A Diagrammatic Approach To Deep Metamodelling

Ole Klokkhammer

Master's Thesis in Informatics - Program Development

Department of Informatics
University of Bergen

Department of Computer Engineering
Bergen University College

June 2014

Contents

Abstract

Metamodelling is used at the core of Model Driven Engineering to define Domain Specific Modelling Languages. Atkinson and Kühne has however pointed out several limitations with the current approach to metamodelling and proposed deep metamodelling as a solution to these limitations. Deep metamodelling has in the later years been recognized by several researches, but there is still a lack of proper formularization of the concepts of deep instantiation, linguistic/ontological classification and linguistic extension. Secondly, current tools for deep metamodelling only facilitates either textual modelling or diagrammatic modelling in a single view. No current tool has support for deep metamodelling in a fully diagrammatic modelling environment.

In this thesis, we present a formularization of a fully diagrammatic approach to deep metamodelling through an extension of the Diagram Predicate Framework. Deep instantiation is implemented with use of potencies. The approach includes support for linguistic and ontological classification in a linear meta-hierarchy. The editor can also be used to model linguistic templates, and we have added support for linguistic extensions based on linguistic templates. Lastly, the editor also has support for modelling in a customizable concrete syntax in addition to the corresponding abstract syntax. The proposed solution is demonstrated in a running example.

Acknowledgements

The work in this thesis has been conducted by myself, but it would not be possible without the invaluable patience and guidance from my supervisor, Yngve Lamo. I would also like to send my greatest appreciations to Juan de Lara, which was my co-supervisor during my stay at the Autonomous University of Madrid. Both the supervision of Yngve and Juan has been of great help to establish the formularization of deep metamodelling in this thesis. I would also like to thank Florian Mantz and Xiaoliang Wang for providing help on the technical side of things.

A special thanks to my good friends Peninnah, Fabien and Uma for proof-reading and my parents Oddvar and Lillian for encouraging me to keep working.

Bergen 1. June 2014

CHAPTER 1

Introduction

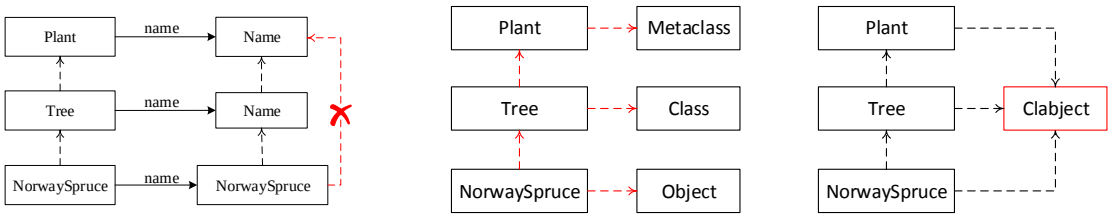
1.1 Motivation

Since the very beginning of software engineering, there has been a continous focus on developing software that is of high quality and low cost. Early programming languages such as FORTRAN [1] for instance, made it possible to shield the programmer from writing machine code. Some years later, more expressive object-oriented programming languages (OOP) appeared [2]. It was now possible for developers to wrap similar concepts into classes, and instantiate them as objects. Developers could also create reusable class libraries, which made it easier to reuse concepts that had been defined earlier. Developers could now reuse mature class libraries of high quality and speed up the development process and provide software of higher quality and a lower cost. However, this has in the later years evolved into a complex web of frameworks. We can now see that large scale systems might contain thousands of interconnected components, which in turn adds up to an ever so increasing software complexity. It is also difficult to correctly and optimally connect these components, and the potential side effects can be difficult to predict and debug. [3]

A promising approach to overcome the increasing platform complexity of frameworks, is the concept of Model Driven Engineering (MDE) [4]. MDE involves the idea of creating domain models, meaning abstract representations of knowledge about the system rather than focusing on underlying mechanisms such as algorithms. MDE is meant to increase productivity and maximize compability between different platforms, as well as simplifying the process of software design. Typically it has been common to express domain models in languages such as the Unified Modelling Language (UML) [5], but only in the process of gaining an overview of the design as well as for documentation purposes. MDE aims at taking this one step further by making the domain modelling a part of the programming process by adding code generation [6]. MDE also aims at using the domain models together with metamodelling, which is the concept of creating models based on models to raise the level of abstraction [7]. In addition to this, MDE aims at including model transformation engines, model checking

mechanisms to prevent errors at an early stage and more [8]. However, MDE does consist of a number of different research areas. In this thesis we will only focus on metamodelling and its limitations.

Even though MDE facilitates many of the weaknesses of OOP through domain models and metamodelling, there are still some major limitations that are holding it back from being fully successful. We are only briefly explaining these weaknesses in this section to facilitate the motivation for this thesis, while a more thorough explanation is provided in chapter 3. Figure 1.1 below, illustrates the three limitations as they were elaborated by Atkinson and Kühne [9].



(a) Replication of concepts. (b) Multiple classification. (c) Classifier Duality.

FIGURE 1.1: Limitations with traditional metamodelling.

The first limitation is the replication of concepts problem, which arises from the need to redefine concepts multiple times. To be able to express concepts across more than two metalevels, we have to redefine concepts at each metalevel until we can use it (see fig 1.1(a)). The second weakness is the multiple classification problem as seen in figure 1.1(b). The multiple classification problem arises from the need to express both linguistic and ontological classification of a model element. The third weakness arises from the need to capture dual type-facets of certain model elements, such as class and object.

To solve these three problems, we follow the definition of deep instantiation as it was established by Atkinson and Kühne [9]. Deep instanitation will make it possible to carry information across several metalevels. There already exists tools with support for deep instantiation such as MetaDepth, but this tool is only textual and not diagrammatic [10]. Modelling diagrammatically means we are modelling by using a graphical representation of the model. A textual model may be easy for a computer to read and generate code from, but from a human perspective it is often easier to imagine a model in a diagrammatic environment. It is more natural for a human being to create models that are more similar to real world objects, instead of modelling based on a computer's perspective.

1.2 Structure Of Thesis

This thesis is structured the following way:

Chapter 2 - Model Driven Engineering
This chapter provides background information of what the concept of Model Driven Engineering means, along with some motivating examples on the way.

Chapter 3 - Problem Statement And Methodology
In this chapter, we explain the current limitations with the approach we presented in chapter 2, and provide some possible solutions to these limitations. This chapter will set the stage for the rest of this thesis by providing a problem description and the research methodology that will be used later in this thesis.

Chapter 4 - Comparison Analysis
This chapter consists of a comparison analysis of existing diagrammatic tools such as the Diagram Predicate Framework, as well as tools with support for deep metamodelling such as Metadepth and Melanee. The results obtained from this comparison is used to establish a starting point for the implementation of a diagrammatic tool for deep metamodelling.

Chapter 5 - Design and Implementation
This chapter consists of the design and implementation phase of abstract syntax of the tool. This chapter presents the design and implementation phase of the concrete syntax of the tool.

Chapter 6 - Demonstration
In this chapter we provide a demonstration of the final editor, illustrating both the abstract and concrete syntax of the editor.

Chapter 7 - Conclusion And Further Work
This chapter concludes the work that has been done in this thesis, and provides some suggestions for possible further work.

CHAPTER 2

Model-Driven Engineering

This chapter provides an introduction to the general situation in software engineering today, and introduce model driven engineering. We are also providing some motivating examples for using domain models over traditional general purpose models. We are presenting domain specific modelling languages and explaining their abstract and concrete syntax. We will also explain diagrammatic modelling and graph based modelling, and last of all provide some examples of existing initiatives for MDE.

2.1 The Current Situation

Since the first computer programs were made, there has been a continuous focus on developing software that is of high quality to low cost. Early programming languages such as assembly and FORTRAN for example, came to replace machine code with programming language constructs aimed at shielding developers from writing complex machine code. Later on, OOP arrived with a goal to raise the abstraction level and facilitate Application Programming Interfaces (API) [11] to provide reuseable class libraries. By developing reusable class libraries and application frameworks, the need to reinvent common and domain specific services such as transactions, security and event notification was minimized [3]. Developers were now shielded from complexities associated with earlier languages with the use of OOP.

However, the process of reusing class-libraries has in later years evolved into a complex web of interconnected components. A large scale system such as .NET [12] and J2EE [13] contains thousands of interconnected components and requires considerable effort to maintain. It is also difficult to connect these components, and the potential side effects can be difficult to predict and debug. In addition to this, since these platforms evolve rapidly, developers are spending considerable amount of time porting code to newer versions of the same platform or to different platforms.

As the platform complexity is increasing, we again see that the software industry is reaching a complexity ceiling. Newer platforms has become so complex that developers spend years trying to master their APIs, while only ending up being familiar with a subset of the platform technologies

out there. Another issue is that OOP languages requires the developer to focus on underlying programming details, resulting in a loss of overview. The result is that developers end up having a hard time maintaining focus on conceptual relationships and overall system correctness. Therefore, the main problems with OOP languages as we see them today, are:

- Platform complexity
- Ability to express domain concepts effectively

A promising approach to overcome these limitations are the concept of Model Driven Engineering (MDE).

2.2 Model Driven Engineering

Model Driven Engineering aims at using Domain models to raise the level of abstraction, such that modellers no longer needs to focus on underlying programming mechanisms such as algorithms or platform specific design patterns. MDE is meant to increase productivity and maximizing compability between different platforms as well as simplifying the process of software design. Typically these domain models have been expressed using General Purpose Languages (GPL) [3], but it has been argued that it does not unfold the full potential of MDE as GPLs are often too generic to express deeper application domain concepts and design. Therefore, a Domain Specific Modelling Language (DSML) can be used to precisely express these domains concepts effectively instead [14].

DSML's can be categorized as *prescriptive* or *descriptive*, where a *prescriptive* model is a model that provides a description of a system before it is produced, and a *descriptive* model is a means of documentation of the system. Traditionally DSML's have been developed for documentation purposes and to gain overview over the system before it is made. MDE aims at utilizing both the *descriptive* and *prescriptive* side of modelling to facilitate the best of both worlds. First by developing domain models, then by generating code based on these models.

These domain models can also be used together with metamodelling, which is the concept of creating metamodels restricting the instances of a model. The definition of a metamodel suggests that modelling has occurred twice: a metamodel and a model [7]. In OOP this has traditionally meant that developers only have been concerned with two metamodels, type and instance, Class and Object. However, the main concept of metamodelling means that we can create models of models at as many levels as we like. The basic idea behind it is to essentially create a type system in form of a meta-hierarchy, where each model conforms to its metamodel and the top most metalevel is the most abstract metalevel. By modelling DSMLs using metamodels we will in turn gain a lower platform complexity, and developers can more easily overcome the biggest disadvantages with current OOP programming languages. However, we will go more into detail

on the concept of metamodelling in section 2.4. To summarize, MDE aims at:

- Making it easier for developers to focus on the problem domain itself rather than writing error prone code,
- Increasing productivity,
- Maximizing compatibility between domains or systems by reusing models,
- Simplifying the design process.

To describe how these advantages are possible, we will first provide a description of how DSMLs are defined.

2.3 Domain Specific Modelling Languages

A Domain Specific Modelling Language (DSML) is a modelling language tailored towards a specific domain of concern rather than for general purpose. It aims at raising the level of abstraction and at the same time providing modelling languages that is more closely related to the domain of concern. As an example, consider that you are developing a system together with a team. Each person working on the system may have different views and perspectives. One colleague may look at the system from an engineering viewpoint, another colleague may look at the same system from an enterprise viewpoint. A DSML will help in this process by making it possible for each modeller to model the system such that the domain concepts of concern are properly described.

For a language workbench to utilize DSML's, it is necessary with a parser that can parse an external representation of the metamodel in form of a source file, into an internal representation. The internal representation can then be visualized in an abstract or concrete syntax [15], either textually or graphically depending on the tool. The abstract and concrete syntax of DSML's will be described in section 2.3.2, but first of all we present how DSMLs are internally represented and persisted in a language workbench.

2.3.1 Internal Representation And Persistence

A modelling tool typically defines DSMLs by means of an internal representation and a persisted model. Figure 2.1 on the next page illustrates a simple overview of a persisted model that can be parsed to an internal representation. The internal representation is handled by the tool and can be edited in a textual or graphical user interface (or both).

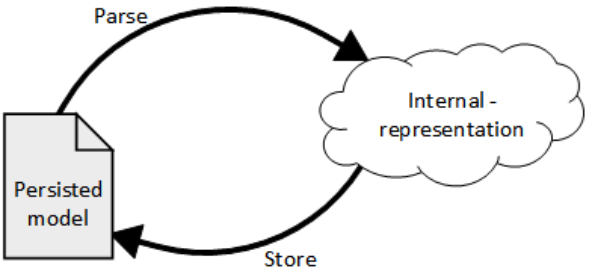


FIGURE 2.1: Internal model and persistence to an external model.

The persisted model is simply a way for the computer to understand and store the model. The internal model can be persisted in a external representation in form of a file. A popular format for this is the XML Metadata Interchange format (XMI), which is a XML format aimed at representing models [16]. The idea is that a persisted model is being parsed into an internal representation, which can be represented in an abstract and concrete syntax.

2.3.2 Abstract And Concrete syntax

Defining a DSML can be decomposed into three parts: the definition of the abstract syntax, concrete syntax and the mapping between abstract and concrete syntax. The abstract syntax basically defines the significant parts of the model. It describes the relationships between concepts and rules for well-formedness, and is parsed for model transformations such as code generation or similar. The concrete syntax on the other hand, defines how the model is presented to a modeller. The concrete syntax is simply a visualization strategy for representing the model in a user friendly way. The concrete syntax may be textual, graphical or a combination of these. As an example, consider the abstract syntax representation of a simple sum expression as illustrated in table 2.1 below.

2 plus 3

TABLE 2.1: Abstract syntax of a sum expression.

The same expression can also be expressed in different concrete syntaxes as illustrated in Table 2.2 on the next page. The concrete syntax is simply different ways to express the abstract syntax. The aim with the concrete syntax is to visually represent the abstract syntax in as intuitive way as possible from a modellers perspective. The concrete syntax does not even have to illustrate all the elements in the abstract syntax.The main idea is that we need an abstract syntax to define the core concepts of DSMLs. The corresponding concrete syntax is made to represent the models in as intuitive way as possible from a modellers perspective.

2 + 3	Infix
(+ 2 3)	Prefix
(2 3 +)	Postfix
The sum of 2 and 3	English language

TABLE 2.2: Different concrete syntaxes of the sum expression.

The illustrated abstract and concrete syntaxes in table 2.1 and table 2.2 are however only expressed in a textual syntax. In this thesis, we argue that a diagrammatic modelling approach is easier for humans to conceptualize. In the next section, we therefore present the aforementioned abstract and concrete syntaxes in a diagrammatic visualization instead.

2.3.3 Diagrammatic Modelling

As mentioned above, a DSML can be expressed in a textual or graphical syntax (or both). The sum expression example we illustrated in Figure 2.1 is one example of a textual syntax for a DSML, but this expression can also be expressed using a graphical syntax. The graphical syntax of a DSML is often specified using diagrams, but the term diagram may have different meanings depending on the context. In the oxford dictionary online [17], the definition of a diagram is *'A simplified drawing showing the appearance, structure, or workings of something; a schematic representation'*. In software engineering, a diagram is simply a structure based on a graph with Nodes and Edges. In this thesis, we will use the term from software engineering to define diagrams, and we also define that a graphical diagrammatic syntax of a DSML is therefore defined using a graph based structure.

Diagrammatic models have already been adopted for use in modelling tools for many years now, from flowcharts in the seventies to Petrinets [18] as well as Computer Aided Software Engineering (CASE) tools [19] in the eighties. The CASE tools can be seen as an early attempt to MDE as we know it today, and focused on developing software methods and tools that made it possible for developers to express software design with GPLs such as business process models, dataflow diagrams and similar. The CASE tools was widely researched, but had some notably problems such as poor mapping between their graphical tools and the underlying platforms [3]. Secondly, the amount of generated code that needed to be generated was of such an amount that it was difficult to grasp for the current technology at the time. The CASE tools was therefore difficult to develop, debug and maintain.

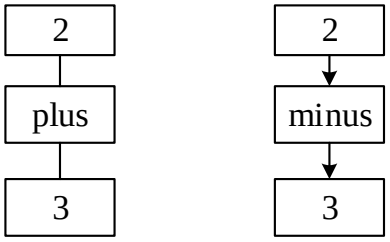
Nowadays, MDE applies lessons learned from what went wrong with the CASE tools, and tries to facilitate these problems in a number of aspects. First of all by utilizing DSMLs to help specifying domain concepts that otherwise would be difficult or impossible to model with general purpose languages. Secondly, MDE aims at tailoring DSMLs together with

metamodelling to match the domain specifications more precisely. MDE also facilitates diagrammatic modelling directly at the domain of concern, which helps flattening out the learning curve. It also ensures that a broader range of perspectives of the domain requirements are fulfilled, such as the perspective of a system architect and a web designer. Thirdly, MDE facilitates transformation engines for model transformations [20], constraints to ensure correctness and perform model checking to prevent errors early in the software lifecycle and much more. As a result it is much easier to develop, debug and maintain models.

Earlier in this section we defined that a diagram is a structure based on a graph. When modelling diagrams, we would by definition also model a graph. This type of modelling is therefore most commonly referred to as graph-based modelling.

2.3.4 Graph-based Modelling

In graph theory, which is the study of graphs, a graph is defined as a structure consisting of nodes and lines called edges connecting them. The formal definition is that a Graph is an ordered pair G = (V, E), where V is a set of vertices or nodes, and E is a set of edges or lines connecting them. Graphs can be either directed or undirected, meaning whether the edges between nodes in a graph has a specific direction or not. An undirected graph does not have a specific direction, meaning the edges can go both ways. In a directed graph, edges can only go a specific direction. Figure 2.2(a) below, illustrates the sum expression example as an undirected graph. In the case of adding two numbers together, it does not matter what direction the edges has. 3 plus 2 has the same meaning as 2 plus 3. However, if we subtract the two numbers instead, we see that 2 minus 3 yields a different result than 3 minus 2.



(a) Undirected graph. (b) Directed graph.

FIGURE 2.2: An illustration of an undirected and a directed graph.

An undirected graph is often what we refer to as a simple graph. There are many more definitions of graphs in the literature, e.g., directed multigraphs, attributes graphs etc. In this thesis we focus on directed multigraphs, which is a directed graph with added support for multiple edges

between nodes. The formal definition of multi-graphs as it was taken from [21] is:

Definition 2.1 (Graph)**.** *A graph $G = (G_N, G_A, src^G, trg^G)$ consists of a set G_N of nodes (or vertices), a set G_A of arrows (or edges) and two maps $src^G, trg^G : G_A \longrightarrow G_N$ assigning the source and target to each arrow, respectively. $f : X \longrightarrow Y$ denotes that $src(f) = X$ and $trg(f) = Y$.*

Earlier in this thesis, we mentioned that MDE aims at utilizing metamodelling together with DSMLs to lower the platform complexity and raising the level of abstraction. In the next section, we present the concept of metamodelling and its practical use in MDE.

2.4 Metamodelling

As briefly mentioned in the beginning of this chapter, the definition of a meta-model suggests that modelling has occurred twice: a metamodel and a model. Specifically, a metamodel describes the set of modelling constructs available to the model, as well as rules for combining these to create valid and well-formed models. A modeller defines a metamodel, which in turn restricts the set of valid instances of the model.

As described in the beginning of this chapter, OOP has traditionally only been concerned with two metalevels, Classes and Objects, types and instances. MDE aims at using metamodelling with as many metalevels as necessary, with the most abstract concepts at the top metalevel, and the least abstract concepts at the bottom metalevel. Each model is an instance of the metamodel adjacent above, and is specified by the corresponding metamodel and its defined rules (e.g. constraints).

In OOP it has been popular to document software systems using languages such as the Unified Modelling Language (UML), which is a family of languages used to describe different structural or behavioural aspects in OOP. UML consists of class diagrams, use case diagrams and sequence diagrams amongst others. Each UML diagram tailored for different documentation purposes. Traditionally MDE has also specified models by using UML based languages as well. As an example, consider the two metalevel hierarchy in figure 2.3 on the next page, which illustrates a two level meta-hierarchy with a partial UML class diagram as its metamodel. The metamodel is a directed graph with two nodes, Class and Attribute, and an edge (attribute) between the Class and the Attribute. The model defines the instances of the elements in the metamodel; in this case the nodes Plant and Name and the edge (name) between them.

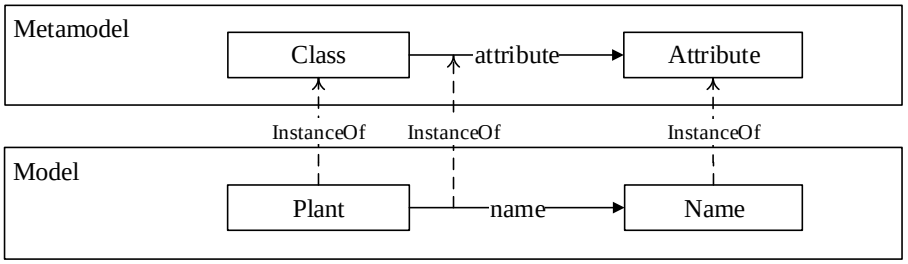
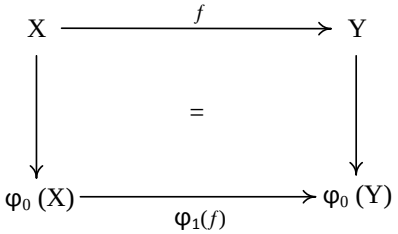


FIGURE 2.3: A basic two-level metamodelling example

Figure 2.3 above illustrates a simplified form of a UML Class diagram and its corresponding instances, modelled in a diagrammatic syntax. As stated earlier in this thesis, a diagram is a graph, and the metamodel in figure 2.3 is thereby modelled with a directed multi-graph. We also realize that the model, which is an instance of the metamodel, also is a directed graph in itself. Each instance in the model (node and edge) is typed by the corresponding model element in the metamodel. If all instances in a model is correctly typed by its corresponding metamodel, we say that the model conforms to the metamodel.

To provide a formalism to the instance of relationship between an instance and its type, we define the instance of relationship by graph homomorphisms. In graph theory, the definition of a graph homomorphism is a mapping between two graphs in respect to their structure. In practice it means it is a mapping between nodes and edges in a model to its corresponding types in the adjacent metamodel above. The formal definition of graph homomorphism as it was taken from [21] is:

Definition 2.2 (Graph homomorphism)**.** *A graph homomorphism $\varphi : G \longrightarrow H$ is a pair of maps $\varphi_0 : G_0 \longrightarrow H_0$, $\varphi_1 : G_1 \longrightarrow H_1$ which preserve the sources and targets, i.e. for each arrow $f : X \longrightarrow Y$ in G we have $\varphi_1(f) : \varphi_0(X) \longrightarrow \varphi_0(Y)$ in H, such that the following diagram commutes:*



This gives a *structure-preserving* mapping between two graphs, in this case between the instance graph and type graph. Through the definition of metamodelling, we can define type-hierarchies in which the element in the metamodel restricts and defines the possible model elements in the corresponding model. In fact the model as illustrated in figure 2.3 above, can be used as a metamodel itself, and modellers can instantiate a new

model based on this metamodel. The result is a metamodelling stack of metamodels, where the topmost metamodel defines the most abstract concepts, and the bottommost metamodel defines the least abstract concepts as illustrated in figure 2.4 below.

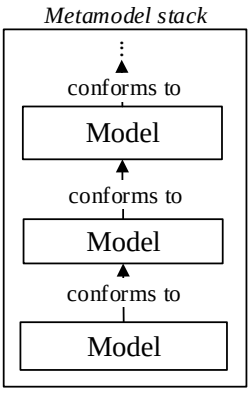


FIGURE 2.4: A linear metamodelling stack

In *traditional metamodelling*, models at each metalevel conforms to the metamodel adjacent above, which is often referred to as *linear metamodelling*. The topmost metamodel is usually defined by itself and is thereby *reflexive*, meaning it conforms to itself. In traditional metamodelling, the depth of a metamodelling stack is also fixed, most commonly numbered from 0 at the bottommost metalevel and increasing upwards the metamodelling stack. The exact role of a metamodel and the instance of relationship has however not been appropriately addressed yet, but there are two main forms of defining the relationship between metamodels; *loose* metamodelling and *strict* metamodelling.

2.4.1 Loose Metamodelling

In loose metamodelling, we do not follow the principles that a model is an instance of a metamodel, but instead we make instances of a type wherever we find a need to mention them. Although this makes the initial specification of the meta-hierarchy much simpler as one can create instances of elements wherever its most natural to create them, it raises some other significant problems. The first problem is that it will blur the level boundaries between the metalevels - a model element is no longer determined by an element in its metamodel. There is no precise meaning to the instance-of relationship either, and all kinds of relationships can by definition cross the boundary between metalevels. Instead of having a well-formed and structured meta-hierarchy we will end up with a web of complex interconnections between different metalevels. The metalevels will instead work as a packaging mechanism for defining common features. This is a good thing in itself as grouping common features into the same package has long been established, but the instance-of relationship not only becomes

confusing, it is also misleading. Defining that a model is an instance of a metamodel, while the same time defining that an element may not even be classified by an instance-of relationship simply provides a misleading meta-hierarchy. The result is that the complexity of relationships between elements is simply too high.

2.4.2 Strict Metamodelling

Strict metamodelling in contrary to loose metamodelling, means that if a model is an instance of a metamodel, every modelling element is an instance of exactly one element in the metalevel above. This means that in strict metamodelling, if a relationship crosses the boundary between metalevels, one holds on to purely instance-of relationships and not by any other type of relationship like those that might occur in loose metamodelling. It is of utmost importance to define that crossing the boundary of a metalevel is synonym to an instance-of relationship to maintain the concept of metamodelling. Otherwise we would no longer have a multi-hierarchy, but it would collapse to a single level [9]. We have therefore decided to focus on strict metamodelling to maintain a consistent and well-defined meta-hierarchy. The formal definition of strict metamodelling is:

Definition 2.3 (Strict Metamodeling)**.** *In an n-level modeling architecture, M_0, M_1, ..., M_{n-1}, every element of an M_m - level model must be an instance of exactly one element of an M_{m+1} - level model, for all $0 \geq m <n - 1$, and any relationship other than the instanceOf relationship between two elements X and Y implies that level(X) = level(Y).*

When it comes to defining the top-level metamodel however, one of the most widely adopted approaches is OMGs Meta-Object-Facility (MOF) [22], which is discussed further in the next section.

