Academy of Montpellier
Montpellier II University
Science and Technique of Languedoc

INTERNSHIP MEMORY OF MASTER M2

Accomplished at LGI2P school of mines of Ales

Speciality : **AIGLE**

Grammar and graphical concrete syntaxes generator assistant for domain specific modeling languages

by **Blazo NASTOV**

Directed by **François PFISTER** and **Vincent CHAPURLAT**

CONTENTS

LIST OF FIGURES

ABSTRACT

Résumé

Les DSMLs graphiques (langages de modélisation métier) sont une alternative aux langages de modélisation d'usage général comme par exemple UML ou SysML. L'objectif de ces langages est de décrire des modèles avec des concepts et des relations spécifiques à un domaine. La définition de tels langages se compose d'une syntaxe abstraite, d'une syntaxe concrète graphique, et d'une correspondance entre les éléments de chacune des deux syntaxes. Cette description forme la grammaire du langage. Le processus de la définition de la grammaire comprend deux phases : la définition de la syntaxe abstraite (1) et la définition d'une syntaxe concrète (2). La syntaxe abstraite est représentée par un méta-modèle conforme à MOF 1.4, et la syntaxe concrète est dérivée de la syntaxe abstraite par une transformation en direction d'un méta-modèle générique de syntaxe concrète graphique (Diagraph). Concevoir la syntaxe concrète revient à définir cette transformation ; cette dernière est inscrite sur la syntaxe abstraite au moyen de méta-données (annotations). Afin de faciliter la conception de la syntaxe concrète, nous proposons d'automatiser partiellement la génération de la transformation, en opérant une reconnaissance de motifs de conception au sein du méta-modèle représentant la syntaxe abstraite.

Abstract

Graphical DSMLs (Domain Specific Modeling Languages) are an alternative to general purpose modeling languages e.g. UML or SysML. The objective of this languages is to describe models with concepts and relations specific to a domain. Defining such languages consist of defining an abstract syntax, a graphical concrete syntax and a correspondence between the elements of each syntax. Such description form the language grammar. The process of defining a grammar is composed of two phases : abstract syntax definition (1) and concrete syntax definition (2). The abstract syntax is represented by a MOF 1.4 conforming meta-model and the concrete one is derived from the abstract syntax by a transformation targeting a generic meta-model of the graphical concrete syntax (Diagraph). Designing the concrete syntax implies the definition of this transformation, the latter is registered into the abstract syntax trough meta-data (annotations). In order to facilitate the concrete syntax conception, we aim to semi-automatize the generation of this transformation, by operating a pattern mining process on the abstract syntax meta-model.

INTRODUCTION

Experts in the field of system modeling use notation languages to provide a model of the system they intend to construct. Such languages create concepts (with a lack of precise semantics) for expressing structural and behavioral views of the system under study. These types of languages are called general purpose modeling languages and their generic expressive power sometimes cannot be used to provide concepts for specific domains. Therefore, experts have to create an extended version of a general purpose language, specifically designed for the system under study, which they intend to model. Such languages are called domain specific modeling languages and their expressive power for the specific system under study is superior to the one of general purpose language. The creation of such languages is divided into two key phases. The first phase consists of defining an abstract syntax. The second phase consists of defining a concrete syntax that is conforming to the previously defined abstract syntax, representing the language statements by a certain form that may be textual or graphical. The conformance of the concrete to the abstract syntax implies a relationship between these two syntaxes usually represented by a graph. Putting all together, i.e. the concrete syntax, the abstract syntax and the mapping relationship, lift up an entity called language description, which is fundamental to software languages. Indeed, it is the language description that separated the artificial nature of software languages from natural languages.

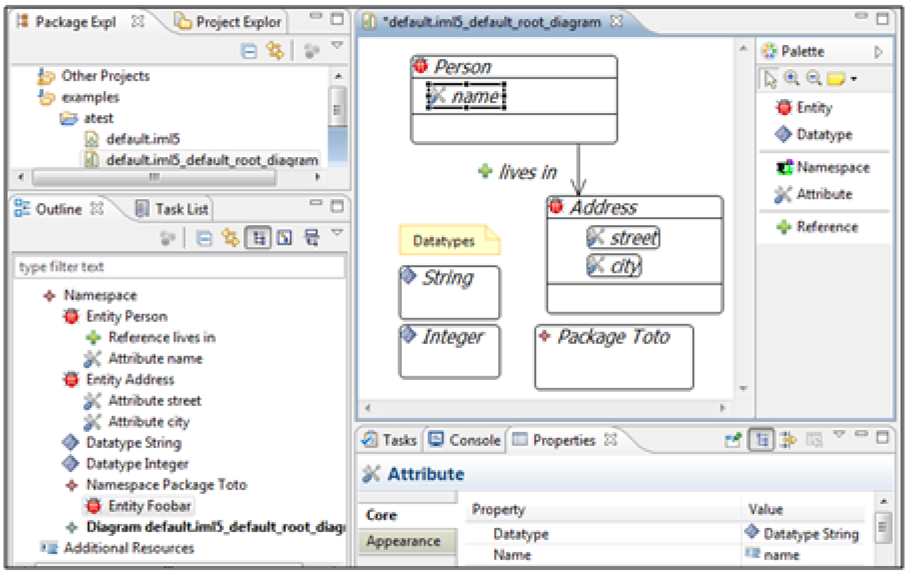
Since we are dealing with modeling software language, our context of study is within the Model Driven Engineering (MDE) which aims to reduce complexity and improve productivity. Among this field of study, the Model Driven Architecture (MDA) is an important approach introduced in 2000 by the Object Management Group, classifying models into different independency levels allowing the possibility of crossing from one to another level by model transformation. The process of designing and implementing a graphical notation for software languages demands important expertise, both in its semiotic and cognitive concerns, and in its technical and operational aspects. Thus, the expertise is used, firstly in the process of implementing the abstract syntax and secondly in the process of implementing the concrete syntax, both represented by meta-models. The meta-model of the concrete syntax is mapped to the one of the abstract syntax. Furthermore "supplementary information" should be added to the concrete syntax meta-model, for the purpose of graphical representation of the language statements, i.e. how these statements are going to be represented graphically, for the end user. My focus aims at automation of grammars and graphical concrete syntaxes for a domain specific modeling language. In other words, a part of the "supplementary information" that should be added to the concrete meta-model follows certain patterns that can be calculated. Hence, I am intending to identify those patterns and to use them in a process of pattern recognition, calculating the maximum possible information, in order to infer a part of the target language grammar i.e. a part of the "supplementary information", from the one side, and from the other, I intend to automatize this process.

Outline of the paper. This paper should be a state of the art for my future internship and an alpha version of my final report that should follows multiple modifications during my internship period. In Chapter 1, I will describe an approach to patterns and pattern recognition in the field of DSMLs. Afterward in Chapter 2, I will present my bibliographic work, that I have done during this month, followed by a synthesis (Chapter 3) that summarizes the strengths and weaknesses of each concept. I will finish with conclusion and perspectives presenting the work proposed during the internship period.

CHAPTER 1 NOTATION PATTERNS, FIRST APPROACH

1.1 Context

In the field of domain specific modeling languages, a language description consists of three fundamental concepts: the abstract syntax, the concrete syntax and the mapping between these two. The process of creating a DSML is divided into two phases. It begins with defining the abstract syntax with a meta-model. Figure 1.2 is an example of a such meta-model, defining a language called "UML reinvented". The second phase consists of defining the concrete syntax or a conforming meta-model to the abstract syntax meta-model and enriching this meta-model with supplementary information i.e. grammar. This is the phase that determines how the language statements i.e. the graphic elements (because we are focused in the field of graphic DSMLs) will be defined in a 2D space. In other words, the graphical representation of these elements, which elements is in relation to other elements, the type of the relationship, etc. Afterwards, we are able to define models expressed in the DSML we just defined, as shown by Figure 1.1.



Property	Value
Datatype	Datatype String
Name	name

Figure 1.1: UML reinvented - a model [Pfi13].

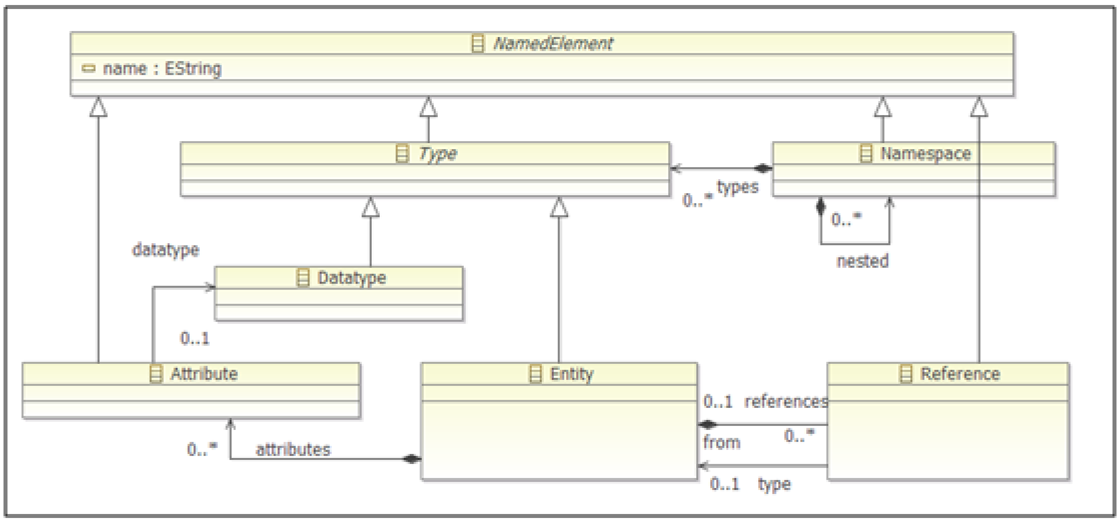


Figure 1.2: UML reinvented - the meta-model [Pfi13].

