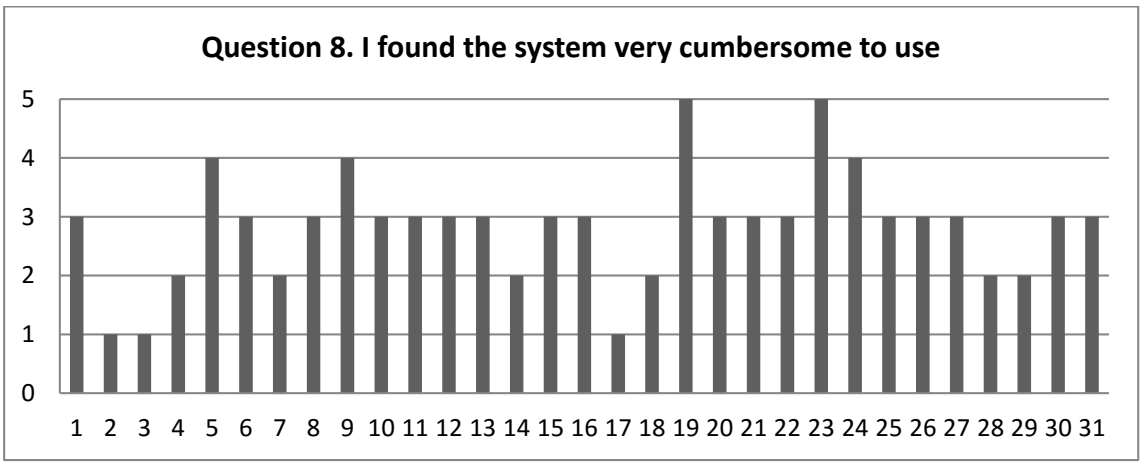


Figure 48: SUS Question 8

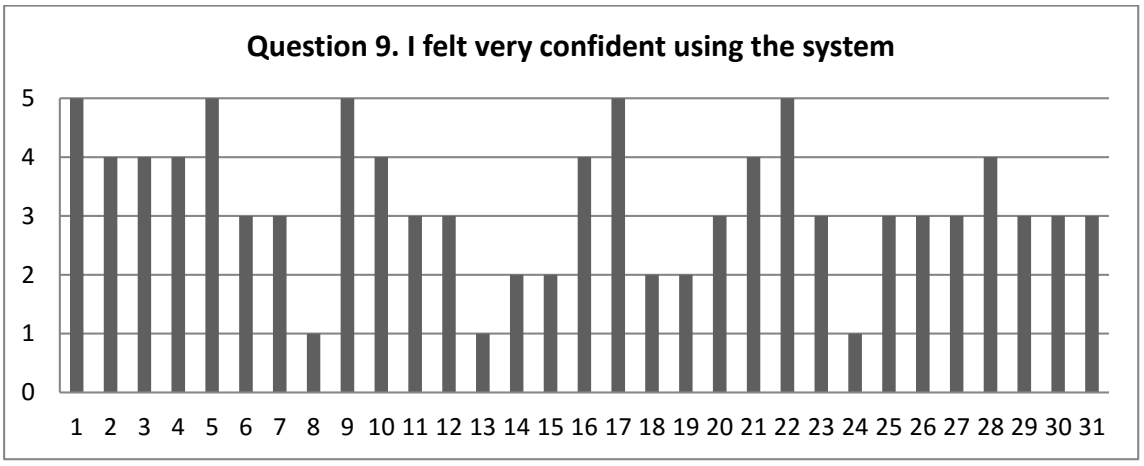


Figure 49: SUS Question 9

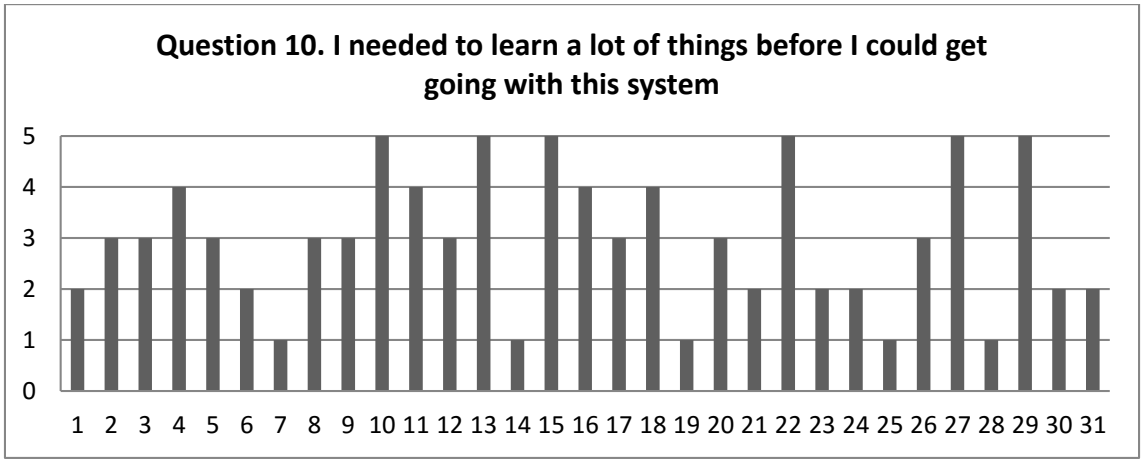


Figure 50: SUS Question 10

Overall evaluation score is acceptable, but additional usability improvements need to be considered in the next version of the MM-DSL IDE. For the overview of all the evaluation results see Appendix G: MM-DSL IDE Evaluation Results Overview.

13 Summary and Outlook

This dissertation project has been focused on improving the state of the art of the metamodeling research and its application in the development of a very specific kind of software – modeling tools. Although, concepts presented in this dissertation cover the realization of any graphical modeling language, the primary focus is set on modeling methods. The language-oriented approach used in this research envelopes all of the facets of a modeling method: abstract and concrete syntax, semantics, algorithms, mechanism and procedures.

MM-DSL, which is a technical implementation of the proposed language-oriented modeling method engineering approach, can be used to describe all of the modeling method facets. Modeling methods coded with MM-DSL can be executed on a metamodeling platform, creating a modeling tool. As a concept, MM-DSL is independent from its execution environment (which is mainly a metamodeling platform).

The similarity between meta^{2}models, which lie at the root of every metamodeling technology, is the secret ingredient that allows MM-DSL to be platform independent and at the same time very expressive in describing modeling methods. Considerable effort has been invested to discover the hidden links between various meta^{2}models. Because of the different syntax that describes the structure of meta^{2}model elements, it was not easy to find similarities between concepts in some of the meta^{2}models. Most important fact that has been discovered is that this different syntax maps to the same semantics. For example, a concept of a relation in one meta^{2}model is a single element, (e.g., a class in UML class diagram notation). In another meta^{2}model it is composed of several UML classes and relationships. But it is still a concept of a relation on an abstract level, therefore, semantically identical. MM-DSL translators (compilers, interpreters or code generators, depending on the execution platform) leverage the similarity of meta^{2}models to map the language concepts to the execution platform meta^{2}model. Adapting an already developed translator to the new platform is a straight forward process where one maps language concepts to the platform's meta^{2}model concepts. In this case, MM-DSL's concept of a class will be mapped to the platform's representation of a class. The same goes for every other MM-DSL concept. If there are MM-DSL concepts that are not supported by the execution platform, they will not be mapped. This means that the language itself still supports them, but the platform doesn't know what to do with them and will ignore them. If there are platform concepts that are not included in MM-DSL, one can include them as extensions. Careful consideration is advised when extending the language, because one does not want to: (1) invalidate the previously written code, and (2) make the language platform-specific.

Although, a lot of effort has been invested in finding the right concepts that are currently included in MM-DSL, there are most certainly some that have been missed or will

need to be included in the future. That is fine, and it is also expected, because every live language evolves over time.

The following are some of the suggested improvements which have been gathered from the community's feedback and throughout the evaluation process.

13.1 MM-DSL as an XML-Based Language

In some scenarios, there may be a necessity to change the concrete syntax of MM-DSL. Because XML-based languages are essentially very simple, they are wide spread and utilized by more people than programming languages. Although, MM-DSL is, from a perspective of a computer scientist and an experienced programmer, very easy to learn and apply, there are metamodeling experts that want to have an alternative way of describing domain concepts. This is why metamodeling community has expressed the need to encapsulate MM-DSL concepts in an XML format.

Additionally, transferring XML-based format messages over the computer network is a standardized procedure, which opens a possibility to provide MM-DSL and its translators as a service over the Internet.

13.2 Reverse Engineering: From Modeling Tools to Code

Currently, there is no support to automatically reverse engineer a modeling tool back to its MM-DSL program. MM-DSL translators only work in one direction: from MM-DSL code to execution platform format.

In a scenario, when one wants to migrate an already realized modeling method to another metamodeling platform, it would be useful to be able to translate the platform format to MM-DSL code. This is considered as reverse engineering. Reduction in effort and time needed to manually perform such an activity is very noticeable. Days or even months of hard work may be accomplished in minutes.

Reverse engineering itself is a very complex process. Therefore, the mapping of platform-specific code to MM-DSL may not produce a code that can be executed without adaption. Such activity would still require some basic knowledge of MM-DSL.

Finally, to produce another modeling tool, MM-DSL code is executed on another metamodeling platform. This entails that a translator for that particular platform already exists.

13.3 Additional Compilers

It has already been mentioned that a translator is required for each metamodeling platform where one wants to execute MM-DSL code. Precisely, a translator needs to be adapted to the underlying meta^{2}model. Basically, platforms that are based around the same meta^{2}model can in theory use identical translators. In most cases, provided

translator needs to be changed according to the format that execution platform understands.

There is also a possibility to use the already present framework to generate a different output. For example, one can use MM-DSL to describe a metamodel and then write a compiler that produces validation rules, which can be in form of first-order logic, descriptive logic, or any other format understandable by the rule system that will be used afterwards. These kind of special purpose compilers open several new MM-DSL application scenarios. We can describe modeling methods, and analyze them in several different ways at the same time. Verification is only one example. A compiler can also generate graphical artifacts, for example graphs where nodes represent classes and edges represent relations for the described metamodel. These graphs can also be attributed, which gives means to indicate if an element of a metamodel has attributes, including the type of these attributes, as well as if it has a graphical representation.

13.4 Towards Standardization

Nowadays, modeling and metamodeling communities depend upon a myriad of different tools and languages. There is no real medium for communication which is understood by everyone. There is no standard. There is no consensus on the terminology, except for the very basic terms such as class, relation or attribute. Prominent examples are the terms modeling method and modeling language. There are communities that do not acknowledge the existence of a modeling method as is described in this work. For these communities, what we consider a modeling method is a modeling language. What we consider a model type is a metamodel, and what we consider a metamodel is an integrated metamodel. There is no right or wrong in any of the terms mentioned here. However, communication is sometimes difficult, especially if the researches from different communities meet for the first time.

Eclipse community, as one of the largest modeling communities that produces metamodeling and modeling tools, has been very influential in the last couple of years, mostly because the tools are simple to use and provided as open source. Almost all of the modeling tools are based on Ecore meta^{2}model which has become de facto a standard. The conceptual roots of Ecore come from OMG and its MOF, precisely EMOF or Essential MOF. MOF is also the source of modeling and metamodeling terminology for several communities. However, there exist some issues with Ecore: (1) Eclipse dependent implementation, and (2) it is not a language, but a meta^{2}model, thus, it only contains the concepts for defining abstract syntax and semantics. Otherwise, it is a solid candidate for standardization.

MM-DSL itself was not developed with the standardization in mind. Nevertheless, it may serve as a communication medium between modeling method engineers until a more suitable language emerges. What makes this possible is the fact that MM-DSL contains all the needed concepts for modeling method description. Its specification is open and accessible, which makes understanding the syntax and semantics of the lan-

guage much easier. And last, but not the least, it is a community effort and community's feedback is reflected in the language itself and it is guiding its evolution. The version of MM-DSL described in this dissertation is not the initial version. There have been three previous versions and an additional IDE prototype. It is also very possible that the currently available version is not the one described in here, because the language itself is constantly changing and growing, as well as its IDE and other tools that support it.