2.5 Meta-Object-Facility

In MDE, the top-level metamodel has typically been defined by the Meta-Object-Facility (MOF), which is a standard originally made by the Object Management Group (OMG) to provide a type system for the CORBA architecture and a set of interfaces to create new types and to edit the created types [23]. The intention with MOF is to create a standardized way to describe data about models, and can be illustrated as a four-level meta-hierarchy such as in figure 2.5 below [24].

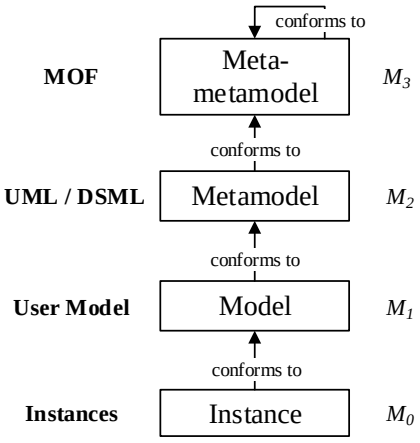


FIGURE 2.5: OMGs four-level meta-hierarchy.

Figure 2.5 above, illustrates OMGs four-level meta-hierarchy (M_3 to M_0). In this hierarchy, models conform to the metamodel of UML, and metamodels conform to MOF. MOF is the topmost model, and is thereby *reflexive* and conforms to itself. OMGs four-level meta-hierarchy is currently one of the most popular practices, and OMG has currently defined two variants of MOF:

1. EMOF for Essential MOF
2. CMOF for Complete MOF

There has also been sent a request for a third viariant, SMOF (Semantic MOF), but since EMOF is the most relevant variant in this thesis, we will not discuss SMOF any further.

Figure 2.6 on the next page illustrates a class overview of the current EMOF implementation. EMOF uses *class* notation to describe model elements in its instances, but the EMOF *classes* must not be confused with OOP *classes*. EMOF *classes* is simply a means to define concepts, while OOP *classes* are used to define objects. The *classes* in EMOF is however often used to describe OOP metamodels such as UML diagrams.

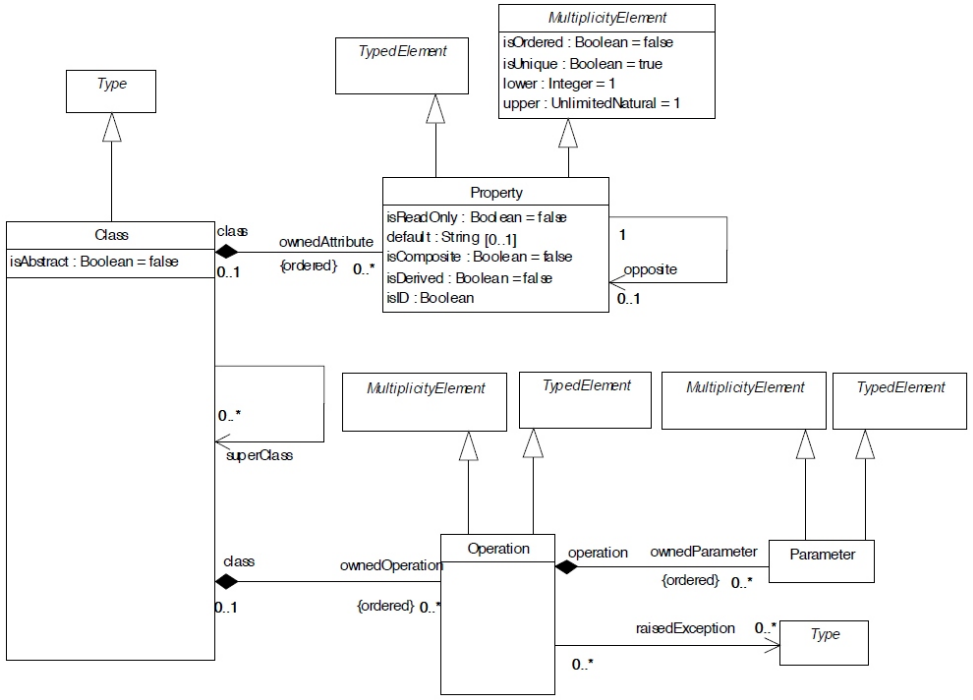


FIGURE 2.6: EMOF classes (meta-metamodel).

We will not describe EMOF in great detail here, but by looking at figure 2.6 above, we can see that EMOF classes contains *Property* that inherits from *MultiplicityElement*. The idea with the multiplicity-element is to make it possible to define multiplicity constraints of model elements directly in the model. This kind of constraints is often referred to as *structural* constraints, meaning the constraints are modelled in the structure of the metamodel. In other cases, structural constraints may not be enough, and we need to define additional attached constraints to define more complex restrictions on the model. Attached constraints are typically written in textual constraint languages such as the Object Constraint Language (OCL) [25], or programmatically in for example Java. The metamodel in figure 2.7 below, illustrates an example of an instance of EMOF in form of a UML Class diagram, along with a corresponding instance of the class diagram with additional constraints.

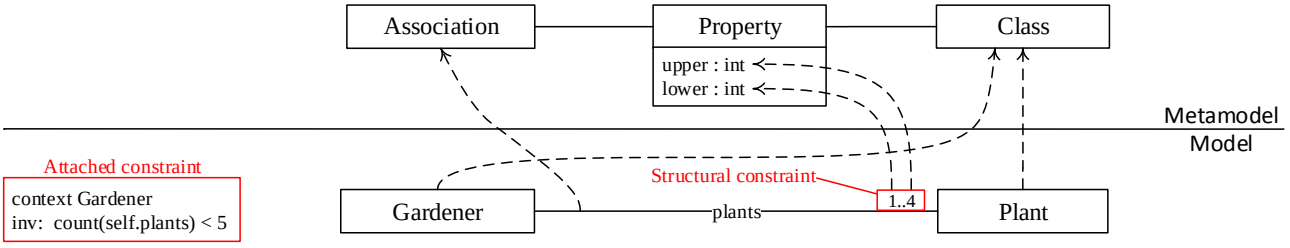


FIGURE 2.7: Structural and attached constraints.

We have developed a model with a *Gardener*, and an association to a *Plant*. To illustrate the restriction of gardeners to have a minimum of 1 plant and a maximum of 4 plants, we have added a structural constraint to the Association named *plants*. We have also added an additional attached constraint written in OCL, which is restricting Gardeners to have more than 4 plants. A language workbench could typically use constraints together with model checking mechanisms to detect errors early in the software lifecycle [3]. We will however not put a large emphasis on constraints in this thesis, but rather focus the main structure of the metamodels instead.

In the next section we are presenting some of the better known existing initiatives for MDE. However, we are only presenting a brief overview of the initiatives here, while a more thorough overview of existing initiatives can be found in [26].

2.6 Existing Initatives

One of the most popular approaches to MDE today is the Model Driven Architecture (MDA) [3]. MDA was initiated by OMG in late 2000 to provide a core foundation for model driven engineering tools, by providing a set of guidelines for structuring models. MDA defines Platform Independent Models (PIM), to provide a modelling platform that is namely platform independent. By specifying PIMs, one can later through model transformations transform the model to a Platform Specific Model (PSM) that a computer can run.

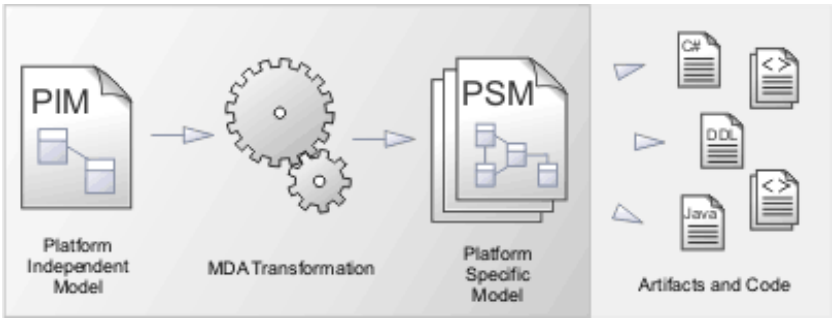
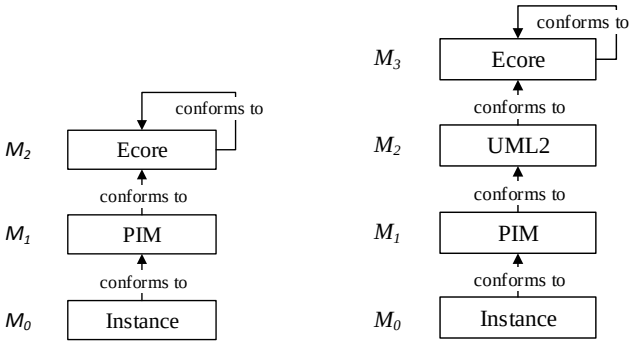


FIGURE 2.8: A Model transformation example from a PIM to a PSM.

Figure 2.8 illustrates a platform independent model and generation of a platform specific model by using model transformation. MDA can also apply to other areas such as Business process modeling and is based on multiple standards such as UML, MOF, the Object Constraint Language (OCL) [25] and XMI. One of the most well-established implementations of the MDA architecture is the Eclipse Modelling Framework.

2.6.1 The Eclipse Modelling Framework

The Eclipse Modelling Framework (EMF) is an Eclipse based modelling framework, at the center of Eclipse's modelling technologies [27]. EMF started as a MOF implementation, and has essentially become an open-source code-generation facility aimed at making the design and implementation of a structured datamodel easier. Figure 2.9(a) below, represents how Ecore fits into the MOF hierarchy as it is described in section 2.4.



(a) Three-level meta-hierarchy. (b) Four-level meta-hierarchy.

FIGURE 2.9: Representation of how Ecore fits the MOF hierarchy.

The top level defines the reflexive Ecore model [27] that conforms to itself, and the M_1 level specifies the platform independent model. If the user want to specify the PIM using UML2, then the UML2 metamodel will be placed between the Ecore metamodel and PIM as seen in figure 2.9(b). EMF provides code generation facilities aimed at building tools based on a Platform Independent Model, in other terms by specifying an PIM (see figure 2.10 below). This PIM can be imported into a generator model, which is used by EMF to generate the codebase for our tool.

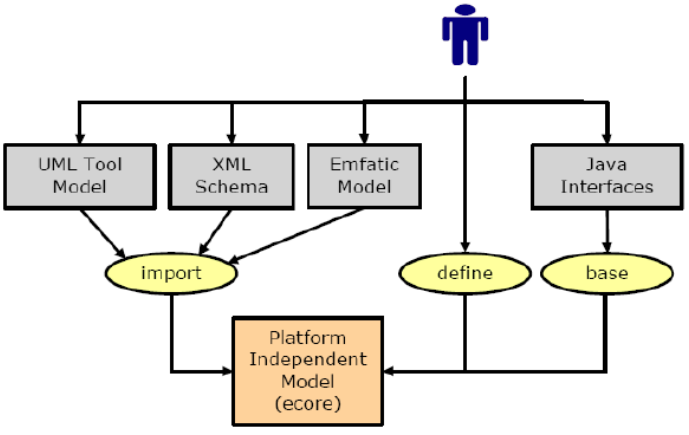


FIGURE 2.10: Definition of a platform independent model in EMF.

As we can see in figure 2.10, a PIM can be specified from a number of sources. The PIM can be imported in form of a UML diagram, XML schema,

it can be based on Java Interfaces or the user can specify it themselves. The PIM can also be specified by an Emfatic model, which is a textual syntax for ecore models. When the platform independent model is created, we can import it to the generator model in EMF, which generates code for our editor. The code-output generated from the generator model consists of three packages: EMF.model, EMF.edit and EMF.editor, as illustrated in figure 2.11 on the next page.

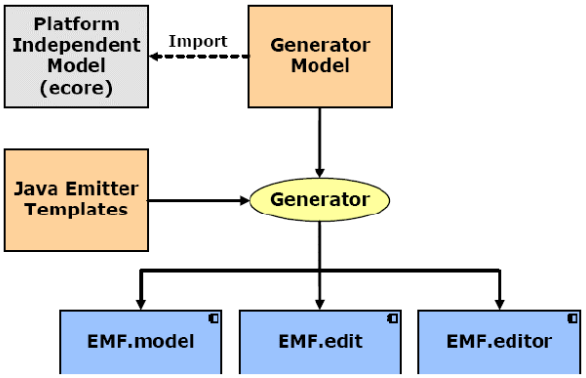


FIGURE 2.11: Generator model in EMF.

EMF also provides tools for interoperability with other tools using a default serialization strategy based on XMI. This means basically what we described in section 2.3.1, we can create instances where the internal representation can be persisted (serialized) into an external model based on XMI. Figure 2.12 below illustrates a simplified view of the metamodelling hierarchy in EMF.

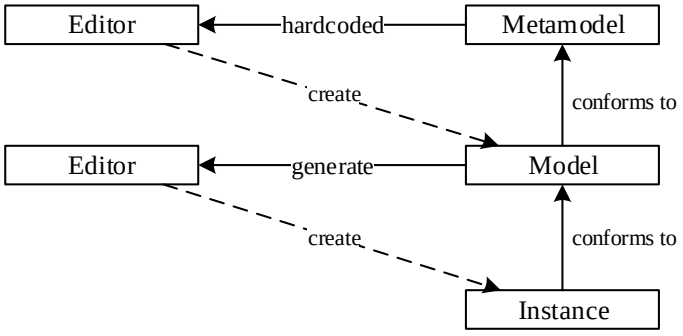


FIGURE 2.12: An illustration of a simplified view of the EMF metamodelling hierarchy.

In EMF, the metamodel is hardcoded and used to define PIM's. We can import the PIM into a generator model and create the model. With the model, we can create instances that represents the software of concern. The eclipse modeling framework is today widely recognized, and has many sub-projects. One of these sub-projects is the Diagram Predicate Framework (DPF) workbench, which we present in the next section.

2.6.2 Diagram Predicate Framework

The Diagram Predicate Framework (DPF) is a research project initiated by Bergen University College (BUC) and University of Bergen (UOB) in 2006, involving several researchers from Norway and Canada [28]. DPF is based on the Generalised Sketches formalism by Zinovy Diskin [29], and is a graph based specification format aiming at formalizing concepts in MDE through category theory and graph transformations [21]. It has previously been branded as Generalized Sketches (GS) and Diagrammatic Predicate Logic (DPL). However, DPF is largely inspired by categorical and First Order Logic (FOL) [30], and aims at using these concepts to facilitate the main concepts of MDE. In DPF, models are represented by diagrammatic specifications in which a specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ consists of an underlying graph S along with a set of atomic constraints $C^{\mathfrak{S}}$ [21]. The graph represents the underlying structure of the specification, and the atomic constraints represents restrictions attached to this structure. DPF provides a completely diagrammatic approach to MDE, and its graph-based nature provides a foundation for one of the greatest strengths in DPF; it can be used to develop patterns of other modelling languages such as UML class diagrams, petri-nets, business process diagrams and many more.

2.6.3 The DPF Workbench

The DPF Workbench is the reference implementation of the Diagram Predicate Framework, and represents a diagrammatic editor for the specification of metamodels [28]. The editor is based on EMF, but is extended to support an arbitrary number of metalevels along with code generation facilities. The DPF workbench supports a fully diagrammatic modelling environment based on the Graphical Editing Framework (GEF). Figure 2.13 below, illustrates the current component architecture of the DPF Workbench.

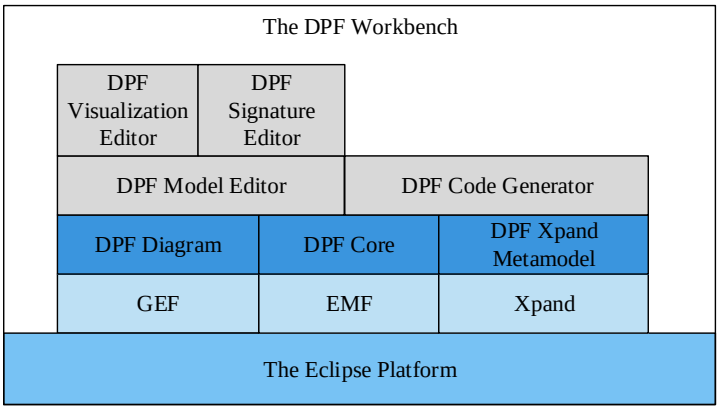


FIGURE 2.13: An illustration of the current architecture of the DPF Workbench.

As we can see from figure 2.13, the DPF Workbench is based on EMF with GEF as visualization framework and XPand for code generation. The DPF

Model Editor is the editor used to diagrammatically develop metamodels in an abstract syntax, which can be further specified with signatures made in the signature editor. There also exists a Visualization editor which aims at providing the workbench with a customizable concrete syntax. We only briefly mention the DPF workbench and its plugins here, while a deeper insight into the DPF Editor and the Visualization Editor is presented in chapter 4.

CHAPTER 3

Problem Description And Methodology

In this chapter, we are going to take a deeper look at traditional metamodelling. We start the chapter by pointing out some major limitations with traditional metamodelling, along with possible solutions from the literature. Finally we present the problem description for this thesis and the research method that will be used to overcome these problems.

3.1 Three Limitations

MDE provides the developer with tools that can lower platform complexity and give the developer ability to express domain concepts in a more effective way than what is possible with OOP. We can now use domain specific modelling languages together with metamodelling, apply constraints to ensure the models fulfill the domain requirements and perform model-checking routines to detect errors early in the software lifecycle. We can run model to model transformations, model to text transformation for code-generation and many more. While this is only a subset of the functionalities a MDE tool could provide, there are still limitations with this approach that needs to be addressed. Atkinson and Kuhne defined three limitations with traditional metamodelling in [9], where modellers preference to strict metamodelling was the underlying reason for the three limitations. The three limitations that needs to be addressed as a result of strict metamodelling are:

1. The replication of concepts problem,
2. The multiple classification problem, and
3. The classifier-duality problem

The replication of concepts problem arises because only a shallow instantiation mechanism is supported in traditional metamodelling. An instance can only be created if its type is contained in the metamodel adjacent above, and deep characterizations of model elements is therefore not possible in traditional metamodelling. This means that certain model elements has

to be replicated for modellers to be able to access them further down the meta-hierarchy. The second problem is the problem of multiple classification, which arises from the need to classify elements with more than one classifier. The third and last limitation arises from a need to capture different meanings of certain model-elements. In OOP this can be the need to capture classlike and objectlike properties of a model-element. These limitations will be discussed in the forthcoming sections, starting with an explanation of the role of metamodels and the reason modellers prefer strict over loose metamodelling.

3.1.1 The Role Of A Metamodel

Even though significant research has been made in the field of model driven engineering, there is currently little agreement on what form and what role metamodelling should play [9]. For example as mentioned in section 2.4, the instance of relationship between metalevels have not yet been fully established. In loose metamodelling, the instance of relationship may not even be the most common relationship between two elements. To illustrate loose metamodelling in practice, we have created an example of a three level meta-hierarchy using loose metamodelling in figure 3.1 below.

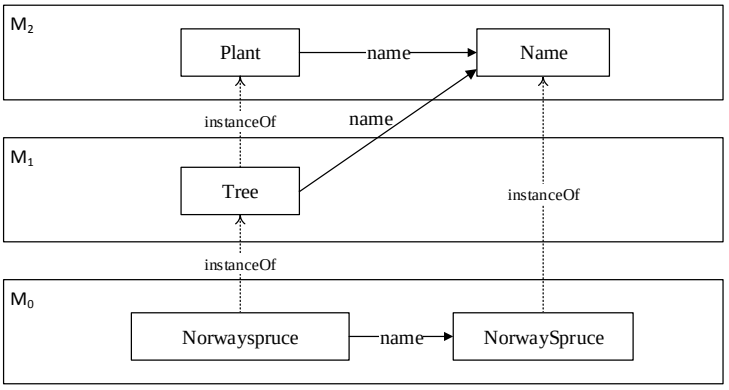


FIGURE 3.1: An example of a three-level meta-hierarchy using loose metamodelling.

In loose metamodelling, both references and instance-of relationships can cross the boundaries between metalevels, thus resulting in a complex web of interconnected components. The metamodels are reduced to simply a packaging mechanism for similar structures, and the meta-hierarchy collapses in the process. Based on this, we see that it is of utmost importance to define that crossing the boundary between metalevels is synonymous to an instance of relationship. Otherwise we will no longer have a multi-level meta-hierarchy, but the hierarchy would collapse into a single level [9].

The immediate solution to this problem is by using strict metamodelling as we defined it in section 2.4.2. By defining strict metamodelling, we thereby keep a consistent meta-hierarchy where it is evident in each case

what kind of relationship we are using. As we can see in figure 3.2 below, only instance-of relationships are allowed to cross the boundary between metalevels.

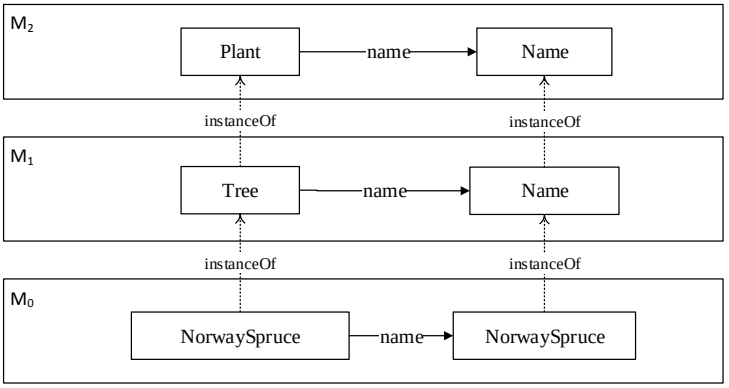


FIGURE 3.2: A three level meta-hierarchy using strict metamodelling.

A model element is an instance of exactly one element in the metalevel adjacent above. If any relationship between two elements is any other than an instance-of relationship, then both elements is at the same metalevel. It is also important to notice that an instance can only be created from an element in the metalevel adjacent above. We can not define a model element in M_2, and then instantiate the model element in M_0. In other words, information can only be carried from one instantiation-step to the next. This is commonly referred to as *shallow instantiation*. However, even though strict metamodelling provides a well-defined definition of the role of metamodels, some problems occur as a result of this. One of these problems is the replication of concepts problem.

3.1.2 The Replication Of Concepts Problem

As an example, let us consider that we want to develop a meta-hierarchy of Plants as the one we created in figure 3.2 above. If we want to model that all sub-sequent instances of Plant should have a *name*, we realize that it is most natural to specify the *name* on the Plant element in the topmost metalevel. However, since traditional metamodelling only supports *shallow instantiation*, it is only possible to specify requirements from one metamodel to the next. To be able to specify that all Plants should have a *name*, we have to re-instantiate the concept of name at each metalevel until the bottommost metalevel. In terms of the Plant example, it means that to be able to specify a *name* on NorwaySpruce, we have to re-instantiate the *name* element at m_1. If we want to specify a requirement across several metalevels, we have to replicate the element at each intermediate metalevel until we reach the metalevel of concern. The limitation of not being able to specify requirements across multiple metalevels leads to the replication of concepts problem.

3.1.3 The Multiple Classification Problem

Traditionally, in OOP languages developers have only been concerned with two metalevels, Class(M_1) and Object(M_0). This has worked fine as all objects are in M_0, and all classes are in M_1 and makes for a distinct separation between the two classifiers. However, when a developer in MDE now introduces additional metalevels, some problem occurs. One of these problems are the multiple classification problem. As an illustration, consider the concepts in figure 3.3.

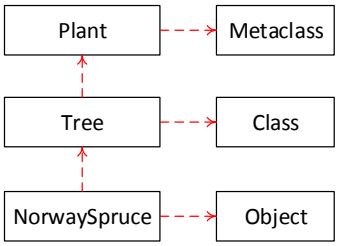


FIGURE 3.3: Multiple classification.

This is the same concepts as described in figure 3.2 above, a hierarchy with Plant at the top level, Tree in middle and NorwaySpruce on the bottom level. As we have seen earlier, NorwaySpruce is an instance of Tree, and Tree is an instance of Plant, but these concepts can also be classified by a second classifier. As in OOP where we are modelling with Classes and Objects, we see that NorwaySpruce in addition to be classified as an instance of Tree, it is also an instance of Object and Tree is an instance of Plant, but it is also an instance of Class. This is what we refer to as multiple classification, which in short means that an element can be classified by more than one type.

One of the more popular attempts to overcome this problem while at the same time adhering to strict metamodelling is through the orthogonal classification architecture [31]. The principles of the orthogonal architecture is to simply capture the two fundamental meta-dimensions and separate them into an ontological and a linguistic meta-dimension. In figure 3.4 on the next page, we have organized the Plant, Tree and NorwaySpruce concepts in each their metalevels on the left side. These metalevels is referred to as the ontological metalevels, each element is an ontological instance of an element in the adjacent metalevel above. To the right in the figure we have aligned the second classifications: Metaclass, Class and Object. These classifiers are put in a single metalevel, the linguistic metalevel. In this case, the classifiers are classifying the OOP oriented types of the elements, determining whether an element is a metaclass, class or an object. This metalevel can consist of other classifiers as well, depending on what type of system we are modelling.