1.2 Grammar Generation

Between phase 1 and phase 2 of the process of creating DSMLs, the abstract syntax meta-model is defined by the designer. This meta-model underlies the concrete syntax i.e. the graphical structure that we are going to generate. The abstract and concrete syntaxes together form the

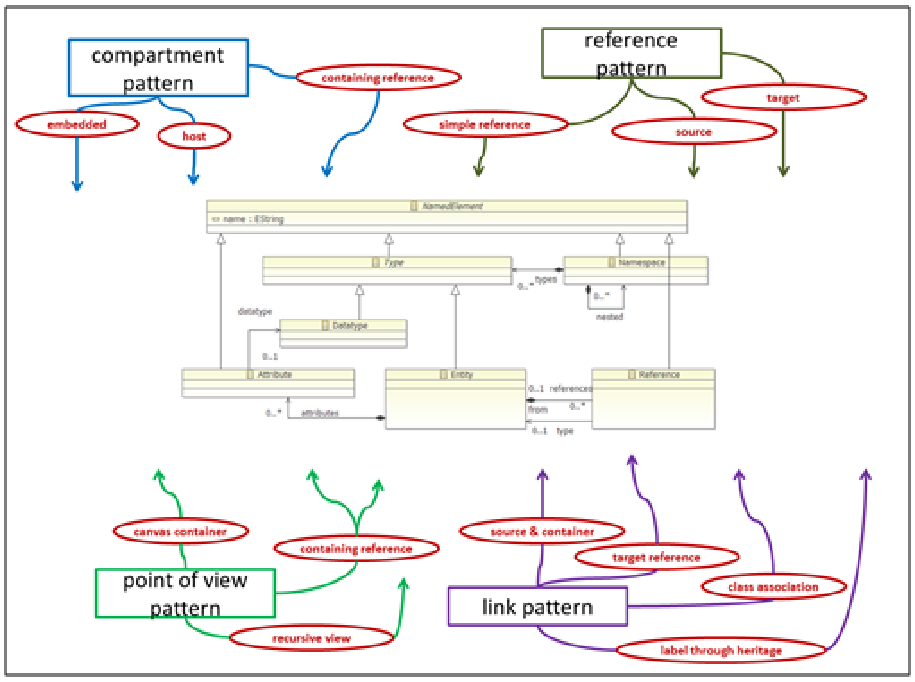


Figure 1.3: Four Notation Patterns which can be found in a meta-model [Pfi13].

visual modeling language. In classic architecture, general and reusable solutions are defined to problems that occur over and over again, even for structures that have nothing in common. Just like the field of classic architecture, in the field of defining DSML, some situations occur often following the same principle. These situations can be identified and partial generation of

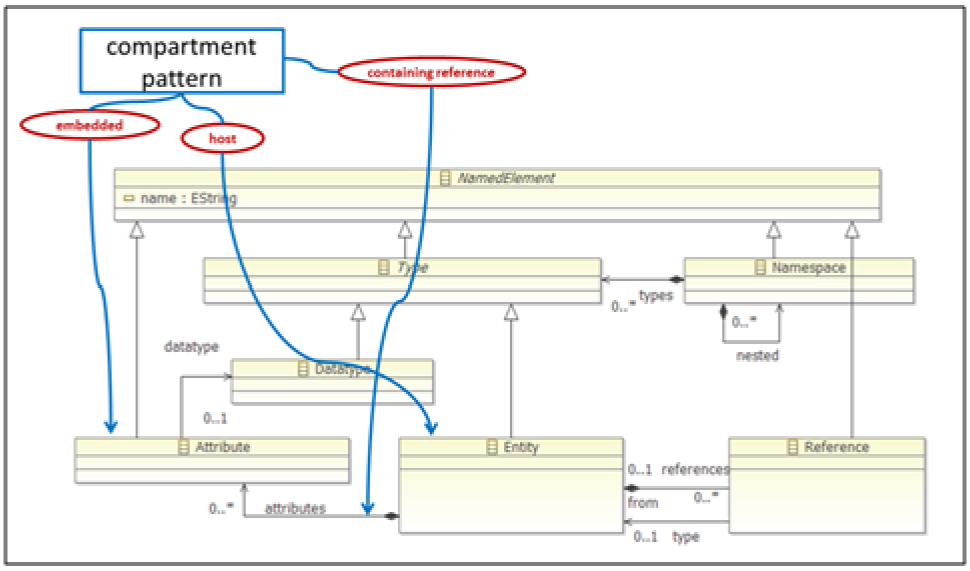


Figure 1.4: Application of the Compartment Pattern [Pfi13].

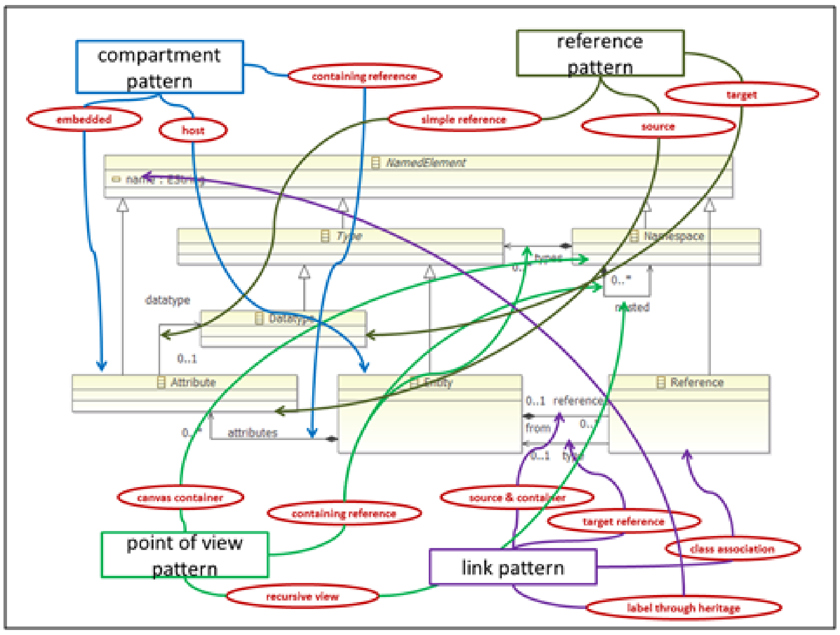


Figure 1.5: Application of several patterns [Pfi13].

grammar can be guided by the detected patterns. For instance, Figure 1.3 identifies four types of patterns illustrated with rectangular shapes and their roles annotated with red elliptical shapes. My aim is to recognize these patterns within the abstract syntax meta-model in order to automatize the generation of grammar for the concrete syntax meta-model. After the process of pattern recognition is complete, the process of pattern applying take place. In this process patterns are applied one by one, for instance on Figure 1.4, the Compartment Pattern has been applied, until all of them are applied, as shown on Figure 1.5, where the pattern application phase is complete. Next, the process of grammar generation occurs. In this process, grammar is automatically generated following the applied patterns (by the means of meta-data generation). Finally, the language designer has a meta-model accompanied by auto-generated concrete-syntax annotations representing, together with the meta-model abstract syntax, the grammar of the target graphic modeling language. In difficult cases, if there is no available or no recognized pattern, the designer has to add manually concrete syntax annotations to the meta-model in order to finalize the domain language. Notation patterns can be applied to any meta-model. However, the latter must match to some well-formed criterions represented by the structure of the notation patterns.

1.3 Conclusion

The process of defining DSML can be automatized, enriching the concrete syntax with automatically calculated grammar. This process of automation occurs between the two main phases of defining a DSML and it can be divided into the following phases:

- Pattern recognition
- Pattern application
- Grammar generation

CHAPTER 2 CONTEXT AND STATE OF THE ART

This chapter defines the context of the works presented in this document. The objective of this work is to propose a grammar and graphical context syntax generator assistant for domain specific modeling languages. Naturally, this state of the art is divided into two domains: (1) model driven engineering, which is the general field of study and (2) software languages engineering, which the more specific field of study.

1) The domain of model driven engineering (MDE) begins describing models and meta-models as the main concepts of the MDE. Afterwards, the origins and the purpose of the MDE are discussed, followed by the model driven architecture (MDA) approach, describing the different model independency levels. At the end, the model transformation approach is discussed.

2) The domain of software language engineering begins describing what a software language is and the different types of software languages. Furthermore, the concept of language description is discussed as a central concept of a software language. Finally, the concepts of domain specific languages and domain specific modeling languages are described.

2.1 Models

2.1.1 Introduction

In order to understand what a model is, we will examine a few definitions.

- The term ”model” is derived from the Latin word modulus, which means measure, rule, pattern, example to be followed [Lud03].
- A model is a set of statements about some system under study (SUS). Here, statement means some expression about the SUS that can be considered true or false [Sei03].
- A model is an abstraction of a (real or language-based) system allowing predictions or inferences to be made [KÖ6].

Ludewig's [Lud03] and Kühne's [KÖ6] definitions refer to Stachowiak [Sta73] definition which describes a model as an entity containing three characteristics :

1. **Mapping:** a model is always representation of an original system. The original system might exist or not[1]. As more appropriately used term by Kühne is subject instead of original.

2. **Abbreviation:** a model does not contain all subject properties. However at least one property of the subject must be mapped.

3. **Pragmatic:** Models are created for a particular purpose and used in a certain context. Therefore, a model is valid for defined subjects or under certain constraints.

A model is always in relationship with a subject who is represented by a system or another model. The relation can be classified along two dimensions: model roles and model usage.

2.1.2 Model Roles

Classification of model roles

A model can have one of the following roles: token model or type model [KÖ6].

Token Model. Token model is what we have in mind when we talk about a model. For example, a plan of a house or a map is a token model. This type of models has a descriptive representation of the subject for illustrative purpose only. They are also called: "representation model", "instance model", singular model" or "extensional model".

Type Model. Contrary to a token model a type model represents the global aspect of the subject by means of classification. Object properties (e.g. "first name", "last name", "age", "address") are used to describe objects (e.g. "person") conform to an object class (e.g. "human"). Other names for type model are "schema model", "classification model", "universal model", or "intentional model". Because of the classification nature of type models, a token model may be represented as an instance of a type model.

Example of model roles

To refine the model roles, Figure 2.1 is an example of both type and token model, using UML. The subject is the real world. The type model classifies the elements into two categories: a city and a road with an association between them. The token model as an instance of the type model represents the real world cities, in this case "Frankfurt", "Munich", "Darmstadt" and "Nurnberg" with the roads between them.

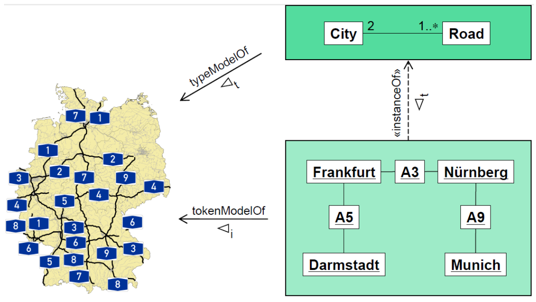


Figure 2.1: Kinds of model roles [KÖ6].

[1]Imaginary system or a system not yet built.

2.1.3 Model Usage

Classification of model usage

Another classification of models based on the model usage can be made. According to [Lud03] models can be classified as descriptive or as prescriptive.

Descriptive Model. The term descriptive model is used when a model represents an existing original like a mirror image to the original

Prescriptive Model. When we are representing something not yet created with a model then we are using prescriptive models. Prescriptive models are used to represent specifications about the original, like a plan for a house.

Transient Model. One more type of model usage can be distinguished which is called transient model. This type of model is used to represent the modifications made on an original. For instance, a model is made representing an original, then changes can be applied to this model, and if they are satisfying then they are applied to the original.

Using models in software engineering

Software engineering in some way depends on models which are prescriptive. *"The requirements specification is double-sided because it describes the user's needs, and it prescribes the product to be developed. It is this double role that makes the specification the most important software component"* [Lud03].

The construction of descriptive model in software engineering is very difficult because it require a precise knowledge of the original, which in the case of software engineering projects is a mess. On the other hand, another problem is that descriptive models are used to describe a simplification on the original, in other words they cover only a small portion of the real world.

In contrary, prescriptive models are easy to construct, but a problem lies in their generality. The risk is a representation of more ideas then needed which remain never tested in practice.