Appendix A: MM-DSL Specification in EBNF

```
root ::=
  methodname (includelibrarytype | embedplatformtype | embedcodetype )* includelibrary* embedcode* method
methodname ::=
  'method' name
includelibrary ::=
  'include' '<' name (':' name-includelybrarytype)? '>'
includelibrarytype ::=
  'def' 'IncludeLibraryType' name
embedcode ::=
  'embed' name '<' name-embedplatformtype (':' name-embedcodetype)? '>' 'start' embeddedcodegoeshere 'end'
embedplatformtype ::=
  'def' 'EmbedPlatformType' name
embedcodetype ::=
  'def' 'EmbedCodeType' name
insertembedcode ::=
  'insert' name-embedcode
method ::=
  enumeration* symbolstyle* symbolclass* symbolrelation* metamodel algorithm* event*
enumeration ::=
  'enum' name '{' enumvalues+ '}'
metamodel ::=
  class+ relation* attribute* modeltype+
class ::=
  'class' name ('extends' name-class)? ('symbol' name-symbolclass)?
  '{' (classattribute | attribute | reference | insertembedcode)* '}'
relation ::=
  'relation' name ('extends' name-relation)? ('symbol' name-symbolrelation)?
  'from' name-class 'to' name-class '{' (attribute | insertembedcode)* '}'
attribute ::=
  'attribute' name ':' type ('acccess' ':' acesstype)?
accesstype ::=
  'write' | 'read' | 'internal'
reference ::=
  'reference' name '->' name-modeltype name-class?
classattribute ::=
  'classattribute' name ':' type
type ::=
  simpletype | enumtype
simpletype ::=
  'string' | 'int' | 'double'
enumtype ::=
  'enum' name-enumeration
modeltype ::=
  'modeltype' name '{' 'classes' name-class+ 'relations' ('none' | name-relation+)
  'modes' ('none' | name-mode+) '}'
mode ::=
  'mode' name 'include' 'classes' name-class+ 'relations' ('none' | name-relation+)
symbolclass ::=
  'classgraph' name ('style' name-symbolstyle)? '{' (svgcommand | insertembedcode)* '}'
symbolrelation ::=
  'relationgraph' name ('style' name-symbolstyle)? '{' 'from' (svgcommand | insertembedcode)*
  'middle' (svgcommand | insertembedcode)* 'to' (svgcommand | insertembedcode)* '}'
svgcommand ::=
  (rectangle | circle | ellipse | line | polyline | poligon | path | text) symbolstyle
rectangle ::=
  'rectangle' 'x' '=' REALNUMBER 'y' '=' REALNUMBER 'w' '=' NUMBER 'h' '=' NUMBER
circle ::=
  'circle' 'cx' '=' REALNUMBER 'cy' '=' REALNUMBER 'r' '=' NUMBER
```

```
ellipse ::=
  'ellipse' 'cx' '=' REALNUMBER 'cy' '=' REALNUMBER 'rx' '=' REALNUMBER 'ry' '=' REALNUMBER
line ::=
  'line' 'x1' '=' REALNUMBER 'y1' '=' REALNUMBER 'x2' '=' REALNUMBER 'y2' '=' REALNUMBER
polyline ::=
  'polyline' 'points' '=' points+
polygon ::=
  'polygon' 'points' '=' points+
path ::=
  'path' 'd' '=' pathdata+
text ::=
  'text' value 'x' '=' REALNUMBER 'y' '=' REALNUMBER ('font-family' '=' fontfamily)?
  ('font-size' '=' fontsize)? ('fill' '=' fillcolor)?
pathdata ::=
  moveto | lineto | horizontallineto | verticallineto | curveto | smoothcurveto |
  quadraticbeziercurveto | smoothquadraticbetiercurveto | ellipticalarc | closepath
moveto ::=
  ('M' | 'm') pathparametersMLT+
lineto ::=
  ('L' | 'l') pathparametersMLT+
horizontallineto ::=
  ('H' | 'h') pathparametersHV+
verticallineto ::=
  ('V'| 'v') pathparametersHV+
curveto ::=
  ('C' | 'c') PathParametersC+
smoothcurveto ::=
  ('S' | 's') pathparametersS+
quadraticbeziercurveto ::=
  ('Q' | 'q') pathparametersQ+
smoothquadraticbeziercurveto ::=
  ('T' | 't') pathparametersMLT+
ellipticalarc ::=
  ('A' | 'a') pathparametersA+
closepath ::=
  ('Z' | 'z')
points ::=
  x ',' y
pathparametersHV ::=
 x
pathparametersMLT ::=
  x ',' y
pathparametersS ::=
  x2 y2 x y
pathparametersQ ::=
  x1 y1 x y
pathparametersC ::=
  x1 y1 x2 y2 x y
pathparametersA ::=
  rx ',' ry xaxisrot largearcflag sweepflag x y
symbolstyle ::=
  'style' name '{' 'fill' ':' ('none' | fillcolor) 'stroke' ':' strokecolor 'stroke-width' ':' strokewidth
  ('font-family' ':' fontfamily)? ('font-size' ':' fontsize)? '}'
fillcolor ::=
  color | HEXCOLOR
color ::=
  standardHTMLcolors
fontfamily ::=
  STRING | font
font ::=
  standardWindowsFonts
fontsize ::=
  NUMBER
algorithm ::=
  'algorithm' name '{' statement* '}'
```

```
statement ::=
  selectionstatement | loopstatement | variable | algorithmoperation | insertembedcode
selectionstatement ::=
  ('if' '(' expression ')' '{' statement* '}') (('elseif' '(' expression ')' '{' statement* '}')*
  'else' '{' statement* '}')?
loopstatement ::=
  whileloop | forloop
whileloop ::=
  'while' '(' expression ')' '{' (statement | ('break' | 'continue'))* '}'
forloop ::=
  'for' '(' start ';' stop ';' interval ')' '{' (statement | ('break' | 'continue'))* '}'
variable ::=
  ('var' name (operatorassign varstatement)?) | (name-variable operatorassign varstatement)
varstatement ::=
  expression | algorithmoperation | ('class' name-class) | ('attribute' name-attribute) |
  ('reference' name-reference) | ('symbolclass' name-symbolclass) | ('symbolrelation' name-symbolrelation) |
  ('symbolstyle' name-symbolstyle) | ('embedded' name-embedcode) | ('modeltype' name-modeltype)
operatorassign ::=
  '=' | '+=' | '-=' | '*=' | '/='
operatorunary ::=
  '!'
operatormultiply ::=
  '*' | '/' | '%'
operatoradd ::=
  '+' | '-'
operatorcompare ::=
  '>=' | '<=' | '>' | '<'
operatorequal ::=
  '==' | '!='
operatorand ::=
  '&&'
operatoror ::=
  '||'
expression ::=
  orexpression
orexpression ::=
  andexpression (operatoror andexpression)*
andexpression ::=
  equalexpression (operatorand equalexpression)*
equalexpression ::=
  compareexpression (operatorequal compareexpression)*
compareexpression ::=
  additionexpression (operatorcompare additionexpression)*
additionexpression ::=
  multiplicationexpression (operatoradd multiplicationexpression)*
multiplicationexpression ::=
  unaryexpression (operatormultiply unaryexpression)*
unaryexpression ::=
  operatorunary? primaryexpression
primaryexpression ::=
  atomicexpression | ( '(' orexpression ')' )
atomicexpression ::=
  'true' | 'false' | name-variable | STRING | RealNumber
algorithmoperation ::=
  fileoperation | diroperation | simpleui | modeloperation | instanceoperation | attributeoperation
fileoperation ::=
  'file' '.' (filecopy | filedelete | filecreate | fileread | filewrite)
filecopy ::=
  'copy' 'source' src 'destination' dest
filedelete ::=
  'delete' filename
filecreate ::=
  'create' filename
fileread ::=
  'read' filename
```

```
filewrite ::=
  'write' filename ('append')?
diroperation ::=
  'dir' '.' (dirsetworking | dircreate | dirdelete | dircreate | dirlist)
dirsetworking ::=
  'set' dirname
dircreate ::=
  'create' dirname
dirdelete ::=
  'delete' dirname
dirlist ::=
  'list' dirname
simpleui ::=
  'ui' '.' (editbox | infobox | errorbox | warningbox | viewbox | itemoperation)
editbox ::=
  'editbox' title text ('button' okbuttontext)?
infobox ::=
  'infobox' title text
errorbox ::=
  'errorbox' title text 'button' buttontype
warningbox ::=
  'warningbox' title text 'button' buttontype
buttontype ::=
  'ok' | 'ok-cancel' | 'yes-no' | 'yes-no-cancel' | 'retry-cancel' | 'def-ok' | 'def-cancel' |
  'def-yes' | 'def-no' | 'def-retry'
viewbox ::=
  'viewbox' title text
itemoperation ::=
  'item' '.' (menuitem | contextitem)
menuitem ::=
  'menu' '.' (insertmenuitem | removemenuitem)
insertmenuitem ::=
  'insert' name 'to' menu
removemenuitem ::=
  'remove' name-menuitem
contextitem ::=
  'context' '.' (insertcontextitem | removecontextitem)
insertcontextitem ::=
  'insert' name 'to' context
removecontextitem ::=
  'remove' name-contextitem
modeloperation ::=
  'model' '.' (modelcreate | modeldelete | modeldiscard | modelsave | modelload | modelisloaded)
modelcreate ::=
  'create' name name-modeltype
modeldelete ::=
  'delete' name-model
modeldiscard ::=
  'discard' name-model
modelsave ::=
  'save' name-model
modelload ::=
  'load' name-model
modelisloaded ::=
  'isloaded' name-model
instanceoperation ::=
  'instance' '.' (classinstance | relationinstance)
classinstance ::=
  'class' '.'
  (classinstancecreate | classinstancedelete | classinstanceget | classinstanceset | classinstancegetall)
classinstancecreate ::=
  'create' name name-class
classinstancedelete ::=
  'delete' name-classinstance
classinstanceget ::=
```

```
  'get' name-classinstance
classinstanceset ::=
  'set' name-classinstance
classinstancegetall ::=
  'getall' name-class
relationinstance ::=
  'relation' '.'
  (relationinstancecreate | relationinstancedelete | relationinstanceget |
  relationinstanceset | relationinstancegetall)
relationinstancecreate ::=
  'create' name 'from' name-classinstancefrom 'to' name-classinstanceto
relationinstancedelete ::=
  'delete' name-relationinstance
relationinstanceget ::=
  'get' name-relationinstance
relationinstanceset ::=
  'set' name-relationinstance
relationinstancegetall ::=
  'getall' name-relation
attributeoperation ::=
  'attribute' '.' (attributeget | attributeset)
attributeget ::=
  'get' '.' attributegetparams
attributegetparams ::=
  ('type' | 'value' | 'name')
attributeset ::=
  'set' '.' attributesetparams
attributesetparams ::=
  'value'
event ::=
  'event' eventname '.' 'execute' '.' name-algorithm
eventname ::=
  'BeforeCreateModel' | 'BeforeCreateRelationInstance' | 'BeforeDeleteInstance' | 'BeforeDeleteModel' |
  'BeforeDiscardModel' | 'BeforeSaveModel' | 'CreateInstance' | 'CreateModel' | 'CreateRelationInstance' |
  'DeleteInstance' | 'DeleteModel' | 'DeleteRelationInstance' | 'DiscardInstance' | 'DiscardModel' |
  'OpenModel' | 'RenameInstance' | 'SaveModel' | 'SetAttributeValue' | 'AfterCreateModelingConnector' |
  'AfterCreateModelingNode' | 'AfterEditAttributeValue' | 'ToolInitialized'
name ::=
  ID
ID ::=
  '^'?('a..z'|'A..Z'|'_') ('a..z'|'A..Z'|'_'|'0..9')*
REALNUMBER ::=
  ('-')? Number
NUMBER ::=
  (HEX | (INT | DECIMAL) ('.' (INT | DECIMAL))?)
HEX ::=
  ('0x'|'0X') ('0..9'|'a..f'|'A..F'|'_')+
INT ::=
  '0..9' ('0..9'|'_')*
DECIMAL ::=
  INT (('e'|'E') ('+'|'-')? INT)?
HEXCOLOR ::=
  '#' ('a..f'|'A..F'|'0..9') ('a..f'|'A..F'|'0..9') ('a..f'|'A..F'|'0..9') ('a..f'|'A..F'|'0..9')
```