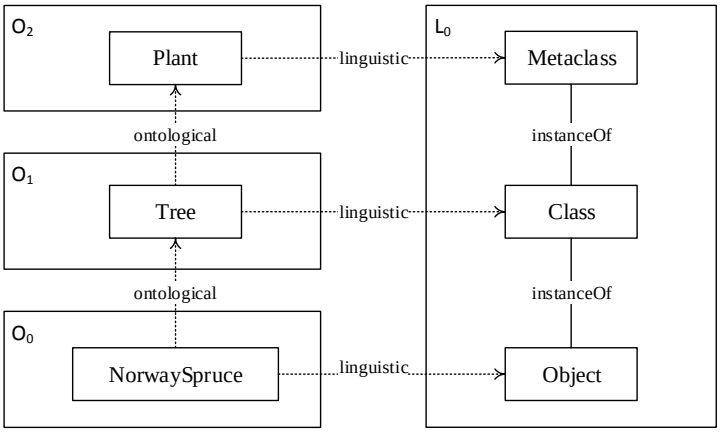


FIGURE 3.4: Two fundamental meta-dimensions - Linguistic and Ontological.

Even though we at this point have enabled both ontological and linguistic classification of model elements, it also means that a tool has to maintain both the ontological metalevels as well as the linguistic metalevels. If a developer wants to add new metalevels, new linguistic classifiers for these elements has to be added as well. The result is that the number of metalevels are restricted by the number of linguistic classifiers available to the modeller. For example if we want to add an instance of NorwaySpruce, we have to add a new linguistic classifier as well, thus adding complexity for the tool. However, by unifying linguistic elements of the same concept into a single unified element, we no longer have to add new linguistic elements if we want to add a new metalevel. The unification of linguistic elements is described in greater detail in the next section.

3.1.4 The Classifier-Duality Problem

One way to overcome the problems of maintaining the linguistic classifiers is by unifying the concepts into a single structural element [9]. Figure 3.5 on the next page illustrates the unification of Metaclass, Class and Object into a single concept: Clabject. (CLAss + oBJECT). We see that Plant, Tree and NorwaySpruce is all classified by Clabject. By unifying the classifiers into a single element, we have now simplified the linguistic metamodel and enabled an arbitrary number of metalevels. Since every ontological element is classified by Clabject, tools no longer need to maintain the linguistic metamodel.

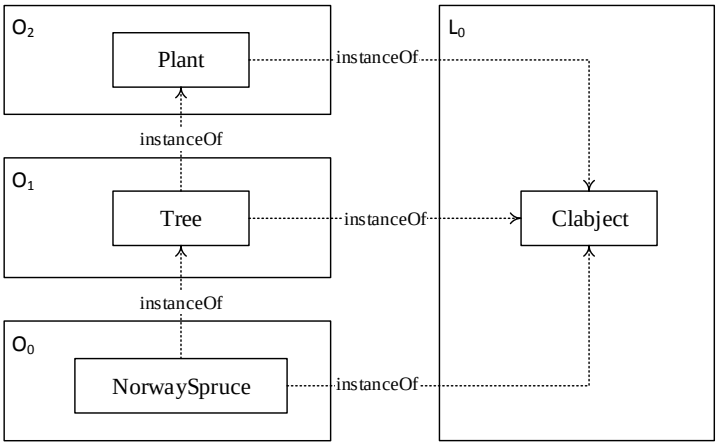


FIGURE 3.5: Unification of linguistic classifiers.

Even though tools no longer need to maintain the linguistic metamodel, there are new problems arising because of the unification of the linguistic classifiers. Looking at the unified element in figure 3.5, we see that the Clabject can be classified as either a Class, Object or both. There is a need to find a way to distinguish between the different types of the unified elements. This is what we refer to as the classifier-duality problem.

3.2 Deep Instantiation

In section 3.1, we presented shallow instantiation as a major limitation in traditional metamodelling, resulting in the replication of concepts problem amongst others. Shallow instantiation is sufficient only when dealing with two metalevels, type and instance, but it should ideally be enhanced for a multilevel environment [32]. If we want to make statements about model elements across more than two metalevels, we need to apply an alternative approach. In this thesis, we will describe two different approaches to this:

1. Powertypes
2. Potency

The first approach is by specifying powertypes to force model elements to be specified the way we want them to be, while the second approach implements deep instantiation through potency. These concepts will be described further in the following sections, starting with powertypes.

3.2.1 Powertypes

Powertypes was introduced by Odell [33] and is a keyword for a specific stereotype in the Unified Modelling Language (UML) [5]. In UML 1.x, a powertype is a classifier whose instances are children of a given parent. In UML 2.x, the stereotype has been removed and is now indicated by generalization as illustrated in figure 3.6 below.

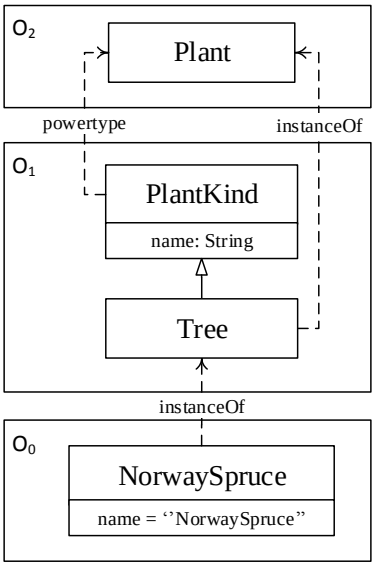


FIGURE 3.6: Plant as a powertype.

The purpose of specifying Plant as a powertype of PlantKind is to enforce that all instances of Plant inherits from PlantKind and thereby aquires the requirements of PlantKind by inheritance. As seen in figure 3.6, we now ensure that all instances of Plant in the meta-hierarchy has a *name* attribute, no matter what type it is.

However, when adding new metalevels we need to ensure the proper powertypes are added to the metamodel, thereby resulting in a more complex solution than satisfactory. Secondly the tool have to support inheritance as well, which we argue is a useful, but not strictly necessary feature in metamodelling. The Powertype concept is a UML specific concept, while metamodelling not only concerns UML diagrams. Metamodelling may concern any type of model. We therefore argue that powertypes is not properly addressing the need to allow deep instantiation. With powertypes we are still working with shallow instantiation, but to achieve a more natural solution we will introduce deep instantiation through potency as it was presented in [9].

3.2.2 Potency

The second approach to allow deep instantiation is through the concept of potency. Potency is simply an integer that is assigned to model elements, and acts as a type of constraint used to specify restrictions on how many sub-sequent metalevels a model element can be instantiated. When the potency of a model element is set, its potency will be reduced by one for each sub-sequent metalevel the model element is instantiated. Figure 3.7 below illustrates the Plant hierarchy with added potency to define deep instantiation, visualized in UML Class diagram syntax.

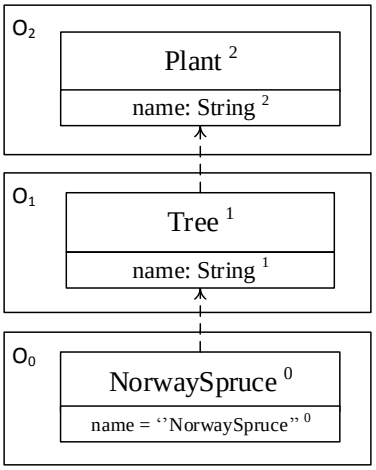


FIGURE 3.7: Deep instantiation.

Traditional classes and attributes have a potency of 1 since they only can be instantiated once, and traditional objects and slots have a potency of 0 since they cannot be instantiated any more [9]. As we see in figure 3.7, there is no longer need for a PlantKind concept any more, since the *name* attribute of Tree is forced as an instance of the *name* meta-attribute in Plant. The same way we also see that the potency 1 *name* attribute of Tree, is instantiated as slot of NorwaySpruce in O_0. When the potency of an attribute is 0, it is instantiated as a slot with a value as seen in figure 3.7 above.

By defining the Plant Clabject with a potency of 2, we restrict the instantiation depth of the Plant to two metalevels below. When adding new requirements such as the *name* attribute in Plant, the *name* attribute will automatically be contained in the instance through the semantics of instantiation. In fact all elements serving to specify some other element will be contained in the instance as well. An attribute or method that is contained in a clabject or any other specification of a model element will automatically be re-instantiated in the instance through the semantics of instantiation. In practice however, tools need proper mechanisms to ensure that it is possible to access model elements across multiple metalevels. Rossini (et.al.) proposed a solution of flattening the metamodels through a set of *replication-rules* [34]. The concept of model flattening will be discussed further in chapter 5.

Secondly, we may also notice how well the potency mechanism fits to describe both requirements across multiple metalevels as well as multiple classification. Through potency we realize that when for example *NorwaySpruce* (Clabject) is of potency 0, it will be classified as a Object, if its potency is 1 it will be classified as an Class, if the potency is 2 it will be classified a MetaClass and so on. The same principle works for any concept with more than one type-facet. With potency we can now precisely define the type-facet of any model element as well as gaining a precise definition of how many sub-sequent metalevels an element can be instantiated.

The third aspect that arises from the definition of potency is that it is becoming evident that there is a need to elaborate for how often model elements are allowed to change their value down the meta-hierarchy. We need to establish a definition of potency together with a specification of the *mutability* of the model elements. The concept of mutability has been discussed in various sources in the literature. Atkinson and Kuhne described the concept of single and dual fields, where a single field is a field in which its value can only be set once, and dual field is a field where its value can be set more than once [32]. In this thesis we will use the term mutability, where we define whether a model element is mutable or not.

3.3 Linguistic And Ontological Instantiation

With added support for deep instantiation through potency in the model, we can now use the linguistic elements and create ontological instances with a user-defined potency. By looking at figure 3.8 below, we see that the elements in the linguistic metamodel have an potency defined with an asterisk symbol, which means it has an undefined potency.

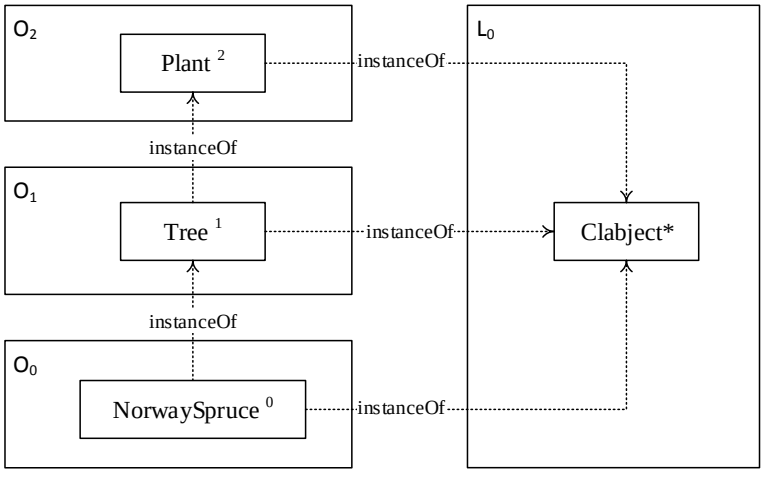


FIGURE 3.8: Deep instantiation with linguistic and ontological typing.

All linguistic elements will by definition have an undefined potency, because the model element has not been instantiated yet, and we thereby

do not know the instantiation depth. The potency can be set whenever the modeller feels it is necessary, so even ontological elements can have an undefined potency. The main idea here is that if the potency is undefined, it behaves the same way as in traditional metamodelling. As soon as the potency is defined, deep instantiation comes into play. The linguistic metamodel can be seen as the core language definition of the domain specific modelling languages we will define, it contains basic elements used to build DSML's with. This metamodel can also define core language libraries, and thereby provide a strong core language foundation for domain specific modelling languages.

Looking back at figure 3.8, we notice that by using a two dimensional meta-hierarchy, we can add new linguistic elements at any ontological metalevel, also referred to as linguistic extension. *Linguistic extensions* are useful as a mechanism to add new requirements to metamodels further down the meta-hierarchy than what could be foreseen at the top metalevel. The two dimensional approach illustrated in figure 3.8 is the most common approach amongst tools today, as we will describe in the next chapter. In this thesis however, we also present an additional approach where we rearrange the meta-hierarchy to a single dimensional hierarchy as illustrated in figure 3.9 below.

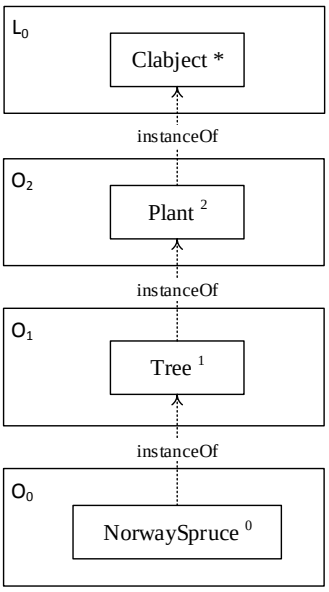


FIGURE 3.9: An illustration of the example given in figure 3.8, rearranged to a linear, one dimensional hierarchy.

Figure 3.9 illustrates the re-arranged meta-hierarchy into a single dimensional hierarchy. We have put the linguistic metamodels on top of the ontological metamodels in the metamodelling stack, and through potency we can still distinguish between dual facets of model elements such as Clabjects. However, by looking at figure 3.9 we also realize that it is no

longer possible to create new Clabjects at metalevels below O_2. If a modeller would like to extend the hierarchy with for example a LumberJack in O_1, it would simply not be possible with the current single dimensional approach. To adhere to strict metamodelling, we can only instantiate elements from the metalevel adjacent above. The linguistic elements were easily accessible using the two dimensional meta-hierarchy as it was described in figure 3.8. However, as we now have rearranged the meta-hierarchy to a linear hierarchy, it is no longer possible to linguistically extend ontological metamodels. It is necessary to find an alternative solution to overcome this problem.

One solution to this problem, is to replicate the linguistic elements at each ontological metalevel except the bottommost metalevel as illustrated in figure 3.10 below.

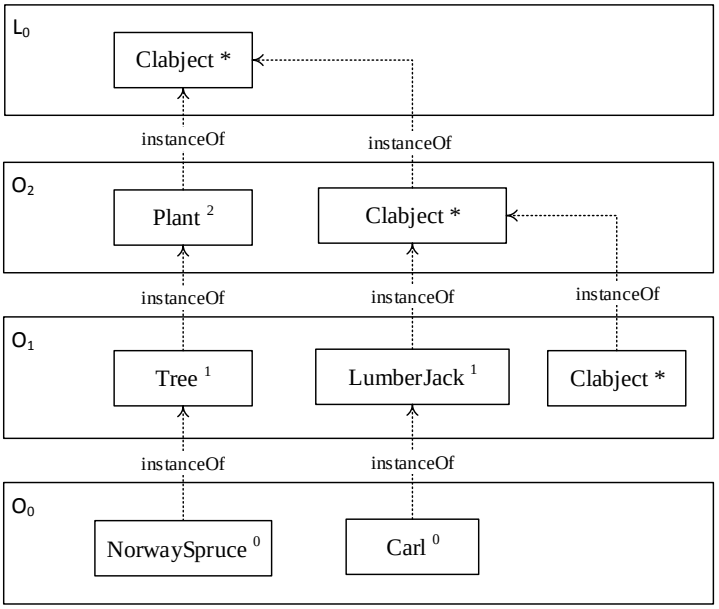


FIGURE 3.10: Deep instantiation with linguistic extension.

Figure 3.10 illustrates the Plant hierarchy with the linguistic elements replicated at each ontological metalevel except the bottommost metalevel. It is not necessary to replicate the linguistic elements in the bottommost metalevel, because the purpose is to create instances of the replicated linguistic elements. If there are no more metamodels below, it is no need to replicate the linguistic elements. Whenever we want to extend a model linguistically, we simply create an instance of the replicated element. This way we adhere to strict metamodelling and makes it possible with linguistic extension at any metalevel, even with only a single dimensional meta-hierarchy.

However, when adding references and connections between ontological and linguistic model elements, we need to find an additional mechanism

that check conformance not on the metalevel adjacent above, but on the linguistic metamodel instead. This means that to add a new linguistic node to the metamodel, we can simply instantiate a replicated linguistic element from the metamodel adjacent above such as *LumberJack* in figure 3.10. However, to be able to add an arrow from the linguistic instance *LumberJack* to the ontological instance *Tree*, we need additional mechanisms that can traverse the metamodelling stack and check conformance on the linguistic metamodel instead.

Even though a linear approach at first sight yields a more complex solution than the two dimensional approach, we still argue for a linear approach, because it makes it possible to model both linguistically and ontologically. With a two dimensional approach, the linguistic metamodel would be hardcoded into the editor and thus restricts us to only modelling with a predefined set of linguistic elements. With a linear approach however, we can use a predefined structure such as a directed multi-graph to model a linguistic meta-hierarchy, and then use the bottommost linguistic metamodel to model an ontological meta-hierarchy. We can thereby use the editor to develop linguistic templates, which can be used to define ontological meta-hierarchies with linguistic/ontological instantiation and linguistic extension.

In the next section, we briefly mention some of the better known initiatives with support for deep metamodelling. A more thorough description of existing initiatives related to deep metamodelling can be found in [31].

3.4 Existing Initiatives

As stated earlier in this thesis, there currently does not exist any fully diagrammatic tool for deep metamodelling. It does however exist a diagrammatic workbench for deep metamodelling called Melanee, which is developed at the same institution as the concepts of potency and the seperation of linguistic and ontological classification was elaborated [31]. Melanee does however only support modelling in a single view, which means that all ontological metalevels is modelled in the same window. Secondly there also exist textual framworks for deep metamodelling such as the Metadepth framework [35]. There exists frameworks such as Nivel, which is a deep metamodelling framework based on the weighted constraint rule language [36]. Nivel does however lack constrains and action languages, which hinders its use in practical MDE. There also exists an extension of the Java programming language with support for deep metamodelling called DeepJava. DeepJave can however not be considered a meta-modelling framework, but merely a programming language with support for an arbitrary number of metalevels [37].

3.5 Summary And Research Methodology

In this chapter, we elaborated that there is a need to overcome three major limitations with traditional metamodelling. To do this, we are first of all going to follow the approach of Atkinson and Kühne to implement deep instantiation through potency [9]. However, whether it is best to stay with a two dimensional hierarchy with deep instantiation, or to keep a linear meta-hierarchy is a design question we will address later in this thesis. Secondly, we need to establish appropriate replication rules to ensure that for example attributes are instantiated along with the clabject it is contained in. Thirdly, we also need to implement mutability to allow restrictions on the number of times a model element can change its value. Lastly, a more human readable visualization of the models is needed as well.

In order to do this in practice, we have identified the design and creation research strategy as the most suitable approach in this thesis [38][39]. The design and research strategy is typically a problem solving approach, which uses an iterative process involving five steps:

1. **Awareness and recognition of the problem.** This is already elaborated for in this chapter and summarized earlier in this section.

2. **Suggestion** involves the step from recognizing the problem to offering a tentative idea for how the problem might be addressed. In this thesis we are going to perform a comparison analysis to establish a starting point for the implementation of a diagrammatic tool for deep metamodelling. The main goal with the comparison analysis is to elaborate suggestions for how the problem might be implemented in practice.

3. **Development** is where the suggestions for the problem we recognized is implemented. The goal is to implement a diagrammatic editor that overcomes the problems we recognized in step 1, with basis in the established suggestions from step 2.

4. **Evaluation** is the step where the implementation is evaluated, and where we assess the implementation we have done so far.

5. **Conclusion** is the final step where we summarize the results from the previous steps, and tie up loose ends by presenting a list of further work.

In practice, these five steps would most likely not be followed in a stepwise fashion, but rather form an iterative cycle. The suggested solution in step 2 leads to a greater awareness of the problem, which is then developed in step 3. By evaluating the solution we developed in step 3, we may discover new and improved design suggestions for the problem, thus repeating the development phase. This way the Design and Creation strategy aims at *learning through making*.

As mentioned earlier in this section, we have at this stage already defined the problem we are going to focus on in this thesis. In the next chapter we are therefore going to perform a comparison analysis to establish suggestions for how we can overcome these problems.

CHAPTER 4

Comparison Analysis

This chapter will analyse current tools that has overcome the three limitations with traditional metamodelling as we described them in chapter 3, and compare them with the DPF Editor. The main goals with this chapter is to establish a starting point for the implementation of a diagrammatic editor for deep metamodelling.

4.1 Expected Outcomes

This chapter will establish a starting point for how we can implement a diagrammatic tool for deep metamodelling. We will analyse Metadepth, Melanee and the DPF Editor with focus on:

1. The three major limitations with traditional metamodelling, and
2. The user interface and useability from a modellers perspective.

To do this, we will start by introducing two of the better known tools for deep metamodelling, namely Metadepth and Melanee. Metadepth is a textual tool, while Melanee is a diagrammatic tool. By analyzing Metadepth and Melanee, we aim at finding their strengths and weaknesses. We will mainly focus on whether the tool has support for an arbitrary number of metalevels, we will analyze how the tool has added support for deep instantiation and we will look at how the tool has managed to seperate between linguistic and ontological classifiers and supported linguistic extension. We will also analyse the user interface of each tool, and determine its useability in terms of how intuitive the tool is to use from a modellers perspective. Next, we will analyse the DPF Editor with basis on how it may be possible to overcome the three major limitations with traditional metamodelling. We will also analyse the useability and diagrammatic syntax of DPF to determine whether its a suitable approach for the implementation of a diagrammatic editor for deep metamodelling or not.

If neither Metadepth, Melanee or DPF is a suitable approach for the implementation, we will elaborate for whether we will analyse additional tools, or whether we will use existing frameworks and plugins such as EMF. In the end, the main objectives of this chapter is to establish a starting point for the rest of this thesis, whether we can extend an existing tool or whether we need to start from scratch.

4.2 Metadepth

Metadepth is as described earlier an textual tool for deep metamodelling, with support for an arbitrary number of metalevels. Metadepth is being built under the METEORIC project sponsored by the Spanish Ministry of Science, and is mainly contributed by: Juan de Lara, Esther Guerra and Jesús Sánchez Cuadrado from Universidad Autónoma de Madrid (UAM) [35]. Metadepth supports the use of the Epsilon language family [40], such as Epsilon Transformation Language (ETL) for model to model transformations and Epsilon Generation Language (EGL) for code generation. Metadepth hosts both Epsilon Object Language (EOL) and Java as constraint and action languages to ensure model consistency and provide a richer set of features used in model-checking. Metadepth also supports derived attributes, meaning the value is not set by the user, but derived by a computed expression instead. Lastly, Metadepth features transactions, meaning the core API calls are recorded in an event list such that previous events can be reproduced, re/un-done and grouped into transactions. However, to elaborate for how Metadepth overcame the three limitations with traditional metamodel, we will first analyse the linguistic metamodel in Metadepth.

4.2.1 The Linguistic metamodel

Figure 4.1 on the next page illustrates a simplified version of the linguistic metamodel in Metadepth. The linguistic metamodel in Metadepth took inspiration from MOF, but with added support for an arbitrary number of metalevels, as opposed to only four in MOF. In Metadepth it is also added potency to add support for deep instantiation as we described it in chapter 3. As seen in figure 4.1, the uncoloured concrete classes are those a modeller would typically instantiate (i.e. Model, Node, Edge, Field and DerivedField). Under closer inspection of the linguistic metamodel in Metadepth, we see that **Clabject** is an element very high up in the inheritance tree - basically everything except from attached constraints is a Clabject. All elements in the metamodel is based on **Clabjects**, taking care of the dual classification of elements such as class/object classification. To do this, the Clabject has a **potency** value, denoting the instantiation depth of the Clabject as well as specifying classification type (Class / Object).

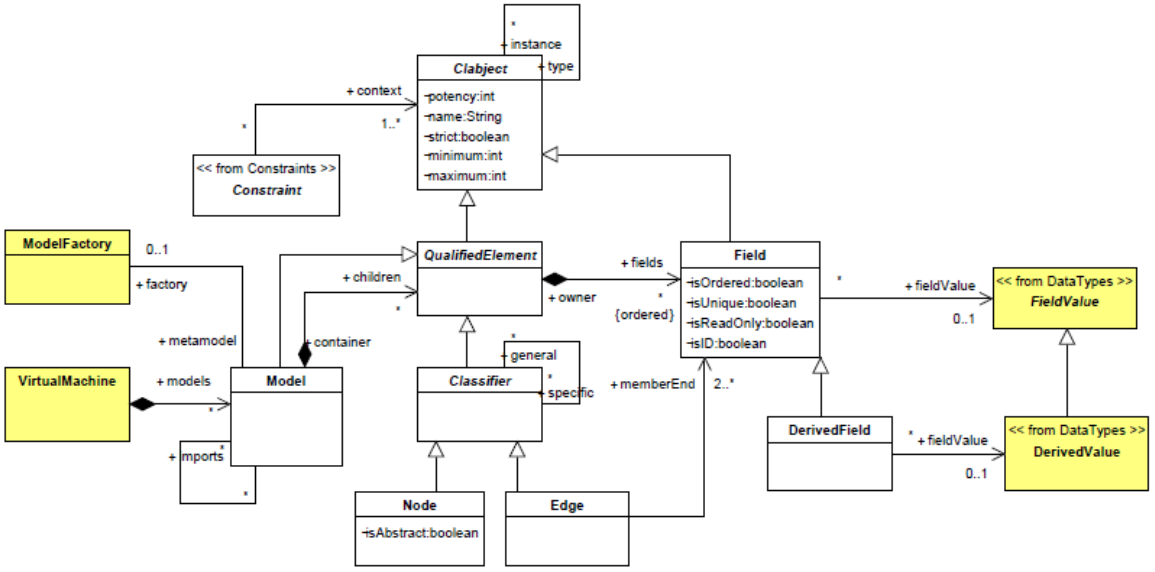


FIGURE 4.1: MetaDepth's linguistic metamodel, taken from [35].

As we see from figure 4.1, Clabjects have a potency attribute, which means all elements except from constraints has potency. This means that both Model, Node, Edge Field and DerivedField are all elements capable of deep instantiation. The potency in Metadepht can be unlimited, such that Clabjects can be instantiated with an arbitrary potency. We also see that Clabjects has **minimum** and **maximum multiplicity** to control the number of instances within a given context. **Constraints** can be attached to Clabjects and have an attached potency as well as the possibility of specifying on what event the constraint should be evaluated(i.e. when creating or deleting the clabject) Metadepth also features both **strict** and **extensible** modelling modes through the strict attribute in Clabject. By specifying strictness, the modeller has control over whether or not the ontological model can be **linguistically extended** or not.