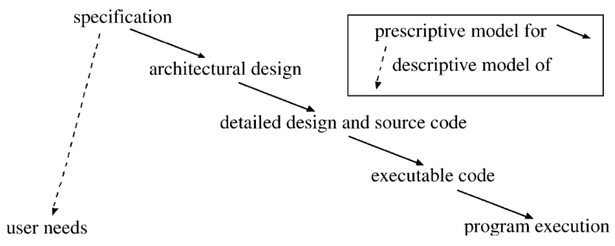


Figure 2.2: The (simplified) document chain. Some documents (like the user's manual, e.g.) and some arrows bypassing some of the documents are not shown [Lud03].

2.2 Meta-models

2.2.1 Introduction

In [MOF02], a meta-model is stated as : "A meta-model is a model that defines the language for expressing a model". In [Fle06], F. Fleurey explains the relationships between different meta-modeling levels. Each meta-modeling level respectively correspond to model, meta-model and meta-meta-model. The system or M_0 level represents a model of the M_1 level. Consequently, the nature of this relationship is different from the relationships between M_1 and M_2 and M_3.Figure 2.3 shows the relationship between different models and languages. A model is expressed in a modeling language, and it is conform to the model that represent that language. Exception to this rule is the M_3 level which is the bootstrap, auto-conforming level. Similar example is shown by Figure 2.4 showing the famous widely used modeling language UML. Figure 2.4 shows how UML is defined and used to describe other classes and instances of those classes. That is the four-layer meta-model hierarchy of UML as explained in [OMG07]. Each level of the hierarchy is an instance of the upstairs level (e.g. M_0 instance of M_1 instance of M_2 instance of M_3) except the last level M_3 which is auto-definable. As schematized in Figure 2.4, M_0 represents "the user data", also called "the run time". Level M_1 is the model of the user data called "user model" while the level M_2 is the model of the user model or the meta-model of the user data which is why it is called "meta-model UML". Level M_3 is the model of the meta-model UML, called meta-meta-model.

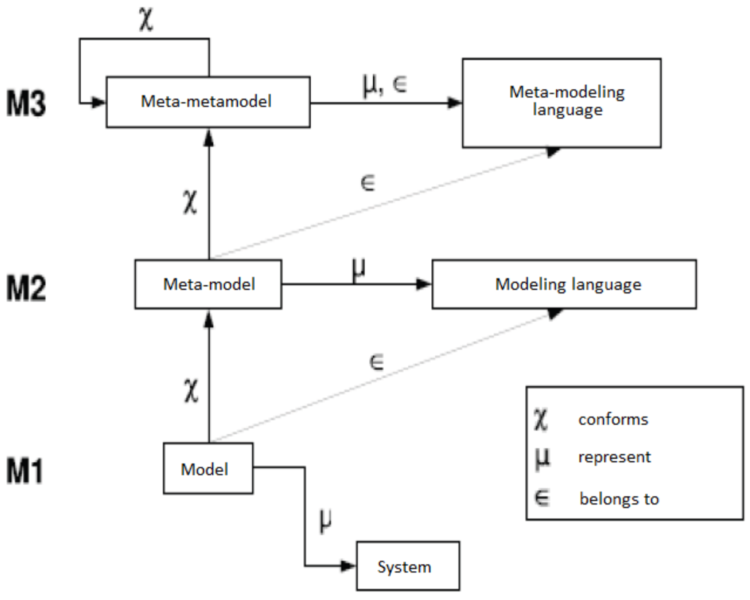


Figure 2.3: Meta-modeling levels [Fle06] page 18.

2.2.2 Meta Object Facility

The M_3 level of Figure 2.4 that we discussed in the previous section is also called MOF or meta object facility, a standard held by the OMG [MOF02]. This is the last level in the four-layer meta-model hierarchy which conforms to itself. This layer is also called meta-meta-model allowing the possibility to define meta-models. A perfect example is UML which is represented as the M_2 level in the hierarchy (see Figure 2.4). This means that MOF is a DSL or domain specific language for defining meta-models, like EBNF which is used to define grammars. In other words, it is a standard for writing meta-models.

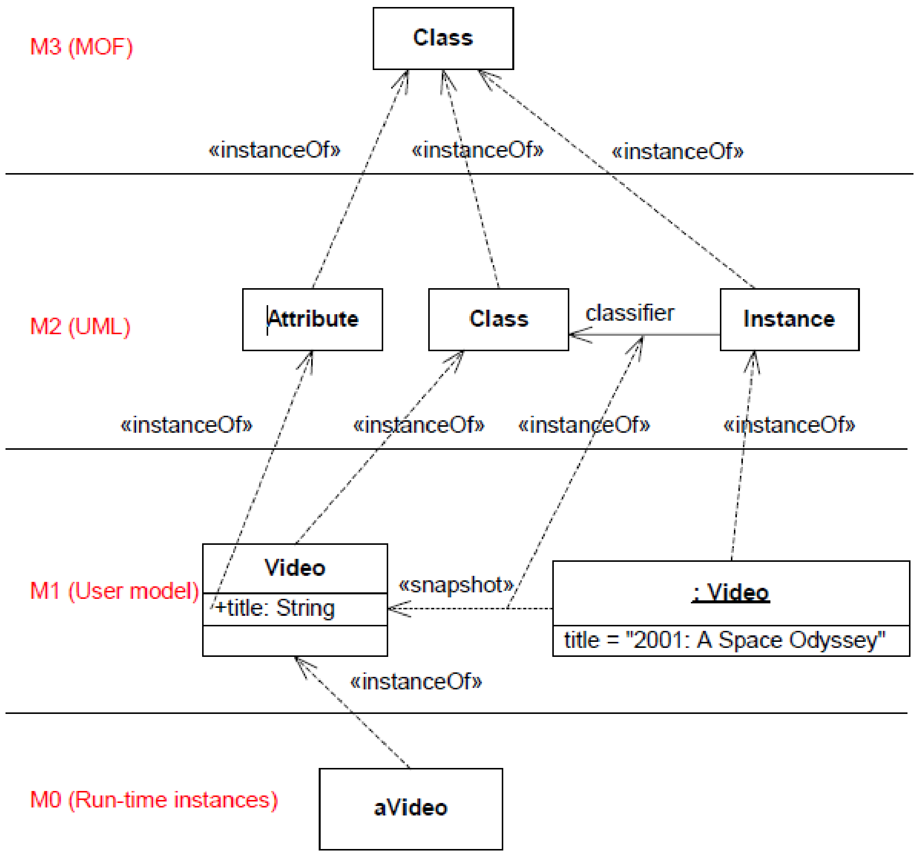


Figure 2.4: An example of the four-layer meta-model hierarchy [OMG07].

Figure 2.5 shows a naive representation of how UML is defined (left) and the auto-definition of the MOF (right), be warned, this is not the real representation, for more details please refer to [MOF02, OMG07, Omg06].

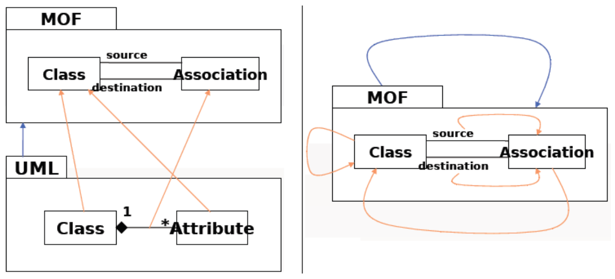


Figure 2.5: Relationships between meta-meta-model and (meta) model [HN12].

In Figure 2.5, the blue arrow represents model to meta-model relationships while the red arrow represents entity to meta-entity relationship.

Two variants of MOF exist [Omg06]:

- Essential MOF or EMOF

- Complete MOF or CMOF

A variant has been defined for EMF (Eclipse Modeling Framework) called ECore that is similar to OMG's EMOF

2.3 Model Driven Engineering

2.3.1 Origins

Over the past five decades, we are witnesses of software complexity problems. Those problems require a new concept in software technologies every ten years. An overview of complexity evolution from 1950s until now is introduces by Huchard and Nebut in [HN12].

In 1950s, assembly languages are used, and the written programs were around 10^3 LOC . Few years later (1960s) a new generation of imperative programming is born with the beginning of FORTRAN (1954). This era is also known by the development of various numbers of compilers, and by the improvements accomplished in language theories, e.g. N. Chomsky in [Cho56]. The length of software was rapidly rising from 10^3 LOC to 10^4 LOC, but still the difference between hardware and software progress was evident. Without any software development methods, it was almost impossible to develop deprived of any hardware capacity issue.

Therefore, the next decade some promising efforts to software complexity are made. In 1968, the first NATO summit [Com69] in Software engineering, structured programming and top-down approaches took place. Also, a significant effort is attained by R. Floyd in [Flo67] trying to assign meaning to program. The software length grown to 10^5 LOC. With the growth of software length and complexity, new solutions to software engineering are proposed, such as modular design and structure, functional decomposition, etc.

In 1980s, a problem caused by requirement changes occurred. For instances, the need of specification changing in a functional program caused complete restructuring of the program. Consequently a functional programming failure was inevitable. With 10^6 LOC, a new way of programming is needed and so object oriented programming took the lead.

The next decade, with the invention of internet and as the software length increased to 10^7 LOC a new prerequisite is required. The distributed systems and software deployment become very popular. Subsequently a new way of programming is developed, the component programming.

The last decade is characterized by the software length which passed from 10^7 to 10^8 LOC, the constant technological changes and a new way of programming called aspect-oriented programming. The rapid technological changes forced the necessity of system migration from one to another technology. This invoked the start of middleware technologies, which made the software integration omnipresent. Also, web services become wieldy used as key of interoperability.

The complexity is mastered by the use of methods, but the various changes of methods cause problems, so the use of another solution is primal, for instance modeling.

2.3.2 Introduction

A significant paradigm shift may occur in the field of software engineering that may have an impact on the maintenance and construction on informatics systems. As explained in [GS03] : *"The software industry remains reliant on the craftsmanship of skilled individuals engaged in labor intensive manual tasks. However, growing pressure to reduce cost and time to market and to improve software quality may catalyze a transition to more automated methods. We look at how the software industry may be industrialized, and we describe technologies that might be used to support this vision. We suggest that the current software development paradigm, based on object orientation, may have reached the point of exhaustion, and we propose a model for its successor."*

Model Driven Engineering or MDE for short is the foundation of the point that we are going to be examining further. Like the basic principle in Object Oriented Programming where "everything is an object", in Model Driven Engineering the basic principle is "everything is a model" [B0́4]. Models and model-elements are first class citizens.

The goal of MDE is to increase the conceptual level in the program specification by the use of models at different conceptual levels for developing systems. Other intention is to increase the automation level in program development by the use of executable model transformations. Therefore, models are transformed using model interpretation or code generator until they are executable.

2.3.3 The purpose of Model Driven Engineering

In [AK03] "the improvement of productivity" is stated as the underlying motivation of MDE. The growing collection of systems and platforms increases the incompatibility of newly found software application. Therefore, MDE aims to increase the business profits from its software development effort. The benefit is provided in two ways [AK03]:

- By improving the short-term productivity of developers.
- By improving the long-term productivity of developers.