Appendix B: MM-DSL – Xtext Language Description

```
/*******************************************************************************
 * Copyright (c) 2015 Niksa Visic.
 * All rights reserved. This program and the accompanying materials
 * are made available under the terms of the Eclipse Public License v1.0
 * which accompanies this distribution, and is available at
 * http://www.eclipse.org/legal/epl-v10.html
 *******************************************************************************/

//grammar org.xtext.nv.dsl.MMDSL
//grammar org.xtext.nv.dsl.MMDSL with org.eclipse.xtext.common.Terminals
//grammar org.xtext.nv.dsl.MMDSL with org.eclipse.xtext.xbase.Xtype
grammar org.xtext.nv.dsl.MMDSL with org.eclipse.xtext.xbase.Xbase

import "http://www.eclipse.org/emf/2002/Ecore" as ecore
import "http://www.eclipse.org/xtext/common/JavaVMTypes" as types
import "http://www.eclipse.org/xtext/xbase/Xbase"

generate mMDSL 'http://www.xtext.org/nv/dsl/MMDSL'

// AST Root
Root:
     methodname=MethodName
     (includelibrarytype += IncludeLibraryType | embedplatformtype += EmbedPlatformType | embedcodetype +=
EmbedCodeType)*
     (includelibrary += IncludeLibrary)*
     (embedcode += EmbedCode)*
     method=Method
;

// used to generate the ADOxx library name
MethodName:
     'method' name=ValidID
;

// standard library defined for specific metamodeling platform, like ADOxx, Eclipse EMP, ...
// eg., include <adoxx:graphrep>

IncludeLibrary:
     'include' '<' name=ValidID (':' includelibrarytype=[IncludeLibraryType|QualifiedName])?  '>'
;

IncludeLibraryType:
     'def' 'IncludeLibraryType' name=ValidID
;

EmbedCode:
     'embed' name=ValidID '<' embedplatformtype=[EmbedPlatformType|QualifiedName] (':' embedcode-
type=[EmbedCodeType|QualifiedName])? '>'
     '{' embeddedcode=STRING '}'
;

EmbedPlatformType:
     'def' 'EmbedPlatformType' name=ValidID
;

EmbedCodeType:
     'def' 'EmbedCodeType' name=ValidID
;

InsertEmbedCode:
     'insert' codesnippetname=[EmbedCode|QualifiedName]
;

// modeling method contains one modeling language aka. metamodel,
// zero or more algorithms,
// zero or more mechanisms,
// zero or more procedures.
Method:
     (enumeration += Enumeration)*
     (symbolstyle += SymbolStyle)*
     (symbolclass += SymbolClass)*
     (symbolrelation += SymbolRelation)*
     metamodel=Metamodel
     (algorithm += Algorithm)*
     (event += Event)*
;
```

```
// enumerations are defined inside method scope and can be used as an attribute type
Enumeration:
     'enum' name=ValidID '{' (enumvalues += STRING)+ '}'
;


 /*
  ******************************************
  * Metamodel Grammar
  ******************************************
  */

// a metamodel is a construct containing:
// 1) at least one class
// 2) zero or more relations
// 3) zero or more attributes
// 4) at least one modeltype
Metamodel:
     (class += Class)+
     (relation += Relation)*
     (attribute += Attribute)*
     (modeltype += ModelType)+
;

// a class is a constructs that:
// 1) can extend other class
// 2) can contain zero or more attributes
Class:
     'class' name=ValidID ('extends' parentclassname=[Class|QualifiedName])? ('symbol' sym-
bolclass=[SymbolClass|QualifiedName])?
     '{' (classattribute += ClassAttribute | attribute += Attribute | insertembedcode += InsertEmbedCode |
reference += Reference)* '}'
;

// a relation is a constructs that:
// 1) can extend other relation (only attributes are inherited from the parent relation)
// 2) can contain zero or more attributes
Relation:
     'relation' name=ValidID ('extends' parentrelationname=[Relation|QualifiedName])? ('symbol' symbolrela-
tion=[SymbolRelation|QualifiedName])?
     'from' fromclassname=[Class|QualifiedName] 'to' toclassname=[Class|QualifiedName]
     '{' (attribute += Attribute | insertembedcode += InsertEmbedCode)* '}'
;

Attribute:
     'attribute' name=ValidID ':' type=Type ('access' ':' access=AccessType)?
;

// specifies if the attribute is visible and modifiable by the user
// default is internal (if access is not specified)
enum AccessType:
     write = 'write' | read = "read" |  internal = 'internal'
;

ClassAttribute:
     'classattribute' name=ValidID ':' type=Type
;

// references modeling object
Reference:
     'reference' name=ValidID '->' refname=RefName
;

// modeling objects that can be referenced
RefName:
     ('modeltype' modeltypename=[ModelType|QualifiedName]) ('class' classname=[Class|QualifiedName])?
;

Type:
     simpletype=SimpleType | enumtype=EnumType
;

enum SimpleType:
     String='string' | Int='int' | Double='double' // double produces a translation error
;

EnumType:
     'enum' name=[Enumeration|QualifiedName]
;

// a modeltype contains the collection of classes and relations
// it must contain at least one class
ModelType:
     'modeltype'   name=ValidID '{'
          'classes' (classname += [Class|QualifiedName])+
          'relations' ('none' | (relationname += [Relation|QualifiedName])+)
          'modes' ('none' | (modename += Mode)+)
```

```
    '}'
;

// a modeltype can contain zero or more modes aka. views in modeling canvas
Mode:
    'mode' name=ValidID  'include' ('classes' (classname += [Class|QualifiedName])+ 'relations' ('none' |
relationname += [Relation|QualifiedName])+)
;


 /*
  ******************************************
  * Graphical Representation Grammar
  ******************************************
  */

// 1) classes (SymbolClass) and relations (SymbolRelation) can be visualized
SymbolClass:
    'classgraph' name=ValidID ('style' globalstyle=[SymbolStyle|QualifiedName] )? '{' (svgcommand += SVGCom-
mand)* '}'
;

SymbolRelation:
    'relationgraph' name=ValidID ('style' globalstyle=[SymbolStyle|QualifiedName] )? '{'
        'from' (svgcommandsfrom += SVGCommand)*
        'middle' (svgcommandsmiddle += SVGCommand)*
        'to' (svgcommandsto += SVGCommand)*
    '}'
;

// SVG coordinate system starts with (0,0) in the top left corner
// it is calculated in pixels (px)
/*
 * (0,0)---(x, 0)
 *   |
 *   |
 * (0, y)
 */
SVGCommand:
/*
 * basic symbol shapes
 * based on SVG notation
 * Rectangle <rect>
 * Circle <circle>
 * Ellipse <ellipse>
 * Line <line>
 * Polyline <polyline>
 * Polygon <polygon>
 * Path <path>
 * Text <text>
 */
    (insertembedcode = InsertEmbedCode) |
    (
        (rectangle=Rectangle |
        circle=Circle |
        ellipse=Ellipse |
        line=Line |
        polyline=Polyline |
        polygon=Polygon |
        path=Path |
        text=Text)
        (symbolstyle=SymbolStyle | ('style' symbolstyleref=[SymbolStyle|QualifiedName]))?
    )
;

Rectangle:
    'rectangle' 'x' '=' x=RealNumber 'y' '=' y=RealNumber 'w' '=' width=Number 'h' '=' height=Number
;

Circle:
    'circle' 'cx' '=' cx=RealNumber 'cy' '=' cy=RealNumber 'r' '=' r=Number
;

Ellipse:
    'ellipse' 'cx' '=' cx=RealNumber 'cy' '=' cy=RealNumber 'rx' '=' rx=RealNumber 'ry' '=' ry=RealNumber
;

Line:
    'line' 'x1' '=' x1=RealNumber 'y1' '=' y1=RealNumber 'x2' '=' x2=RealNumber 'y2' '=' y2=RealNumber
;

Polyline:
    'polyline' 'points' '=' (points += Points)+ // format x1,y1 x2,y2 ... xn,yn
;

Polygon:
    'polygon' 'points' '=' (points += Points)+ // format x1,y1 x2,y2 ... xn,yn
```

```
;

Path:
     'path' 'd' '=' (pathdata += PathData)+
;

Text:
     'text' value=STRING 'x' '=' x=RealNumber 'y' '=' y=RealNumber
     ('font-family' '=' fontfamily=FontFamily)?
     ('font-size' '=' fontsize=FontSize)?
     ('fill' '=' fillcolor=FillColor )?
;

PathData:
     /*
    M=moveto
    L=lineto
    H=horizontal lineto
    V=vertical lineto
    C=curveto
    S=smooth curveto
    Q=quadratic Bézier curve
    T=smooth quadratic Bézier curveto
    A=elliptical Arc
    Z=closepath
*/
// All of the commands above can also be expressed with lower letters.
// Capital letters means absolutely positioned, lower cases means relatively positioned.

     moveto=MoveTo |
     lineto=LineTo |
     horizontallineto=HorizontalLineTo |
     verticallineto=VerticalLineTo |
     curveto=CurveTo |
     smoothcurveto=SmoothCurveTo |
     quadraticbeziercurve=QuadraticBezierCurve |
     smoothquadraticbeziercurveto=SmoothQuadraticBezierCurveTo |
     ellipticalarc=EllipticalArc |
     closepath=ClosePath
;

MoveTo:
     ('M' | 'm') (parameters += PathParametersMLT)+
;

LineTo:
     ('L' | 'l') (parameters += PathParametersMLT)+
;

HorizontalLineTo:
     ('H' | 'h') (parameters += PathParametersHV)+
;

VerticalLineTo:
     ('V'| 'v') (parameters += PathParametersHV)+
;

CurveTo:
     ('C' | 'c') (parameters += PathParametersC)+
;

SmoothCurveTo:
     ('S' | 's') (parameters += PathParametersS)+
;

QuadraticBezierCurve:
     ('Q' | 'q') (parameters += PathParametersQ)+
;

SmoothQuadraticBezierCurveTo:
     ('T' | 't') (parameters += PathParametersMLT)+
;

EllipticalArc:
     ('A' | 'a') (parameters += PathParametersA)+
;

ClosePath:
     ('Z' | 'z')
;

Points:
     x=RealNumber ',' y=RealNumber
;

// 1 parameter - H, V
PathParametersHV:
```

```
    x=RealNumber
;

// 2 parameters - M, L, T
PathParametersMLT:
    x=RealNumber ',' y=RealNumber
;

// 4 parameters - S
PathParametersS:
    x2=RealNumber y2=RealNumber x=RealNumber y=RealNumber
;

// 4 parameters - Q
PathParametersQ:
    x1=RealNumber y1=RealNumber x=RealNumber y=RealNumber
;

// 6 parameters - C
PathParametersC:
    x1=RealNumber y1=RealNumber x2=RealNumber y2=RealNumber x=RealNumber y=RealNumber
;

// 7 parameters - A
PathParametersA:
    rx=RealNumber ',' ry=RealNumber xaxisrot=RealNumber largearcflag=Number sweepflag=Number x=RealNumber
y=RealNumber
;

SymbolStyle:
    'style' name=ValidID '{'
        'fill' ':' ('none' | fillcolor=FillColor)
        'stroke' ':' strokecolor=StrokeColor
        'stroke-width' ':' strokewidth=StrokeWidth
        ('font-family' ':' fontfamily=FontFamily)?
        ('font-size' ':' fontsize=FontSize)?
        (insertembedcode += InsertEmbedCode)*
    '}'
;

FillColor:
    {FillColor}
    color=Color | hexcolor=HEXCOLOR
;

StrokeColor:
    {StrokeColor}
    color=Color | hexcolor=HEXCOLOR
;

StrokeWidth:
    Number
;

FontFamily:
    {FontFamily}
    fontstr=STRING | font=Font
;

FontSize:
    Number
;

// standard Windows Fonts
enum Font:
    arial='Arial' |
    arialblack='Arial_Black' |
    comicsansms='Comic_Sans_MS' |
    couriernew='Courier_New' |
    georgia='Georgia' |
    impact='Impact' |
    lucidaconsole='Lucida_Console' |
    lucidasansunicode='Lucida_Sans_Unicode' |
    palatinolinotype='Palatino_Linotype' |
    tahoma='Tahoma' |
    timesnewroman='Times_New_Roman' |
    trebuchetms='Trebuchet_MS' |
    verdana='Verdana' |
    symbol='Symbol' |
    webdings='Webdings' |
    windings='Wingdings' |
    mssansserif='MS_Sans_Serif' |
    msserif='MS_Serif'
;

// standard HTML Colors
enum Color:
```

```
aliceblue='aliceblue' |
antiquewhite='antiquewhite' |
aqua='aqua' |
aquamarine='aquamarine' |
azure='azure' |
beige='beige' |
bisque='bisque' |
black='black' |
blanchedalmond='blanchedalmond' |
blue='blue' |
blueviolet='blueviolet' |
brown='brown' |
burlywood='burlywood' |
cadetblue='cadetblue' |
chartreuse='chartreuse' |
chocolate='chocolate' |
coral='coral' |
cornflowerblue='cornflowerblue' |
cornsilk='cornsilk' |
crimson='crimson' |
cyan='cyan' |
darkblue ='darkblue' |
darkcyan='darkcyan' |
darkgoldenrod='darkgoldenrod' |
darkgray='darkgray' |
darkgreen='darkgreen' |
darkkhaki='darkkhaki' |
darkmagenta='darkmagenta' |
darkolivegreen='darkolivegreen' |
darkorange='darkorange' |
darkorchid='darkorchid' |
darkred='darkred' |
darksalmon='darksalmon' |
darkseagreen='darkseagreen' |
darkslateblue='darkslateblue' |
darkslategray='darkslategray' |
darkturquoise='darkturquoise' |
darkviolet='darkviolet' |
deeppink='deeppink' |
deepskyblue='deepskyblue' |
dimgray='dimgray' |
dodgerblue='dodgerblue' |
firebrick='firebrick' |
floralwhite='floralwhite' |
forestgreen='forestgreen' |
fuchsia='fuchsia' |
gainsboro='gainsboro' |
ghostwhite='ghostwhite' |
gold='gold' |
goldenrod='goldenrod' |
gray='gray' |
green='green' |
greenyellow='greenyellow' |
honeydew='honeydew' |
hotpink='hotpink' |
indianred='indianred' |
indigo='indigo' |
ivory='ivory' |
khaki='khaki' |
lavender='lavender' |
lavenderblush='lavenderblush' |
lawngreen='lawngreen' |
lemonchiffon='lemonchiffon' |
lightblue='lightblue' |
lightcoral='lightcoral' |
lightcyan='lightcyan' |
lightgoldenrodyellow='lightgoldenrodyellow' |
lightgreen='lightgreen' |
lightgray='lightgray' |
lightmagenta='lightmagenta' |
lightpink='lightpink' |
lightsalmon='lightsalmon' |
lightseagreen='lightseagreen' |
lightskyblue='lightskyblue' |
lightslategray='lightslategray' |
lightsteelblue='lightsteelblue' |
lightyellow='lightyellow' |
lime='lime' |
limegreen='limegreen' |
linen='linen' |
magenta='magenta' |
maroon='maroon' |
mediumaquamarine='mediumaquamarine' |
mediumblue='mediumblue' |
mediumorchid='mediumorchid' |
mediumpurple='mediumpurple' |
mediumseagreen='mediumseagreen' |
```

```
    mediumslateblue='mediumslateblue' |
    mediumspringgreen='mediumspringgreen' |
    mediumturquoise='mediumturquoise' |
    mediumvioletred='mediumvioletred' |
    midnightblue='midnightblue' |
    mintcream='mintcream' |
    mistyrose='mistyrose' |
    moccasin='moccasin' |
    navajowhite='navajowhite' |
    navy='navy' |
    oldlace='oldlace' |
    olive='olive' |
    olivedrab='olivedrab' |
    orange='orange' |
    orangered='orangered' |
    orchid='orchid' |
    palegoldenrod='palegoldenrod' |
    palegreen='palegreen' |
    paleturquoise='paleturquoise' |
    palevioletred='palevioletred' |
    papayawhip='papayawhip' |
    peachpuff='peachpuff' |
    peru='peru' |
    pink='pink' |
    plum='plum' |
    powderblue='powderblue' |
    purple='purple' |
    red='red' |
    rosybrown='rosybrown' |
    royalblue='royalblue' |
    saddlebrown='saddlebrown' |
    salmon='salmon' |
    sandybrown='sandybrown' |
    seagreen='seagreen' |
    seashell='seashell' |
    sienna='sienna' |
    silver='silver' |
    skyblue='skyblue' |
    slateblue='slateblue' |
    slategray='slategray' |
    snow='snow' |
    springgreen='springgreen' |
    steelblue='steelblue' |
    tan='tan' |
    teal='teal' |
    thistle='thistle' |
    tomato='tomato' |
    turquoise='turquoise' |
    violet='violet' |
    wheat='wheat' |
    white='white' |
    whitesmoke='whitesmoke' |
    yellow='yellow' |
    yellowgreen   = 'yellowgreen'
;


 /*
  ******************************************
  * Algorithm Grammar
  ******************************************
  */

Algorithm:
    'algorithm' name=ValidID '{' stmnt += Statement* '}'
;

// all possible statements for algorithm implementation
Statement:
    selection=SelectionStatement |
    loop=LoopStatement |
    variable=Variable |
    algorithmoperation = AlgorithmOperation |
    insertembedcode = InsertEmbedCode
;

// selection: if-elseif-else
SelectionStatement:
    ('if' '(' ifcondition=Expr ')' '{' ifblock+=Statement* '}')
    (('elseif''(' elseifcondition+=Expr ')' '{' elseifblock+=Statement* '}')* 'else' '{'
elseblock+=Statement* '}')?
;

// loops
LoopStatement:
    whiletloop=WhileLoop | forloop=ForLoop
;
```

```
WhileLoop:
     'while' '(' condition=Expr ')' '{' (whileblock+=Statement | breakcontinue+=BreakContinue)* '}'
;

ForLoop:
     'for' '(' start=INT ';' stop=INT ';' interval=INT ')' '{' (forblock+=Statement | breakcontin-
ue+=BreakContinue)* '}'
;

BreakContinue:
     break='break' | continue='continue'
;

// variable declaration and initialization
Variable:
     ('var' name=ValidID (opassing=OperatorAssign varstatement=VarStatement)?) |
     (variable=[Variable|QualifiedName] opassing=OperatorAssign varstatement=VarStatement)
;

// list of statements that can be used as variable assignments
VarStatement:
     expression=Expr |
     algorithmoperation = AlgorithmOperation |
     ('class' class=[Class|QualifiedName]) |
     ('attribute' attribute=[Attribute|QualifiedName]) |
     ('reference' reference=[Reference|QualifiedName]) |
     ('symbolclass' symbolclass=[SymbolClass|QualifiedName]) |
     ('symbolrelation' symbolrelation=[SymbolRelation|QualifiedName]) |
     ('symbolstyle' symbolstyle=[SymbolStyle|QualifiedName]) |
     ('embedded' embeddedcode=[EmbedCode|QualifiedName]) |
     ('modeltype' modeltype=[ModelType|QualifiedName])
;

// precednece - last
OperatorAssign:
     assign='=' | multyassign=OperatorMultyAssign
;

OperatorMultyAssign:
     addassign='+=' | subassign='-=' | multiassign='*=' | divassign='/='
;

// precednece 1
OperatorUnary:
     not='!'
;

// precednece 2
OperatorMultiply:
     multiply='*' | divide='/' | modulo = '%'
;

// precednece 3
OperatorAdd:
     add='+' | subtract='-'
;

// precednece 4
OperatorCompare:
     greaterequal='>=' | lesserequal='<=' | greater='>' | lesser='<'
;

// precednece 5
OperatorEqual:
     equal='==' | notequal='!='
;

// precednece 6
OperatorAnd:
     and='&&'
;

// precednece 7
OperatorOr:
     or='||'
;

// lowest precedence operation
Expr:
     expr=OrExpression
;

// 7
OrExpression returns Expression:
     AndExpression (=>({OrExpression.left=current} op=OperatorOr) right=AndExpression)*
;
```

```
// 6
AndExpression returns Expression:
     EqualExpression (=>({AndExpression.left=current} op=OperatorAnd) right=EqualExpression)*
;

// 5
EqualExpression returns Expression:
     CompareExpression (=>({EqualExpression.left=current} op=OperatorEqual) right=CompareExpression)*
;

// 4
CompareExpression returns Expression:
     AdditionExpression(=>({CompareExpression.left=current} op=OperatorCompare) right=AdditionExpression)*
;

// 3
AdditionExpression returns Expression:
     MultiplicationExpression (=>({AdditionExpression.left=current} op=OperatorAdd)
right=MultiplicationExpression)*
;

// 2
MultiplicationExpression returns Expression:
     UnaryExpression (=>({MultiplicationExpression.left=current} op=OperatorMultiply) right=UnaryExpression)*
;

// 1
UnaryExpression returns Expression:
     (op=OperatorUnary)? operand=PrimaryExpression
;

PrimaryExpression returns Expression:
     atomic=AtomicExpression | ('(' expression=OrExpression ')')
;

AtomicExpression returns Expression:
     true='true' | false='false' | variable=[Variable|QualifiedName] |
     valueString=STRING | valueRealNumber=RealNumber
;


 /*
  ******************************************
  * Algorithm Operations - ADOscript
  ******************************************
  */

AlgorithmOperation:
     fileoperation=FileOperation |
     diroperation=DirOperation |
     simpleui=SimpleUI |
     modeloperation=ModelOperation |
     instanceoperation=InstanceOperation |
     attributeoperation=AttributeOperation
;

// file
FileOperation:
     'file' '.' (filecopy = FileCopy | filedelete = FileDelete | filecreate = FileCreate | fileread = FileRead
| filewrite = FileWrite)
;

FileCopy:
     'copy' 'source' src=STRING 'destination' dest=STRING
;

FileDelete:
     'delete' filename=STRING
;

FileCreate:
     'create' filename=STRING
;

FileRead:
     'read' filename=STRING
;

FileWrite:
     'write' filename=STRING 'text' text=STRING (append='append')?
;

// directory
DirOperation:
     'dir' '.' (dirsetworking=DirSetWorking | dirgetworking=DirGetWorking | dircreate=DirCreate |
dirdelete=DirDelete | dirlist=DirList)
```

```
;

DirSetWorking:
     'set' dirname=STRING
;

DirGetWorking:
     {DirGetWorking}
     'get'
;

DirCreate:
     'create' dirname=STRING
;

DirDelete:
     'delete' dirname=STRING
;

DirList:
     'list' dirname=STRING
;

// simple UI
SimpleUI:
     'ui' '.' (editbox=EditBox | infobox=InfoBox | errorbox=ErrorBox | warningbox=WarningBox | viewbox=ViewBox
| itemoperation=ItemOperation)
;

EditBox:
     'editbox' 'title' title=STRING 'text' text=STRING ('button' okbuttontext=STRING)?
;

InfoBox:
     'infobox' 'title' title=STRING 'text' text=STRING
;

ErrorBox:
     'errorbox' 'title' title=STRING 'text' text=STRING 'button' buttontype=ButtonType
;

WarningBox:
     'warningbox' 'title' title=STRING 'text' text=STRING 'button' buttontype=ButtonType
;

enum ButtonType:
     ok='ok' | okcancel='ok-cancel' | yesno='yes-no' | yesnocancel='yes-no-cancel' | retrycancel='retry-
cancel' |
     defok='def-ok' | defcancel='def-cancel' | defyes='def-yes' | defno='def-no' | defretry='def-retry'
;

ViewBox:
     'viewbox' 'title' title=STRING 'text' text=STRING
;

// menu item manipulation (application) - is a part of SimpleUI
ItemOperation:
     'item' '.' (menuitem=MenuItem | contextitem=ContextItem)
;

MenuItem:
     'menu' '.' (insertmenuitem=InsertMenuItem | removemenuitem=RemoveMenuItem)
;

InsertMenuItem:
     'insert' name=ValidID 'to' menu=ValidID
;

RemoveMenuItem:
     'remove' menuitemname=[InsertMenuItem|QualifiedName]
;

ContextItem:
     'context' '.' (insertcontextitem=InsertContextItem | removecontextitem=RemoveContextItem)
;

InsertContextItem:
     'insert' name=ValidID 'to' context=ValidID
;

RemoveContextItem:
     'remove' contextitem=[InsertContextItem|QualifiedName]
;

// model manipulation (core)
ModelOperation:
     'model' '.' (modelcreate=ModelCreate | modeldelete=ModelDelete | modeldiscard=ModelDiscard |
```

```
        modelsave=ModelSave | modelload=ModelLoad | modelisloaded=ModelIsLoaded)
;

ModelCreate:
    'create' name=ValidID modeltype=[ModelType]
;

ModelDelete:
    'delete' modelname=[ModelCreate|QualifiedName]
;

ModelDiscard:
    'discard' modelname=[ModelCreate|QualifiedName]
;

ModelSave:
    'save' modelname=[ModelCreate|QualifiedName]
;

ModelLoad:
    'load' modelname=[ModelCreate|QualifiedName]
;

ModelIsLoaded:
    'isloaded' modelname=[ModelCreate|QualifiedName]
;

// instance manipulation (core)
InstanceOperation:
    'instance' '.' (classinstance=ClassInstance | relationinstance=RelationInstance)
;

ClassInstance:
    'class' '.' (classinstancecreate=ClassInstanceCreate | classinstancedelete=ClassInstanceDelete |
classinstanceget=ClassInstanceGet | classinstanceset=ClassInstanceSet | classinstancege-
tall=ClassInstanceGetAll)
;

ClassInstanceCreate:
    'create' name=ValidID nameofclass=[Class|QualifiedName]
;

ClassInstanceDelete:
    'delete' nameofclassinstance=[ClassInstanceCreate|QualifiedName]
;

ClassInstanceGet:
    'get' nameofclassinstance=[ClassInstanceCreate|QualifiedName]
;


ClassInstanceGetAll:
    'getall' nameofclass=[Class|QualifiedName]
;

ClassInstanceSet:
    'set' nameofclassinstance=[ClassInstanceCreate|QualifiedName]
;

RelationInstance:
    'relation' '.' (relationinstancecreate=RelationInstanceCreate | relationinstancede-
lete=RelationInstanceDelete | relationinstanceget=RelationInstanceGet | relationinstance-
set=RelationInstanceSet | relationinstancegetall=RelationInstanceGetAll)
;

RelationInstanceCreate:
    'create' name=ValidID nameofrelation=[Relation|QualifiedName] 'from' classinstance-
from=[ClassInstanceCreate|QualifiedName] 'to' classinstanceto=[ClassInstanceCreate|QualifiedName]
;

RelationInstanceDelete:
    'delete' nameofrelationinstance=[RelationInstanceCreate|QualifiedName]
;

RelationInstanceGet:
    'get' nameofrelationinstance=[RelationInstanceCreate|QualifiedName]
;

RelationInstanceGetAll:
    'getall' nameofrelation=[Relation|QualifiedName]
;

RelationInstanceSet:
    'set' nameofrelationinstance=[RelationInstanceCreate|QualifiedName]
;
```

```
// attribute manipulation (core)
AttributeOperation:
     attributename=[Attribute|QualifiedName] '.' (attributeget=AttributeGet | attributeset=AttributeSet)
;

AttributeGet:
     'get' '.' attrgetparams=AttrGetParams
;

enum AttrGetParams:
     type='type' |
     value='value' |
     name='name'
;

AttributeSet:
     'set' '.' attrsetparams=AttrSetParams (valueString=STRING | valueRealNumber=RealNumber | valueVaria-
ble=[Variable|QualifiedName])
;

enum AttrSetParams:
     value='value'
;

 /*
  *******************************************
  * Events - ADOscript
  *******************************************
  */

Event:
     'event' '.' name=EventName '.' 'execute' '.' algorithmname=[Algorithm|QualifiedName]
;

enum EventName:
     beforecreatemodel='BeforeCreateModel' |
     beforecreaterelationinstance='BeforeCreateRelationInstance' |
     beforedeleteinstance='BeforeDeleteInstance' |
     beforedeletemodel='BeforeDeleteModel' |
     beforediscardmodel='BeforeDiscardModel' |
     beforesavemodel='BeforeSaveModel' |
     createinstance='CreateInstance' |
     createmodel='CreateModel' |
     createrelationinstance='CreateRelationInstance' |
     deleteinstance='DeleteInstance' |
     deletemodel='DeleteModel' |
     deleterelationinstance='DeleteRelationInstance' |
     discardinstance='DiscardInstance' |
     discardmodel='DiscardModel' |
     openmodel='OpenModel' |
     renameinstance='RenameInstance' |
     savemodel='SaveModel' |
     setattributevalue='SetAttributeValue' |
     aftercreatemodelingconnector='AfterCreateModelingConnector' |
     aftercreatemodelingnode='AfterCreateModelingNode' |
     aftereditattributevalue='AfterEditAttributeValue' |
     toolinitialized='ToolInitialized'
;

 /*
  *******************************************
  * Expressions and Operators Grammar (Xbase)
  *******************************************
  */

//
https://github.com/eclipse/xtext/blob/master/plugins/org.eclipse.xtext.xbase/src/org/eclipse/xtext/xbase/Xbase
.xtext


 /*
  *******************************************
  * Terminals
  *******************************************
  */

//terminal ID :
//  '^'?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*
//  ;
//
//terminal STRING :
//            '"' ( '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') | !('\\'|'"') )* '"' |
//            "'" ( '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') | !('\\'|"'") )* "'"
//;
//
//terminal ML_COMMENT  : '/*' -> '*/';
//terminal SL_COMMENT  : '//' !('\n'|'\r')* ('\r'? '\n')?;
```

```
//
//terminal WS          : (' '|'\t'|'\r'|'\n')+;
//
//terminal ANY_OTHER: .;

// int with negative values
//terminal INT returns ecore::EInt:
//   ('-')? ('0'..'9')+
//;

RealNumber:
     ('-')? Number
;

// hex color representation: #123456
terminal HEXCOLOR:
     '#'
     ('a'..'f'|'A'..'F'|'0'..'9')
     ('a'..'f'|'A'..'F'|'0'..'9')
     ('a'..'f'|'A'..'F'|'0'..'9')
     ('a'..'f'|'A'..'F'|'0'..'9')
     ('a'..'f'|'A'..'F'|'0'..'9')
     ('a'..'f'|'A'..'F'|'0'..'9')

;
```