Clabjects has one subclass, **QualifiedElements**, that can own **Fields**, meaning Nodes, Edges and even the Model itself can contain Fields. The **Models** are contained in a **VirtualMachine**. The Model can contain **Nodes**, **Edges**, **Fields**, as well as **nesting of Models** inside other Models. It is possible to **import** other models, nest models together, **bind a model** to other Models and more. Nodes and Edges are **Classifiers**, which can form general/specific hierarchies, meaning the concept of **inheritance**. **Edges** are modelled in Metadepth with an input and a output field in each node we want to create an edge between. The Edge itself has two or more associations ends specified by the input/output fields in the Nodes we want to create an edge between. The **Nodes** can also be specified as **abstract** to support abstraction of nodes. Through these features it is possible to create abstract nodes and inherit from abstract Nodes to simplify the models.

Through these mechanisms, Metadepth has overcome all the three major limitations with traditional metamodelling. The next question is how the user interface in Metadepth is organized.

4.2.2 The User Interface

Figure 4.2 below illustrates the plant example as it would be modelled in metadepth. As we see, we begin by making an instance of a model with a defined potency. Inside the model, we create instances of Node and Edges, which each can have their own attached fields. By making instances of nodes, we can define new concepts and link them together using edges.

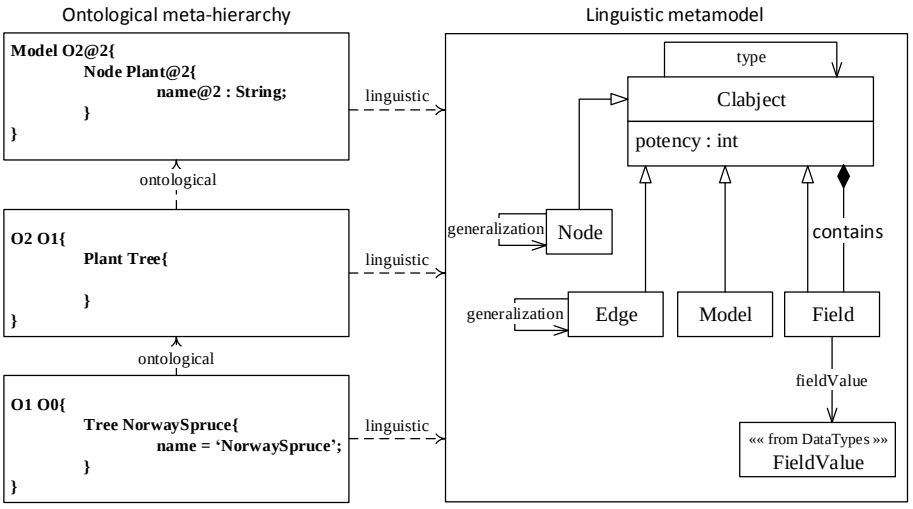


FIGURE 4.2: The plant model as it would be modelled in Metadepth.

As we can see, Metadepth has support for deep instantiation of Nodes, Edges, Fields, Derived Fields, Models and Constraints. It solved the dual classification problem as well through potency and provides a rich set of features for deep metamodelling. However, at the same time we see that it only has a textual interface. This may not be any problem for programmers that are used to develop in a textual interface, but the learning curve is certainly higher than that of diagrammatic editors. The models can be developed in external text-based tools, but even though the syntax is not complicated, we argue that it still can not compete with diagrammatic editors. For a humans it is easier to understand elements that is more similar to real world objects, and this is to a large extent made possible by using diagrammatic editors. Another aspect worth mentioning is that Metadepth is a standalone application, and not based on EMF such as for example DPF and Melanee as we will describe later. This means we can develop models using a text-based modelling tool and run them in a lightweight standalone console-application. However, it also means it wont benefit from the broad range of plugins provided in EMF and its various sub-projects.

4.3 Melanee

Melanee is a multi-level and ontology engineering environment, that has support for both textual and diagrammatic modelling. Melanee was developed by the Software Engineering Group at University of Mannheim in Germany, and supports model transformations, code-generation, querying of model contents among others. Melanee is based on EMF for the support of visualization and enhancability of models. This is because of EMF's established metamodel technology and because EMF has a broad range of plugins. Melanee is shipped with a default general purpose notation with UML and Entity-Relationship (ER) diagrams in mind [24]. The diagrammatic syntax is extensible as well, which makes it possible to use both the built-in general purpose notations as well as user specified domain specific notations. Important aspects for this thesis is however Melanee's capabilities as an editor for diagrammatic modelling of deep elements. To investigate how Melanee overcomes the three limitations with traditional metamodel, we will first analyse the top-level linguistic metamodel in Melanee, also called the Pan Level Metamodel (PLM).

4.3.1 The Linguistic Metamodel

Figure 4.3 below illustrates a simplified version of the linguistic metamodel in Melanee, where we have left out concepts such as Classification, Complement, Equal, Inversion and Enumeration for illustration purposes.

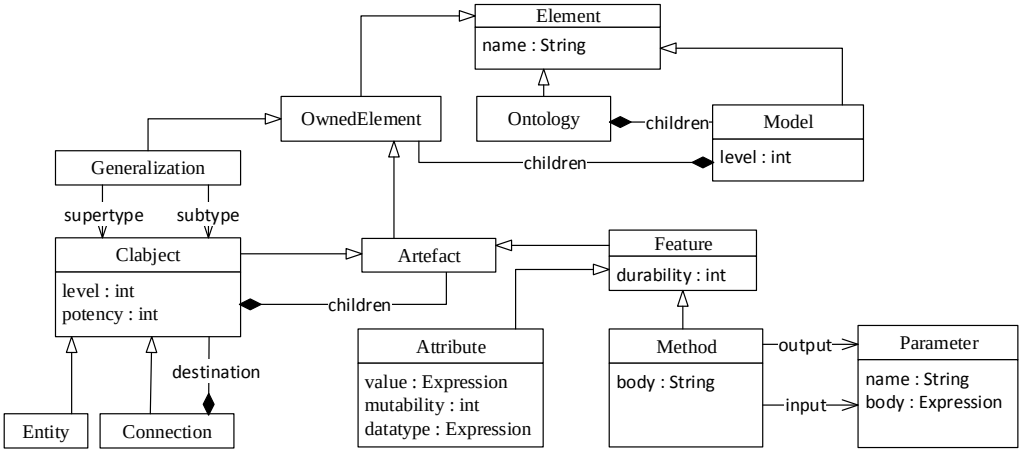


FIGURE 4.3: Melanee's linguistic metamodel, also called the Pan Level Metamodel (PLM).

By looking at figure 4.3, we see that every concept in Melanee is based on the **Element** class, where the **Ontology** class is the most general type of Elements. As VirtualMachine is holding the Models in Metadepth, Ontology is holding all Models In Melanee. Most of the times when users are modelling in Melanee, the user will model at one ontology at a time.

However, it is also possible to model multiple ontologies at the same time. Metamodels in Melanee is modelled in a single view only, where all metamodels are open at the same time focusing on the whole ontology at a time. From a useability point of view, this architecture works well as long as the models are small, but as the models are growing bigger and bigger, it is getting more and more difficult to get an overview of the system. In this thesis, we will therefore argue that a single modelling view is not a satisfactory approach for our implementation, but rather that a seperation of the modelling process such that each model is modelled in a seperate view. In the latter approach, a modeller will have full control over each seperate model at a time, and other models can be opened and closed depending on the modellers needs.

By looking at figure 4.3 on the previous page, we know that an Ontology contains **Models**, and each Model represents a single ontological metalevel. Each Model in Melanee contains **OwnedElements**, which is an Element that is owned by another element, such as Entity, Connection, Generalization, Attribute and Method. Entities and Connections are subtypes of **Clabject**, where Clabjects is the main building blocks of PLM models [31]. A Clabject may contain **Artefacts** such as **Attributes** and **Methods**, but as Clabject is a subtype of Artefact, a Clabject can also contain other Clabjects to model ontological composition. Clabjects also support Generalization, which allows creating a general/specific hierarchy similar to that of Metadepth. Looking at the Clabject class in figure 4.3, we also see that Clabjects is the only element that can have potency defined. Clabjects will however contain other Features such as Attributes and Methods, and their potency will be defined by the Clabject it is contained in.

Features in Melanee is used to extend Clabjects with either data or behaviour such as Attributes and Methods. By looking at the PLM in figure 4.3, we see that Features do not have potency, but have **durability** instead. This means that the Features potency is determined by the Clabject it is contained in, while durability means how many sub-sequent metalevels the Feature will last. In Melanee, the durability of a Feature can be higher than the potency of the Clabject it is contained in. This allows for the Features to endure longer than the Clabject it was initially contained in. A Clabject may inherit from a Clabject that has lower potency than the current Clabject, and by providing durability, containing Features can possibly endure even though the supertype can not be instantiated any more. The concept of durability is however not within the scope of this thesis, but we will discuss the concept further in chapter 7 under future work.

Attributes in Melanee is containing three fields: **value**, **mutability** and **datatype**. The value and datatype of the Attribute will be stored in the value and datatype fields. Mutability is another form of potency, namely value potency as we introduced it in section 3.2.2. The **mutability** controls whether or not attributes created from the attribute can change its value or not. If the mutability of an Attribute is zero, it can not change its value but

must retain the value given upon creation. In other words, if the mutability of an Attribute is zero, its value can not be changed at any subsequent-metalevel.

Through these mechanisms, Melanee has overcome all the three major limitations with traditional metamodelling. The next question is how the user interface in Melanee is organized.

4.3.2 The User Interface

As the PLM defines the core structure of the modelling concepts in Melanee, it was necessary to find a way to render these models in a diagrammatic syntax. In Melanee, this is done by introducing the Level Agnostic Modeling Language (LML), which is used to visualize the multilevel models of the PLM [31][41]. The LML provides concrete syntax visualization data for every PLM concept, such as Ontology, Model, Entity, Connection, Generalization, Attributes and Methods.

Ontology
An Ontology is the outermost container in Melanee, containing Models and visualized as a rounded rectangle with the name of the ontology at the top.

Model
Models are the only elements that is possible to add directly under Ontologies. The Models is forming a stack inside Ontologies, and contains ontological metamodel elements.

Entity
An Entity in Melanee is representing the Nodes in the Model graph, and is visualized the same was classes is visualized in UML Class diagrams. Entities is thereby a rectangle with three components; A header compartment, Attribute compartment and a Method compartment.

Connection
To realize edges in the model graph, it is used Connection elements in Melanee. To connect Entities to other Entities like Classes have references in UML Class diagrams, Melanee has Connections to realize this feature. In Melanee, Connections is modelled as Clabjects themselves, such that every connection is the visual counterpart to association-classes in UML Class diagrams. To distinguish between Entities and Connections, the shape of a Connection is a flattened hexagon.

Attribute
Attributes is visualized as text-elements inside the attribute compartment of Clabjects, illustrating name, datatype, value, durability and mutability.

Method
Methods is visualized much the same as Attributes, where the textual representation is the method-signature.

Generalization
As in UML Class diagrams, Generalizations in Melanee is visualized as a line between connected Clabjects (Entity or Connection)

Figure 4.4 below illustrates the Plant hierarchy as it would be modelled in Melanee. The linguistic metamodel is simplified, and we have left out Generalizations for illustration purposes.

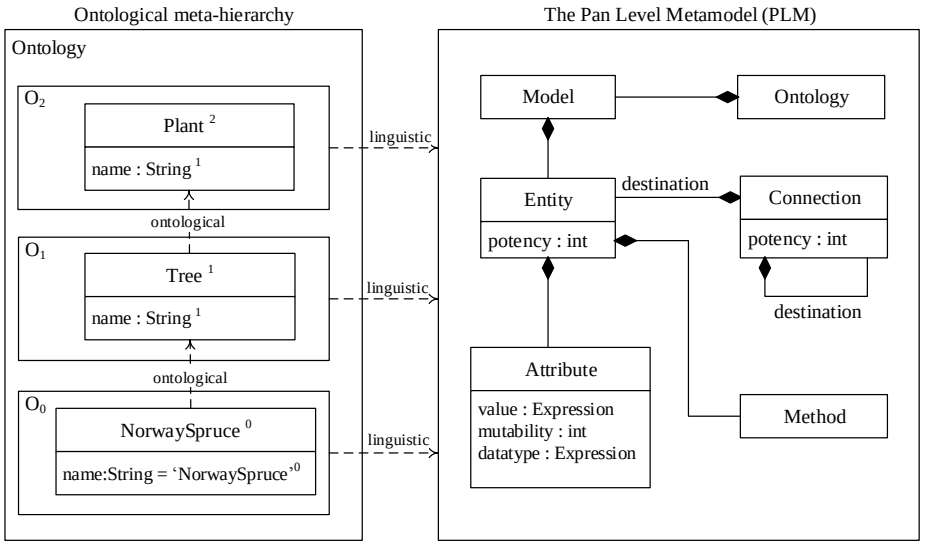


FIGURE 4.4: The plant model as it would be modelled in Melanee.

To work out the multiple classification problem in Melanee, the *orthogonal classification architecture* is used as in Metadepth, with two seperate meta-dimensions. The PLM is illustrated to the right in figure 4.4, which is used when modelling the ontological metamodels to the left in the figure. The visual representation to the left in figure 4.4 is represented by the LML, where we start modelling by adding an Ontology that will contain the Models we want to create. First we added a Model O_2, with an Entity presenting a Plant. In this example, we chose to define the Plant Entity with a potency of two, and a *name* Attribute with a mutability of one. The potency restricts the plant Entity to be instantiated a maximum of two sub-sequent metalevels below, while the mutability defines that the value of the *name* Attribute can be set only once. If the value of an attribute is changed, the mutability will be decreased by one for each sub-sequent metalevel the value is changed.

By analysing Metadepth and Melanee, we realize that both solutions has overcome all three limitations as they were described in chapter 3. Both tools support an arbitrary number of metalevels and deep instantiation.

We also found out that both Metadepth and Melanee has seperated the linguistic and ontological metamodels into a two dimensional hierarchy of models (orthogonal classification). However, while Metadepth focuses on one model at a time, Melanee focus at one ontology at a time. In practice this means that modellers will model at all ontological metalevels in a single view. We argue that this is satisfactory for small models, but as the size of the models are growing, so is the complexity of the ontology as a whole. We will therefore focus on developing a solution that focus on modelling on a model by model basis, rather than ontology by ontology. Therefore, we will introduce the DPF Editor in the next section to elaborate for whether DPF is a suitable starting point for the rest of this thesis or not.

4.4 Diagram Predicate Framework

This section will introduce the Diagram Predicate Framework and its supported plugins, starting with an investigation of the useability of the DPF Editor as a possible approach for further implementation.

4.4.1 The DPF Editor

Looking back at DPF as we presented it in section 2.6.2, we recall that with the DPF Editor, it is possible to diagrammatically develop models at an arbitrary number of metalevels, viewing a single model at a time [42]. DPF is based on a directed multi-graph, which makes it suitable for developing patterns such as UML Class diagrams. We could use the DPF Editor to model our linguistic metamodel, then replicate it down the meta-hierarchy. This would result in a linear, single dimensional meta-hierarchy as we presented it in section 3.3. Another solution to implement dual classification in the DPF Editor is to alter the top-level metamodel in DPF, also called the core metamodel, and add linguistic elements directly in the metamodel as in Metadepth or Melanee. However, this means we will alter the graph such that it may not even be a graph as we know it in DPF anymore. In turn, the solution may render many of the sub-projects of DPF useless. In this thesis, we will therefore argue that a linear approach to dual classification is the most suitable approach in DPF.

Regarding deep instantiation and how we can implement it in the DPF Editor, we will first have a look at the core metamodel in DPF as illustrated in figure 4.5 on the next page. With inspiration from Metadepth, we can add a potency integer to the graph-elements such as Node, Arrow and the Graph itself. We can specify the default potency to -1 such that by default, the elements will behave as in traditional metamodelling. As soon as the potency of an element is defined, deep instantiation comes into play. We can add programmatic constraints, restricting the model elements from being instantiated any deeper than what is specified through the attached

potency. When potency is defined, the dual classification problem is solved as well as we explained it in section 3.3.

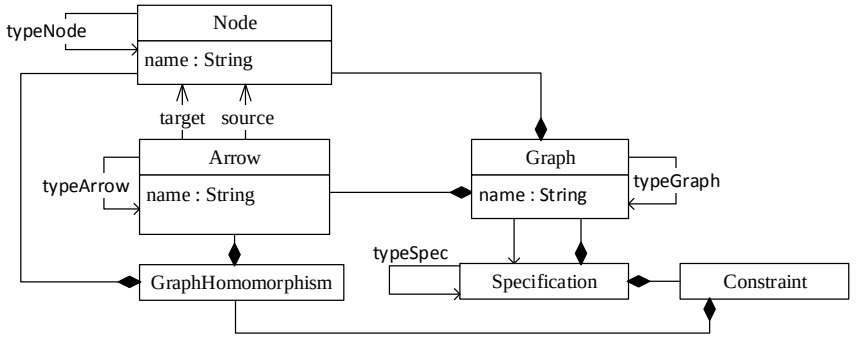


FIGURE 4.5: A simplified illustration of the core metamodel in DPF.

Figure 4.5 illustrates the core metamodel in DPF without potency. After adding potency, we will have overcome the three limitations with traditional metamodelling, and we only need to find a solution for the concrete syntax for our editor. As we recall from Melanee, the PLM is specifying the linguistic metamodel, while the LML is specifying the visualization of these elements. The DPF Editor is structured in a similar way, where the core metamodel is the main representation which is persisted in an external representation in form of a .xmi file. To visualize the graph elements in the abstract syntax it was created a diagram metamodel, which contains visualization data for each element in the core metamodel, much like the LML in Melanee. Figure 4.6 below illustrates a simplified version of the diagram metamodel in DPF.

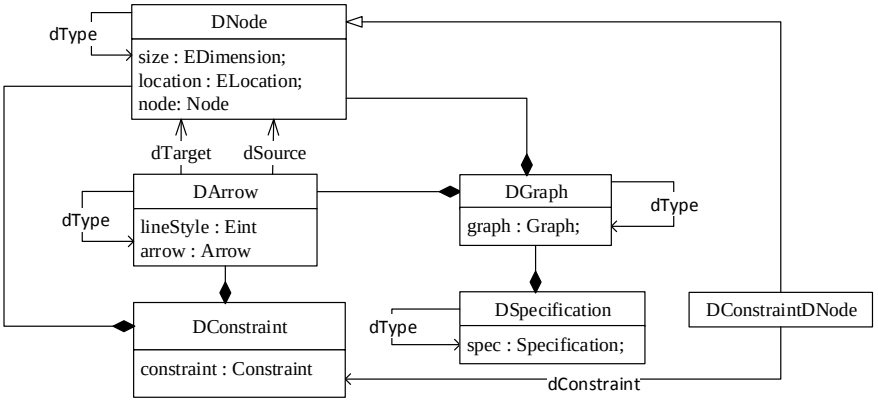


FIGURE 4.6: A simplified illustration of the diagram metamodel in DPF.

The main elements in the diagram metamodel, mainly the elements a modeller would use when developing new metamodels is DNode, DArrow and DConstraint. DNode and DArrow is visual representations of Node and

Arrow as they were defined in the core metamodel, and contains pointers to the core element as illustrated in figure 4.6 above. With DNodes and DArrows we can specify visualization data such as location and size of the Nodes and Arrows. DConstraints on the other hand is used to visualize Constraints on Arrows in the model. The diagram metamodel is thereby representing the abstract syntax in DPF, and the diagram metamodel is persisted in an external file in form of a .dpf file. The diagram file can then be imported into the DPF Editor and visualized as illustrated in figure 4.7 below.

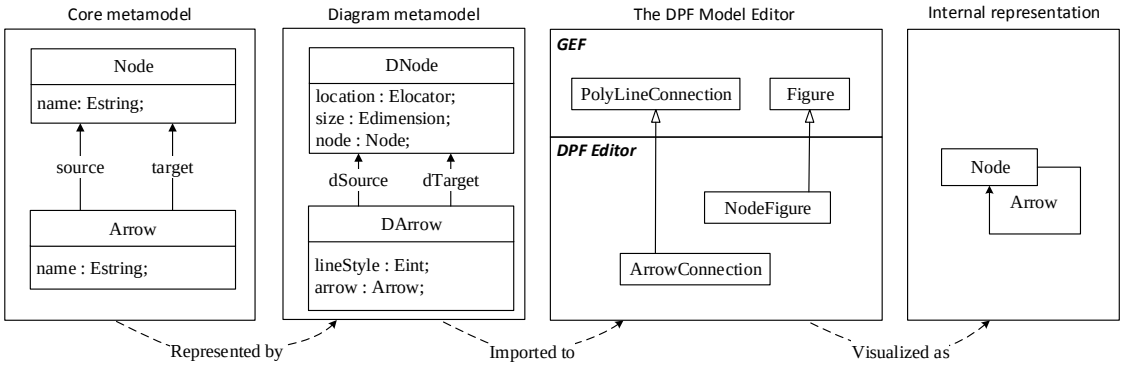


FIGURE 4.7: An illustration of the process of visualizing models in the DPF Editor.

As we see from figure 4.7 above, the visualization in the DPF Editor is built on the Graphical Editing Framework (GEF), which provides technology to create rich graphical editors and views in EMF [43]. The diagram metamodel is imported into the DPF Editor and parsed such that DNodes and DArrows visualized with the appropriate structure. In the DPF Editor, its currently only necessary to visualize two types of visualizations, Node and Arrow. To visualize Nodes and Arrows, it was decided to extend the Figure and PolyLineConnection classes in GEF to two classes; NodeFigure and ArrowFigure. NodeFigure represents a DNode, is capable of rendering the visualization attributes that is specified in the DNode, and visualized as a rectangle. ArrowConnection represents a DArrow, its defined attributes, and is visualized as an arrow.

However, the DPF Editor is only capable of modelling with Nodes and Arrow with attached Constraints. There is no clear seperation between an abstract and concrete syntax. Basically modelling with Nodes and Arrows is the only possible syntax in the DPF Editor as we know it, and the abstract syntax is therefore used as a concrete syntax as well. However, by looking at figure 2.13 in section 2.6.3, we recall that there has been developed a visualization editor by the previous master student [44]. This visualization editor aims at providing the DPF editor with a concrete syntax, which is also synchronized with the abstract syntax. This is done by

providing a third metamodel, the visual metamodel, which is a concrete syntax representation of the abstract syntax in the DPF Editor.

4.4.2 The DPF Visualization Editor

While the diagram model in the DPF Editor is enough to specify the abstract syntax, it was necessary to implement compositions of elements as well in the concrete syntax. A new metamodel for the concrete syntax called the visual metamodel was implemented, containing additional visualization data and capability of nesting components. A simplified version of the visual metamodel is illustrated in figure 4.8 below.

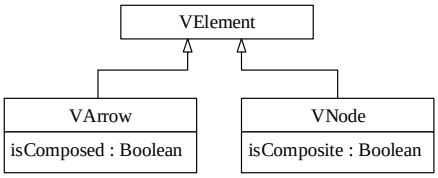


FIGURE 4.8: A simplified illustration of the current visual metamodel.

The concrete syntax metamodel consists of visualization data about the abstract syntax. Currently the concrete syntax metamodel consists of a VNode and a VArrow, which is used to represent the DNodes and DArrows in the diagram metamodel. VNodes can specify a broad range of customizable parameters such that the DNodes are visualized exactly as modellers want. In the current visual metamodel however, VNodes has a isComposite attribute, meaning whether the DNode is composite or not, or in other words if the DNode can contain other DNodes or not. VArrows contains a isComposed attribute, which means whether the DArrow is composed inside a DNode or not. If a VNode is composite, the outgoing composed VArrows and the target VNodes will be contained in the composite VNode. Figure 4.9 below, illustrates a simple example of a clabject with containing attributes and its mapping to the visual metamodel.

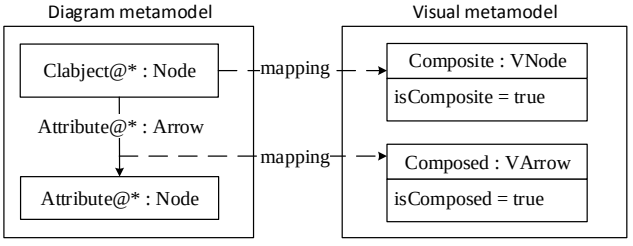


FIGURE 4.9: An illustration of a mapping between an instance of the diagram metamodel and an instance of the visual metamodel.

By creating an instance of the visual metamodel as seen to the right in figure 4.9 above, we can specify different types of visualization elements for

the abstract syntax. In addition to specifying whether a DNode is composite or an DArrow is composed, we can also specify color, rounded corners and similar, but these are left out for illustration purposes. When the instance of the visual model is created, we can map the model elements in the diagram metamodel to the model elements in the visual metamodel as illustrated in figure 4.9. The idea is to first create concrete syntax visualization elements in the visual metamodel, and then map the elements in the diagram metamodel to its appropriate elements in the visual metamodel. When the model elements in the diagram metamodel is mapped to the visual metamodel, we can create instances of the visual metamodel in the visualization editor. When creating the new instances of the visual metamodel, the elements will be visualized as described in the metamodel adjacent above. The way it works is that when modelling elements in the visualization editor, we will create diagram elements, that is visualized as specified in the visual metamodel in a concrete syntax. While modelling the concrete syntax, the abstract syntax and its containing elements will be automatically created, resulting in a synchronization between the diagram and visual metamodel as illustrated in figure 4.10 below.

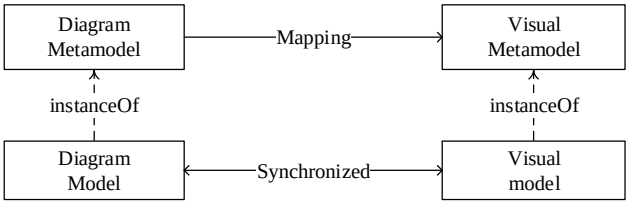


FIGURE 4.10: An illustration of the current model mapping solution in the visual editor.