The level of short-term productivity is the primary concern of most tool vendors. This level increases the rate of primary artifacts in terms of how much functionality they offer. These artifacts are generated from models in order to assist developers and increase their productivity.

As previously explained, software technologies are rapidly changing. Consequently the level of long-term production is particularly important. This level implies that the profit of the primary artifact depends from the longevity of the artifact. In other words, as much longer a primary artifact is used, as much bigger the profit from that artifact is. Therefore, another even more fundamental aspect is the sensitivity to change of primary artifacts. In [AK03], four main basic forms of particular importance to this sensitivity are detailed:

Personnel. Vital development knowledge should be made easy accessible to others then the software programmer that created the artifact. It should also get easy understandable form if possible. For instance, primary software artifacts can be expressed using a presentation.

Requirements. The fast changing business environments of enterprises implies the necessity of developing new functionality. These changes should not have vast effects on existing systems as a function on disrupting online-systems and system maintenance. Ideally, changes are made while the system keep running, but this is rarely the case.

Development platforms. Like software technologies, development platforms are in constant evolution. Primary artifacts are based on the tool that created them i.e. the lifetime of an artifact is related to the lifetime of the corresponding tool. In order to decouple the lifetime of artifacts from the lifetime of the tool, it is necessary to decouple the artifact from the tool.

Deployment platforms. In order to increase the duration of primary artifacts, it is necessary to shield them from changes in deployment platforms. This form of change is the main concept behind the OMG's - MDA approach[KCXc03], which is examined in the following sections.

2.4 Model Driven Architecture

2.4.1 Introduction

In 2000, OMG adopted a new framework, Model Driven Architecture, or MDA for short. The idea of MDA is to separate the specifications of the operations of a system from the details of the way that system uses the capabilities of its platform [KCXc03]. It is necessary to distinguish MDE from MDA. According to Favre in [Fav04], MDA is a specific representation of the MDE approach.

2.4.2 Architecture

Figure 2.6 shows the four-layer architecture of MDA. In the center, three standards are placed: UML (Unified Modeling Language), MOF (Meta Object Facility) and CWM (Common Warehouse Meta-model). In the next layer, the XMI (XML Metadata Interchange) standard is placed. This standard enables the communication between middleware (CORBA, JAVA, .NET and Web services). The third layer contains the services that manage events, security, transactions and directory. Finally, the last layer proposes a framework that is specific to the application domain.

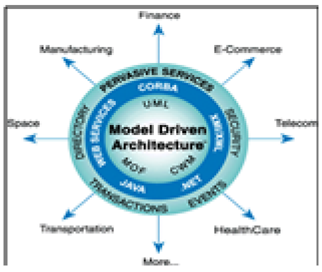


Figure 2.6: MDA architecture [HN12].

In order to create an application, the architect follows the four-layer architecture of MDA, progressing from layer 1 through all layers, starting with a schema described in layer 1 standard.

2.4.3 Independency model levels

MDA is composed of different models for modeling an application and for code generation by model transformation.

Computation Independent Models. The CIM model is independent to all informatics systems, representing a system vision of the environment where it operates, detailing neither a system structure nor a system implementation. The purpose of CIM is to represent what exactly a system should do, to understand a problem and to share certain vocabulary with other models. The CIM model is also known as domain model. It is modified only if the needs or the comportment of that domain change. The expertise is represented on a CIM specification instead on an implementation technology [KCXc03].

Platform Independent Model. This model is independent to all platform techniques (CORBA, .NET, EJB...) without any information on the technology that is going to be used for deploying the application allowing the possibility of deploying the model to a number of different platforms of similar type. The PIM model represents a particular view of a CIM describing the system without showing any detail of its use on the platform. It also represents the conception model or domain logic specific to the system. It needs to be sustainable. An UML diagram combined with a constraint language like OCL (Object Constraint Language) can be used to describe a PIM. Multiple levels of PIM exist. The PIM can hold additional information on persistence, transaction and security [KCXc03, Jul02].

Platform Specific Model. The PSM is based on the platform specified by an architect. A PSM combines the specifications in the PIM with the details that describe how that system uses a specific type of platform. Multiple levels of PSM exist. The first level is derived from a PIM transformation represented by a specific UML schema to a platform. The others are obtained from successive transformations until the generation of a specific language code (C++, Java...). For instance, a PSM can contain information like the program code, deployment description, related programs etc [KCXc03, Jul02].

Platform Model. A platform model delivers concepts representing the different kinds of parts that structure a platform and the services provided by that platform. It also provides concepts representing the different kinds of elements to be used in specifying the use of the platform by an application. A platform model also specifies requirements on the connection and use of the parts of the platform, and the connections of an application to the platform. A generic platform model can amount to a specification of a particular architectural style [KCXc03].

2.4.4 The basic process of model driven architecture

After defining the basic concepts of MDA, we can discuss how MDA is used to design a software system.

First, the computation independent model is defined by an analyst in cooperation with the business user. By model transformation, the CIM model is transformed into a platform independent model. This phase should be prepared by an enterprise architect, who should add the architectural knowledge to the CIM and shouldn't leave any used platform details. In order to finalize the process, the resulting model has to be a target of a platform. Therefore, a transformation from PIM to PSM should be produced by a platform specialist. The resulting PSM is detailed model of the platform providing the necessary system constructing information

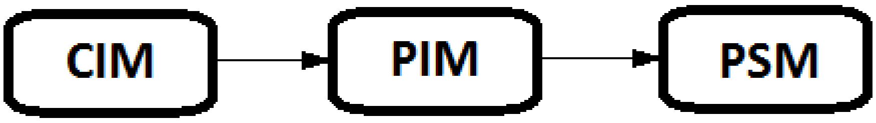


Figure 2.7: MDA basic process.

to put it into operation. The process is illustrated by Figure 2.7. The main principle is model transformation between different models adding knowledge at each transformation by different professionals.

2.5 Model Transformation

2.5.1 Introduction

As explained in [SK03] *"The objective of model-driven development is to increase productivity and reduce time-to-market by enabling development at a higher level of abstraction and by using concepts closer to the problem domain at hand, rather then the ones offered by programming languages. The key challenge of modeldriven development is in transforming these higher-level models to so-called platform-specific models that can be used to generate code."*, the key challenge in MDD (Model Driven Development) is the model transformation. Figure 2.8 illustrates a model transformation process which transforms a source model into a target model following some transformation rules.

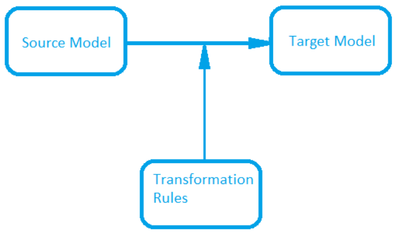


Figure 2.8: Model transformation process [NH12].

2.5.2 Features of Model Transformation Approaches

A classification of model transformation approaches is given in [CH03]. The top-level areas of variation in model transformation are: transformation rules, rule application scoping, source-target relationship, rule application strategy, rule scheduling, rule organization, tracing and directionality.

Transformation Rules. A transformation rule is composed of left-hand side (LHS) and right-hand side (RHS). The left-hand side accesses the source model while the right-hand side expends in the target model. The goal of these transformation rules is the description of a transformation process, in other words, how a source model is transformed into target model.

Rule Application Scoping. Rule application scoping allows a transformation to restrict the parts of a model that participate in the transformation. Target scope of that transformation can be defined, which is the scope of the target model. Some approaches support source model scoping, which are important for performing reasons.

Source-Target Relationship. The purpose of model transformation is the transformation of a source model into a target model. The target model can be a newly created model, based on information presented by the source model and the transformation rules, or it can be an updated version of the source model by removing or adding elements to the source model.

Rule Application Strategy. A rule application strategy is needed in order to determine where to apply a rule in a given source scope while more then one match are possible. The rule application strategy can be deterministic, non-deterministic or interactive.

Rule Scheduling. This approach determines the order in which individual rules are applied. Rule scheduling can vary in the following main areas: form, rule selection, rule interaction and phasing.

Rule Organization. Transformation rules can be structured and composed by the rule organization. In this context, three areas of variation exist: modularity mechanism, rules mechanism and organization structure.

Traceability Links. This approach consists of saving links between source and target elements, which may be useful in performing impact analysis, synchronization between models, model-based debugging and determining the target of a transformation.

Directionality. Transformations may be unidirectional, executed in one direction only (e.g. from source to target, but not the other way around) and bidirectional, executed in both ways (e.g. from source to target and from target to source).

2.5.3 Categories of model transformation approaches

More then thirty transformation approaches exist in the literature. Czarnecky and Helsen in [CH03] present the major categories of existing transformation approaches. Depending on the generated entity, two principal categories can be distinguished: model-to-code category and model-to-model category.

Model-To-Code Category. The model to code category is a code generation transformation, which from a source model generates code. This category can be divided into visitor-based approach and template-based approach.

- **Visitor-Based Approach** consists in providing some visitor mechanism to traverse the internal representation of a model and write code to a text stream.
- **Template-Based Approach** consists of the target text containing splices of meta-code to access information from the source and to perform code selection and iterative expansion (see [Cle01] for an introduction to template-based code generation).

Model-To-Model Category. The model to model category aims to transform source model to target model. Both, source and target may be instance of the same or different meta-model (see Source to target relationship paragraph in the Features of Model Transformation Approaches section). This category can be divided into: direct-manipulation, relational, graph-transformation-based, structure-driven and hybrid approaches.

- **Direct-Manipulation Approach** offers an internal model representation plus some API to manipulate it.
- **Relational Approach** is a declarative approach based on mathematical relationships.
- **Graph-Transformation-Based Approach** is based on graph-transformation theory.
- **Structure-Driven Approach** contains two phases; the first phase is concerned with creating the hierarchical structure of the target model while the second sets the attributes and references in the target.
- **Hybrid Approach** combines different approaches from the previous categories.
- **Other Model-To-Model Approaches.** For instance the transformation framework defined in the OMG's Common Warehouse Meta-model (CWM) Specification [CI02] and transformations implemented using XSLT [Kay07].

2.5.4 Graph Transformation and Graph Grammars

The most widely used model transformation technique with relevant impact in the literature and industry is graph transformation. In graph transformation, the most costly operation is to find a sub-graph matching the precondition of the transformation rules [SV09].

In the context of EMF ECore, models are represented by hyper-graphs (abstract-syntax graphs). The QVT (Query/View/Transformation) standard [Kur08, Omg08] is based on a graph grammar describing the transformation. A graph transformation contains transformation rules. Each rule describes a correspondence between the LHS and the RHS. A set of transformation rules is applied on an input graph in order to perform a graph transformation. When a correspondence is found between the LHS of a rule and a part of the input graph, the corresponding graph is replaced by the RHS of that rule. A rewriting system iteratively applies these rules until the end of this process which means that correspondences were not found. The rules can be extended with: application priority level, conditions that need to be satisfied in order to apply the rule and actions that are going to be executed when applying that rule. Besides this declarative aspect, some systems propose imperative rules.

The use of graph transformations based on graph grammar, have some advantages over an implicit (imperative) representation. Firstly, a high-level, abstract and declarative representation of the model quality which is a target of graph operations like transformations, reasoning and analyze and secondly, the foundation graph theory can be used through proves and propriety validation of the model transformation.