Appendix C: MetaDSL – Irony Language Description

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Irony.Parsing;

namespace Irony.Samples.MetaDSL {
  // Loosely based on Metamodeling concepts - research project.

  [Language("MetaDSL", "0.1", "MetaDSL grammar")]
  public class MetaDSLGrammar : Grammar {
      public MetaDSLGrammar()
          : base(true) { //MetaDSL is case insensitive

      //TERMINALS
      var comment = new CommentTerminal("comment", "/*", "*/");
      var lineComment = new CommentTerminal("line_comment", "//", "\n", "\r\n");
      NonGrammarTerminals.Add(comment);
      NonGrammarTerminals.Add(lineComment);
      var number = new NumberLiteral("number");
      var string_literal = new StringLiteral("string", "'", StringOptions.AllowsDoubledQuote);
      var identifier = new IdentifierTerminal("identifier");

      var COMMA = ToTerm(",");
      var DOT = ToTerm(".");
      var SEMICOLON = ToTerm(";");
      var COLON = ToTerm(":");
      var LBR1 = ToTerm("{");
      var RBR1 = ToTerm("}");
      var LBR2 = ToTerm("(");
      var RBR2 = ToTerm(")");
      var LBR3 = ToTerm("[");
      var RBR3 = ToTerm("]");

      var METHOD = ToTerm("method");

      var PACKAGE = ToTerm("package");
      var MODELTYPE = ToTerm("modeltype");
      var EVENT = ToTerm("event");
      var ENUM = ToTerm("enum");

      var ABSTRACT = ToTerm("abstract");
      var CLASS = ToTerm("class");
      var RELATION = ToTerm("relation");

      var NULL = ToTerm("null");
      var NOT = ToTerm("not");
      var CONSTRAINT = ToTerm("constraint");
      var FROM = ToTerm("from");
      var TO = ToTerm("to");
      var DATATYPE = ToTerm("datatype");

      var ATTRIBUTE = ToTerm("attribute");
      var OPERATION = ToTerm("operation");
      var REFERENCE = ToTerm("reference");
      var CONTAINER = ToTerm("container");
      var NOTATION = ToTerm("notation");

      var RECTANGLE = ToTerm("rectangle");
      var ELLIPSE = ToTerm("ellipse");
      var POLYGON = ToTerm("polygon");

      var STRING = ToTerm("string");
      var INT = ToTerm("int");
      var FLOAT = ToTerm("float");
      var DOUBLE = ToTerm("double");
      var BOOL = ToTerm("bool");

      var FOR = ToTerm("for");
      var FOREACH = ToTerm("foreach");
      var WHILE = ToTerm("while");
      var SWITCH = ToTerm("switch");

      var SET = ToTerm("set");
      var GET = ToTerm("get");
      var USE = ToTerm("use");
      var ADD = ToTerm("add");
```

```
var IF = ToTerm("if");
var ELSEIF = ToTerm("elseif");
var ELSE = ToTerm("else");
var RETURN = ToTerm("return");
var VAR = ToTerm("var");
var THROW = ToTerm("throw");

//include concepts
var INCLUDE = ToTerm("include");

//embed concepts
var EMBED = ToTerm("embed");

//CONSTANTS:
var constants = new ConstantsTable();
constants.Add("pi", Math.PI);
constants.Add("true", true);
constants.Add("false", false);

//Non-terminals:
//method
var method = new NonTerminal("method");
var methodDeclaration = new NonTerminal("methodDeclaration");
var methodBody = new NonTerminal("methodBody");
var methodStatement = new NonTerminal("methodStatement");

//package
var packages = new NonTerminal("packages");
var package = new NonTerminal("package");
var packageDeclaration = new NonTerminal("packageDeclaration");
var packageBody = new NonTerminal("packageBody");

//modeltype
var modeltypes = new NonTerminal("modeltypes");
var modeltype = new NonTerminal("modeltype");
var modeltypeDeclaration = new NonTerminal("modeltypeDeclaration");
var modeltypeBody = new NonTerminal("modeltypeBody");

//statement
var statements = new NonTerminal("statements");
var statement = new NonTerminal("statement");
var statementList = new NonTerminal("statementList");
var block = new NonTerminal("block");

//abstract
var abstractStatement = new NonTerminal("abstractStatement");
var abstractDeclaration = new NonTerminal("abstractDeclaration");
var abstractBody = new NonTerminal("abstractBody");

//class
var classStatement = new NonTerminal("classStatement");
var classDeclaration = new NonTerminal("classDeclaration");
var classBody = new NonTerminal("classBody");
var classNoParent = new NonTerminal("classNoParent");
var classExtends = new NonTerminal("classExtends");

//relation
var relationStatement = new NonTerminal("relationStatement");
var relationDeclaration = new NonTerminal("relationDeclaration");
var relationBody = new NonTerminal("relationBody");
var relationFrom = new NonTerminal("relationFrom");
var relationTo = new NonTerminal("relationTo");
var relationNoParent = new NonTerminal("relationNoParent");
var relationExtends = new NonTerminal("relationExtends");

//acrStatement - used in abstract, class and relation
var acrStatements = new NonTerminal("acrStatements");
var acrStatement = new NonTerminal("acrStatement");

//listOfIdentifiers - separated by comma
var identifiers = new NonTerminal("identifiers");

//listOfNumbers - separated by comma
var numbers = new NonTerminal("numbers");

//attributes
var attributeStatement = new NonTerminal("attributeStatement");
var attributeDeclaration = new NonTerminal("attributeDeclaration");
var attributeBody = new NonTerminal("attributeBody");

//operation
var operationStatement = new NonTerminal("operationStatement");
var operationDeclaration = new NonTerminal("operationDeclaration");
var operationBody = new NonTerminal("operationBody");

//notation
```

```
var notationStatement = new NonTerminal("notationStatement");
var notationDeclaration = new NonTerminal("notationDeclaration");
var notationBody = new NonTerminal("notationBody");
var notationType = new NonTerminal("notationType");

//notatyon types
//rectangle
var rectangleStatement = new NonTerminal("rectangleStatement");
var rectangleParam = new NonTerminal("rectangleParam");
//ellipse
var ellipseStatement = new NonTerminal("ellipseStatement");
var ellipseParam = new NonTerminal("ellipseParam");
//polygon
var polygonStatement = new NonTerminal("polygonStatement");
var polygonParam = new NonTerminal("polygonParam");