As we see from figure 4.10 above, we can use the original DPF Editor to develop new diagram metamodels in the abstract syntax, and we can open the diagram metamodel in the visualization editor and continue editing on the same model in the concrete syntax. Figure 4.11 below illustrates modelling on both the diagram and the visual metamodel, which is synchronized with each other.

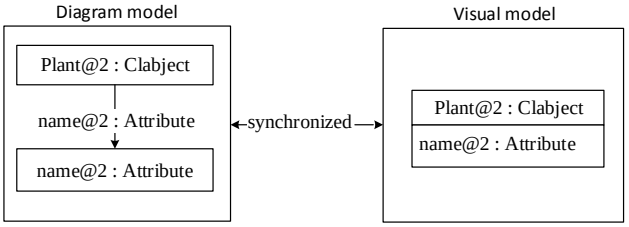


FIGURE 4.11: A concrete example of synchronization between the abstract and concrete syntax as it was mapped in figure 4.9.

In the diagram metamodel as seen to the left in figure 4.11, we recall that the visualization components for the model elements are based on GEF. In the DPF Editor we have created NodeFigure and ArrowConnection classes, representing the DNodes and DArrows in the diagram metamodel. To visualize the visual metamodel, we additionally needed a structure that allows nesting of components such as DNodes with containing DNodes. To make this possible, it was decided to create a class hierarchy with the following components as illustrated in figure 4.11 below.

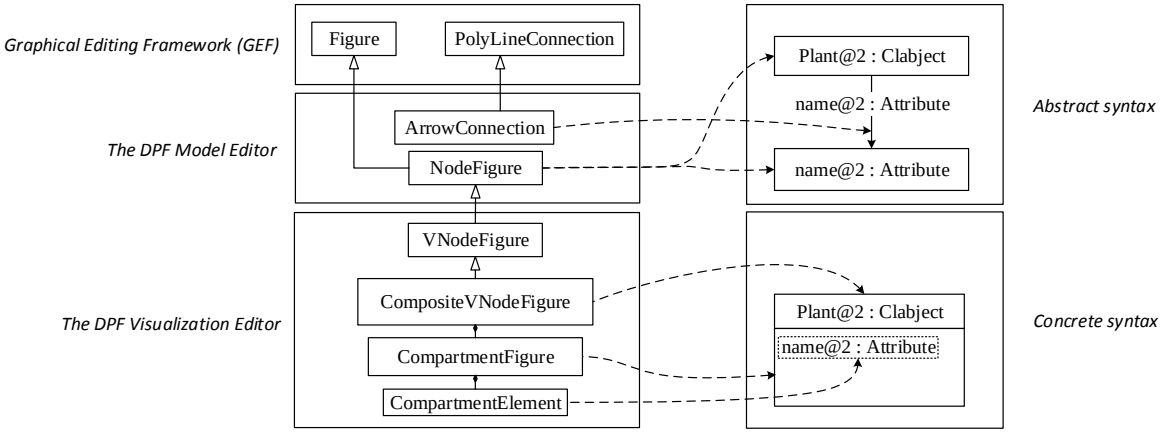


FIGURE 4.12: The structure of the visualization components in the DPF Editor and the visualization editor.

To visualize VNodes that is not composite, meaning DNodes that does not have any containing DNodes, it was decided to create a subclass of NodeFigure called VNode as it is represented in the abstract syntax, named VNodeFigure. VNodeFigure will inherit the same capabilities as a NodeFigure, but with added visualization elements such as rounded corners, node color and similar. To allow nesting of DNodes, it was created a subclass of VNodeFigure named CompositeVNodeFigure, representing the composite VNodes and how they are visualized. CompositeVNodeFigures contains CompartmentFigures holding CompartmentElements. The CompartmentFigures is a container of all instances of the composed DArrows and its targets such as Attributes. Instances of Attributes will be contained in the CompartmentFigure in each their CompartmentElement.

The current implementation of the Visualization editor is however currently not working properly, and has several bugs that needs to be fixed. Many of the mechanisms in the visualization editor that is supposed to build on the mechanisms in the model editor, is simply copy-pasted and not working properly. Other bugs such references to the core metamodel when there should be references to the diagram metamodel instead also needs to be fixed. In order to extend the visualization editor to support deep metamodelling, we first need to fix these bugs.

4.5 Summary

By analysing Metadepth, Melanee and DPF, we have established a starting point for the implementation of a diagrammatic editor for deep metamodelling. The general nature of DPF, its ability to be used to model pattern languages and its diagrammatic syntax made it clear that DPF makes an excellent starting point for the rest of this thesis.

	Metadepth	Melanee	DPF
Metalevels	∞	∞	∞
Deep Metamodelling	✓	✓	
Linguistic/ontological instantiation	✓	✓	
User Interface	Textual	Diagrammatic + Textual	Diagrammatic
Framework	Standalone	EMF	EMF
Platform	JavaVM	JavaVM	JavaVM

TABLE 4.1: Summary of the comparison analysis between Metadepth, Melanee and DPF.

CHAPTER 5

Design And Implementation

This chapter is structured into five parts, starting with an implementation of the most basic solutions and building up more and more functionality later on. Part one consists of the implementation of deep instantiation and dual classification. Secondly we present the implementation of a code generator to evaluate the solution we implemented in part one. We will continue building on what we learned from the code generator to implement the final solution of the abstract syntax for the editor. This chapter will end with a presentation of the implementation of a concrete syntax so that the editor is as intuitive from a human modellers perspective as possible.

5.1 Extending DPF - Part One

In the previous chapter, we established that DPF makes a good starting point for a diagrammatic editor, it only needs to address the three limitations as they were described in chapter 3. To address the three limitations, we are first going to present an implementation of deep instantiation, and then describe how we distinguish between linguistic and ontological classification, also called dual classification.

5.1.1 Deep Instantiation

As mentioned in the comparison analysis, we can start with adding a potency integer to the Node and Arrow as well as the graph itself in the core metamodel as illustrated in figure 4.5. We decided that to allow flexibility and to make it easier to determine exactly what elements that is classified as *deep*, the best solution is to create a DeepElement interface with a potency integer in the core metamodel. By making the Node, Arrow and Graph classes in the core metamodel inherit from the DeepElement interface, these elements will acquire the potency attribute through inheritance. However, adding a potency integer alone is not enough to add support for deep instantiation. We also need to define the behaviour of the potency, such as decrementing it for each instantiation-step and to make sure it can not be instantiated any further if the potency is zero. To support deep

instantiation in DPF, we decided to first implement the DeepElement interface and make Node, Arrow and Graph inherit from the DeepElement interface as illustrated in figure 5.1.

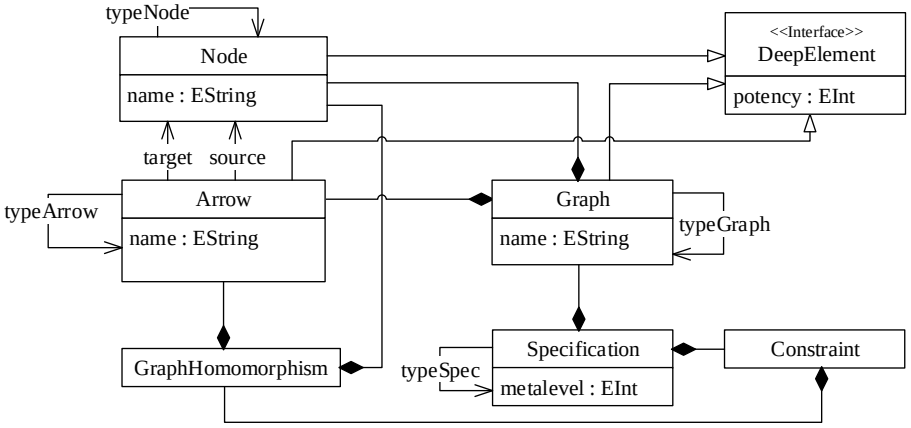


FIGURE 5.1: An illustration of the core metamodel in DPF with added potency.

Secondly, we decided to decrement the potency at each sub-sequent metalevel by decrementing the potency when a model element is added from the palette in the editor. The potency attribute will be available in a property sheet as illustrated in figure 5.2 below. The property-sheet is a table, containing the properties of model elements such as name, potency, attached constraints outgoing arrows and similar. By selecting a model element, the properties of the selected model element will appear in the property-sheet as illustrated in figure 5.2 below.

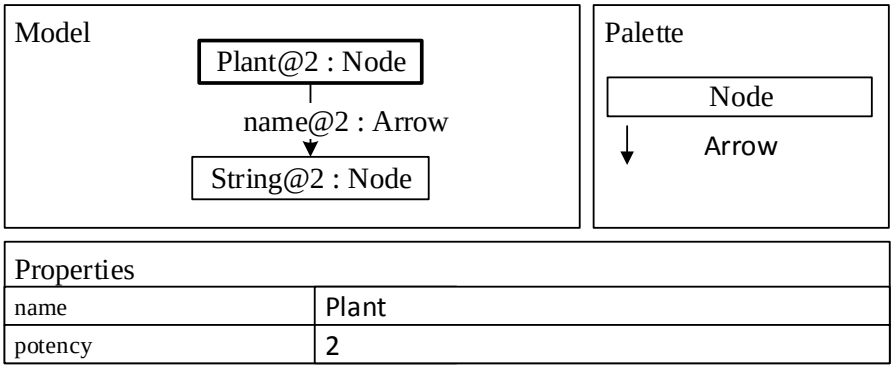


FIGURE 5.2: The current layout in DPF, with added potency.

Figure 5.2, illustrates a simplified view of the current layout in DPF. As we can see, we are modelling in DPF by adding elements from the palette to the right in the figure and placing them in the model window in the

center of the figure. The elements in the palette are elements from the metamodel adjacent above, which in this case is the hardcoded directed multi-graph from the diagram metamodel as illustrated in figure 4.6. When a model element is added, modellers can select the model element and modify its properties in the property sheet as illustrated in the bottom of figure 5.2 above. The figure illustrates editing of the Plant element with a name attribute. We can set the potency to two in the property sheet, such that instances of Plant will have a potency of one, and then instances of instances of Plant will have a potency of zero. When the potency of a modelling element reach zero, it will not be contained in the palette any more.

In section 3.3, we recall that the potency of all linguistic model elements has an undefined potency of -1, which is visualized in the model as an asterisk symbol. If an element is denoted an undefined potency, it will behave as in traditional metamodelling, but as soon as the potency is defined, deep instantiation comes into play. Each time a new element is added from the palette, the potency will be checked for whether it is set or not. If the potency is set, it will be decreased by one, and when its potency reaches zero it will not be added to the palette any more. This way we make it possible to define potency on model elements, and restricts modellers from adding model elements that should not be added any more.

Even though we now have implemented deep instantiation, the implementation still needs improvements to address aspects of potency such as mutability and model flattening semantics. With the current solution, the modeller has to redefine model elements at each instantiation-step to be able to access them further down the meta-hierarchy. In the case of adding attributes in clabjects, we would in most cases prefer not to redefine the attributes at each metalevel. It would be more intuitive to instantiate an attribute at one metalevel, and specify its value further down the modelling-hierarchy. The concept of defining when it is necessary to re-define model elements to be able to access it further down the modelling-hierarchy is often referred to as value-potency, and will be discussed further in section 5.3.1. Before that, we are going to define linguistic and ontological typing and linguistic extension.

5.1.2 Dual Classification And Linguistic Extension

One way to distinguish between the linguistic and ontological meta dimensions is by modifying the core metamodel in DPF as seen in figure 4.5 with added linguistic elements such as Clabject and Fields as in Metadepth or Melanee. However, this means the basic structure will not be a graph as we know it in DPF anymore, which may render many of the subprojects of DPF useless. Another approach as discussed in section 4.4, is to keep the original directed multi-graph, and use the editor as a pattern editor to model a linguistic model instead. With this approach, we can rearrange the linguistic and ontological metalevels in a linear fashion and replicate

the linguistic classifiers as we described it in section 3.3. Interestingly, this opens up for the possibility to use the model editor in two ways, firstly as an editor for developing new linguistic metamodels, secondly by modelling the ontological metamodels based on the linguistic metamodel. We can replicate the linguistic metamodel at each metalevel and thereby support linguistic extension, and at the same time keep the original directed multi-graph.

Figure 5.3 below illustrates deep metamodelling in DPF, with the top-most metalevel hardcoded in DPF as a directed multi-graph. As we illustrated the original layout in figure 5.2, the directed multi-graph is at the top-most metalevel and is the basic modelling elements in DPF. We can use the editor to create a default linguistic metamodel such as a UML class diagram, and replicate this metamodel at each ontological metalevel except from the bottom-most metalevel as illustrated in figure 5.3 below. The default linguistic metamodel will thereby always be at the default metalevel, which we decided to be at metalevel -1. Typically when we want to add new model elements, graph homomorphisms are checked in the metamodel adjacent above. However, when adding new linguistic elements, we have to check the default linguistic metamodel instead. By setting the default linguistic metamodel at metalevel -1, it will be easier to traverse the metamodelling stack since we know exactly what metalevel the default metalevel is at.

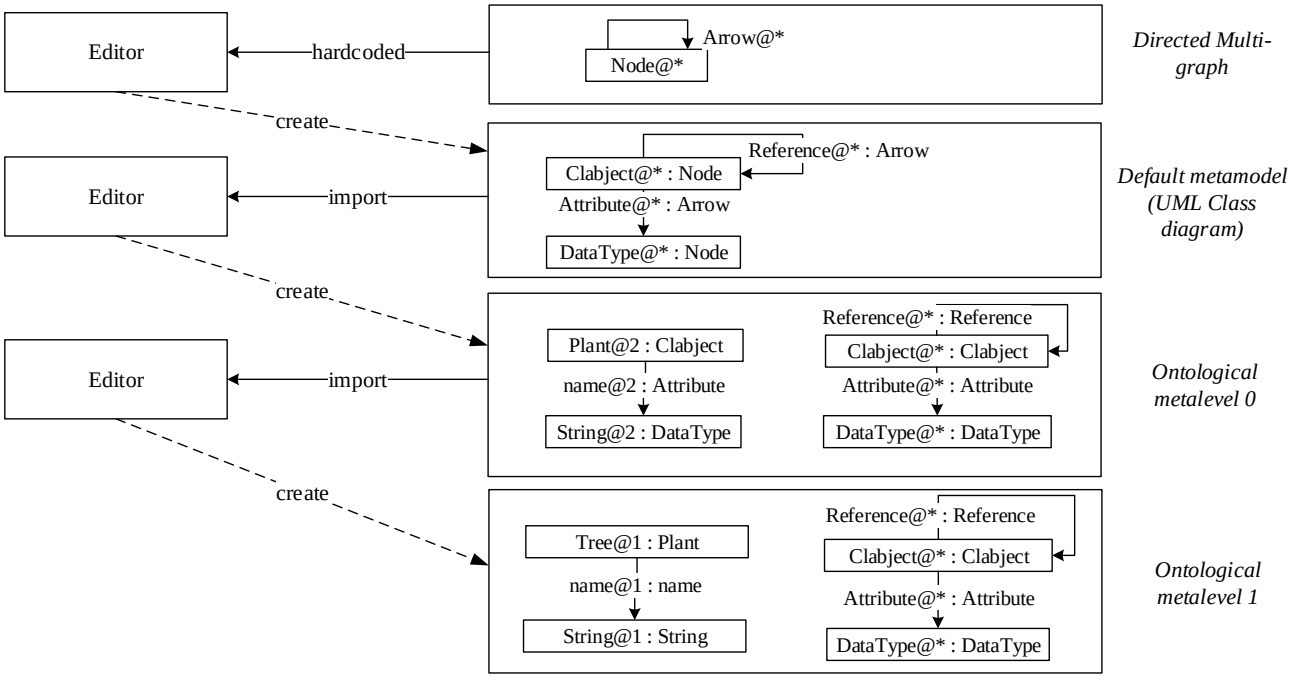


FIGURE 5.3: An illustration of deep metamodelling in DPF, with linguistic / ontological typing.

By looking at figure 5.3, we also see that this solution leads to that instead of having metalevel zero at instance level, we are now starting with the topmost ontological metalevel as metalevel zero. The reason for this is first of all because we do not necessarily know from the beginning how many metalevels we will develop. Secondly, it makes for a better seperation between linguistic and ontological metalevels. Setting the linguistic metalevels to negative integers and ontological metalevels positive integers, makes it easier to separate between linguistic and ontological metalevels. To implement such a solution, we need to:

1. Create a default linguistic metamodel such as a simplified UML Class diagram,
2. Replicate the default linguistic metamodel at each ontological metalevel,
3. Add a new palette group in the DPF palette for the default linguistic elements, and
4. Add mechanisms for distinguishing between ontological extension and linguistic extension.

First of all, we need to create the default linguistic metamodel that modellers will use to create ontological instances from. As stated in chapter 4, we will focus on UML class diagram notations as in the previous examples in this thesis. Figure 5.4 illustrates a simple default linguistic metamodel based on the directed multi-graph in DPF. The potency of the model elements in the linguistic metamodel should not be set before it has been ontologically instantiated, and it is therefore important that all linguistic model elements have an undefined potency.

FIGURE 5.4: A simple default linguistic metamodel in DPF.

The next step is to replicate the default metamodel in each ontological metamodel except from the bottom-most metalevel. By replicating the default linguistic elements, we can create instances of the default linguistic elements at any adjacent metalevel below and thus keeping strict metamodelling.

Replication Rule 5.1 (Default Metamodel)**.** *For each metamodel at metalevel m where* $m >= 0$*, create replications of each linguistic element from the metalevel above. This means if* $m = 0$*, then create instances of each model element in metalevel -1 with the exact same name. If* $m > 0$*, create instances of the replicated linguistic elements in the metalevel adjacent above with the exact same name.*

Even though we have now made it possible to create linguistic extensions, it is still not possible to connect ontological elements with linguistic elements. There may be arrows between ontological nodes in the metalevel adjacent above, but the replicated linguistic nodes is not directly connected to any of the ontological nodes in the metamodel adjacent above. All ontological nodes are however indirectly typed by a linguistic node, but to add an arrow between an ontological node-instance and a linguistic node-instance, we need to check the linguistic metamodel for conformance instead of the metamodel adjacent above.

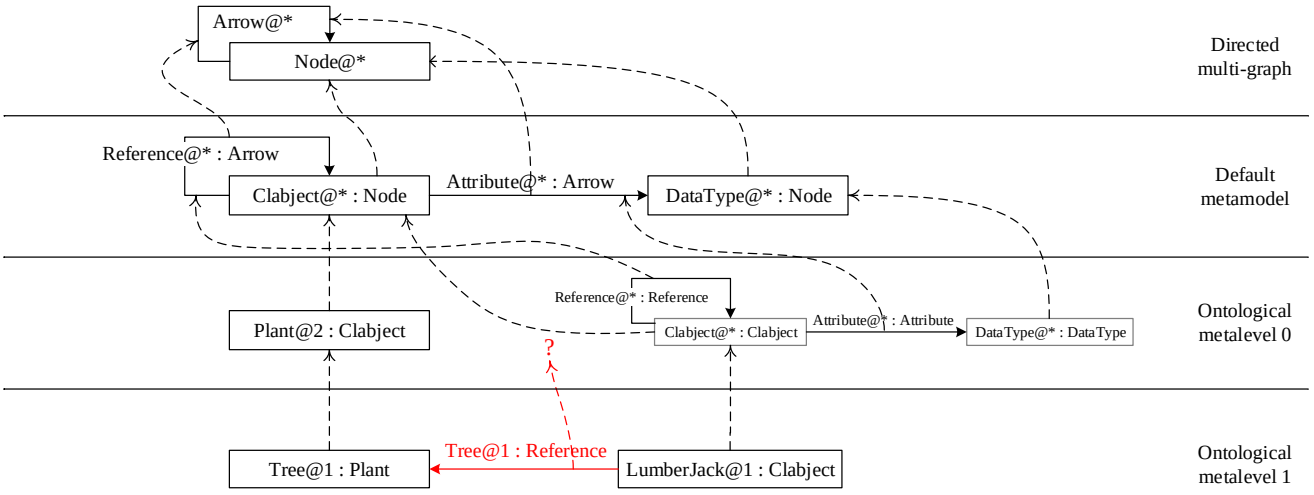


FIGURE 5.5: An illustration of the Plant example where it is currently not possible to linguistically extend ontological elements.

Figure 5.5 above illustrates that the current implementation of the model editor does not support linguistic extension of ontological elements. It is not possible to add a *Reference* arrow between *LumberJack* and *Tree* at metalevel 1. Looking at metalevel 0, we see that there is no *Reference* arrow between *Clabject* and *Plant*. However, *Plant* is indirectly typed by *Clabject*, and as we can see in the default metamodel *Clabjects* contains *References* to other *Clabjects*. Since both *Tree* and *LumberJack* is a *Clabject*, it should still by definition be possible to add a *Reference* arrow between *LumberJack* and *Tree*. To make it possible to connect ontological elements with linguistic extensions, we therefore need a mechanism to distinguish whether we are extending an element ontologically or linguistically. When adding a new *arrow* between two model elements, we need to find out whether the *arrow* we are instantiating is an ontological or linguistic instance. In practice, we need to:

1. Iterate over the arrows in the default metamodel and add the names of the linguistic arrows in a list, and
2. Check if the name of the arrow we instantiating matches any of the names of the linguistic arrows.

This means however that modellers are not allowed to create ontological instances with the same name as any linguistic element, and a proper naming constraint should be added to avoid confusion.

5.1.3 Summary

In the current solution we have implemented deep instantiation and dual classification. We have created a default metamodel which acts as a linguistic metamodel, and we have added support for linguistic extension. What we need to do next, is to elaborate for appropriate mechanisms such that we do not have to re-instantiate concepts at each metalevel to be able to access them further down the meta-hierarchy. To allow instantiations across several metalevels, we will elaborate for whether we need to flatten the metamodel with a set of replication rules, or whether we will investigate other possible solutions.

As briefly mentioned in section 3.2.2, it is also becoming apparent that we need to elaborate for where and how often model elements are allowed to change their value. We need to establish a definition of mutability along with the definition of potency. When it comes to Attributes, the name of the attribute should be kept consistent throughout the meta-hierarchy, which is something the specification of mutability can take care of. We also realize that there are no proper definition of datatypes yet. We currently have to define these manually even though the datatypes should be accessed from a predefined database such as the predefined datatypes in EMF.

To evaluate the current implementation and to elaborate suggestions for how we can overcome the aforementioned problems, the next section consists of the implementation of a lightweight code generator. The aim with this code generator is to make it easier to find new solutions, and maybe reveal unforeseen difficulties with the current implementation.

5.2 Evaluation by Code Generation

Even though we now have implemented deep instantiation, dual classification and support for linguistic extension, the current implementation does not support mutability and the definition of attributes and datatypes should be improved as stated in section 5.1.3. To find a solution to these limitations, we decided to implement a lightweight code generator that transforms DPF models to Metadepth models. In addition to making it easier to implement a solution for mutability and improve the definition of attributes, we hope the code generator will help us reveal unexpected bugs and discover new functionality that may benefit the current implementation. It is important to notice however, that the code generator is not intended for use, but is rather as a proof of concept to make limitations with the current implementation in DPF more apparent. To implement the code generator, we will first explain the design and implementation, then present the results of our findings.

5.2.1 Design And Implementation

To implement a solution that generates code based on DPF models, we need a tool that can take DPF models as input and generate a text based language based on these models. We remember from section 2.6.2 that DPF is based on EMF, and we know that DPF models are structured based on Nodes and Arrows which is persisted as a XMI file. Looking back at the DPF family as it was illustrated in figure 2.13, we recall that it is already developed a code-generator for DPF models, namely the DPF Code Generator developed by previous master student Anders Sandven [45]. This code generator is specifically created to generate code from DPF Models and would seem like a natural choice for our code generator. The DPF code-generator can take any DPF metamodel as source model and specify templates for each metamodel that can produce target models.

In our case, the main structure of the models will have single target structure in form of the linguistic metamodel in Metadepth. Since the linguistic metamodel equivalent in the current model editor is the default metamodel, we found it natural to create a mapping between the default metamodel to the linguistic metamodel. However, this also means that we need a code-generator that can take any source model that is either directly or indirectly typed by the default metamodel. We need to take a DPF source-model as input, find out what default element it is directly or indirectly typed by and then generate Metadepth code based on what default element it is. Therefore we need a code-generator framework that is capable of traversing the metamodel stack up till the default metamodel and then generate Metadepth code based on what linguistic element it is. Unfortunately, the current implementation of the DPF code-generator does not have support for such a feature. It is only possible to model on a model to model basis, meaning we have to create new mappings for each metalevel.

The second aspect is that the new model editor now has added support for deep instantiation through potency. The current DPF code-generator is not able to access the potency, which makes it difficult to create Metadepth code. It is possible to indirectly access the potency through utility classes, but we argue that it is not a suitable approach for our code-generator. The conclusion is therefore that as the DPF Code generator currently does not support the possibility of traversing the metamodel stack and does not provide access to the potency attribute, we decided that the current implementation of the DPF code-generator is not suitable for our implementation.

Another possible framework we can use is Acceleo [46], which is an open source code generator from the Eclipse Foundation. Acceleo is a pragmatic implementation of the MOF Model to Text Transformation Language (MOFM2T) standard for performing model to text transformations, and can generate code from any kind of metamodel compatible with EMF to any textual language. With Acceleo, we can define templates and generate code based on these. These templates can be specified exactly as the code is generated in the editor, meaning it is possible to access the potency attribute and traversing the metamodel stack. As we are also already familiar with Acceleo and how it works, it makes a good choice for the implementation of our code generator.