However, the use of graph grammars is constraint to efficiency considerations. In fact, the decision of graphs isomorphism is a NP-complete problem.

2.6 Software Language

2.6.1 Introduction

In the past few years, an increasing number of languages used for designing and creating software have been created. Along with that has come an increasing interest in language engineering, in other words, the act of creating languages.

First, Domain Specific Languages (DSL) have become more and more dominant, describing certain aspect of a software system or the software system from particular view. Second, modeling languages, especially Domain Specific Modeling Languages (DSML), a specific type of DSLs, have become very important within the field of MDE and MDA. Each model is written in specific language, and the model is transformed into another model written in (in most cases) in another language (see Model Transformation) [Kle08].

Generally, two types of software languages can be distinguished: modeling and programming languages. Traditionally, these types of languages are viewed to be different. As shown in the following table, some characteristics of modeling languages are different to programming languages. However, there are more similarities then differences. In essence, both describe software, and after transformation or compilation both need to be executable. In the course of time, the differences of modeling and programming languages are becoming less and less distinctive. For instance, Executable UML [MB02] and The UML Profile for Java [SG04], where models are becoming precise and executable, while programming languages are adopting high-level graphical syntax. Furthermore, MDE implies the use of models as core products.

Modeling Language	Programming Language
Imprecise	Precise
Not executable	Executable
Overview	Detailed view
High level	Low level
Visual	Textual
Informal semantic	Execution semantic
By-product	Core-product

Table 2.1: Differences between Modeling and Programming languages [Kle08] page 3.

Therefore, the term software language is used to indicate any language that is created to describe and create software systems. Languages like query, data-definition and process languages are also software languages.

Over the past two decades the nature of software languages has changed in at least two important ways. First, software is increasingly being built using graphical (visual) languages instead of textual ones. Second, more and more languages have multiple syntaxes [Kle08].

2.6.2 Graphical versus Textual

"It is helpful to compare the linear structure of text with the flow of musical sounds. The mouth as the organ of speech has rather limited abilities. It can utter only one sound at a time, and

the flow of these sounds can be additionally modulated only in a very restricted manner, e.g. by stress, intonation, etc. On the contrary, a set of musical instruments can produce several sounds synchronously, forming harmonies or several melodies going in parallel. This parallelism can be considered as nonlinear structuring. The human had to be satisfied with the instrument of speech given to him by nature. This is why we use while speaking a linear and rather slow method of acoustic coding of the information we want to communicate to somebody else"[BG91].

In the area of sound, the difference between spoken words with music is clearly explained in the previous quote. Similarly, in the context of software languages, textual languages can be distinguished from graphical languages. An expression in textual languages has linear structure while in graphical languages have much more complex structure.

The fundamental difference between graphical and textual languages is the connection of expressions which in graphical languages represented by graphical symbols (e.g. rectangle or arrow) can be poly-dimensional whereas in textual languages represented by symbols (e.g. tokens in the field of parse design) is one-dimensional. Figure 2.9 shows what an UML class diagram would look like while expressed in linear way (a) and in normal non-linear way (b). Automatically, duplication problem rises with the linear representation where same entity is repeated [Kle08].

Many languages support hybrid textual/graphical syntax. For instance, in UML class diagram the notation of properties (attributes and operations) is a textual syntax embed in a graphical one.

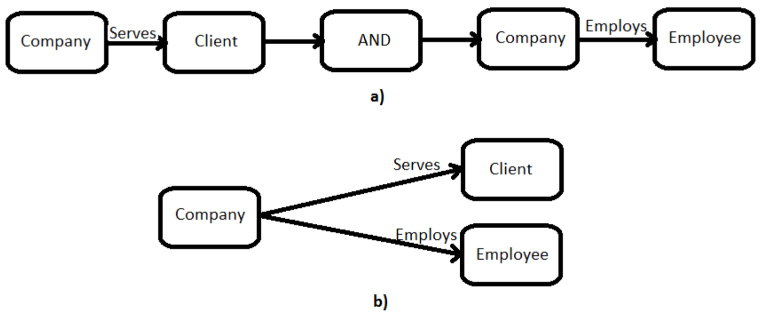


Figure 2.9: A linear and a non-linear expression [Kle08] page 6.

2.6.3 Multiple syntaxes

The previous example of the UML hybrid syntax shows that languages can have multiple (concrete) syntaxes. Another example is languages that have an interchange format (e.g. XML), which means that they have an interchange syntax and a normal syntax. The need of languages with both textual and graphical syntaxes is growing, such tool is TogetherJ which uses UML as a graphical notation for Java.

Consequently, as a result of multiple (concrete) syntaxes, the (concrete) syntax cannot be the focus of language design. The focus of language design should be on a common representation which is independent of the outward appearance in which it is entered by or shown to the language user. This common representation is called abstract syntax [Kle08].

2.7 Language Description

2.7.1 Introduction

All software languages, no matter if they are modeling, mark-up, programming or any other type, are created artificially (compared to natural languages). This artificial nature of the languages implies the impotency of a language description which has the same role as grammar in natural languages. Before going any further, let's focus on what a language is, and why the language description is fundamental.

In [Kle07a], language is defined as *"A language L is the set of all linguistic utterances of L"*.

The term utterance in linguistic refers to complete communicative unit, which may consist of single words, phrases, clauses and clause combinations spoken in context. Utterances are created according to rules. These rules define how the utterances should be structured. A set of these rules is called language description. For the sake of formality, Kleppe uses the term utterance in the general case of language, while in the case of software language she uses the term **mogram** which is short for model/program.

Kleppe defines language description in [Kle07a] as *"A language description of language L is the set of rules according to which the linguistic utterance of L are structured, optionally combined with a description of the intended meaning of linguistic utterances."*

A language description should be complemented with a description of the language's semantics. This part of the language description is optional but in my opinion it is very important. What is the point of speaking a syntactically perfect sentence if the meaning is unknown?

2.7.2 Abstract and Concrete syntaxes

In a software language, the superficial structure is not forced to be identical to the underlying structure, in contrary it might be entirely different. This implies the necessity of dividing the language syntax into two levels: abstract level which contains the abstract syntax and concrete level which contains one or multiple concrete syntaxes. For instance, a mogram is presented to a language user in a concrete form (e.g. text or diagram), this is the concrete level. A mogram also should have a representation that is hidden in the tool that handles it. This is in-memory representation in the abstract form of the mogram, or abstract syntax graph, or abstract syntax tree [Kle08, Kle07a].

Abstract Syntax. The abstract syntax plays a central role in a language specification. It is the pivot between concrete syntax and semantics. The original meaning from the term abstract syntax comes from natural language, where it means the hidden, underlying, unifying structure of a number of sentences [Cho65]. In computer science, the abstract syntax is hidden, presented as in-memory form. When shown on a screen, the mogram is obtaining a concrete form. Language users can handle abstract syntax only trough concrete forms. The abstract syntax representation is the underlying structure of a mogram. The multiple concrete syntax of software language implies the unifying nature of the abstract syntax. For instance the same abstract syntax representation can be presented concretely in different concrete formats, e.g. in graphical and textual formats. The abstract syntax also plays a role in the relationship of concrete syntax and semantic interpretation. When creating a software language, understanding the used concepts is fundamental. These concepts are specified in the abstract syntax model, together with the possible relationships between concepts and also the meaning of each concept. Therefore, the abstract syntax is the central element in a language specification [Kle08, Kle07b].

Concrete Syntax. A crucial element to a language design is the concrete syntax. The concrete form that we mentioned in the Abstract Syntax section is an instance of the concrete syntax, so its role is to represent a mogram to humans, usually via tool. Such tool is the editor, which is used to create or change a mogram. The type of editor used influences the way we think about concrete syntax and its mapping to abstract syntax. Another large influence is whether the language is textual or graphical [Kle08].

2.7.3 Software Language and Mogram Relationships

How do we decide whether the mogram created by a language user is correct according to the language specification? In traditional language theory, the language specification is a grammar and there should be derivation to that mogram according to that grammar. When a language specification is a meta-model, then the mogram should be instance or conforms to that meta-model. This relationship is formally explained in [Kle08] page 78 introducing some concepts.

Abstract Syntax Model. *"The abstract syntax model (ASM) of L is a meta-model whose nodes represent the concepts in L and whose edges represent relationships between these concepts."*

Abstract Form. *"The abstract form of a linguistic utterance of language L is a labeled graph typed over the abstract syntax model of L."*

Concrete syntax Model. *"The concrete syntax model (CSM) of L is a meta-model whose nodes represent elements that can be materialized to the human senses and whose edges represent spatial relationship between these elements."*

Concrete Form. *"The concrete form of a linguistic utterance of language L is a labeled graph typed over the concrete syntax model of L"*

Linguistic Utterance. *"A linguistic utterance belongs to language L when (1) both its concrete and its abstract forms are instances of (one of) the concrete syntax model(s) and the abstract syntax model, respectively, and (2) there exists a transformation from the concrete form into the abstract form."*

To conclude, a mogram is an instance of the abstract syntax model which is a meta-model. A mogram has one or multiple concrete forms which humans can see and manipulate. The abstract form should contain relevant information to the linguistic expression yet leave out all the syntactical sugar associated with a particular concrete syntax.

2.7.4 Parts of Language Description

As defined in [Kle08] a language description is composed of the following parts:

- One abstract syntax model (**ASM**)

- One or multiple concrete syntax models (**CSM**)
- **Syntax mapping** – from each CSM, a model transformation that defines how the concrete form of a mogram is transformed into its abstract form.
- **Semantic description** (optionally) – describing the meaning of the mograms, including a model of the semantic domain.
- **Required language interfaces** (optionally) – a definition of what the language's mograms needs from others mograms written in another language.
- **Offered language interfaces** (optionally) – a definition of what the language's mograms offers to other mograms written in another language.

The last two parts of the language description are especially important for establishing communication between mograms in order to create a complete application. In [Kle08] an example is cited combining DSL dedicated to the design of Web-based user interfaces with a DSL for the design of services in server oriented architecture.

2.7.5 Formalisms to specify languages

An overview of formalisms to specify languages is given in [Kle08] page 47. Some of them partially define the language description.

Context-Free Grammars. The context-free grammar is a specification of the concrete syntax. In 1956, Noam Chomsky formalized grammars and classified them into types known as Chomsky Hierarchy. The third level in this hierarchy is called Type-2 or context-free grammars. A context-free grammar is a set of production rules, where every rule is of the form: V -¿ v where the left side is represented by a single non-terminal symbol while the right side is a string of a terminals and non-terminals. The languages that could be generated by context-free grammars are called context-free languages. The context-free grammar's underlying structure is a derivation tree, whereby each node represents either the application of a production rule during the recognition of a mogram or a basic symbol (non-terminal). In computer science, one of the main notation techniques for context-free grammars is the Backus Normal Form or Backus Naur Form (BNF). To simplify, the context-free grammar specify the symbols and keywords that must be present in a concrete form of a mogram. For instance, the while loop in C, contains the keyword "while" and two types of brackets, a condition and a statement as well. The context-free grammar allows you to specify only textual languages. [Cho56, Kle08, Kle07a]

Attributed Grammars. A context-free grammar that has attributes associated with each of the non-terminals is called an attributed grammar. This type of grammar holds the information on a mogram within their attributes. The attributes can have the form of a tree, graph or a more complex structure. The attributed grammars allow the specification of any type of languages (graphical too). The abstract form of a mogram can be created as an attribute of a concrete element, such as keyword. The graphical languages that use this type of grammar are concerned with scanning and parsing language expressions and diagrams. The idea is to recognize a basic graphical symbol and group it into more meaningful elements i.e. the alphabet. The graphical symbols hold information on the concrete representation of that symbol to the language users i.e. color, position, and style. Various types of spatial relationships are defined based on the position, attaching points or attaching areas. Together, the alphabet and the spatial relationships are used to state the grammar rules that define grouping of grammatical symbols [Kle08].