//type
var typeDeclaration = new NonTerminal("typeDeclaration");
var builtInType = new NonTerminal("builtInType");
var enumType = new NonTerminal("enumType");

//enumeration
var enumStatements = new NonTerminal("enumStatements");
var enumStatement = new NonTerminal("enumStatement");
var enumDeclaration = new NonTerminal("enumDeclaration");
var enumBody = new NonTerminal("enumBody");
var enumParameter = new NonTerminal("enumParameter");

//include
var includeStatements = new NonTerminal("includeStatements");
var includeStatement = new NonTerminal("includeStatement");
var includeDeclaration = new NonTerminal("includeDeclaration");
var includeBody = new NonTerminal("includeBody");

//embed
var embedStatements = new NonTerminal("embedStatements");
var embedStatement = new NonTerminal("embedStatement");
var embedDeclaration = new NonTerminal("embedDeclaration");
var embedBody = new NonTerminal("embedBody");

//BNF Rules:

//symbols
var commaOpt = new NonTerminal("commaOpt", Empty | COMMA);
var commasOpt = new NonTerminal("commasOpt");
commasOpt.Rule = MakeStarRule(commasOpt, null, COMMA);

//Number comma separated list
numbers.Rule = MakePlusRule(numbers, COMMA, number);

//Identifier comma separated list
identifiers.Rule = MakePlusRule(identifiers, COMMA, identifier);

//program root element
this.Root = method;

//Method
method.Rule = methodDeclaration;
methodDeclaration.Rule = METHOD + identifier + LBR1 + methodBody + RBR1;
methodBody.Rule = methodStatement;
methodStatement.Rule = includeStatements + packages + modeltypes + enumStatements + embedStatements;
includeStatements.Rule = MakeStarRule(includeStatements, includeStatement);
packages.Rule = MakePlusRule(packages,package);
modeltypes.Rule = MakePlusRule(modeltypes, modeltype);
enumStatements.Rule = MakeStarRule(enumStatements, enumStatement);
embedStatements.Rule = MakeStarRule(embedStatements, embedStatement);

//Include
includeStatement.Rule = includeDeclaration + includeBody + SEMICOLON;
includeDeclaration.Rule = INCLUDE;
includeBody.Rule = identifiers;

//Package
package.Rule = packageDeclaration + LBR1 + packageBody + RBR1;
packageDeclaration.Rule = PACKAGE + identifier;
packageBody.Rule = statements;
statements.Rule = MakePlusRule(statements, statement);
statement.Rule = abstractStatement | classStatement | relationStatement;

//ModelType
modeltype.Rule = modeltypeDeclaration + LBR1 + modeltypeBody + RBR1;
modeltypeDeclaration.Rule = MODELTYPE + identifier;
modeltypeBody.Rule = identifiers;

//Abstract
abstractStatement.Rule = abstractDeclaration + LBR1 + abstractBody + RBR1;
abstractDeclaration.Rule = ABSTRACT + identifier;
```

```
abstractBody.Rule = acrStatements;

//Class
classStatement.Rule = classDeclaration + LBR1 + classBody + RBR1;
classDeclaration.Rule = classNoParent | classExtends;
classNoParent.Rule = CLASS + identifier;
classExtends.Rule = classNoParent + COLON + identifier;
classBody.Rule = notationStatement + acrStatements;

//Relation
relationStatement.Rule = relationDeclaration + LBR1 + relationBody + RBR1;
relationDeclaration.Rule = relationNoParent | relationExtends;
relationNoParent.Rule = RELATION + identifier;
relationExtends.Rule = relationNoParent + COLON + identifier;
relationBody.Rule = relationFrom + relationTo +  notationStatement + acrStatements;
relationFrom.Rule = FROM + COLON + identifiers + SEMICOLON;
relationTo.Rule = TO + COLON + identifiers + SEMICOLON;

//Attributes
attributeStatement.Rule = ATTRIBUTE + identifier + COLON + typeDeclaration;

//Operations
operationStatement.Rule = OPERATION + identifier + LBR2 + identifiers + RBR2 + COLON + typeDeclaration;

//Notations
notationStatement.Rule = notationDeclaration + notationBody + SEMICOLON;
notationDeclaration.Rule = NOTATION + COLON;
notationBody.Rule = notationType;
notationType.Rule = rectangleStatement | ellipseStatement | polygonStatement;

//Notation Types
rectangleStatement.Rule = RECTANGLE + rectangleParam;
rectangleParam.Rule = LBR2 + numbers + RBR2;
ellipseStatement.Rule = ELLIPSE + ellipseParam;
ellipseParam.Rule = LBR2 + numbers + RBR2;
polygonStatement.Rule = POLYGON + polygonParam;
polygonParam.Rule = LBR2 + numbers + RBR2;

//ACR Statement
acrStatements.Rule = MakeStarRule(acrStatements, acrStatement);
acrStatement.Rule = attributeStatement + SEMICOLON | operationStatement + SEMICOLON;

//Types
builtInType.Rule = INT | FLOAT | DOUBLE | BOOL | STRING;
enumType.Rule = identifier;
typeDeclaration.Rule = builtInType | enumType;

//Enumeration
enumStatement.Rule = enumDeclaration + LBR1 + enumBody + RBR1;
enumDeclaration.Rule = ENUM + identifier;
enumBody.Rule = identifiers;

//Embedded
embedStatement.Rule = embedDeclaration + embedBody + embedDeclaration + SEMICOLON;
embedDeclaration.Rule = EMBED;
embedBody.Rule = string_literal;

//Operators
RegisterOperators(1, "||");
RegisterOperators(2, "&&");
RegisterOperators(3, "|");
RegisterOperators(4, "^");
RegisterOperators(5, "&");
RegisterOperators(6, "==", "!=");
RegisterOperators(7, "<", ">", "<=", ">=", "is", "as");
RegisterOperators(8, "<<", ">>");
RegisterOperators(9, "+", "-");
RegisterOperators(10, "*", "/", "%");
RegisterOperators(11, ".");

RegisterOperators(-3, "=", "+=", "-=", "*=", "/=", "%=", "&=", "|=", "^=", "<<=", ">>=");
RegisterOperators(-2, "?");
RegisterOperators(-1, "??");

this.Delimiters = "{}[](),:;+-*/%&|^!~<>=";
this.MarkPunctuation(";", ",", "(", ")", "{", "}", "[", "]", ":");

//Whitespace and NewLine characters
//TODO:
// 1. In addition to "normal" whitespace chars, the spec mentions "any char of unicode class Z" -
//   need to create special comment-based terminal that simply eats these
//   category-based whitechars and produces comment token.
// 2. Add support for multiple line terminators to LineComment

//CR, linefeed, nextLine, LineSeparator, paragraphSeparator
this.LineTerminators = "\r\n\u2085\u2028\u2029";
//add extra line terminators
```

```
      this.WhitespaceChars = " \t\r\n\v\u2085\u2028\u2029";

    }//constructor

  }//class
}//namespace
```

Appendix D: Exercise Used to Evaluate MM-DSL

The following is the exercise used in the evaluation of usability and understandability of MM-DSL. It has been performed by twenty-two students within the Metamodeling course in summer semester of 2014.

Task Summary

Describe the Car Parking modeling method with MM-DSL. You have 60 minutes to complete the exercise. It is allowed to use any standard text editor like Notepad++ to write your code.

Preparation:

To be adequately prepared for the exercise it is advised that you study the MM-DSL Specification document available at http://www.omilab.org/web/guest/mm-dsl. You can use this document as a reference while doing the exercise.

Submission

You need to turn in the file (.txt) containing the code for the given modeling method. It should look similar to the code that can be found in the Appendix of the *"MM-DSL: An EBNF Specification"*.

Task Details

Figure 51 describes the abstract (metamodel) and concrete syntax (graphical objects) of the Car Parking modeling method:

Concrete syntax assigned to modeling elements is depicted in the table on the left side. Associations between various elements are given as an UML class diagram (right side). Classes and relations without graphical representations are *Vehicle*, *City*, *Park*, *belongs to*. The resulting diagram (model type) needs to contain the following classes: *Car*, *Truck*, *Motorcycle*, *Bicycle*, *Parking Lot*, *Parking Garage*, and relation *is parked*. The attributes need to have a type assigned to them, either int, string, double or enum. Enums or enumerations can be specified for attributes such as type, payment or color. For example, attribute type can have values: {"car" "truck" "motorcycle" "bicycle"}, attribute payment can have values: {"ticket machine" "mobile phone" "cash"}.

	Class	Super Class	Attributes
-	Vehicle	-	name, color, length, width, height, weight
	Car	Vehicle	-
	Truck	Vehicle	-
	Motorcycle	Vehicle	-
	Bicycle	Vehicle	-
-	City	-	name, latitude, longitude
-	Park	-	name, address, city, size, type, payment
	Parking Lot	Park	-
	Parking Garage	Park	-

	Relationship	From / To	Attributes
	is parked	Vehicle / Park	-
-	belongs to	Park / City	-

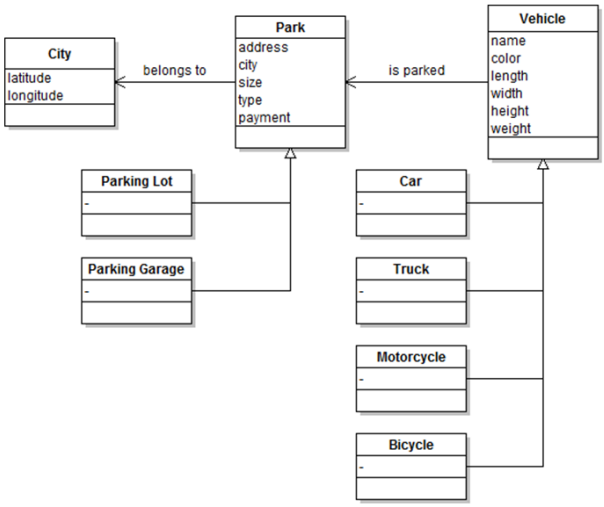
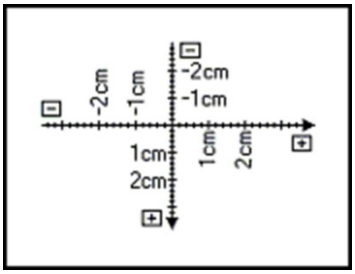


Figure 51: Concrete and Abstract Syntax of a Car Parking Modeling Method

Helpful Hints

The MM-DSL program has a certain structure. We begin by defining the method name, def and embed statements, followed by user data types or enums, styles, class graphs, relation graphs, then classes, relations, model types and at the end algorithms (and events).

Identifiers or names of elements must not have empty spaces (valid identifier: CarPark, Car_Park; invalid identifier: Car Park).



Graphical objects use the following coordinate system:

(0, 0) coordinate is in the middle. Try drawing objects in this coordinate system before writing the code for graphical representation. The units for all graphical objects are points (only integers are allowed, no decimal numbers). 1 centimeter has 28.34 points.

It is also possible to combine several SVG commands (one after another) to define complex graphical objects.

Solution

```
method CarParkModelingLanguage

enum EnumParkType { "car" "truck" "motorcycle" "bicycle" }
enum EnumPayment { "ticket machine" "mobile phone" "cash" }
```

```
style Blue {fill:blue stroke:black stroke-width:1}
style Green {fill:green stroke:black stroke-width:1}
style Black {fill:black stroke:black stroke-width:1}
//style Orange {fill:orange stroke:black stroke-width:1}
//style Red {fill:red stroke:black stroke-width:1}

classgraph CarGraph style Blue {
    rectangle x=-10 y=-10 w=20 h=20
}

classgraph TruckGraph style Blue {
    rectangle x=-5 y=-15 w=20 h=20
    rectangle x=-10 y=-10 w=20 h=20
}

classgraph MotorcycleGraph style Green {
    circle cx=0 cy=0 r=10
}

classgraph BicylceGraph style Green {
    circle cx=-5 cy=0 r=10
    circle cx=5 cy=0 r=10
}

classgraph ParkingLotGraph {
    rectangle x=-20 y=-20 w=40 h=40 style Orange {fill:orange stroke:black stroke-
width:1}
    polygon points=-20,20 0,20 -20,0 style Red {fill:red stroke:black stroke-width:1}
    polygon points=0,-20 20,-20 20,0 style Red {fill:red stroke:black stroke-width:1}
    text "L" x=-2 y=-2
}

classgraph ParkingGarageGraph {
    rectangle x=-20 y=-20 w=40 h=40 style Orange {fill:orange stroke:black stroke-
width:1}
    polygon points=-20,-20 0,-20 -20,0 style Red {fill:red stroke:black stroke-width:1}
    polygon points=0,20 20,20 20,0 style Red {fill:red stroke:black stroke-width:1}
    text "G" x=-2 y=-2
}

relationgraph IsParkedGraph style Black {
    from
    rectangle x=-2 y=-2 w=4 h=4
    middle
    text "is parked in" x=0 y=0
    to
    polygon points=-2,2 2,0 -2,-2
}

class Vehicle {
    attribute color:string access:write
    attribute lenght:int access:write
    attribute width:int access:write
    attribute height:int access:write
    attribute weight:int access:write
}

class Car extends Vehicle symbol CarGraph {}
class Truck extends Vehicle symbol TruckGraph {}
class Motorcycle extends Vehicle symbol MotorcycleGraph{}
class Bicycle extends Vehicle symbol BicylceGraph {}

class City {
    attribute latitude:int
    attribute longitude:int
}
```

```
class Park {
    attribute address:string
    attribute city:string
    attribute size:int
    attribute parktype:enum EnumParkType
    attribute payment:enum EnumPayment
}

class ParkingLot extends Park symbol ParkingLotGraph {}
class ParkingGarage extends Park symbol ParkingGarageGraph {}

relation isParked symbol IsParkedGraph from Vehicle to Park {}
relation belongsTo from Park to City{}