As a starting point, we first need to decide how to structure the templates in Acceleo. We need to decide how we should map the elements in DPF against the elements in Metadepth. By analysing the linguistic metamodel in Metadepth, we decided the following mapping as illustrated in figure 5.6 below:

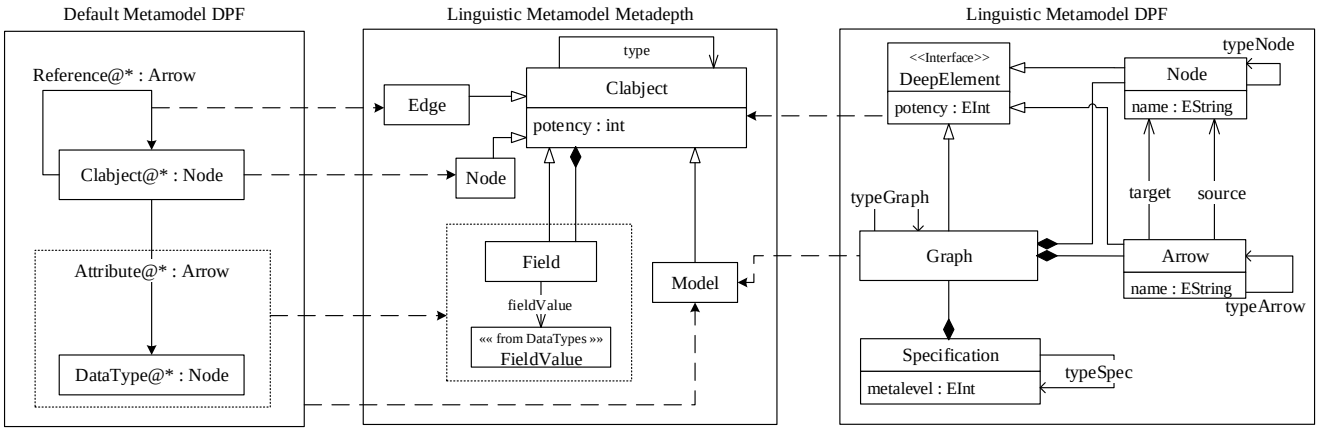


FIGURE 5.6: Mapping between the linguistic metamodel in Metadepth as seen in the centre, with the core metamodel in DPF to the right and the default metamodel in DPF to the left.

As we can see in figure 5.6 above, we have the linguistic metamodel in Metadepth in the centre of the figure, with the core metamodel in DPF

to the right. Elements such as potency and graph in the core metamodel in DPF is mapped to the model element and potency under Clabject in Metadepth. On the left side of the figure, we have the default linguistic metamodel in DPF, where we have mapped the Reference arrow to Edge, Clabject to Node and Attribute to Field. The whole graph consisting of all the elements in a metamodel is equivalent to a model in Metadepth, and the model itself, or the graph in DPF is mapped to a Model in Metadepth.

To be able to do this in practice, there are some factors we need to be aware of. In terms of Associations, it is in DPF simply modelled as a Reference arrow, while in Metadepth it is necessary with a field in the source-node and target-node, and an edge connecting them with one another. When it comes to attributes, in DPF the name of the attribute will be at the Attribute-arrow and the datatype on the DataType-node, as in a Entity-Attribute-Value model. When it comes to assigning default values to attributes, it is possible in metadepth at any metalevel, but in the current implementation in DPF, it is only possible to specify the value of an attribute when potency is zero, as the value is being set on the DataType-node. As a result of this, we decided to create the templates based on algorithm 1 below:

```
for each DPF metamodel, starting with the bottom-most metamodel and traversing the metamodels till we
reach the topmost metamodel do
    for each Clabject do
        generate a Node header with the following body:
        for each Attribute do
            generate a Field for the attribute;
        end
        for each incoming Reference do
            generate a Field for the incoming reference;
        end
        for each outgoing Reference do
            generate a Field for the outgoing reference;
        end
    end
    for each Reference do
        generate an Edge with source and target based on the reference Fields in the Clabjects;
    end
end
```

Algorithm 1: Code generation algorithm used to generate code from DPF models to Metadepth models

By creating templates in Acceleo based on the parameters described in algorithm 1 above, we can input the DPF metamodels, starting with the bottom-most metamodel. We will then iterate over the Nodes in the bottom-most metamodel in DPF and check the name of the Node if its Clabject or not. If its a Clabject, the algorithm will check its outgoing Arrows and their names, generating its corresponding Attributes and Reference fields. When all the Clabjects is generated, the algorithm will iterate over the Arrows in the DPF metamodel, checking its names if its a Reference or not. If its a Reference, we will generate its corresponding Edges in Metadepth. As briefly mentioned in section 4.2, it is possible to import metamodels in Metadepth, and therefore we will add an import statement on the top of each Metadepth model, loading the adjacent metamodel above. This way

the generator will start by generating code from the bottom up, and when all the models are generated, the bottom-most metamodel in Metadepth can be loaded, while the rest of the models are loaded automatically.

Figure 5.7 below illustrates the results from the code-generator with the DPF source-models to the left and the generated code to the right.

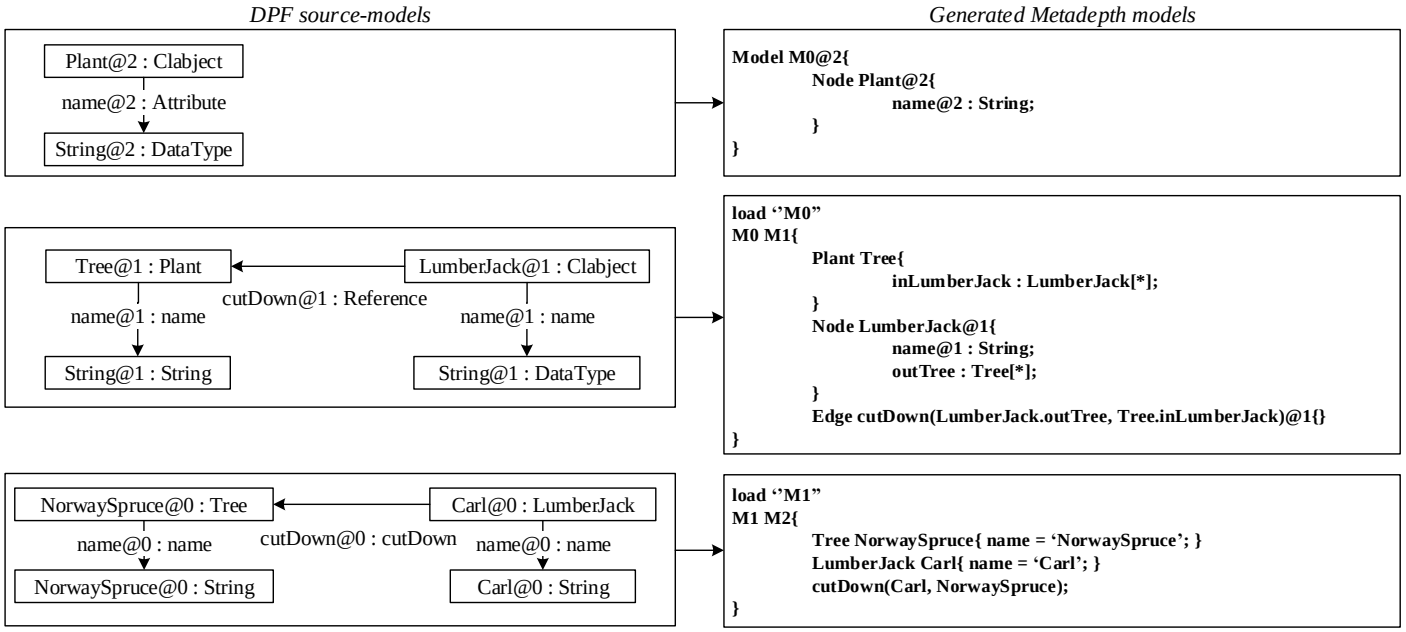


FIGURE 5.7: An illustration of the DPF source-models to the left and the generated Metadepth models to the right.

As we can see, we started the code generation by iterating over the bottommost metalevel in DPF as seen to the bottom left in figure 5.7 above. As the code generator is reading the potency of the model elements to be zero, it produces Metadepth code accordingly. When potency is zero, the fields in Metadepth will have their values set and the Edges will be instantiated with its appropriate syntax as seen to the bottom right in figure 5.7. The code generator also puts a line of code before the model code, telling Metadepth to load the metamodel adjacent above such as M_1. The Models in DPF does currently not have a name, so the Model names in Metadepth is set based on the number of metalevels in the loaded meta-hierarchy, which in this case is three. When the bottom-most metamodel is generated, the code geneartor will traverse up the DPF metamodel stack to the metamodel adjacent above. By following this process till we reach the top-most ontological metamodel O_0, the generator will output Metadepth code as illustrated to the right in figure 5.7. The bottom-most metamodel in Metadepth can then be loaded into Metadepth, while the metamodels adjacent above will be loaded automatically through the load statement.

As this section has elaborated a solution for the code generator, the next section will analyse the results and present some possible solutions for improvement of the current implementation of deep metamodelling in the model editor.

5.2.2 Results And Possible Improvements

Generally the code generation from DPF models to Metadepth models went straight forward except from some minor difficulties, such as making a proper definition of Attributes and its DataType and the syntax of References. As the current implementation of the default metamodel in DPF only concerns Clabjects, Attributes and References, this section will only focus on the limitations of these concepts. Language dependent factors such as proper naming and additional features will not be mentioned in great detail here, but elaborated for in the conclusion as possible further work instead.

Model
The current definition of metamodels in DPF is defined by the Graph itself. Currently the Graph has potency, but Graphs does not have proper naming. The Graph class has a name attribute, but this is not implemented in practice. When analysing Metadepth we also realized that a feature of importing external models could be useful in the future along with binding models together and relate metamodels at different metamodels. Also the definition of strict and extendible modes for linguistic extension could enhance the definition of deep metamodelling in DPF in the future. These features is however currently outside the scope of this thesis, and will be discussed as future work instead.

Clabject
When investigating Nodes in Metadepth, we realized that properties such as abstract Nodes and added support for Generalization should be discussed as added features in DPF in the future. These properties will not have a high priority, but will be discussed later in this thesis as the DPF Editor is reaching a more mature state.

Attribute
The current implementation of Attributes as we see in the default metamodel is based on the Entity-Attribute-Value model, as illustrated in figure 5.8 below.

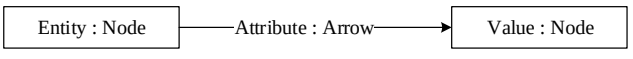


FIGURE 5.8: The Entity-Attribute-Value model, modelled in the DPF Model Editor

This means that in the default metamodel in DPF, an Attribute consists of a Clabject with an outgoing Attribute Arrow pointing at a Node containing the value of the Attribute. In metamodelling, this means we will first define the DataType of the Attribute, then instantiate it with its value. This solution works fine as long as we are only concerned with two metalevels, but as soon as we add intermediate metalevels the limitations of this approach is becoming more apparent. While creating the code-generator for Metadepth, it was becoming clearer that a proper naming constraint should be added on Attributes. In the current implementation it is possible to set different names of the Attributes at each instantiation step, when it should rather be kept consistent. This can be solved through mutability and setting the mutability of Attribute names to 1. This way, the name of the Attribute can only be set once, and the mutability will act as a constraint that ensures a consistent name throughout the modelling hierarchy.

When modelling the Plant hierarchy, it was also becoming more apparent that it is currently not possible to ensure specifications across multiple metalevels. If we want to specify a *name* attribute on Plant, we currently have to re-instantiate it at each metalevel until the metalevel of concern. Otherwise it would not be possible to instantiate the attribute, since strict metamodelling dictates that the instance of relationship crosses the boundary between two metalevels only. To overcome this limitation, we need to develop a solution that can automatically re-instantiate requirements at intermediate metalevels. To implement such a solution in practice, we can follow the replication rules as Rossini (et. al.) defined it [34] [47]. This way, the containing attribute will automatically be replicated, following the semantics of instantiation.A more thorough explanation of how this is done in practice will be presented in section 5.3.2.

Other limitations with the current definition of Attributes include that currently we have to define DataTypes manually, when they ideally should be defined from a predefined storage of standardized datatypes such as EMF instead. This can be made possible by extending the current directed multi-graph to an E-Graph [48]. In practice, this means we keep the current graph, but each element will have the possibility of an added attribute in form of a DataNode. With an added DataNode defined in the core metamodel in DPF, we can access predefined DataTypes that is predefined in EMF. Through the implementation of DataNodes, we also realize that it becomes easier to define OOP properties such as visibility of attributes and similar. The implementation of the E-Graph along with the formal definition will be presented in section 5.3.3.

Reference
In Metadepth it is possible to add Fields to Edges, but the current implementation of Associations in DPF is made up of a Reference arrow only. This solution yields a simple model of situations where Attributes are not needed on References, but on the downside it means to be able to model

compositions of Nodes we would have to add another type of Arrow, or otherwise investigate alternative solutions such as using Nodes as References instead of only an Arrow. A third solution is by modelling References by using an E-Graph. By adding attributes to the Reference Arrow, we can easily make it possible to define properties such as composition of Nodes. In Metadepth it is also possible to define properties such as ordered and unique on Edges. This can be made possible as well by extending the current graph to an E-Graph as we will explain in greater detail in section 5.3.3.

Other Possible Extensions
Methods is a central concept in UML Class diagrams, and we are therefore adding support for this in the default metamodel. We will also need to add enums, but this will be elaborated under future work.

5.2.3 Summary

By developing a code-generator and transforming DPF models to textual Metadepth models, we have revealed limitations with the current approach as described earlier in this section. Currently we still have to re-instantiate Attributes to be able to access them further down the meta-hierarchy. The semantics of instantiation states that containing Attributes will be instantiated as the Clabject is instantiated, and to make this possible in practice, we will implement a set of replication rules for Attributes.

Secondly, we revealed that under certain situations the value of model elements should not be allowed to change such as the name of an attribute. To allow restrictions on how when the value of model elements can change, we will introduce the concept of mutability.

Thirdly, to make it easier to define DataTypes amongst others, we will extend the directed multi-graph to an E-Graph.

5.3 Extending DPF - Part Two

By implementing the code generator, we made it clear that the current implementation of the model editor can specify the instantiation depth of model elements, but for example it is currently not possible to instantiate a model element at metalevel 2 and instantiate it at metalevel 0. We currently still need manually to re-instantiate model elements at intermediate metalevels to be able to access them further down the meta-hierarchy. To solve this problem, we will follow a set of replication rules with inspiration from Rossini (et.al.) [34] [47]. In the code Generator it also became more apparent that certain values such as the name of attributes should not change throughout the meta-hierarchy, but stay consistent instead. This will be solved by the concept of mutability. Last but not least, we will extend the current multi-graph to an E-Graph to (amongst others) allow easier access to predefined datatypes in EMF.

5.3.1 Model Flattening Semantics

The semantics of instantiation is intimately related to the related elements and its containing elements. When an element x is instantiated to create an element y, the semantics of instantiation dictates that every containing element of x becomes an element of y with the same name and value as the appropriate type [32]. In terms of attributes, this means that for a class containing attributes, every instance of the class (object) should contain the attribute (slot) with the same name and value as its type. In order to provide a solution that ensures this in practice, we will flatten the metamodel with basis on a set of replication rules for attributes. The idea is to create linguistic instances of Clabjects, add attributes and define potency. When the instances of these Clabjects are created, its containing attributes are automatically replicated as well. The definition of the replication rule for Attributes in this thesis is:

Replication Rule 5.2 (Attribute)**.** *When instantiating Clabjects, iterate over the Attributes contained in the Clabject we are instantiating. If the contained Attribute has a potency > 0 then re-instantiate the Attribute with the same name as its type, and add it to the instantiated Clabject.*

By following the replication rule for Attributes, we will automatically replicate the attributes at each instantiation-step until potency is zero. This way the Attribute is made available for modellers further down the meta-hierarchy without having to explicitly redefine attributes at intermediate metalevels. The Attribute is also hidden from the palette, such that from a modellers perspective it will look like we specify an Attribute at one metalevel, and instantiate it several metalevels below. In fact the models are still the same, only that Attributes are replicated at intermediate metalevels to allow access further down the meta-hierarchy.

When modelling requirements across multiple metalevels, we also realize that the value of the model element might change. The next section will discuss the concept of mutability, which provides a well-defined solution to restrict how and when a model element can change its value.

5.3.2 Mutability

As potency is a way to restrict the instantiation depth of model elements, mutability is a way to restrict the number of times it is possible to change the value of a model element. To implement mutability, we first need to have a look at current definition of the core metamodel in DPF. We want to add mutability to all model elements with a name such as Node and Arrow. To do this we will implement a MutableElement interface containing a mutability integer, much like the DeepElement interface is used to define potency. We decided to set the default value of mutability to -1, which means that the value of a model element can change as many times as the potency of the element.

As potency is defining how many sub-sequent metalevels a model element can be instantiated, it also means that the potency sets the boundary for how many times the value of the model element can be changed. When the potency of a model element has been set, it can only be instantiated at a certain amount of sub-sequent instantiation steps, which also means it is only possible to change its value as many times as the number of its potency. This means the mutability can not be higher than the potency. However, whether the mutability is equal or higher than the potency of the model element does not make any difference in the system. If the mutability is higher than the potency, it might only appear confusing for modellers, and will not impact the system. For this reason, a constraint on mutability will be discussed under future work instead.

Figure 5.9 below illustrates the core metamodel in the model editor with the MutableElement interface added to the Node and Arrow.

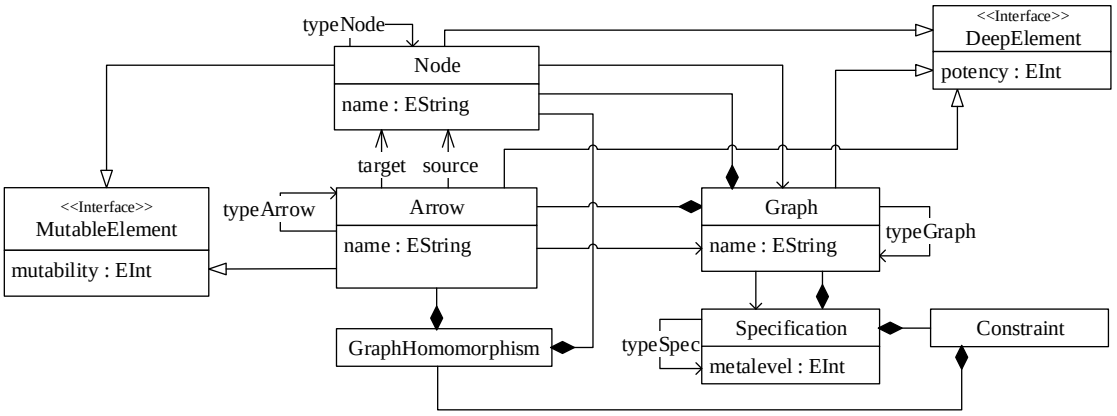


FIGURE 5.9: The core metamodel in DPF with added potency and mutability

Figure 5.9 illustrates the modified core metamodel in the model editor with added mutability. All linguistic model elements will have a default mutability which is set to -1, except from the default metamodel where we specify the behaviour of the ontological elements. When it comes to **Clabjects** and **References**, we have decided to keep an undefined mutability. This means in practice that References and Clabjects are mutable, and its value can change as many times as we would like.

The mutability of **Attributes** will have an initial mutability of one, while DataTypes will have a mutability of two. By defining Attributes with a mutability of one, we will restrict the possibility of changing the name of the Attribute further down the meta-hierarchy. When an Attribute is instantiated, its name is being set and the mutability is decreased to zero. The mutability will then act as a constraint that ensures that the name of the Attribute has the same name at every sub-sequent metalevel. Regarding **DataTypes**, the idea is that first the name of the DataType Node is specifying the DataType of the Attribute, then at instance level we can set its value.

In practice this means that the value has to be set twice, and a mutability of two is the most natural choice.

It is now possible to access Attributes further down the meta-hierarchy without having to re-instantiate it at intermediate instantiation-steps. Secondly, to ensure naming consistency and provide a well-formed solution to restrict the number of times a model element can change its name, we have now implemented mutability. Thirdly, we will now improve on the definition of DataTypes. Currently we have to define datatypes manually, which naturally is prone to error. The next section will therefore provide an explanation of how we will extend the current directed graph to an E-Graph.

5.3.3 E-Graphs

In this section we will present the original directed multi-graph extended with node- and arrow-attributes. We may recall definition 2.1 where we defined a directed multi-graph as a set $G = (G_N, G_E, src^G, trg^G)$. In this section, we will extend the directed multi-graph by following the approach in [48] to define a new kind of graph, called E-Graph.

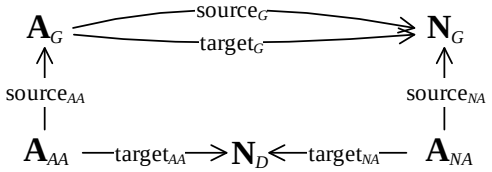
Definition 5.3 (E-Graph and E-Graph Morphism)**.** *An E-graph* $G = (N_G, N_D, E_G, A_{NA}, A_{AA}, (source_j, target_j)_{j=G,NA,AA})$ *consists of sets*

— N_G *and* N_D *called graph resp. data nodes,*
— A_G, A_{NA}, A_{AA} *called graph, node attribute and arrow attribute arrows respectively,*

and source and target functions:

— $source_G : A_G \to N_G$, $target_G : A_G \to N_G$ *for graph arrows;*
— $source_{NA} : A_{NA} \to N_G$, $target_{NA} : A_{NA} \to N_D$ *for node attribute arrows;*
— $source_{AA} : A_{AA} \to N_G$, $target_{AA} : A_{AA} \to N_D$ *for arrow attribute arrows;*

such that the following diagram commutes:



Let $G^k = (N^k_G, N^k_D, A^k_G, A^k_{NA}, A^k_{AA}, (source^k_j, target^k_j)_{j \in (G,NA,AA)})$ *for* $k = 1, 2$ *be two E-graphs. An E-graph morphism* $f : G^1 \to G^2$ *is a tuple* $(f_{N_G}, f_{N_D}, f_{A_G}, f_{A_{NA}}, f_{A_{AA}})$ *with* $f_{N_i} : {N_i}^1 \to {N_i}^2$ *and* $f_{A_j} : {A_j}^1 \to {A_j}^2$ *for* $i \in (G, D)$, $j \in (G,NA, AA)$ *such that* f *commutes with all source and target functions, e.g.* $f_{N_G} \circ {source^1}_G = {source^2}_G \circ f_{A_G}$.

By following definition 5.3 taken from [48], we form category **EGraphs**. With this new type of graph, we will be able to model with two additional

types of arrows (NodeAttribute and ArrowAttribute), as well as one additional type of node (DataNode). With NodeAttributes, we can specify additional properties of Nodes, such as values of Attributes with datatype specified directly from EMF. We are also able to add additional properties of for example Clabjects, such as visibility, isAbstract, isInterface, isFinal and similar. With ArrowAttributes it also becomes easier to specify References between Clabjects, as we now are able to add attributes to References similar to adding Fields to Edges in Metadepth.

The next section will present the extension of the core metamodel in the model editor to an E-Graph.

5.3.3.1 Implementation

To extend the current definition of a directed multi-graph in DPF to an E-Graph, we need three components according to definition 5.3. We need a NodeAttribute(E_{NA}), ArrowAttribute(E_{EA}) and a DataNode(V_D) as illustrated in figure 5.10 below. We have decided that to extend the current Graph in DPF to an E-Graph, we have added a NodeAttribute, ArrowAttribute and a DataNode class. The DataNode class will represent a new type of Node with three attributes: *value*, *datatype* and *isDatatype*. As we recall from the current definition of attributes, we first define the DataType, then instantiate the DataType node with its corresponding value. The idea is to create a DataNode that works two ways, first as a DataType node where we can specify datatypes directly from the predefined set of datatypes in EMF. Secondly, we aim at using the DataNode to contain values. To distinguish between whether the DataNode is specifying datatype or datavalue, we have added a *isDatatype* boolean. If the *isDatatype* attribute is true, it is a datatype, otherwise it is a datavalue.

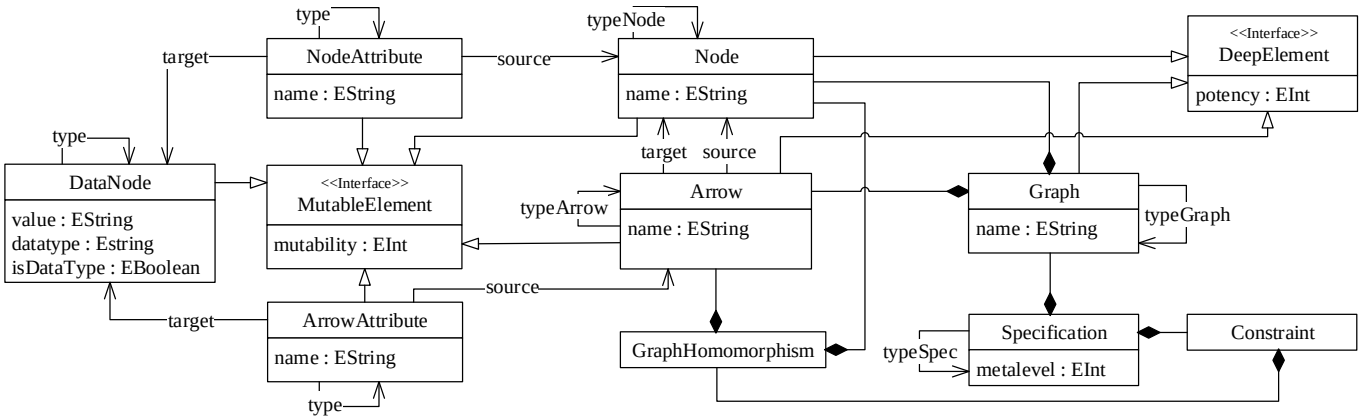


FIGURE 5.10: The new core metamodel in DPF with added potency, mutability, NodeAttribute, ArrowAttribute and DataNode.