Graph Grammars. Another way to specify graphical languages is with the use of graph grammar. This is the most complex type of grammar. Graphs are used to represent diagrams. The graph edges represent relationships between the parts of the diagram and attributes or labels to the edge (or nodes) represent the concrete representation of the element. Another structure is also needed for the application of the grammar rules which is a graph called derivation graph. The nodes of a derivation graph are graphs representing the diagram [Kle08].

[RS97] shows how graph grammars can be used as syntax definition formalism for graphical languages, and develops a graphical parsing algorithm based on these grammars. Discusses the internal representations necessary to support editing and execution of visual programs, by showing how graph grammars fit in, and by arguing why graph parsing would be useful for users of visual languages. (Top Down approach starting from already existing visual notations).

[DRLP11] The GMF framework, as it stands, lacks support for evolution. In particular, there is no support for propagating changes from the domain model (i.e., the abstract syntax of the visual language) to other editor models. The paper analyzes the resulting co-evolution challenge, and we provide a solution by means of GMF model adapters, which automate the propagation of domain-model changes. These GMF model adapters are special model-to-model transformations that are driven by difference models for domain-model changes.

UML Profiling. Very powerful way of specifying new languages or extending an existing generic ones is by creating UML profiles. In this case, stereotypes and the UML specification represent the concrete syntax model and the mapping to the abstract syntax model, while the abstract syntax is represented by the profile itself. The language designer should additionally specify the language semantics [Kle08].

Meta-modeling. Meta-models are used to specify languages (see Meta-models for more details). Some consider that meta-models are more powerful for specifying languages then traditional BNF grammars [GS03, Kle07a].

The context-free grammar has limited expression for language description because of the underlying structure which is a tree, while the graph grammar is too complex. The attributed grammar simply obscures the true structure of the mogram and a UML profile is too restrictive. Consequently, meta-modeling in the most practical choice for specifying software languages.

2.8 Domain Specific Language

2.8.1 Introduction

Dimension	Value
System Aspects	Workflow, Data, Security, Business rules, Presentation...
Subject Areas	Customer portal, Order entry, Back-end administration...
Abstraction Level	CIM, PIM, PSM
Stakeholders	Programmer, Domain expert

Table 2.2: Dimensions for MDE projet

Important point in software development process is the establishment of a dimension to the project. Except versioning and authorship, other dimensions are also important like which system aspect is of importance or which subject area plays a role. The chosen level of abstraction in the process and the chosen stakeholders are also important decisions to be made. In software development process, models can be placed at an intersection of the dimensions. For instance:

- Computation independent model (CIM) of the security of the order entity part of the software artifact aim at a domain expert.
- Platform independent models (PIM) of the date of the order entity part of the software artifact also aim at domain expert.
- Platform specific model (PSM) of the data of the customer portal part of the software artifact aim to a programmer.

For each particular model, a modeling language should be chosen influenced by the level of abstraction, the subject area and the stakeholders. This type of language is also called DSL (Domain Specific Language).

2.8.2 What is a Domain Specific Language?

What exactly domain specific language is, still rests a subject to debate. In [DKV00], a DSL is stated as : *"A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain"*, while in [Voe09], a DSL is: *"Language that is custom defined to describe aspects of software system"*.

Two types of DSL exist: Graphical and Textual. Graphical DSLs have graphical concrete syntax, while textual DSLs have textual.

These languages are used to create models which are further used as validation, simulation, input for code generation or interpretation.

DSLs are not only made to be used by architects and developers, but also by business users who are not necessary considered as "developers".

2.8.3 Model Driven Architecture methodology and Domain Specific Languages

Languages have a degree of domain specificity. Some are more dedicated to a certain domain then others. Two general types of domains can be targeted by a DSL.

Knowledge domain. A set of concept and terminology that form a field of knowledge understood by experts in that field.

System family domain. A set of similar functionality software systems, which can be seen as a kind of knowledge domain.

Compared to the dimensions represented by Table 2.2, system aspect dimension corresponds to knowledge domain, while the subject area dimension corresponds to system family domain.

In the MDA methodology, the choice of the kind of DSL to be defined is very important. If we define for each intersection between dimensions a DSL, the number of different DSLs will be way too high. For instance in Table 2.2, 90 different domain-specific languages can be defined (i.e. the multiplication of the number of values of each dimension). Therefore, critters need to be added in order to reduce this number:

- Directly executable DSLs on a specific platform
- Extensible DSLs by en exiting GPLs (Global Purpose Language)
- DSL for each aspect
- Internal DSL (vs External DSL)

Applying these critters to the previously considered example reduces the number of DSLs to 5 (for the following aspects: security, data, business rules, presentation and workflows).

2.8.4 Should we use Domain Specific Languages? – Advantages and Disadvantages

The comparison of languages can be based on code statements number needed for implementing a function point. In [Jon], languages are compared into a table, separating low level languages (e.g. natural language, assembly language or machine language) from high level languages targeting at a specific domain. However, the difference between DSLs and GPLs is a matter of degree. In my opinion, only the most focused languages as HTML or SQL should be considered DSLs.

K. Czarnecki in [Cza05] describes some significant advantages of DSLs over GPLs:

- Domain-specific abstractions: a DSL provides pre-defined abstractions to directly represent concepts from the application domain.
- Domain-specific concrete syntax: a DSL offers a natural notation for a given domain and avoids syntactic clutter that often results when using a general-purpose language.
- Domain-specific error checking: a DSL enables building static analyzers that can find more errors then similar analyzers for a general-purpose language and that can report the errors in a language familiar to the domain expert.
- Domain-specific optimizations: a DSL creates opportunities for generating optimized code based on domain-specific knowledge, which is usually not available to a compiler for a general-purpose language.
- Domain-specific tool support: a DSL creates opportunities to improve any tooling aspect of a development environment, including, editors, debuggers, version control, etc. The domain-specific knowledge that is explicitly captured by a DSL can be used to provide more intelligent tool support for developers.

In addition, A. Van Deursen in [DKV00] cites *"DSLs embody domain knowledge and thus enable the conservation of this knowledge"*. Eric Evans in [Eva03] underlines that language should be understandable by both domain experts and developers in order to decrease the complexity of software development.

However, DSLs also have some disadvantages, as explained in [DKV00]:

- The cost of designing, implementing and maintaining a DSL.
- The cost of education for DSL users.
- The limited availability of DSLs.
- The difficulty of finding the proper scope for a DSL.
- The difficulty of balancing between domain-specificity and general-purpose programing language constructs.
- The potential loss of efficiency when compared with hand-coded software.

2.9 Domain Specific Modelling Language

2.9.1 Introduction

In the section Domain Specific Language the concept of DSL is explained. In this section we are going to examine similar concept called Domain Specific Modeling Languages which is a modeling DSL and an example of a tool called Diagraph, allowing the possibility to defining such languages.

DSLs are alternative to general purpose languages while the alternative to general purpose modeling languages (e.g. UML, SysML) are DSMLs. General purpose modeling languages provide concepts with a lack of precise semantics for expressing structural and behavioral views of the system under study. The use of general concepts have limited expression, for instance they cannot precisely express every detail of a specific domain. Therefore, when a model cannot be described using generic concept, specially tailored languages for that problem are proposed. This type of languages promotes the domain concepts as first-class citizens. Such concepts are classified according to the following points of view (proposed in [PCHN12]):

- Behavioral point of view
- Functional point of view
- Qualitative, non-functional point of view
- Operational point of view
- Organizational point of view
- Protocol point of view
- Structural point of view
- Semantic point of view

Tools like Microsoft DSL Tools and EMF-GMF support the design of domain specific modeling languages, and they are integrated in platforms such as Visual Studio and Eclipse. These platforms can generate graphic editors from meta-model and parameters given by the domain expert as input. The editor can create instances of the given meta-model. This process is shown in the section Notation Patterns, first approach where Figure 1.2 represents the meta-model and Figure 1.1 represents an instance of the meta-model.

2.9.2 Meta-modeling layers comparisons

The understanding of the whole process is a little complicated because of the mixed meta-modeling levels of the domain meta-modeling compared with the classis UML meta-modeling that we discussed in section Meta-models. Therefore, Figure 2.10 compares the meta-modeling layers of UML from the one site, and domain specific modeling language from the other. The meta-meta-model layer is common for both UML and DSML, while the M_2 and M_1 layer are different respectively represented by input domain specific meta-model (for DSML), meta-model (for UML) and instance of the domain specific model (for DSML) and instance of the UML meta-model (e.g. class, activity, use case...). The M_1 layer represented by class instances (for UML) is empty for the DSMLs.

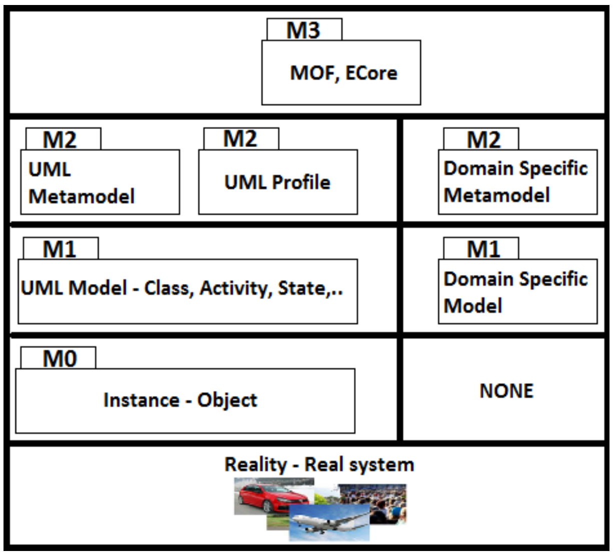


Figure 2.10: Comparing Domain Specific and UML meta-modeling layers [DGD06].

2.9.3 Design Process for Domain Specific Modeling Languages

The design of a software language involves the creation of language description. As software language, DSMLs also need to define the language description which is shown by the workflow described in Figure 2.11. The process of defining the abstract syntax is actually the process of creating the abstract form represented by the domain specific meta-model (explained in the previous section) while the process of defining the concrete syntax refers to the process of creating the concrete form which may be represented by textual or graphical form. The mapping between the abstract and concrete syntax is an obligatory part of the language description, as explained in the section Parts of Language Description. Therefore, this mapping is established as a graph of correspondences between the abstract and concrete syntaxes which are also represented by graphs. *"This correspondences graph is noted on the abstract syntax graph by annotating the* *latter whit a very simple language, like a layer that would provide additional information to a map, by transparency"* F. Pfister in [PCHN12].