modeltype CarPark {
    classes Car Truck Motorcycle Bicycle ParkingLot ParkingGarage
    relations isParked
    modes none
}
```

Appendix E: Exercise Used to Evaluate MM-DSL IDE

The following is the exercise used to test the usability of MM-DSL IDE prototype. The participants had to use MM-DSL IDE to extend the given modeling method with additional features. The exercise included utilizing the ADOxx platform and Linked Open Data concepts (RDF) after the artifacts have been created with MM-DSL. These additional parts are not directly relevant to the MM-DSL IDE. However, as a whole the exercise demonstrates one of many application scenarios where multiple metamodeling technologies are chained together.

Thirty-one participants, which all had different degrees of knowledge about metamodeling and computer science, have completed the exercise. Some of them had a strictly business background, with no previous experience in programming.

Next Generation Enterprise Modeling: A Case Study

Business View

A courier company needs to keep track of available parking areas in the various geographical areas where it provides services and to provide its employees (couriers) the possibility to quickly reserve the necessary parking spaces when a task is assigned to them.

Its couriers have different types of tasks allocated to them, with transportation activities assigned to different cities.

Conceptualization

Choice of technology

Because the courier company has good experience with business process modeling, it would prefer to use modeling tools to design and communicate the tasks assigned to its couriers;

Because parking space availability is already published for free by a third party company that manages parking spaces, as Linked Open Data, RDF is chosen as a technology to store and retrieve this information;

Because the courier company implements a bring-your-own-device IT policy and couriers need high mobility, it would prefer to have a mobile app provided to its couriers.

Requirements for the IT support

Design-time. A task coordinator must be able to describe through models:

a) different types of tasks for its couriers;

b) the allocation of parking areas to the cities where it provides services;
c) a mapping of different types of tasks on the cities where they must be performed;

Run-time. A courier must be able:

a) to see his list of tasks;
b) to see the cities where a task will take him, as well as the parking space availability in those cities;
c) to reserve a parking space if available.

Design decisions

The design of the IT system covers several aspects pertaining to a proposed architecture:

a) An app must be designed, which should be able to query parking space availability information filtered by the current route;
b) A modeling method must be designed to enable the description of tasks and the allocation of parking areas to relevant cities. This includes the design of a metamodel that enables models to capture the necessary information;
c) The RDF vocabulary of the parking managing company must be investigated to understand what data structure is provided and how can the models be linked with the available Linked Open Data.
d) A bridge between the model information and the available Linked Open Data must be designed, to allow filtering legacy data by model information or vice versa. It has been decided that this will be in the form of an RDF vocabulary and a mechanism for serializing model information in RDF format.

Implementation decisions

Implementation will rely on the following environments:

a) ADOxx and the MM-DSL language will be used for the implementation of a modeling tool;
b) Sesame will be used as back-end storage; this implies that RDF will be used as a data model;
c) The bridge between the modeling environment and the Sesame storage will be implemented as a serialization mechanism that will represent model information in RDF format;
d) Android is the platform of choice for the mobile app and it will use SPARQL as a query technology, to retrieve the necessary information from the back-end data store.

Exercise Overview

Figure 52 shows the sequence of steps required to complete the exercise. The steps which were directly related with the MM-DSL IDE evaluation are 1, 2, 3 and the loop for method extensions.

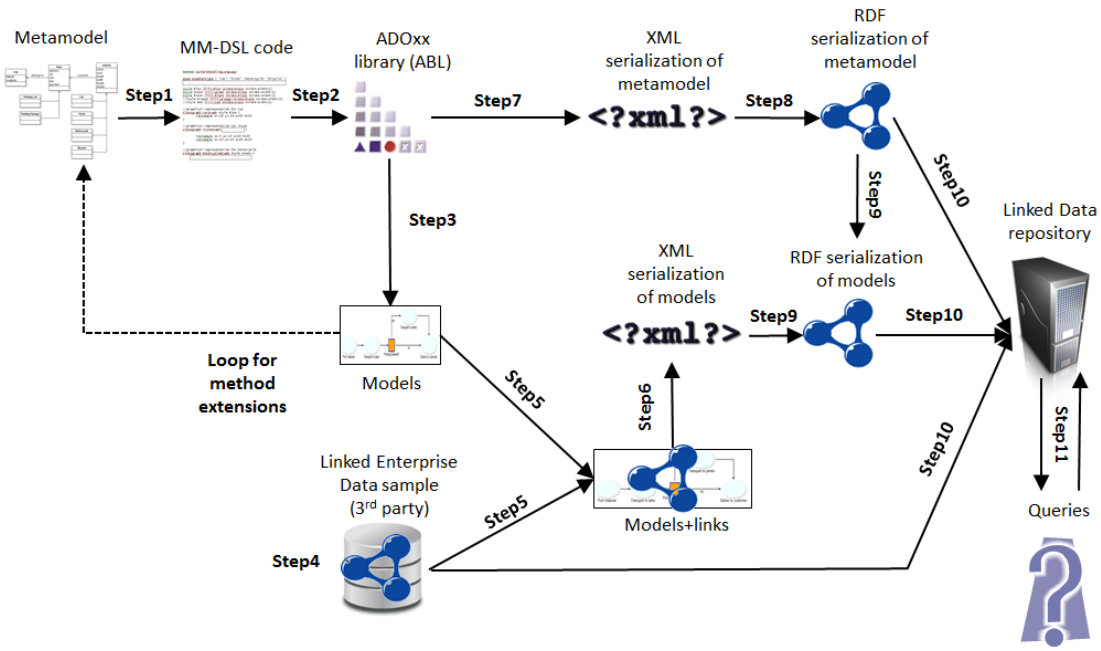


Figure 52: Sequence of Actions to Fulfill the Exercise

The exercise is structured in two parts:

Part 1. Iterative metamodeling (steps 1-3 loop in Figure 52) deals with the evolution of a modeling method due to evolving requirements. Some assumptions will be made about evolving requirements and the method will be built in an iterative loop;

Part 2. Bridging the technology gap (steps 4-11 in Figure 52) deals with the interoperability between the third party enterprise data (park spaces availability) and the model information, to enable queries that make use of both.

Part two is irrelevant for the evaluation. Therefore, the focus is placed on part one. Most of the details related to part two are excluded.

Part 1 – Iterative metamodeling

For the purposes of this exercise, in order to emphasize the benefits of metamodeling for method evolution, it will be assumed that the requirements for the modeling method evolve in three phases

Phase I – Initial requirements

The modeling method must enable users to describe the allocation of parking areas to different cities, as well as the types of vehicles that each parking area accepts.

Phase II – Requirements for enriching the modeling method (extension 1)

The method must enable the modeling of different types of courier tasks, mapped on geographical locations (cities). A courier task is a sequence of transportation actions, but decisions may occur to change the sequence (therefore also the route).

Phase III – Requirements for linking models with enterprise data (extension 2)

A courier must be able to see the list of courier tasks, and for each of them to see the cities involved in the task, as well as the (third party) park space availability information (in order to reserve space in case of availability).

The steps for this part (see Figure 52) are as follows:

1. Create method definition (MM-DSL) code (*.MML file):
   a) Input: the method metamodel (driven by requirements);
   b) Means: Eclipse MM-DSL editor, support from MM-DSL language specification.
2. Create ADOxx-specific method library (*.ABL file):
   a) Input: the MM-DSL code;
   b) Means: Eclipse MM-DSL compiler.
3. Create models:
   a) Input: ADOxx library as input for modeling tool, scenarios as input for models;
   b) Means: import of library in ADOxx to create modeling tool, use modeling tool to create models.

Repeat steps 1-3 to extend the modeling method based on evolving requirements.

Part 2 – Bridging the technology gap (to third party Linked Open Data)

The steps for this part (see Figure 52) are as follows:

4. Create enterprise data sample:
   a. Input: provided sample in a RDF format (TriG);
   b. Means: any text editor;
5. Create model-data links:
   a. Input: Global identifiers (URIs) for modeling objects;
   b. Means: edit URI attributes in modeling objects (or generate modeling objects from a list of available URIs);
6. Serialize models in XML:
   a. Input: models created in modeling tool;
   b. Means: XML export of models from modeling tool;
7. Serialize metamodel in XML:
   a. Input: ADOxx library;
   b. Means: XML export of library from ADOxx;
8. Transform metamodel in RDF format:
   a. Input: XML serialization of metamodel;
   b. Means: RDF Transformer (upper side of the user interface);
9. Transform models in RDF format:
   a. Input: XML serialization of models, RDF representation of metamodel;
   b. Means: RDF Transformer (lower side of the user interface);
10. Upload metamodel, models and enterprise data in Linked Data repository:
    a. Input: RDF representation of models, metamodel and 3rd party enterprise data;
    b. Means: Sesame user interface;
11. Query examples:
    a. Means: Sesame user interface.

Exercise Details: Part I – Iterative Metamodeling

Phase I – Create Method Definition

The initial requirements can be fulfilled with the metamodel depicted in Figure 53. The diagram indicates a single model type (ParkingMap) which allows modeling of Cities, ParkingAreas and four types of Vehicles.

Cities may contain ParkingAreas, ParkingAreas may accept various Vehicle types. For the goals of this exercise, the cardinality of relations is irrelevant and will not be considered. The method name will be CityTransport.

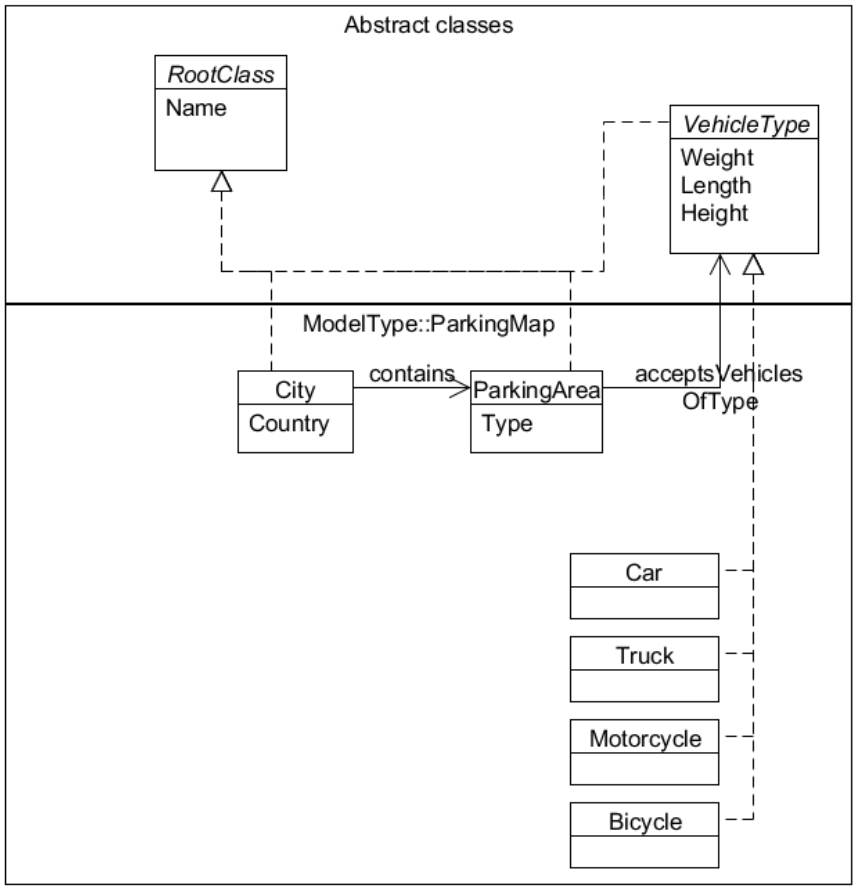


Figure 53: Metamodel for ParkingMap Model Type

The following MM-DSL code describes this version of the method. Comments in the code indicate placeholders where additional code must be added later, to fulfil the requirements in phase II and III of the exercise.

```
method CityTransport

def EmbedPlatformType ADOxx
def EmbedCodeType Notebook

// Additional Code Required for Extensions

// optional extension for displaying object names
```

```
// only use if you define graphs yourself
embed ShowNameGraph <ADOxx:Notebook> {
"ATTR \\\"Name\\\" x:0pt y:9pt w:c"
}

// Additional Code Required for Extensions

style Blue {fill:blue stroke:black stroke-width:1}
style Green {fill:green stroke:black stroke-width:1}
style Black {fill:black stroke:black stroke-width:1}
style Orange {fill:orange stroke:black stroke-width:1}
style Red {fill:red stroke:black stroke-width:1}

classgraph GraphCar style Blue {
   rectangle x=-10 y=-10 w=20 h=20
   insert ShowNameGraph
}

classgraph GraphTruck style Blue {
   rectangle x=-5 y=-15 w=20 h=20
   rectangle x=-10 y=-10 w=20 h=20
   insert ShowNameGraph
}

classgraph GraphMotorcycle style Green {
   circle cx=0 cy=0 r=10
   insert ShowNameGraph
}

classgraph GraphBicylce style Green {
   circle cx=-5 cy=0 r=10
   circle cx=5 cy=0 r=10
   insert ShowNameGraph
}

classgraph GraphParkingArea {
   rectangle x=-20 y=-20 w=40 h=40 style Orange
   polygon points=-20,20 0,20 -20,0 style Red
   polygon points=0,-20 20,-20 20,0 style Red
   text "P" x=-2 y=-2
   insert ShowNameGraph
}

relationgraph GraphAcceptsVehiclesOfType style Black {
   from
   rectangle x=-2 y=-2 w=4 h=4
   middle
   text "accepts vehicle of type" x=0 y=0
   to
   polygon points=-2,2 2,0 -2,-2
}

// Additional Code Required for Extensions

class Root {
   // Additional Code Required for Extensions

}

class City extends Root {
```

```
    attribute Country:string
}

class ParkingArea extends Root symbol GraphParkingArea {
    attribute Type:string
}

class VehicleType extends Root {
    attribute Weight:int access:write
    attribute Lenght:int access:write
    attribute Height:int access:write
}

class Car extends VehicleType symbol GraphCar {}
class Truck extends VehicleType symbol GraphTruck {}
class Motorcycle extends VehicleType symbol GraphMotorcycle {}
class Bicycle extends VehicleType symbol GraphBicylce {}

// Additional Code Required for Extensions

relation acceptsVehiclesOfType symbol GraphAcceptsVehiclesOfType from
ParkingArea to VehicleType {}
relation contains from City to ParkingArea {}

// Additional Code Required for Extensions

modeltype ParkingMap {
    classes City ParkingArea Car Truck Motorcycle Bicycle
    relations acceptsVehiclesOfType contains
    modes none
}