Through the separation of the DataNode concept, we aim at specifying a deep hierarchy of datanodes by first defining a hierarchy of the desired datatype, then defining the datavalue of the attribute. Datavalues can also be instantiated further, thus providing a meta-hierarchy of datavalues. This hierarchy of datavalues can be seen in relation to dual fields as it was defined by Atkinson and Kühne [32]. A single field is a field containing a certain value, while a dual field is a set of single fields. Each time the value of a datanode is changed, it will represent the equivalent of a single field as it was defined in [32]. With inspiration from the linguistic metamodel in Melanee, we will not define potency on DataNodes. This is because DataNodes will be a property of Nodes or Arrows much like Attributes is contained in Clabjects in Melanee. Therefore we have decided that the instantiation depth of a datanode will be determined by the Node or Arrow it is contained in instead.

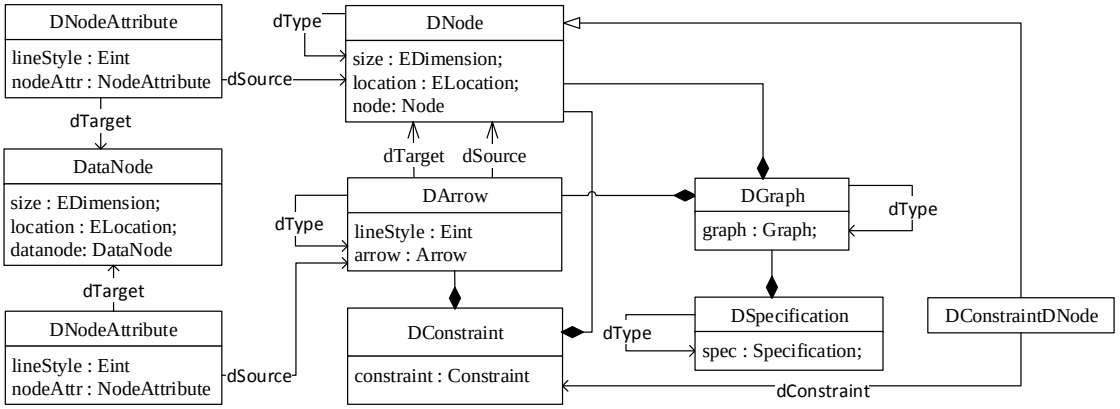


FIGURE 5.11: The extended diagram metamodel in DPF with added support for NodeAttribute, ArrowAttribute and DataNode.

Figure 5.11 above, illustrates the extended diagram metamodel with added support for *NodeAttribute*, *ArrowAttribute* and *DataNode*. By adding visual representations of the core elements in the diagram metamodel, we can visualize DataNodes, NodeAttributes and ArrowAttributes. The abstract syntax visualization of these elements has been made with inspiration from [48], with DataNodes visualized as ellipses with dotted lines, and NodeAttributes and ArrowAttributes as dotted arrows.

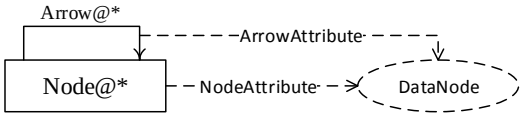


FIGURE 5.12: A visualization of the E-Graph as it is implemented in the DPF Editor.

With this new E-Graph, we can redefine the definition of Attributes in the default metamodel from Entity-Attribute-Value to an Attribute Arrow

and single Attribute Node as illustrated in figure 5.13. With support for attributes on Nodes, we have now added NodeAttributes with DataNodes holding the datatype, value and default value of the Attribute. We have also added support for methods with containing parameters, with inspiration from [48]. With NodeAttributes we can also define properties of method parameters such as direction and order. By defining direction, we can specify whether the parameter is an input parameter or return parameter, and its order is telling us the order of the input parameters as some programming languages such as Java require this.

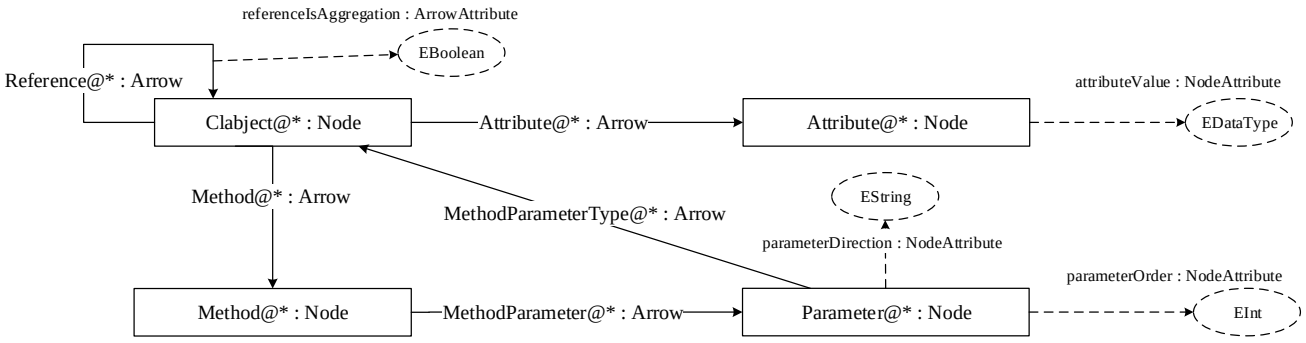


FIGURE 5.13: The extended default metamodel in DPF.

As we can see from figure 5.13, we can now specify attributes on Nodes and Arrows. In fact, we can specify properties of the Nodes and Arrows which follow the definition of their attached Node or Arrow. The attributes does therefore not have any potency, but will only have a mutability defined. As explained earlier, through mutability we can define hierarchies of datatypes and values that can form a structure like dual fields as Atkinson and Kühne described it [32]. The potency of the attributes is thereby defined by its attached element, and replicated as the Node or Arrow is being instantiated. In fact, the attributes in the graph will follow replication rules much like the one used to replicate Attributes in Clabjects. When a new instance of for example a Node is created, its NodeAttributes will be automatically replicated as corresponding to the outgoing NodeAttributes of its type in the type-graph.

Replication Rule 5.4 (Node- / ArrowAttributes)**.** *Instantiate all Node and ArrowAttributes along with its corresponding Node / Arrow. When instantiating a new Node / Arrow, instantiate all Node- / ArrowAttributes of the corresponding type as well.*

5.3.4 Summary

At this stage we have implemented deep instantiation with replication rules ensuring requirements across multiple metalevels without the need to explicitly redefine concepts. Secondly we have implemented the concept of mutability, which allows a concise definition of how and when the value

of a model element can change. Thirdly we have extended the original graph to an E-Graph, which makes it easier to specify certain details of the models. Extending the graph to a E-Graph also makes it possible to use datatypes that is predefined in EMF. Another interesting aspect as we briefly mentioned when introducing the model editor in section 2.6.2, is that one of the greatest strengths of the model editor is that it can be used to model templates (patterns). The default metamodel is one example of a template of a simplified UML Class diagram, but the editor can be used to model basically any kind of template, and utilize it with deep metamodelling. The next section will illustrate how we made this possible.

5.4 Templates

In chapter 2, we briefly introduced the model editor with its general nature and ability to be used to develop pattern languages. The default metamodel we have developed so far in this thesis is one type of a pattern language, namely a UML Class diagram. However, deep metamodelling is not only about UML Class diagrams, it is about multi-level metamodelling in general. This can also mean other types of diagrams such as Expression diagrams, Business Process Modelling diagrams, UML Package diagrams, State diagrams and so on. Modelling new types of diagrams is unfortunately not possible in the current implementation of the model editor, but would greatly benefit the usability of the model editor as a whole.

The second aspect of the model editor is that we want to make the DPF models as generic as possible, meaning that the modelling hierarchy we end up developing should be possible to be run in other systems as well. It should be possible to import any modelling hierarchy that is modelled by the model editor into a target system. The general nature of EMF is to create new platform independent models and create editors based on the platform independent model. It should be possible to map the top-level metamodel in DPF to any platform independent model of concern, so that the DPF models can be used in other systems as well.

To make it easier to model pattern languages (templates), and to make it easier to map the DPF models to other systems, we decided to create a new instance of the core metamodel in DPF. We decided to develop a metamodel of an enriched graph that resides between the default metamodel and the core metamodel. Secondly, with this new metamodel we can also use the NodeAttributes and ArrowAttributes to extend the enriched graph with model elements that can represent specific functionality such as *Inheritance* and *Containment*. With *Containment*, we can nest Nodes inside each other, and provide a more generic approach to the replication rules of Attributes. We know that Attributes is contained in Clabjects, and we have developed a replication rule specifically for Attributes. There might however be other types of elements contained in Clabjects such as Methods, or any type of nesting of DeepNodes such as Clabjects inside Clabjects. With

the *Containment* feature, we can establish a generic replication rule for this, and allow modellers to define directly in the template what elements that should be contained in other elements.

5.4.1 Enriched Graph

As explained in the previous section, we decided that by creating an instance of the core metamodel, we can create an enriched graph with additional functionalities. An additional metamodel between the core metamodel and the default metamodel also makes it easier to map the top-level metamodel in DPF with other types of core metamodels; this would allow the meta-hierarchy of metamodels to be run on other systems as well. This new metamodel of an enriched graph will consist of the basic building-blocks of our editor, as illustrated in figure 5.14 below.

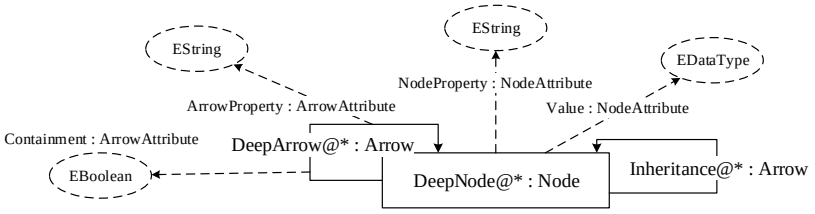


FIGURE 5.14: The current enriched graph in the DPF Model Editor

The main elements considering deep metamodelling, is *DeepNode* along with *DeepArrow*. Both these elements are capable of deep instantiation, and the instantiation depth is defined by its attached potency. Both *DeepNode* and *DeepArrow* can also be defined with additional properties in form of *NodeProperty* and *ArrowProperty*. These properties are defined by the EString datatype, and can be instantiated in the template metamodel with default values for a given property(see figure 5.16). We also decided to add *Value* to the *DeepNode*. This is to ensure a generic way for the editor to access values that might be assigned to DeepNodes such as Attributes. By defining the *Value* here, it is easier to ensure that the value of a given element is instantiated when it is supposed to, such as for *slots* at potency 0 in OOP. Lastly and with inspiration from Melanee, we decided to add *Containment* on *DeepArrow* to support nesting of *DeepNodes*. With this feature we can define Attributes as containment inside Clabjects in a generic way, and replicate elements accordingly. This will generalize the replication rule for Attributes we defined in replication rule 5.2 to a generic replication rule for containments:

Replication Rule 5.5 (Containment)**.** *When instantiating a Clabject, check the type of the instantiated Clabject if it has any outgoing arrows that is containment arrows. If there is any containment arrows, create instances of the containment arrows along with instances of the target node of the containment arrow.*

In practice this means if the type of an instantiated *DeepNode* has any contained *DeepNodes*, they will be automatically instantiated along with the new *DeepNode*. Instead of hardcoding that all Attributes should be replicated along with Clabjects, we can now define outgoing Attribute arrows from Clabject as containment, and they will be instantiated automatically along with the instantiation of the Clabject. For example if we would introduce Methods - instead of hardcoding in the model editor that all Methods should be replicated according to some replication rule for Methods, it will follow the generic replication rule for containment instead. Lastly, we also decided to add *Inheritance* in the enriched graph, but this is currently regarded as outside the scope of this thesis, and will be discussed under further work instead. Adding additional functionality in the enriched graph also makes us avoid that other plugins in the DPF family are getting corrupted. The enriched graph will be hardcoded in the model editor, and each type of model element will have their own functionality, yielding a solution where we adhere to a graph structure as the core metamodel, and adding more functionality as we add new metamodels. The result is a platform independent modelling hierarchy with the most abstract concepts at the top level, and the least abstract concepts at the bottommost metalevel.

5.4.2 Platform Independent Modelling Hierarcies in DPF

Figure 5.15 on the next page illustrates the new hierarchy of metamodels in the model editor and its intended use. As we defined earlier, the core metamodel in DPF can be defined by a broad range of external resources such as UML, XML, databases, ontologies, OO interfaces and more. The core metamodel is the meta-metalanguage in the DPF meta-hierarchy, and is currently consisting of an E-Graph. The idea is that this meta-metalanguage can be interchanged with any type of language, as long as there exist elements that can be mapped to each element in the metamodel of the enriched graph we defined in figure 5.14.

Earlier in this thesis we have focused on a default metamodel of a UML Class diagram as a template for our editor. We have used the default metamodel as a linguistic metamodel, which is replicated at sub-sequent instantiation-steps to allow linguistic extension. With this new structure, each template is still at metalevel -1, and will be replicated at each subsequent metalevel except the bottommost ontological metalevel. The difference now is that it is possible to create new templates other than UML Class diagrams. Modellers can use the enriched graph (metalevel -2) to model almost any kind of graph based on the modellers needs, with the additional possibility to linguistically extend ontological metamodels with elements from the template metamodel. It will be possible to model almost any kind of metamodelling hierarchy with support for deep instantiation and linguistic extension. It will be possible to define mutability on any model element in the new template, specification of requirements will be possible across an arbitrary number of metamodels.

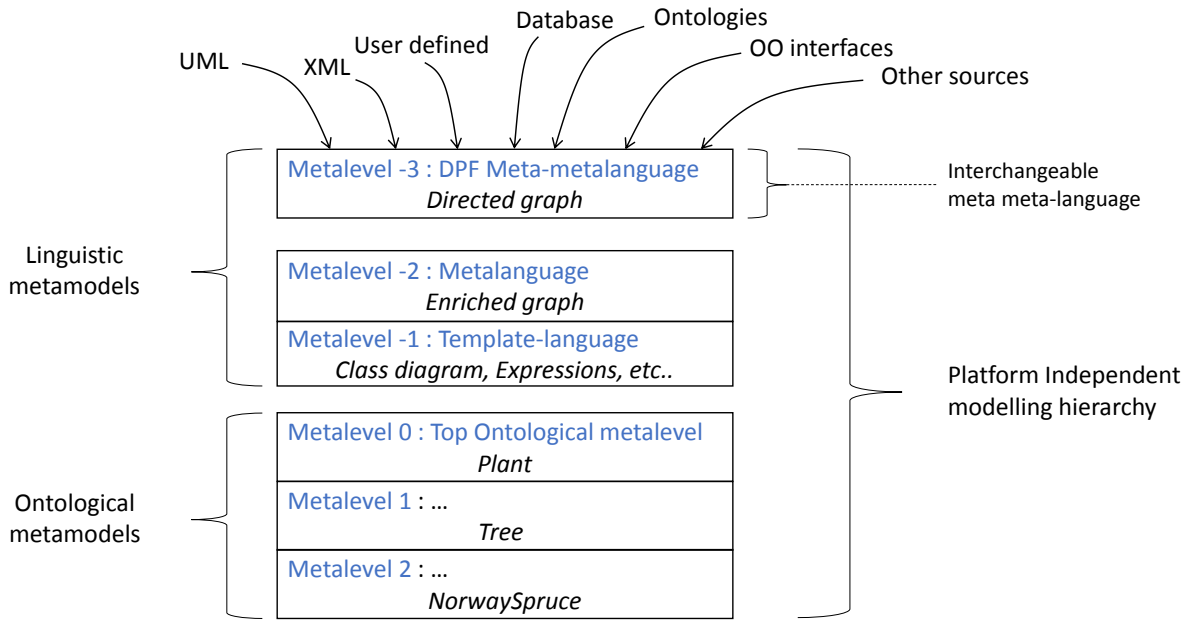


FIGURE 5.15: A Platform Independent Modelling Hierarchy in the DPF Model Editor.

Figure 5.15 illustrates an example of a Platform Independent Modelling Hierarchy (PIMH) in the DPF Model Editor. As we see, the meta-metalanguage at metalevel -3 can be defined by a broad range of external resources. In the current model editor the top-level meta-metalanguage consists of an E-Graph, but other tools may consist of other types of structures. The idea with the PIMH in figure 5.15 is to provide a meta-hierarchy of metamodels that can be used in other systems as well. The top-level meta-metalanguage should therefore be as interchangeable as possible. As we described earlier in this section, we have therefore added a metalanguage between the template metamodel and the meta-metalanguage to make this process easier. The metalanguage resides on metalevel -2, and is used as the main building-blocks of the meta-hierarchy in DPF with added support for functionalities such as inheritance and nesting of nodes. The metalanguage will be used to model linguistic metamodels in form of templates, which is replicated at the ontological metalevels to allow linguistic extension.

5.4.3 Summary

Even though we have implemented a well-defined meta-hierarchy as illustrated in figure 5.15 , we realize that the models are growing bigger and bigger and it is getting increasingly difficult for modellers to gain an overview of the models. The abstract syntax we have defined up till now is not very intuitive for modellers to use either, and therefore a more user-friendly concrete syntax is necessary as well. Figure 5.16 below illustrates

the current version of the UML Class diagram template that is used in this thesis.

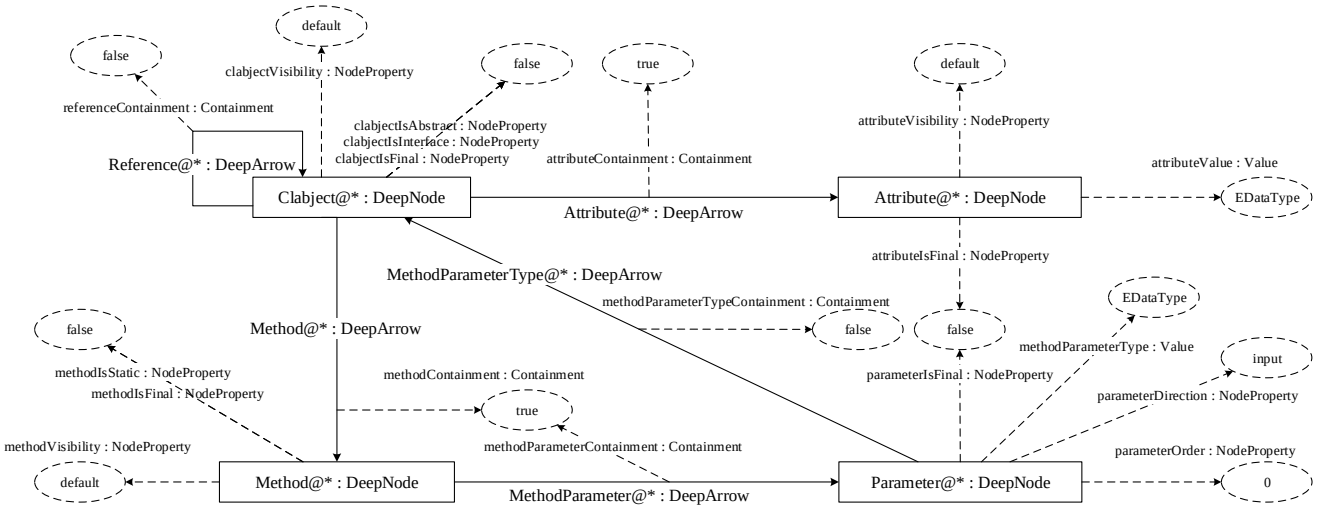


FIGURE 5.16: An illustration of the current template of a UML Class diagram in DPF.

As we see, it is not easy for modellers to develop new models based on this diagram in its abstract syntax. In the next section, we are therefore going to improve on the definitions made so far in this thesis, by adding a more visually appealing and user-friendly concrete syntax. We will present the implementation of the corresponding concrete syntax, which in the end is the syntax that modellers will use when modelling in DPF.

5.5 The Concrete Syntax

As illustrated in figure 5.16 in the previous section, it is becoming apparent that modelling larger models is a both difficult and cumbersome process. The purpose of MDE is to simplify the design process and increase productivity, but as the models are growing bigger, the models are getting increasingly harder to understand and maintain. A more user-friendly syntax is therefore necessary to improve the useability of the system. This section will introduce the development of a new concrete syntax for the abstract syntax we have defined earlier in this thesis. To do this, we will first of all need a visual metamodel, describing the structure of the model elements in the concrete syntax. Secondly, we need to specify the mapping between the abstract and concrete syntax. First of all, we will introduce the new visual metamodel that is used in this thesis.

5.5.1 The Visual Metamodel

In chapter 2, we briefly mentioned that the syntax of a DSML can be divided into three parts: the definition of the abstract syntax, the concrete syntax and the mapping between the abstract and the concrete syntax. To implement the concrete syntax, we will use the DPF Visualization Editor and extend it to visualize template metamodels in a concrete syntax. To do this, we first need to create a visual metamodel of the concrete syntax. We need to elaborate for how the template is going to be visualized. In terms of the current template as illustrated in figure 5.16, UML Class diagrams already has a fixed visual syntax in which we should follow. Classes are visualized as rectangles with compartments for Attributes and Methods. This means we will create a visual metamodel consisting of the visual elements as illustrated in figure 5.17.

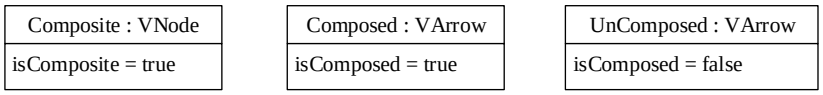


FIGURE 5.17: An illustration of the visual metamodel used to visualize UML Class diagrams in a concrete syntax.

The *Composite* VNode will be used to visualize Clabjects with containing Attributes and Methods. We have also created a *Composed* VArrow, which will be used to determine what elements that will be contained in the *Composite* VNode. The last element is a *UnComposed* VArrow, and will be used to visualize Arrows between concrete syntax elements, such as References between Clabjects. The next step is then to elaborate for how we will map the abstract syntax with the concrete syntax, the diagram metamodel with the visual metamodel. A problem with the current Visualization Editor is however that the mapping between the diagram metamodel and the visual metamodel only works on two metalevels at a time. This means that if we want to model more than two metalevels, we need to redefine the mapping between the diagram metamodel and the visual metamodel at each metalevel. Instead we want a consistent solution where we dont have to redefine the visualization of each model element at each metalevel. A UML Class diagram already have a defined visualization which should not change at each metalevel, but rather be kept consistent throughout the meta-hierarchy.

To do this we need to specify how the model mapping will be done, and secondly we will create a new wizard that follows these principles to create a new concrete syntax diagram. The next section will elaborate for how this is done in this thesis.

5.5.2 The Model Mapping

For the mapping between the diagram metamodel and the visual metamodel we created in figure 5.17, we decided that we are going to specify

the mapping only once. We want to specify the mapping based on the template metamodel, and then make the editor automatically map sub-sequent instances based on what template element it is typed by. What we want is a solution where the only thing modellers need to do is to load a diagram metamodel and the concrete syntax mapping will be made automatically.

When modelling new template diagrams, we currently need to specify the mapping either programatically or through an in-built wizard in the DPF Visualization editor. This can be simplified by adding visualization data in the template metamodel, and thereby make it possible for modellers to consisely specify how each model element in the abstract syntax should be visualized. By extending the template metamodel with support for visualization data, we can make the editor treat a specific kind of arrow in a specific way such as holding visualization data. However, if we would add visualization data in the template metamodel, we would have ensure the names of the model elements that is holding visualization data is named in a specific manner. If we instead extend the enriched graph with two additional types; *NodeVisualization* and *ArrowVisualization*, we can ensure that these specific types of arrows are only used to hold visualization data.

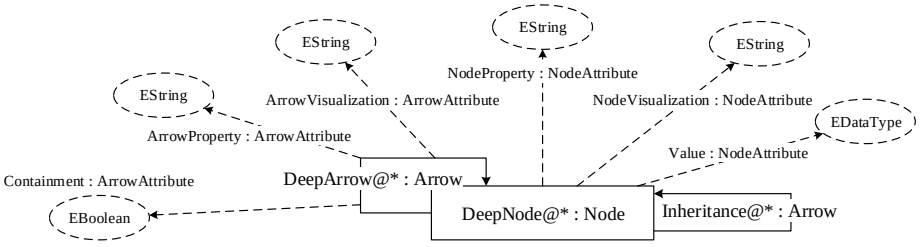


FIGURE 5.18: An illustration of the enriched graph in the DPF Model Editor, with added support for visualizations

With the new types of arrows, we can now add instances of these in the template metamodel which can be used to specify how each element in the abstract syntax should be visualized as illustrated in Figure 5.19 on the next page illustrates a simplified example of a template of a UML Class diagram with attached visualization data. We have stripped off additional Node- and ArrowAttributes for illustration purposes. We have added a NodeAttribute representing the concrete syntax visualization of the Clabject: *clabjectVisualization*. In this case, we want to map the Clabject to the Composite VNode in the visual metamodel, and the value of the *clabjectVisualization* is therefore set to 'Composite'. The idea is that modellers can specify what visual element each Node and Arrow in the template should be mapped to, and is specified by the name of the visual element. To allow Attributes and Methods to be composed inside Clabjects, we have specified the Attribute and Method Arrows as 'Composed'. By defining the Attribute and Method Arrows to be mapped to the 'Composed' VArrow in the visual metamodel, we will tell the Visualization editor that the target nodes should be composed inside the Clabject.

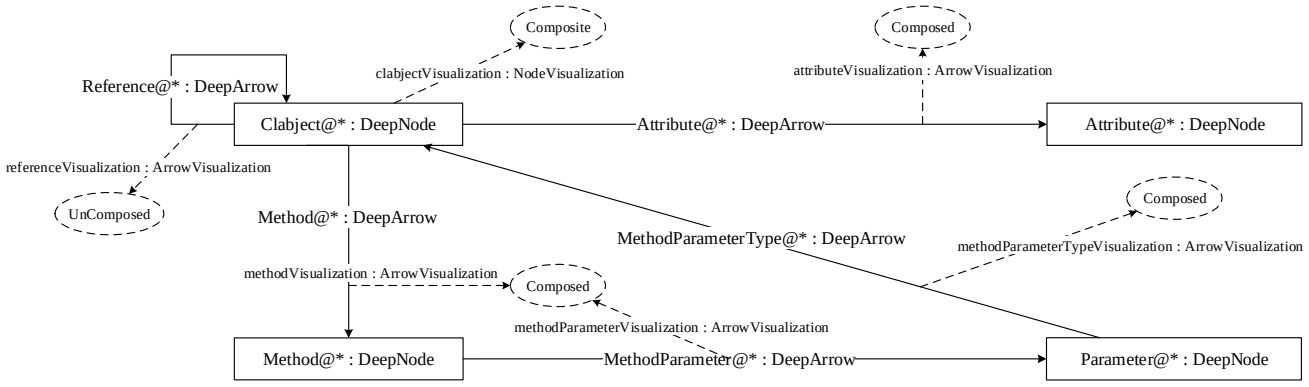


FIGURE 5.19: An illustration of a simplified UML Class diagram template with visualization data.