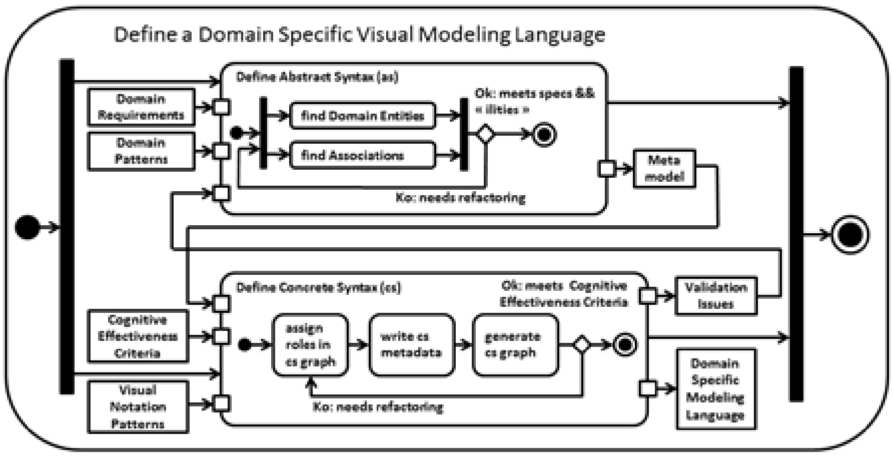

Define a Domain Specific Visual Modeling Language
Domain Requirements
Domain Patterns
Define Abstract Syntax (as)
find Domain Entities
find Associations
Ok: meets specs && « ilities »
Ko: needs refactoring
Meta model
Define Concrete Syntax (cs)
Ok: meets Cognitive Effectiveness Criteria
Validation Issues
Cognitive Effectiveness Criteria
assign roles in cs graph
write cs metadata
generate cs graph
Visual Notation Patterns
Ko: needs refactoring
Domain Specific Modeling Language

Figure 2.11: A Design Process for Domain Specific Modeling Languages [PCHN12].

This process involves many small and loosely coupled meta-models which may contribute to a library of reusable meta-models, foreshadowing of the concept of reusable pattern. These fragments are located in an area called Megamodel [BJV04].

2.9.4 The Modeling Framework Diagraph

The modeling framework named Diagraph is based on Ecore meta-models which imply the class-relationship paradigm. Diagraph is a tool that assists the design of graphical DSMLs using non-intrusive annotations of the meta-model to identify the concrete syntax elements (e.g. edges, nodes, nested structures and other graphical information). The EMF-GMF instance editor is used for immediate validation of the meta-models. This framework is used as an Eclipse plugin on the top of legacy tools including the design process described by Figure 2.12, an annotation language and a megamodel manager. Elements in pink form the context of the Diagraph environment. In the middle, the diagraph's DSML process is described. The diagraph abstract syntax is a domain specific meta-model that conforms to ECore shown by the upper part of the figure. The concrete syntax is represented by the layer named "Generic Concrete Graphical Syntax Layer" associated to a transformation noted "to" in front of a graph description language [PCHN12].

2.9.5 Designed Graphical Domain Specific Modeling Language Typology

Diagraph is a tool that is able to generate editors targeting the whole range of languages used in complex system modeling. Several types of (graphical modeling) languages can be distinguished:

- Languages describing structures: type graphs, object graphs, component graphs, class diagrams, ontologies.
- Languages describing interactions: sequence diagrams, communication diagrams (protocols, operational scenarios)
- Languages describing transformations of "things": process languages, functional block languages.

- Languages describing states and modes: state diagrams, languages for verification of system properties.
- Languages describing physical quantities (variables, units).

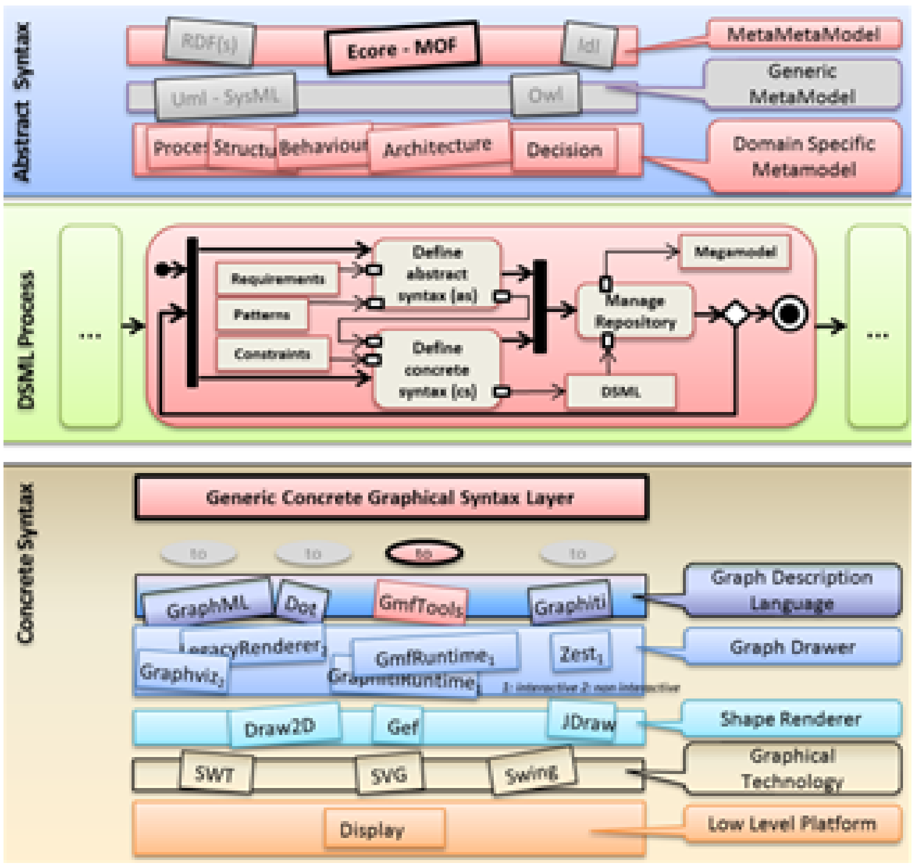


Figure 2.12: Diagraph in the context [PCHN12].

The languages describing interactions, transformations and states have operational semantics, in other words, a process model or a state machine can be run through them. Contrary to this type of languages, the languages describing structures are of a ”static” semantic nature and they are used to describe entities. Every described entity has calculable values that can be described by languages describing physical quantities. They are not to be executed, but they are subject to carry physical ”things” in order to predict the successive states of the real system they represent by the way of computation and model simulation[MV04]. Transformational languages have operation semantics that are ”executable”, in other words that semantics are denoted by the states the system can reach.

The BWW model [Bun77, WW88] could illustrate such a language typology: The most central phenomena every upper-level model are thing, property, state, and transformations (functions). According to that typology, we can build models that represent a real system.

This work is not concerned with operation semantics, even if target models can be executable models such as Petri-nets or eFFBDs. It aims to define the process of building the concrete syntax, and also to define meta-concepts and thus a meta-model of visual language notation.

2.9.6 Cognitive Effectiveness Criteria

Since my focus aims on the graphical DSMLs, an important concern is how to graphically denote or connote the semantics of a graphical language and which graphical structure (nested or associated structures, complexity hiding paradigm or view separation) to use. Such concerns were first examined by Bertin [Ber83] in the field of cartography. Some of these concepts were adopted and published by Moody [Moo09] in the field of visual notation. Similar work is done in [LPDc12] where the following criteria is proposed:

Semiotic Clarity. In order to avoid the redundancy, the overload, the excess or the symbols deficit the set of semantic (abstract) elements and the set of graphic (concrete) elements should be in bijective relationship.

Perceptual Discriminability. The visual discrimination level between elements should be as elevated as possible. This improves the perception of the diagram and the cognitive processing as well. The visual discrimination consists of : the position, the size, the color, the orientation and the form.

Semantic Transparency. This is one of the most difficult critters to evaluate. The semantic transparency is related to the graphic representation of the elements. It consists of representing the element with a form as logically closer to the human logical vision of that element as possible. For instance, if we want to represent a ball we should choose a circle and not a triangle because the circle form is logically closer to the ball then the triangle.

Complexity Management. When we are constructing a very complex structure we should try to decrease the complexity of that structure as much as possible. In the case of visual representation the complexity implies a large number of graphical elements. In order to manage that complexity we should try to arrange the elements by changing the size and position, by embedding elements in other or by constructing multiple diagrams (e.g. a diagram that will show the one half of the elements and another that will show the other half).

Cognitive Integration. Information mechanisms should be integrated in the diagrams. For instance, when a language user navigates from one diagram into another by the mechanism of complexity management, visual elements should describe the context in which the diagram we are manipulating is. In other words, the studied part of the model should be integrated into the mental representation of the global model.

Visual Expressiveness. A concrete syntax is visually expressive if it operates with a large number of visual variables and it uses a large number of values for each variable. The visual expressiveness improves the cognitive efficiency of the syntax.

Dual Coding. A textual annotation is very useful for boosting a geometric form or an icon. The dual coding implies the association of visual annotation with abstract elements. We should watch out of the complexity because the dual coding intensively increases the number of graphic elements.

Cognitive Fit. This critter takes in consideration the users. An experienced user should not have the same writing support as a beginner. This is often chosen by the language designer.

CHAPTER 3 SYNTHESIS AND ANALYZE OF THE STATE OF THE ART

This chapter indents to synthetize the work I presented in the Context and State of the art chapter.

The fundamental concept of the presented work is model. Therefore, models took the first place in the chapter. I start introducing the concept of models with several definitions. In my opinion only the definition of Stachowiak is a complete one. However, I would like to propose my own definition here : "A model is an entity allowing to represent particular characteristics of the system under study, for both cognitive and reducing complxity representation purposes". Afterwards, two classification are proposed, by roles, dividing the models into token and type model and by usage, dividing the models into descriptive, prescriptive and transient model. Here, the type model clasification can be seen as representation of conforming models to some properties, given by the mode, while the type model is a ono-to-one representation of the system. I compare this representation with classes and instance of classes concepts of OO progrmming where the typemodel coresponds to a class, whereas the instance of that class coresponds to the token model. In my opinion the usage classification of models is related to the field of architecture, representing systems that already exists (descriptive model), systems that do not exist (prescriptive model) and systems that exists and that we intend to modify (transient model).

Furthermore, I introduce the meta-model concept and the OMG's meta object facility (MOF). Here again, I would like to compare the "meta" of modeling by the one of OO programming. In fact, like meta-classes, which are used to instantiate classes in OO programming, in model driven engineering, meta-models are used to instantiate models. The MOF stands at the top of the modeling architecture, defining the fundamental concepts of models i.e. class and reference.