// Additional Code Required for Extensions
```

The code is syntactically and semantically correct and can be compiled to a modeling tool. In this exercise the ADOxx platform has been used to execute MM-DSL code.

Phase II – first extension

It is assumed that the requirements evolve by adding those indicated for Phase2 (the addition of task modeling capabilities). The extended requirements can be fulfilled by the metamodel depicted in Figure 54. The extensions to the metamodel are as follows:

A new model type is added (CourierTask) to depict the sequence of transportation Actions necessary for a type of task. Besides actions, Decisions may also be used to split the sequence. Next is the name of the relation which links subsequent nodes in the task (it has a Condition attribute to describe, after a decision, what determines each path choice). Actions can be mapped on Cities where they need to be performed (suggesting that a courier would need to find parking in those cities in order to perform the task).

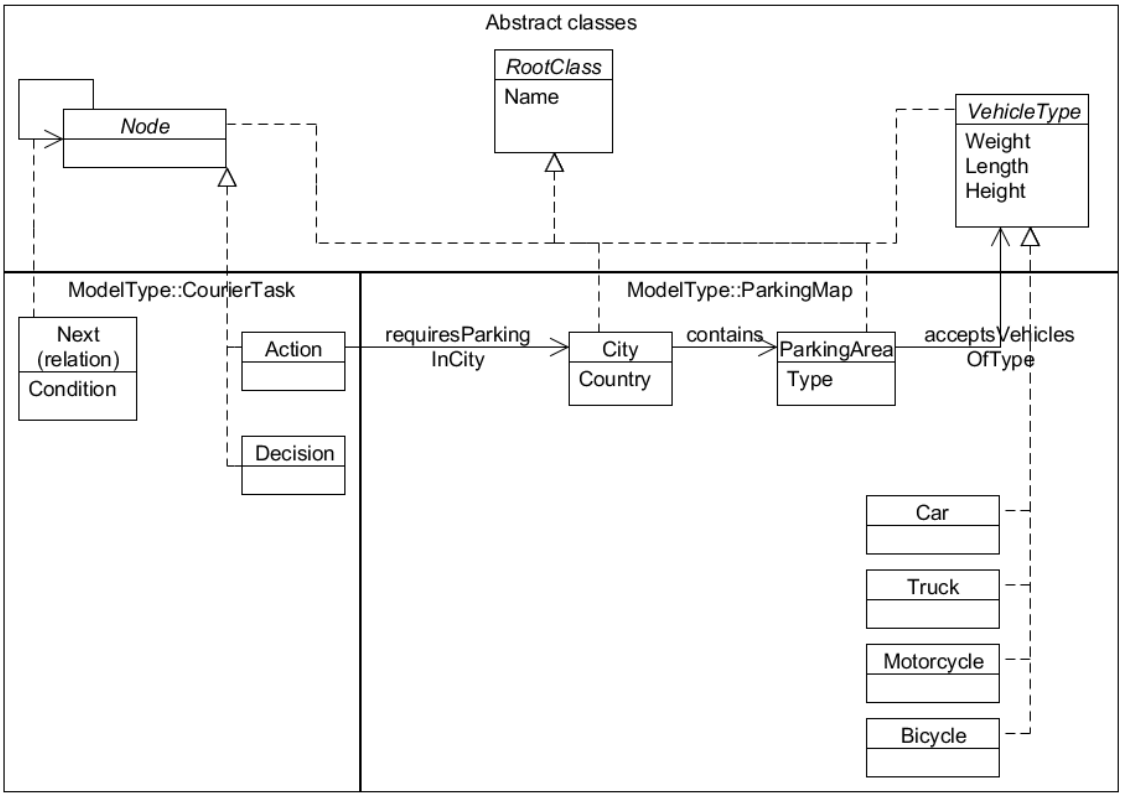


Figure 54: Extended Metamodel Containing the new Model Type CourierTask

Table XIX indicates some code snippets that must be added in some of the placeholders reserved during the first iteration. It is left as an exercise for the students to identify, based on the language specification, where exactly should each snippet be inserted.

Table XIX: Code Extensions for Phase II

1	**relationgraph** GraphNext **style** Red {
	from
	middle
	insert DynConditionGraph
	to
	polygon points=-2,2 2,0 -2,-2
	}

2	**relation** next **symbol** GraphNext **from** Node **to** Node {
	attribute Condition:**string**
	}

3	**modeltype** CourierTask {
	classes Decision Action
	relations next
	modes none
	}

4	**embed** DynConditionGraph <ADOxx:Notebook> {

	`"ATTR \\\"Condition\\\""` `}`
5	`class Node extends Root {}`
6	`class Decision extends Node {}`

7	`class Action extends Node {`
	`  reference requiresParkingInCity -> modeltype ParkingMap class City`
	`}`

After the code insertion is performed, the code needs to be compiled again.

Phase III – Second Extension

Now the requirements of Phase3 are added, for bridging the technological gap between the modeling environment and the third party Linked Open Data available. The new metamodel is shown in Figure 55. Its only addition is the URI attribute to the Root class, thus to be inherited by all modeling elements. This URI will act as a global identifier, to allow the indication that a certain element of a model (e.g. a ParkingArea) is the same thing as some third party resource (e.g. a ParkingArea record from the third party space availability data).

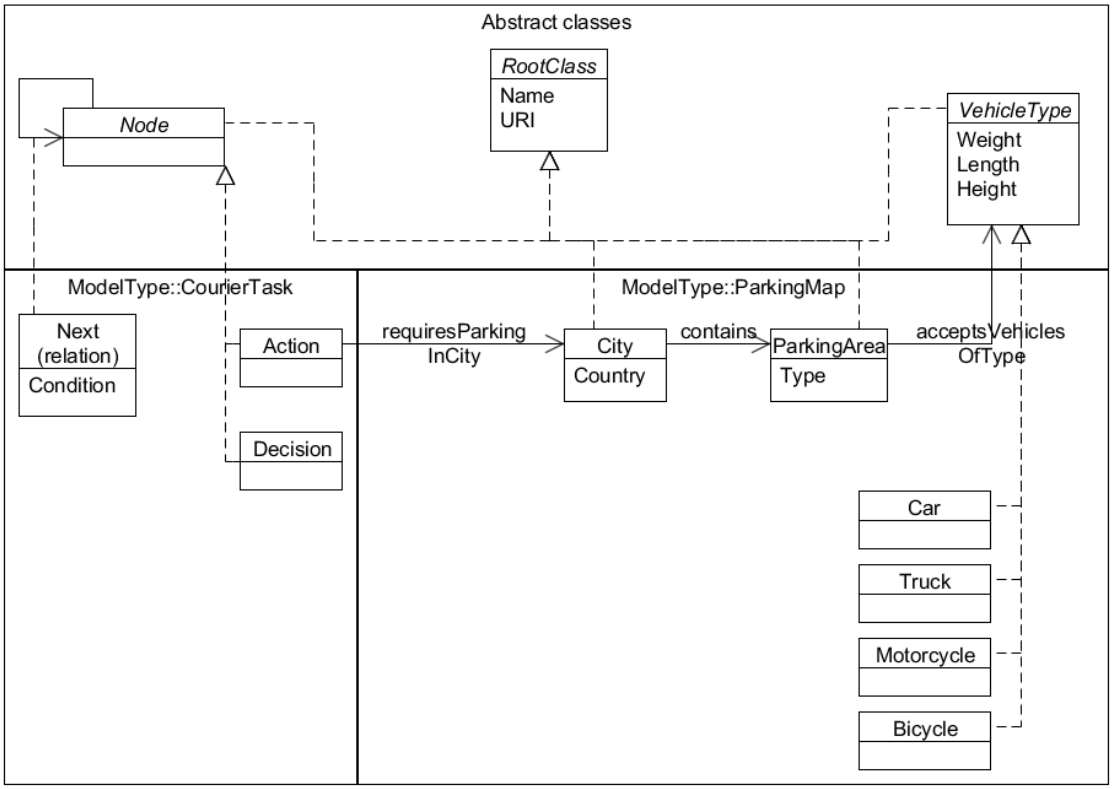


Figure 55: Extended Metamodel Containing a new URI attribute in the Root Class

In addition to the metamodel change, functionality will be added in the form of embedded AdoScript code. This code will facilitate the model-to-data linking by importing

URIs from a list given by the user (it is not mandatory to use, the URIs can also be typed in one by one, in every model element).

Table XX indicates some code snippets that must be added in some of the placeholders reserved during the first iteration. It is left as an exercise for the students to identify, based on the language specification, where exactly should each snippet be inserted.

Table XX: Code Extensions for Phase III

8	**embed** URIImportItem <ADOxx:AdoScript> {
	"ITEM \\\"URI import\\\" modeling:\\\"~AdoScripts\\\" pos2:1"
	}

9	**attribute** URI:**string**
10	**def EmbedCodeType** AdoScript

11	**algorithm** URIImport {
	  // embed the AdoScript code
	  insert URIImportItem
	  insert URIImportAlgorithm
	}

12	**embed** URIImportAlgorithm <ADOxx:AdoScript> {
	// always put \\\ before special characters and quotations (e.g. \\\\n, \\\")
	"
	SETL cls_name:(\\\"ParkingArea\\\")
	SETL mod_type_name:(\\\"ParkingMap\\\")
	SETL attr_uri_name:(\\\"URI\\\")
	SETL obj_cnt:(0)
	
	CC \\\"Modeling\\\" GET_ACT_MODEL
	SETL pm_id:(modelid)
	IF (pm_id = -1) {
	  CC \\\"AdoScript\\\" ERRORBOX (\\\"The selected model could not be determined.\\\\nMake sure a model of type \\\" + mod_type_name + \\\" is opened and selected.\\\")
	  EXIT
	}
	
	CC \\\"Core\\\" GET_MODEL_MODELTYPE modelid:(pm_id)
	IF (modeltype != mod_type_name) {
	  CC \\\"AdoScript\\\" ERRORBOX (\\\"The selected model is of the wrong type.\\\\nMake sure a model of type \\\" + mod_type_name + \\\" is opened and selected.\\\")
	  EXIT
	}

```
CC \\\"AdoScript\\\" EDITBOX text:(\\\"\\\") title:(\\\"Enter URIs\\\")
oktext:(\\\"Create\\\")
IF (endbutton != \\\"ok\\\") {
  EXIT
}
SETL uris:(text)

CC \\\"Core\\\" GET_CLASS_ID classname:(cls_name)
SETL cls_id:(classid)

CC \\\"Core\\\" GET_ATTR_ID classid:(cls_id) attrname:(attr_uri_name)
SETL attr_uri_id:(attrid)