Next, the model driven engineering (MDE) is discussed. This paradigm is created as an answer of the increasing software complexity over the years. Models are first class citizens. The underlying purpose of MDE is the improvement of productivity. As specific representation of the MDE approach, the model driven architecture is discussed. This is the OMG's approach as a result to interoperability, productivity and compatibility. Different independency model levels are presented, in order to define furthermore the process of designing software systems by this approach. This process is finalized via diver model transformations, transforming models from one independency level to another. Consequentially, the next section defines the model transformation approach, the features and different categories of the model transformation. This approach aims to transform a model into another or into code i.e. code generation. In my opinion, along with models which are the first class citizens, they represent the core concepts of the MDE.

Within the next sections, I start discussions about another point of view, which is the one of a language. I start introducing what a software language is and I discus about the classification to modeling and programming languages proposed by A. Kleppe. Indeed, differences are shown between these two types of languages, but some disagree with this classification, giving another point of view based on the fact that models can be formal, which brings us to the point that they can have precise, executable semantics. Nevertheless, A. Kleppe was my main source, so I guarded her classification. Afterwards, I discuss about visual classification of software languages, which discriminates languages with textual from languages with graphical syntax, which brings up to the last part of this section, where hybrid languages with both textual and graphical syntaxes are discussed. Defining software languages implies the necessity of defining a language description. Having multiple syntaxes in a language involves the separation of the language syntax into abstract and concrete syntaxes. Actually, it is the language description that holds these syntaxes, with a mapping of correspondences between them. I noticed that: "Since the concrete model is conforming to the abstract, instead of defining completely new concrete model, the abstract model can be proposed for extending to the language designer". At the end I discuss about several formalisms to specify languages. The context free grammar is the first highlighted formalism. In my opinion, this is the closest approach to natural language grammar. So, having in mind this fact, brings us to a conclusion: "Language description is more general than natural language grammar", which guide us to another: "Since language description is the fundamental part of a software language, and it is more general then a natural language grammar, a software language differs from natural language in the way of generality i.e. a software language stands at superior general level". Remember, this is only my opinion.

The next section tries to explain what a domain specific language is, relating it to the MDA approach. One of the basic principles of marketing is: "being specialized in a field raises the profits". I think that if the same principle is used in software engineering, the production rate of the enterprise will rise, compared to the production rate during the usage of a general purpose language. Another advantages and disadvantages are discussed at the end of this section.

Finally, domain specific modeling languages and a tool allowing the possibility of defining a DSML are discussed. The general process of defining a DSML is described. Similar process is used by the tool Diagraph. At the end, a graphical DSMLs classification is proposed and since, the focus is on graphical concrete syntaxes, a cognitive effeteness criteria is proposed.

CONCLUSION

I have discussed the existing works, from the most general to the most specific point of view. My focus aims at semi-automation of grammars and graphical concrete syntaxes for a domain specific modeling language. In other words, a part of the "supplementary information" that should be added to the concrete meta-model follows certain patterns that can be calculated. Hence, I am intending to identify those patterns and to use them in a process of pattern recognition, calculating the maximum possible information, in order to infer a part of the target language grammar i.e. a part of the "supplementary information", from the one side, and from the other, I intend to semi-automatize this process.

The assistance to the grammar generation can occur between the phase of abstract and concrete syntax definition. At this specific point, only the abstract syntax is defined, which we are going to use as an entry to the pattern recognition phase. After the pattern application and grammar generation phase are complete, a concrete syntax meta-model should be generated, enriched with grammar for the language designer.

The process, I am intending to accomplish during internship period, should help the language designer creating the concrete syntax forming together with the abstract syntax a visual, relational grammar by the mean of declarative meta-data (annotation on the M_2 layer). However, the meta-data generated by the previously discuses semi-atomize process is not sufficient for a complete language description. Therefore, the language designer should manually add some meta-data in order to complete the process of creating a DSML.

The purpose of my work is to facilitate the process of creating DSMLs by semi-automatizing the process of generation of the language description i.e. the language grammar.

BIBLIOGRAPHY

[AK03] C Atkinson and T Kuhne. Model-driven development: a metamodeling foundation, 2003.

[B0́4] Jean Bézivin. In Search of a Basic Principle for Model-Driven Engineering. *Novatica Journal Special issue*, V(2):1–5, 2004.

[Ber83] Jacques Bertin. *Semiology of Graphics: Diagrams, Networks, Maps.* University of Wisconsin Press, 1983.

[BG91] Igor A Bolshakov and Alexander Gelbukh. *Computational Linguistics: Models, Resources, Applications*, volume 1 of *Ciencia de la Computación.* INSTITUTO POLITÉCNICO NACIONAL, 1991.

[BJV04] Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. On the Need for Megamodels. In *Proceedings of the OOPSLAGPCE Best Practices for ModelDriven Software Development workshop 19th Annual ACM Conference on ObjectOriented Programming Systems Languages and Applications.* ACM, 2004.

[Bun77] Mario Bunge. *Treatise on Basic Philosophy: Volume 3: Ontology I: The Furniture of the World.* Reidel, Dordrecht, 1977.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. *Architecture*, 45(3):1–17, 2003.

[Cho56] N Chomsky. Three models for the description of language, 1956.

[Cho65] Noam Chomsky. *Aspects of the Theory of Syntax*, volume 119 of *Special technical report (Massachusetts Institute of Technology. Research Laboratory of Electronics).* MIT Press, 1965.

[CI02] D Chang and S Iyengar. Common warehouse metamodel (CWM) UML model. *Object Management Group*, 1(formal/02-05-04), 2002.

[Cle01] C. Cleaveland. Program Generators with XML and Java. *Prentice-Hall*, 2001.

[Com69] Nato Science Committee. Software engineering. (October 1968), 1969.

[Cza05] Krzysztof Czarnecki. Overview of Generative Software Development. *Unconventional Programming Paradigms*, 3566(Unconventional Programming Paradigms):326–341, 2005.

[DGD06] Dragan Djuric, Dragan Gaševic, and Vladan Devedžic. The Tao of Modeling Spaces. *Engineering*, 5(8):125–147, 2006.

[DKV00] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[DRLP11] Davide Di Ruscio, Ralf Lämmel, and Alfonso Pierantonio. Automated co-evolution of gmf editor models. *Software Language Engineering*, pages 143–162, 2011.

[Eva03] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*, volume 70. Addison-Wesley, 2003.

[Fav04] Jean-Marie Favre. Towards a Basic Theory to Model Model Driven Engineering. *System*, 1(4):262–271, 2004.

[Fle06] Franck Fleurey. *Langage et méthode pour une ingénierie des modèles fiable.* PhD thesis, 2006.

[Flo67] R W Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):19–32, 1967.

[GS03] Jack Greenfield and Keith Short. Software factories: assembling applications with patterns, models, frameworks and tools. In Ron Crocker and Guy L Steele Jr., editors, *Companion of the 18th annual ACM SIGPLAN conference on Objectoriented programming systems languages and applications*, volume 9 of *OOPSLA '03*, pages 16–27. ACM, ACM, 2003.

[HN12] Marianne Huchard and Clémentine Nebut. IDM Master 2 : Modèles et métamodèles. 2012.

[Jon] Capers Jones. Programming Languages Table.

[Jul02] Etienne Juliot. Presentation de MDA. 2002.

[K0̈6] Thomas Kühne. Matters of (Meta-) Modeling. *Software Systems Modeling*, 5(4):369–385, 2006.

[Kay07] Michael Kay. XSL Transformations (XSLT) Version 2.0, 2007.

[KCXc03] Alan Kennedy, Kennedy Carter, and William Frank X-change. MDA Guide Version 1.0.1. *Object Management Group*, 234(June):51, 2003.

[Kle07a] Anneke Kleppe. A Language Description is More than a Metamodel. *Syntax*, (612):1–9, 2007.

[Kle07b] Anneke Kleppe. Towards the Generation of a Text-Based IDE from a Language Metamodel. In David Akehurst, Régis Vogel, and Richard Paige, editors, *ECMDAFA*, volume 4530 of *Lecture Notes in Computer Science*, pages 114–129. Springer, Springer Berlin / Heidelberg, 2007.

[Kle08] Anneke Kleppe. *Software language engineering: creating domain-specific languages using metamodels.* Addison-Wesley, 2008.

[Kur08] Ivan Kurtev. State of the Art of QVT : A Model Transformation Language Standard. *Data Engineering*, 5088(ii):377–393, 2008.

[LPDc12] Xavier Le Pallec and Sophie Dupuy-chessa. intégration de métriques de qualité des diagrammes et des langages dans l'outil ModX. *Conference en ingenierie du logicielle (CIEL)*, pages 1–6, 2012.

[Lud03] Jochen Ludewig. Models in software engineering ? an introduction. *Software and Systems Modeling*, 2(1):5–14, 2003.

[MB02] Stephen J Mellor and Marc J Balcer. *Executable UML: A Foundation for Model-Driven Architecture.* Object Technology Series. Addison-Wesley Professional, 2002.

[MOF02] OMG MOF. OMG Meta Object Facility (MOF) Specification v1. 4. (April), 2002.

[Moo09] Daniel L Moody. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, 35(6):756–779, 2009.

[MV04] Pieter J Mosterman and Hans Vangheluwe. Computer Automated Multi-Paradigm Modeling: An Introduction. *Simulation*, 80(9):433–450, 2004.

[NH12] Clémentine Nebut and Marianne Huchard. IDM Master 2 : Transformation des modèles avec Kermeta. 2012.

[Omg06] Omg. Meta Object Facility (MOF) Core Specification. *Management*, 080907(January):1–76, 2006.

[OMG07] OMG. OMG Unified Modeling Language (OMG UML). *Language*, (November):1 – 212, 2007.

[Omg08] Qvt Omg. Meta Object Facility (MOF) 2 . 0 Query / View / Transformation Specification. *Transformation*, (January):1–230, 2008.

[PCHN12] François Pfister, Vincente Chapurlat, Marianne Huchard, and Clémentine Nebut. A proposed tool and process to design domain specific modeling languages. *Advances on cognitive automation at LGI2P / Ecole des Mines d'Alès Doctoral research snapshot 2011-2012*, 47(June):31–35, 2012.

[Pfi13] François Pfister. *Heuristiques architecturales et patrons de conception métier pour une approche fédérée de l'interopérabilité des systèmes : application à un processus d'Ingénierie Système.* PhD thesis, Ecole des Mines d'Alès, 2013.

[RS97] Jan Rekers and Andy Schürr. Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages and Computing*, 8(1):27–55, 1997.

[Sei03] E Seidewitz. *What models mean*, volume 20. IEEE Computer Society, 2003.

[SG04] An Adopted Specification and Object Management Group. Metamodel and UML Profile for Java and EJB Specification. (February), 2004.

[SK03] S Sendall and W Kozaczynski. Model transformation: the heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*, volume formatik-S. Springer, 1973.

[SV09] Eugene Syriani and H Vangheluwe. Matters of model transformation. *School of Computer Science, McGill*, pages 1–20, 2009.

[Voe09] Voelter Markus. Best Practices for DSLs and Model-. *Journal of Object Technology*, 8(6):79–102, 2009.

[WW88] Y Wand and R Weber. An Ontological Analysis of Some Fundamental Information Systems Concepts. In *Proceedings of the Ninth International Conference on Information Systems*, pages 213–224, Minneapolis, Minnesota, USA, 1988.