FOR uri in:(uris) sep:(\\\"\\\\n\\\") {
  IF (LEN uri > 1) {
    CC  \\\"Core\\\"  CREATE_OBJ  modelid:(pm_id)  classid:(cls_id)  ob-
jname:(cls_name + \\\"-\\\" + STR obj_cnt)
    SETL obj_id:(objid)
    CC  \\\"Core\\\"  SET_ATTR_VAL  objid:(obj_id)  attrid:(attr_uri_id)
val:(uri)
    CC \\\"Modeling\\\" SET_OBJ_POS objid:(obj_id) x:(2cm) y:(1cm+CM (1.5
* obj_cnt))
    SETL obj_cnt:(obj_cnt + 1)
  }
}
"
}
```

After the code insertion is performed, the code needs to be compiled once more. This is the final compilation for this exercise, which continues in part 2, by using the created modeling tool, models created by it, and RDF to bridge the technological gap to third party Linked Open Data. As already mentioned, the details of part 2 are not included because they are not relevant to the current evaluation of MM-DSL IDE. Full exercise can be found in [65].

Appendix F: Standard SUS Questionnaire

	Strongly disagree				Strongly agree
1. I think that I would like to use this system frequently	1	2	3	4	5
2. I found the system unnecessarily complex	1	2	3	4	5
3. I thought the system was easy to use	1	2	3	4	5
4. I think that I would need the support of a technical person to be able to use this system	1	2	3	4	5
5. I found the various functions in this system were well integrated	1	2	3	4	5
6. I thought there was too much inconsistency in this system	1	2	3	4	5
7. I would imagine that most people would learn to use this system very quickly	1	2	3	4	5
8. I found the system very cumbersome to use	1	2	3	4	5
9. I felt very confident using the system	1	2	3	4	5
10. I needed to learn a lot of things before I could get going with this system	1	2	3	4	5

Appendix G: MM-DSL IDE Evaluation Results Overview

Question / Participant	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
1	4	5	4	3	3	3	3	2	4	4	3	3	1	3	2	4	5	4	2	3	2	4	4	2	3	4	5	3	5	3	3
2	3	5	1	2	4	2	1	3	3	3	3	2	5	2	4	2	1	5	4	3	2	3	4	3	1	3	3	1	2	4	4
3	5	5	5	4	3	4	3	2	3	3	4	3	2	3	3	5	1	2	2	3	4	4	1	1	2	3	2	5	3	2	2
4	2	1	3	2	2	4	3	3	5	4	2	2	4	2	2	1	1	3	5	3	2	4	4	3	1	2	4	1	5	2	2
5	3	5	5	3	4	4	4	2	5	4	3	4	2	3	3	4	5	2	2	3	4	5	3	3	2	4	4	4	3	4	4
6	1	1	1	1	1	2	1	2	4	2	3	3	2	2	2	2	1	1	2	3	2	4	4	3	2	3	2	2	1	2	1
7	4	5	4	4	5	2	4	3	4	5	2	5	1	3	1	5	5	1	2	2	3	4	2	2	4	4	3	4	3	2	2
8	3	1	1	2	4	3	2	3	4	3	4	3	3	2	3	3	1	2	5	3	4	5	5	4	3	3	3	2	2	3	3
9	5	4	4	4	5	3	3	1	5	4	3	3	1	2	2	4	5	2	2	3	4	5	3	1	2	3	3	4	3	3	3
10	2	3	3	4	3	2	1	3	3	5	4	3	5	1	5	4	3	4	1	3	2	5	2	2	1	3	5	1	5	2	2
SUS Score	75.00	82.50	82.50	67.50	65.00	57.50	72.50	40.00	55.00	50.00	50.00	62.50	20.00	62.50	37.50	70.00	95.00	42.50	32.50	47.50	65.00	62.50	35.00	35.00	67.50	60.00	50.00	82.50	55.00	52.50	55.00
Average SUS Score: 57.66	Median SUS Score: 57.50											Best SUS Score: 95							Worst SUS Score: 20						Standard Deviation: 16.7						

Bibliography

[1] “Research in Programming Languages | Tagide.” [Online]. Available: http://tagide.com/blog/2012/03/research-in-programming-languages/. [Accessed: 03-Aug-2012].

[2] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, “The Evolution of an Extension Language: A History of Lua,” in *IN PROCEEDINGS OF V BRAZILIAN SYMPOSIUM ON PROGRAMMING LANGUAGES*, 2001, pp. 14–28.

[3] T. Clark, P. Sammut, and J. Willans, “Superlanguages: developing languages and applications with XMF.,” 2008. [Online]. Available: http://itcentre.tvu.ac.uk/~clark/docs/Superlanguages.pdf. [Accessed: 15-Jan-2013].

[4] M. Fowler and R. Parsons, *Domain Specific Languages*, 1st ed. Addison-Wesley Longman, Amsterdam, 2010.

[5] F. Jouault and J. Bézivin, “KM3: A DSL for Metamodel Specification,” in *Formal Methods for Open Object-Based Distributed Systems*, R. Gorrieri and H. Wehrheim, Eds. Springer Berlin Heidelberg, 2006, pp. 171–185.

[6] T. Clark and L. Tratt, “Language factories,” in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, New York, NY, USA, 2009, pp. 949–955.

[7] S. Gunther, M. Haupt, and M. Splieth, “Agile Engineering of Internal Domain-Specific Languages with Dynamic Programming Languages,” in *2010 Fifth International Conference on Software Engineering Advances (ICSEA)*, 2010, pp. 162–168.

[8] A. van Deursen, P. Klint, and J. Visser, “Domain-specific languages: an annotated bibliography,” *SIGPLAN Not*, vol. 35, no. 6, pp. 26–36, Jun. 2000.

[9] M. Mernik, J. Heering, and A. M. Sloane, “When and how to develop domain-specific languages,” *ACM Comput. Surv.*, vol. 37, pp. 316–344, Dec. 2005.

[10] D. Karagiannis and H. Kühn, “Metamodelling Platforms,” in *E-Commerce and Web Technologies*, vol. 2455, K. Bauknecht, A. M. Tjoa, and G. Quirchmayr, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 182–182.

[11] “Systems and software engineering – Vocabulary,” *ISOIECIEEE 247652010E*, pp. 1–418, 2010.

[12] T. Kühne, “Matters of (Meta-) Modeling,” *Softw. Syst. Model.*, vol. 5, no. 4, pp. 369–385, Dec. 2006.

[13] “MOF 2.4.1.” [Online]. Available: http://www.omg.org/spec/MOF/2.4.1/. [Accessed: 05-Nov-2013].

[14] E. Guerra, J. de Lara, D. Kolovos, and R. Paige, “A Visual Specification Language for Model-to-Model Transformations,” in *2010 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2010, pp. 119–126.

[15] A. J. Perlis, “Special Feature: Epigrams on Programming,” *SIGPLAN Not*, vol. 17, no. 9, pp. 7–13, Sep. 1982.

[16] “UML 2.4.1.” [Online]. Available: http://www.omg.org/spec/UML/2.4.1/. [Accessed: 20-Nov-2013].

[17] “Unified Modeling Language,” *Wikipedia, the free encyclopedia*. 22-Aug-2014.

[18] “Introduction to the Diagrams of UML 2.X.” [Online]. Available: http://www.agilemodeling.com/essays/umlDiagrams.htm. [Accessed: 20-Nov-2013].

[19] "UML graphical notation overview, UML diagram examples, tutorials and reference." [Online]. Available: http://www.uml-diagrams.org/. [Accessed: 20-Nov-2013].

[20] S. Kelly, *Domain-specific modeling: enabling full code generation*. Hoboken, N.J: Wiley-Interscience : IEEE Computer Society, 2008.

[21] M. Fowler, "Language Workbenches: The Killer-App for Domain Specific Languages?" [Online]. Available: http://www.martinfowler.com/articles/languageWorkbench.html. [Accessed: 26-Mar-2012].

[22] J.-P. Tolvanen and S. Kelly, "MetaEdit+: defining and using integrated domain-specific modeling languages," in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, New York, NY, USA, 2009, pp. 819–820.

[23] N. Chomsky, *Syntactic structures*, 2nd ed. Berlin ; New York: Mouton de Gruyter, 2002.

[24] N. Chomsky, "On certain formal properties of grammars," *Inf. Control*, vol. 2, no. 2, pp. 137–167, Jun. 1959.

[25] N. Chomsky, "Three models for the description of language," *IRE Trans. Inf. Theory*, vol. 2, no. 3, pp. 113–124, 1956.

[26] T. Jiang, M. Li, B. Ravikumar, and K. W. Regan, "Algorithms and Theory of Computation Handbook," M. J. Atallah and M. Blanton, Eds. Chapman & Hall/CRC, 2010, pp. 20–20.

[27] D. Grune, *Modern compiler design*. New York: Springer, 2012.

[28] R. Cole, "Converting CFGs to CNF (Chomsky Normal Form)." [Online]. Available: http://cs.nyu.edu/courses/fall07/V22.0453-001/cnf.pdf. [Accessed: 27-Nov-2013].

[29] W. Goddard, "Chomsky Normal Form." [Online]. Available: http://people.cs.clemson.edu/~goddard/texts/theoryOfComputation/9a.pdf. [Accessed: 27-Nov-2013].

[30] W. Goddard, *Introducing the theory of computation*. Sudbury, Mass: Jones and Bartlett Publishers, 2008.

[31] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger, "Report on the Algorithmic Language ALGOL 60," *Commun ACM*, vol. 3, no. 5, pp. 299–314, May 1960.

[32] "ISO/IEC 14977:1996 - Information technology -- Syntactic metalanguage -- Extended BNF." [Online]. Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=26153. [Accessed: 26-Nov-2013].

[33] "Extensible Markup Language (XML) 1.0 (Fifth Edition)." [Online]. Available: http://www.w3.org/TR/REC-xml/#sec-notation. [Accessed: 27-Nov-2013].

[34] D. E. Knuth, "Semantics of context-free languages," *Math. Syst. Theory*, vol. 2, no. 2, pp. 127–145, Jun. 1968.

[35] D. E. Knuth, "The genesis of attribute grammars," in *Attribute Grammars and their Applications*, P. Deransart and M. Jourdan, Eds. Springer Berlin Heidelberg, 1990, pp. 1–12.

[36] J. Paakki, "Attribute Grammar Paradigms&Mdash;a High-level Methodology in Language Implementation," *ACM Comput Surv*, vol. 27, no. 2, pp. 196–255, Jun. 1995.

[37] K. Slonneger and B. Kurtz, *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[38] F. Essalmi and L. J. B. Ayed, “Graphical UML View from Extended Backus-Naur Form Grammars,” in *Sixth International Conference on Advanced Learning Technologies, 2006*, 2006, pp. 544–546.

[39] Y. Xia and M. Glinz, “Rigorous EBNF-based definition for a graphic modeling language,” in *Software Engineering Conference, 2003. Tenth Asia-Pacific*, 2003, pp. 186– 196.

[40] “OMG’s MetaObject Facility (MOF) Home Page.” [Online]. Available: http://www.omg.org/mof/. [Accessed: 15-Jan-2013].

[41] “School and Training Plans Solution | ConceptDraw.com.” [Online]. Available: http://www.conceptdraw.com/solution-park/building-school-training-plans. [Accessed: 07-Sep-2014].

[42] D. H. Lorenz and B. Rosenan, “Cedalion: A Language for Language Oriented Programming,” in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, New York, NY, USA, 2011, pp. 733–752.

[43] M. E. Lesk and E. Schmidt, *Lex: A lexical analyzer generator*. Bell Laboratories Murray Hill, NJ, 1975.

[44] S. C. Johnson, “Yacc: Yet Another Compiler-Compiler,” 1978.

[45] T. Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[46] “ANTLR.” [Online]. Available: http://www.antlr.org/. [Accessed: 14-Sep-2014].

[47] S. Efftinge and M. Völter, “oAW xText: A framework for textual DSLs,” in *Workshop on Modeling Symposium at Eclipse Summit*, 2006, vol. 32, p. 118.

[48] “Xtext - Community Web Page.” [Online]. Available: http://www.eclipse.org/Xtext/community.html. [Accessed: 27-Sep-2014].

[49] M. Voelter and V. Pech, “Language modularity with the MPS language workbench,” in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 1449–1450.

[50] “mbeddr - engineering the future of embedded software.” [Online]. Available: http://mbeddr.com/index.html. [Accessed: 27-Sep-2014].

[51] “Irony - .NET Language Implementation Kit. - Home.” [Online]. Available: http://irony.codeplex.com/. [Accessed: 15-Jan-2013].

[52] “MetaEdit+ Workbench User’s Guide.” [Online]. Available: http://www.metacase.com/support/50/manuals/mwb/Mw.html. [Accessed: 02-Nov-2014].

[53] “GME: Generic Modeling Environment.” [Online]. Available: http://www.isis.vanderbilt.edu/Projects/gme/. [Accessed: 19-Aug-2013].

[54] Z. Molnár, D. Balasubramanian, and A. Lédeczi, “An Introduction to the Generic Modeling Environment,” in *TOOLS Europe 2007 Workshop on Model-Driven Development Tool Implementers Forum*, Zurich, Switzerland, 2007.

[55] “GME - ISIS Forge Development Community.” [Online]. Available: https://forge.isis.vanderbilt.edu/gme/. [Accessed: 09-Nov-2014].

[56] “Eclipse Modeling - EMF.” [Online]. Available: http://www.eclipse.org/modeling/emf/. [Accessed: 19-Aug-2013].

[57] "Graphical Editing Framework." [Online]. Available: http://www.eclipse.org/gef/. [Accessed: 09-Nov-2014].

[58] "Graphical Modeling Framework." [Online]. Available: http://www.eclipse.org/modeling/gmp/. [Accessed: 09-Nov-2014].

[59] S. Kelly, *Towards a Comprehensive MetaCASE and CAME Environment: Conceptual, Architectural, Functional and Usability Advances in MetaEdit+*. University of Jyvaskyla, 1997.

[60] J. L. Wynekoop and N. L. Russo, "Studying system development methodologies: an examination of research methods," *Inf. Syst. J.*, vol. 7, no. 1, pp. 47–65, 1997.

[61] J. Nunamaker, J.F. and M. Chen, "Systems development in information systems research," in *, Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences, 1990*, 1990, vol. iii, pp. 631–640 vol.3.

[62] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A design science research methodology for information systems research," *J. Manag. Inf. Syst.*, vol. 24, no. 3, pp. 45–77, 2007.

[63] A. Hevner and S. Chatterjee, *Design science research in information systems*. Springer, 2010.

[64] S. Espana, N. Condori-Fernandez, A. Gonzalez, and O. Pastor, "Evaluating the Completeness and Granularity of Functional Requirements Specifications: A Controlled Experiment," in *Requirements Engineering Conference, 2009. RE '09. 17th IEEE International*, 2009, pp. 161 –170.

[65] "OMILAB.org." [Online]. Available: http://www.omilab.org/. [Accessed: 19-Aug-2013].

[66] M. Glinz, "On Non-Functional Requirements," in *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, 2007, pp. 21 –26.

[67] M. Völter and E. Visser, "Language extension and composition with language workbenches," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, New York, NY, USA, 2010, pp. 301–304.

[68] H. Kühn and M. Murzek, "Interoperability Issues in Metamodelling Platforms," in *Proceedings of the 1st International Conference on Interoperability of Enterprise Software and Applications*, Geneva, 2006, pp. 215–226.

[69] K. Sledziewski, B. Bordbar, and R. Anane, "A DSL-Based Approach to Software Development and Deployment on Cloud," in *2010 24th IEEE International Conference on Advanced Information Networking and Applications (AINA)*, 2010, pp. 414–421.

[70] D. Karagiannis and N. Visic, "Very Lightweight Modeling Language (VLML): A Metamodel-based Implementation," in *Modelling and quality in requirements engineering*, N. Seyff and A. Koziolek, Eds. Münster: Monsenstein und Vannerdat, 2012.

[71] M. Glinz, "Very Lightweight Requirements Modeling," in *Requirements Engineering Conference (RE), 2010 18th IEEE International*, 2010, pp. 385–386.

[72] M. Glinz, "Problems and Deficiencies of UML as a Requirements Specification Language," in *Proceedings of the 10th International Workshop on Software Specification and Design*, Washington, DC, USA, 2000, p. 11–.

[73] M. Glinz, S. Berner, and S. Joos, "Object-oriented modeling with ADORA," *Inf Syst*, vol. 27, no. 6, pp. 425–444, Sep. 2002.

[74] M. Glinz and D. Wüest, "A Vision of an Ultralightweight Requirements Modeling Language. TR IFI-2010.06," University of Zurich.

[75] H.-G. Fill, “On the Conceptualization of a Modeling Language for Semantic Model Annotations,” in *Advanced Information Systems Engineering Workshops*, vol. 83, C. Salinesi and O. Pastor, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 134–148.

[76] D. Karagiannis, H.-G. Fill, S. Zivkovic, and W. Utz, “From Model Editors to Modelling Tools: Operationalizing Modelling Methods with ADOxx,” *MODELS 2012, Innsbruck, Austria*. [Online]. Available: http://models2012.info/index.php?option=com_content&view=article&id=14&Itemid=19#T2. [Accessed: 06-Nov-2012].

[77] D. Karagiannis and N. Visic, “Next Generation of Modelling Platforms,” in *Perspectives in Business Informatics Research*, vol. 90, J. Grabis and M. Kirikova, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 19–28.

[78] J. Steiff, *The Complete Idiot’s Guide To Independent Filmmaking*. Alpha Books, 2005.

[79] N. Visic and D. Karagiannis, “Developing Conceptual Modeling Tools Using a DSL,” in *Knowledge Science, Engineering and Management*, R. Buchmann, C. V. Kifor, and J. Yu, Eds. Springer International Publishing, 2014, pp. 162–173.

[80] M. Jarke, R. Gallersdörfer, M. A. Jeusfeld, M. Staudt, and S. Eherer, “ConceptBase — A deductive object base for meta data management,” *J. Intell. Inf. Syst.*, vol. 4, no. 2, pp. 167–192, Mar. 1995.

[81] H. Kern, A. Hummel, and S. Kühne, “Towards a comparative analysis of meta-metamodels,” in *Proceedings of the compilation of the co-located workshops on DSM’11, TMC’11, AGERE!’11, AOOPES’11, NEAT’11, & VMIL’11*, New York, NY, USA, 2011, pp. 7–12.

[82] E. Yu, P. Giorgini, N. Maiden, and J. Mylopoulos, *Social Modeling for Requirements Engineering*. The MIT Press, 2011.

[83] “HUTN.” [Online]. Available: http://www.omg.org/spec/HUTN/. [Accessed: 29-Aug-2013].

[84] “Emfatic Language Reference.” [Online]. Available: http://www.eclipse.org/epsilon/doc/articles/emfatic/. [Accessed: 29-Aug-2013].

[85] “Epsilon.” [Online]. Available: http://www.eclipse.org/epsilon/. [Accessed: 29-Aug-2013].

[86] T. Horn, “Model Querying with FunnyQT,” in *Theory and Practice of Model Transformations*, K. Duddy and G. Kappel, Eds. Springer Berlin Heidelberg, 2013, pp. 56–57.

[87] J. Ebert and T. Horn, “GReTL: an extensible, operational, graph-based transformation language,” *Softw. Syst. Model.*, pp. 1–21.

[88] “Graphiti.” [Online]. Available: http://www.eclipse.org/graphiti/. [Accessed: 21-Mar-2014].

[89] “XModeler.” [Online]. Available: http://www.eis.mdx.ac.uk/staffpages/tonyclark/Software/XModeler.html. [Accessed: 21-Mar-2014].

[90] M. P. Ward, “Language Oriented Programming,” *Software—Concepts Tools*, vol. 15, pp. 147–161, 1995.

[91] T. Clark, P. Sammut, and J. Willans, “Applied metamodelling: a foundation for language driven development.,” 2008. [Online]. Available: http://eprints.mdx.ac.uk/6060/. [Accessed: 23-Sep-2011].

[92] D. S. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley, 2003.

[93] J. Greenfield and K. Short, *Software factories: assembling applications with patterns, models, frameworks, and tools*. Wiley Pub., 2004.

[94] D. Ameller, X. Franch, and J. Cabot, "Dealing with Non-Functional Requirements in Model-Driven Development," in *Requirements Engineering Conference (RE), 2010 18th IEEE International*, 2010, pp. 189 –198.

[95] M. Eysholdt and H. Behrens, "Xtext: implement your language faster than the quick and dirty way," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, New York, NY, USA, 2010, pp. 307–309.

[96] "Microsoft Visual Studio 2010 Visualization & Modeling SDK," *Microsoft Download Center*. [Online]. Available: https://www.microsoft.com/en-us/download/details.aspx?id=23025. [Accessed: 16-Jan-2013].

[97] "OMiLAB: The Modelling Methods Booklet." [Online]. Available: http://www.omilab.org/web/guest/booklet. [Accessed: 25-Dec-2014].

[98] L. McIver and D. Conway, "Seven deadly sins of introductory programming language design," in *Software Engineering: Education and Practice, 1996. Proceedings. International Conference*, 1996, pp. 309–316.

[99] B. Stroustrup, *The C++ Programming Language*. 1995.

[100] B. Stroustrup, *The design and evolution of C++*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1994.

[101] "The SVG path element - SVG Tutorial." [Online]. Available: http://tutorials.jenkov.com/svg/path-element.html. [Accessed: 07-May-2014].

[102] N. Visic, "MM-DSL Source Code Repository," *GitHub*. [Online]. Available: https://github.com/niksavis/mm-dsl. [Accessed: 13-Jan-2015].

[103] "Evaluating Programming Languages." [Online]. Available: http://courses.cs.washington.edu/courses/cse341/02sp/concepts/evaluating-languages.html. [Accessed: 19-Mar-2014].

[104] J. Brooke, "SUS-A quick and dirty usability scale," *Usability Eval. Ind.*, vol. 189, pp. 189–194, 1996.

[105] A. Bangor, P. Kortum, and J. Miller, "Determining what individual SUS scores mean: Adding an adjective rating scale," *J. Usability Stud.*, vol. 4, no. 3, pp. 114–123, 2009.

Index of Terms