When the appropriate visualization data is added to the template metamodel, the editor will run algorithm 2 as illustrated below. By first iterating through all the DNodes in the model, the algorithm aims at finding the specified visualization of each node and map it to the corresponding visual element. If the name of the datanode that is containing the visualization data is equal to the name of a VNode in the visual metamodel, the algorithm will map the DNode with the visual element. When the algorithm has iterated over all the DNodes in the DGraph, it will do the same process for the DArrows in the DGraph.

```
Data: Insert a diagram model (dgraph) along with a visual metamodel (visual)
foreach dnode in dgraph do
    foreach outgoing dnodeAttribute from dnode do
        get targetDatanode from dnodeAttribute;
        foreach visualElement in visual do
            if visualElement.getName().equals(targetDatanode.value) then
                map dnode → visualElement;
            end
        end
    end
end
foreach darrow in dgraph do
    foreach outgoing darrowAttribute from darrow do
        get targetDatanode from darrowAttribute;
        foreach visualElement in visual do
            if visualElement.getName().equals(targetDatanode.value) then
                map darrow → visualElement;
            end
        end
    end
end
```

Algorithm 2: The abstract to concrete syntax mapping algorithm used in the DPF Visualization editor.

As we can see from algorithm 2, all we need to do is to add the correct visualization data in the abstract syntax. As long as each appropriate node and arrow is containing the correct visualization data as seen in figure 5.19, each element will be mapped automatically. The clue is however that the visualization data must be specified with the exact same name as

the corresponding visual element we want to map the diagram element to. Unfortunately this currently makes the system prone to error, and the names of the visualization elements should be accessed from a list instead.

At this stage, we have defined both the visual metamodel and an algorithm that will map the abstract syntax to the concrete syntax. The next step is to implement a solution that initiates algorithm 2, and automatically maps the abstract syntax to the concrete syntax. There already exists a model mapping wizard in the visualization editor, but this wizard require a re-mapping at each meta-level. As stated earlier in this section, we only want to define the mapping once, and keep the visualization at sub-sequent metalevels. We are therefore going to implement a new wizard that initiates the model mapping instead, namely the Template Visualization Wizard.

5.5.3 The Template Visualization Wizard

We may recall that the abstract syntax is externally represented in two files, an .xmi file for the core metamodel and a .dpf file for the diagram metamodel. The concrete syntax is represented in a third .visualization file, holding the mapping and a reference to the visual metamodel as well as the diagram metamodel. To initiate algorithm 2, we have implemented a wizard that can take a diagram metamodel and a visual metamodel as input, and create new metamodels as a result. The wizard will create a .xmi file for the core metamodel, a .dpf file for the diagram metamodel and a .visualization file for the visual metamodel. The template visualization wizard is fairly simple, meaning it only needs to execute algorithm 2, and save the result in three files (.xmi, .dpf and .visualization). We are however not going into any further detail about the wizard here, but rather demonstrate it in chapter 6 instead.

When the concrete syntax is created and the model is opened, we realize that every single model element from the abstract syntax is added to the palette (Nodes, NodeAttributes, Arrows, ArrowAttributes and DataNodes). This is obviously not suitable for our implementation, as it makes it difficult to find the model element we want to use. The next section will explain what we can do to filter out the necessary model elements such that only the model elements we would model with, is added to the palette. The rest of the model elements are automatically generated through the replication rules as stated earlier in this chapter.

5.5.4 Filtering Model Elements In The Palette

As all the model elements are added to the palette, it naturally results in a palette that is very difficult to use for modellers. For starters, Node- and ArrowAttributes are re-instantiated along with the Node or Arrow it is contained in. It will therefore not be necessary to have neither NodeAttributes, ArrowAttributes nor DataNodes in the palette.

Secondly, instantiated Attributes and Methods contained in Clabjects are re-instantiated by replication rules, so these elements should not be in the palette either. In practice, all elements that is re-instantiated through a replication rule should not be in the palette (except from the template elements). As an example, lets consider the addition of a *name* Attribute in a Clabject. The *name* Attribute will be re-instantiated along with the instantiation of the Clabject. Allowing to add new instances of *name* in the Clabject would result in duplicate elements of the same Attribute in a single Clabject, and is clearly something we want to avoid.

Figure 5.20 below, illustrates the plant example with the current layout in the visualization editor. To the right, we can see the palette containing the template elements from the template we created in figure 5.16, only with added visualization data. As we can see, the template metamodel is containing far more elements than what is added to the palette. The palette is only containing the elements we need to model with.

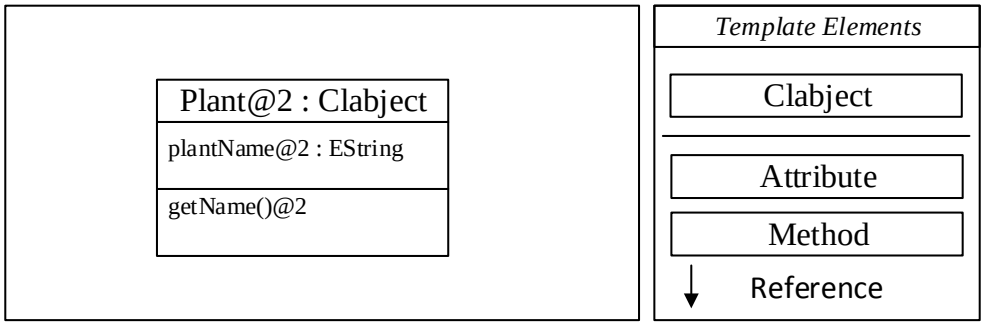


FIGURE 5.20: A simple illustration of metalevel 0 of the plant example, illustrated in the visualization editor.

To seperate between ontological instances and linguistic instances, we have created a seperate palette-drawer for the template elements, named *Template Elements*. When instances of this metamodel is created, the elements will be added to a new palette-drawer named *DSL Elements*.

The current implementation of the visualization editor does however not support nesting of nodes at more than one level. As an example, lets consider Methods. Methods are nodes that is contained in Clabjects, but Methods is also containing Parameter nodes. It is currently possible to nest Methods in Clabjects, but there are no structure that allows nesting

of Parameters in Methods as well. This means we have to create Parameters manually, and we have created a Method dialog especially for this purpose. Only Method nodes will be in the palette, but when adding new Method nodes in Clabjects, a method dialog will pop up with the possibility to define properties of methods such as name, return parameter and potency. However, the method dialog is currently very simple, and only supports modelling of methods with a return type. The method dialog will be presented under the demonstration of the tool in chapter 6.

Figure 5.21 below, illustrates an instance of the model we presented in figure 5.20. As we can see, the only element we need to model with is the template elements for linguistic extension, and the *Plant* element. The *Plant* element is an ontological element, and is added to a new palette-drawer named *DSL Elements*, as described above. The palette-drawer for *Template Elements* is closed for illustration purposes. By clicking on the *Template Elements* drawer, it will slide down and provide modellers with template elements for linguistic extensions.

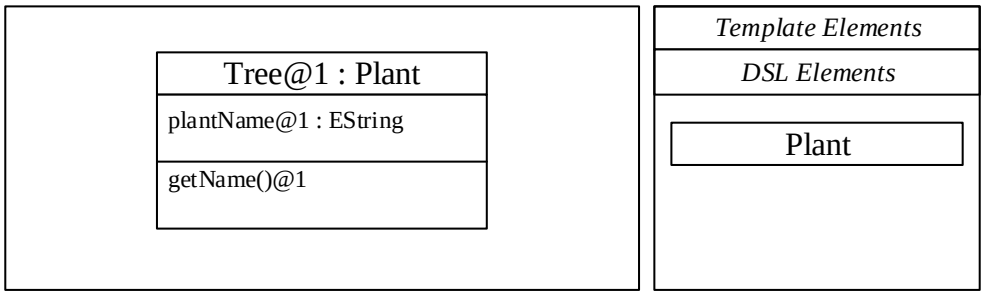


FIGURE 5.21: A simple illustration of metalevel 1 of the plant example, illustrated in the visualization editor.

Modellers can linguistically extend ontological models by adding template elements from the palette, and connect them with the ontological instances at any metalevel. As we see from figure 5.20 above, it is easy for modellers to reach the tools they need to model with. Model elements that is replicated and simply not necessary to add to the model directly, is not added to the palette. These elements such as NodeAttributes and ArrowAttributes can be modified through a *Set Property* dialog instead. This dialog will work on any node or arrow, and is dynamically filling a table with the outgoing NodeAttributes or ArrowAttributes. Modellers can then modify any element in the table, such as the datatype or value of an Attribute. This property dialog will be presented under the demonstration of the tool in chapter 6.

5.6 Current Shortcomings

In short, the editor implemented in this thesis should be regarded as a prototype, as it has not been tested properly. Secondly, it is necessary with proper naming constraints for potency to ensure for example that an attribute is instantiated with a datatype, and a slot is instantiated with a value. We need proper naming constraints to ensure that no ontological model element has the same name as any linguistic element. The visualization editor is not updating its visualizations properly after replicating elements, and currently the visualizations offered in the visualization editor is quite limited. We only briefly mention the most apparent limitations here, while a more thorough listing of the shortcomings is presented section 7.2.1.

5.7 Summary

By following the research methodology as we defined it in section 3.5, we have now managed to implement a fully diagrammatic editor for deep metamodelling in both an abstract and concrete syntax. The editor supports deep instantiation through potency, linguistic and ontological instantiation and linguistic extension. We have also added support for mutability to define restrictions on how many times the value of a node or arrow can change. With the implementation of an e-graph, we can also define datatypes directly from the predefined datatypes in EMF. Lastly, we have created an enriched graph with additional functionality such as containment and added visualization data. With this enriched graph it is also easier to create new template models, and it is easier to map different modelling-hierarchies in DPF to other systems of concern.

CHAPTER 6

Demonstration

In this chapter, we present the abstract and concrete syntax in DPF as we implemented it in this thesis. Through this chapter, we aim at demonstrating the current features as we implemented them in chapter 5. We begin this chapter with a demonstration of the development of templates in the Model Editor, building up larger models throughout this chapter. Finally we will demonstrate an example of the development of a visual metamodel in the Visualization editor to visualize the models in a concrete syntax.

6.1 The Abstract Syntax

As a starting point, we begin the demonstration by presenting the process of creating new templates. We have implemented a new wizard for templates as illustrated in figure 6.1 below.

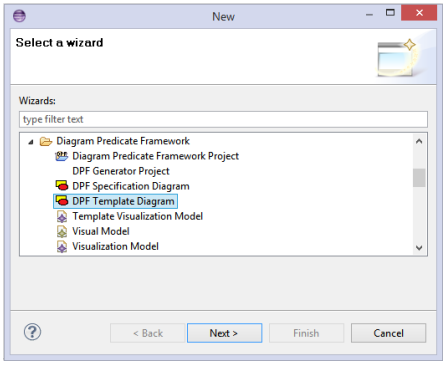


FIGURE 6.1: An overview of the wizards provided in DPF.

Figure 6.1 illustrates the wizards that is currently available in DPF, such as the Template-, Visual- and Template Visualization wizards amongst others. The template wizard is simple, basically the only thing it is doing is to ensure that the new metamodel is based on the enriched graph we defined in section 5.4, and that the new metamodel resides on the default metalevel (metalevel -1). To make it easier to conceptualize the template metamodel, we present the enriched graph as illustrated in figure 6.2.

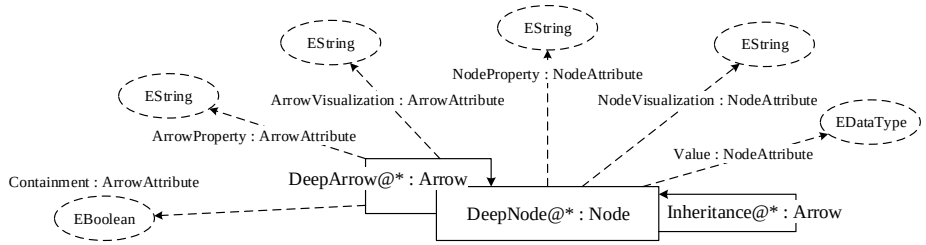


FIGURE 6.2: An illustration of the enriched graph in the DPF Editor

As explained in section 5.4.1, the enriched graph contains extensions of the core metamodel with functionalities such as containment, support for visualization data, value and inheritance. When running the template wizard, we create an instance of the enriched graph such that we can develop new templates. Figure 6.3 below, illustrates the current modelling environment in the Model Editor, where we have excluded the propertysheet for illustration purposes.

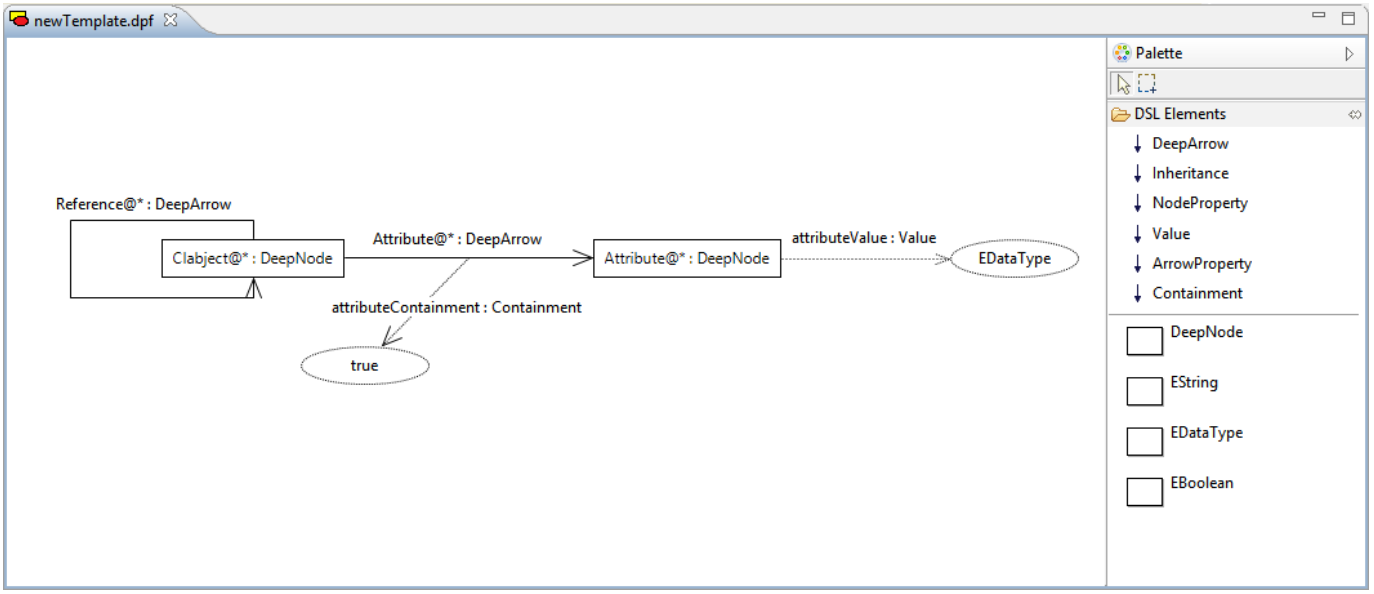


FIGURE 6.3: An illustration of a simple UML Class diagram template with Attributes specified as containment-nodes.

To the right in the figure we see the palette containing elements from the metamodel adjacent above, which in this case is the enriched graph as seen in figure 6.2. In the center of the figure we see the model window, which is where we create the instances of the elements in the palette. The potency and mutability of model elements can be set by selecting the model element of concern, and the properties of that model element can be set in the propertysheet below the model window.

The model we have created in figure 6.3 above, is a simplified class diagram of a Clabject with support for Attributes and References. By specifying

the Attribute as *containment*, we were able to specify that Attributes is contained inside the Clabject much like Features is contained in Clabjects in Melanee. This allows nesting of Attributes inside Clabjects and ensure that the Attribute is replicated according to the replication rules as defined in chapter 5. We have also defined that Attributes will have a mutability of 1, which means that name of the Attribute can only be set once. When adding Attributes to Clabjects, the mutability of the Attribute will be decreased to 0, and it is thereby not possible to change the name of the Attribute any more. This ensures a consistent name of the Attribute throughout the meta-hierarchy. Because we defined the Attribute arrow as *Containment*, the Attribute will be automatically generated at sub-sequent metalevels, and the value of the Attribute can be set whenever the modeller wants to.

Figure 6.4 below illustrates an instance of the template metamodel in the abstract syntax. To model the uppermost metamodel in the Plant example, we begin by dragging a new Clabject from the palette to the right, and into the modelling window in the center as seen in figure 6.4.

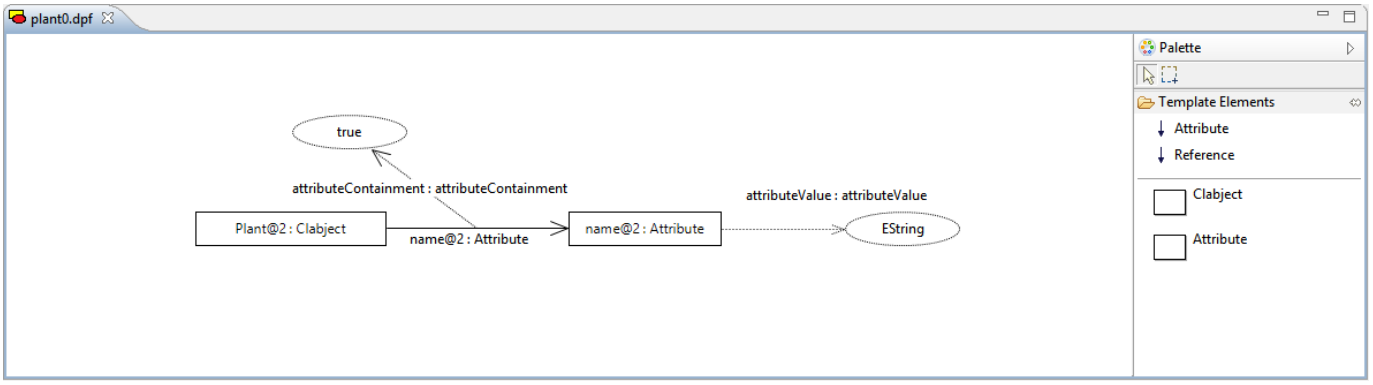


FIGURE 6.4: An illustration of metalevel 0 in the Plant example.

As we have seen in the plant example earlier in this thesis, we will begin modelling by defining a *Plant* Clabject with a potency of 2. We also add a *name* Attribute to the *Plant* with a potency of 2. As the *name* Attribute is being instantiated, its mutability decreases to 0, which means we can not change its name at any sub-sequent metalevel.

By looking at figure 6.4, we may realize that when creating new Clabjects, none of its containing Attribute or Method Nodes are replicated along with it. This is because Clabjects are template elements. When creating instances of template elements, it would not make any sense to replicate the containing elements such as Attribute and Method Nodes. Instead we only replicate the whole template metamodel to allow linguistic extension, which is hidden from modellers for improved useability.

Figure 6.5 on the next page, illustrates an instance of the metamodel we created in figure 6.4 above. The potency of the Plant instance named *Tree* has been decreased to 1, as well as its containing attribute has been

automatically replicated with its potency decreased to 1 as well.

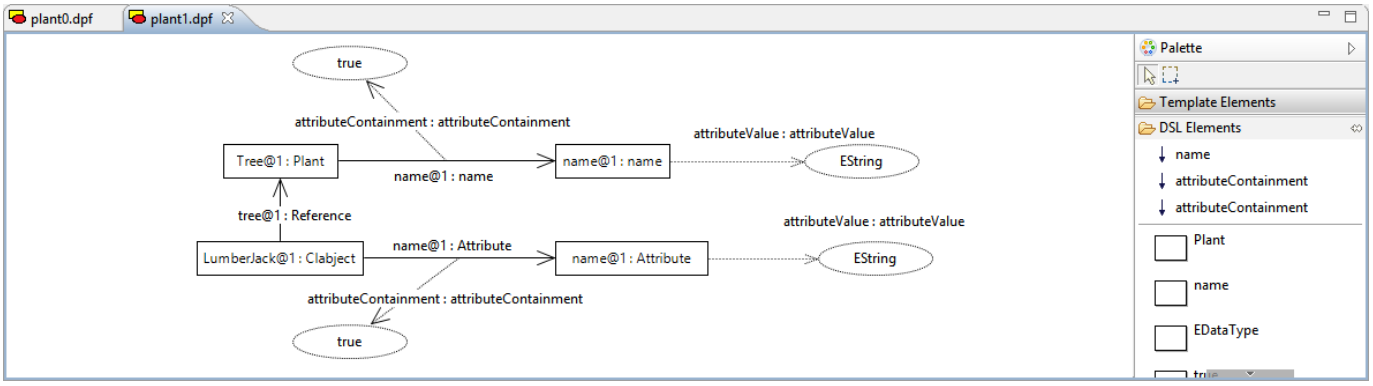


FIGURE 6.5: An illustration of metalevel 1 in the Plant example.

As we can see in figure 6.5 above, the name of the attribute stays the same, and can not change because its mutability is 0. At this metalevel, we have also added a linguistic extension of named *LumberJack*, which is an instance of the replicated Clabject in the metalevel adjacent above (metalevel 0). It is also possible to add connections between linguistic and ontological elements as we described it in section 5.1.2, which is realized in form of a Reference from *LumberJack* to *Tree*. By instantiating this metamodel we are finally at instance level, and the following metamodel takes place:

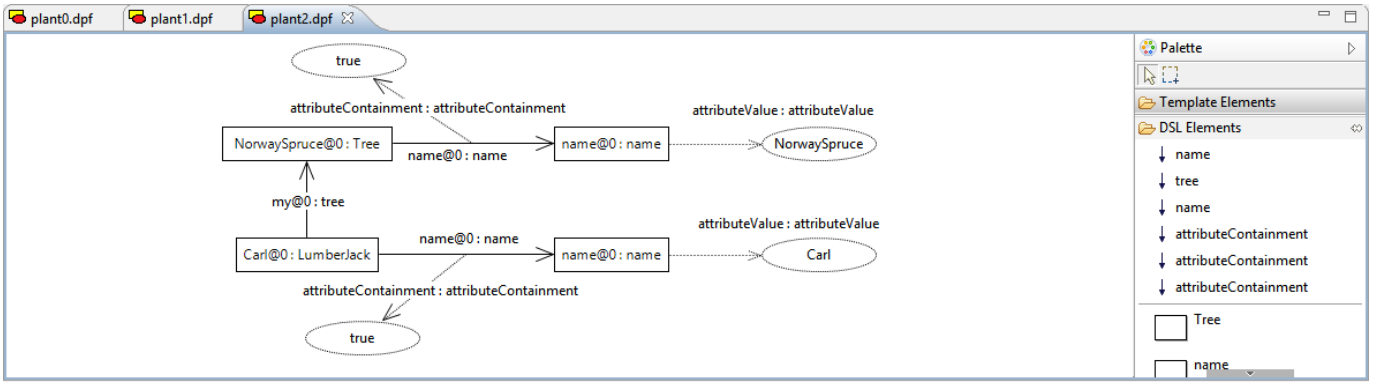


FIGURE 6.6: An illustration of the metalevel 2 in the Plant example.

The potency of each model element is now 0, which means we are at instance level and values must be set on the attributes. However, by looking at the metamodels in the Plant example as illustrated above, it is becoming evident that even with such small models as in the Plant example, it is becoming increasingly difficult to model with the abstract syntax. The next section will therefore demonstrate how we can visualize the Plant hierarchy in a concrete syntax instead.

6.2 The Concrete Syntax

To model the Plant hierarchy in a concrete syntax, we first need a visual metamodel to define how the model elements will be visualized in the concrete syntax. The idea is that we create a mapping from each element in the diagram metamodel (abstract syntax) to the corresponding element in a visual metamodel (concrete syntax). The intent of the visual metamodel si that it will describe how each model element in the diagram metamodel should be visualized in the corresponding concrete syntax. Figure 6.7 below, illustrates an overview of the process of the mapping between the diagram metamodel and the visual metamodel, the mapping between th abstract and the concrete syntax.

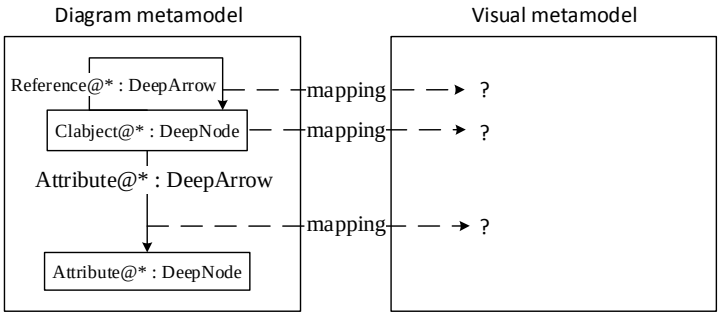


FIGURE 6.7: Abstract to Concrete syntax mapping.

To visualize the abstract syntax in a concrete syntax, we therefore need to create a visual metamodel that describes the visualization of the abstract syntax. There already exists an editor for modelling visual elements which we can use to visualize the template in a concrete syntax. For example to visualize the Clabjects, we need a structure that allows compositions of nodes. To do this, we can create a composite Node as illustrated in figure 6.8 below.

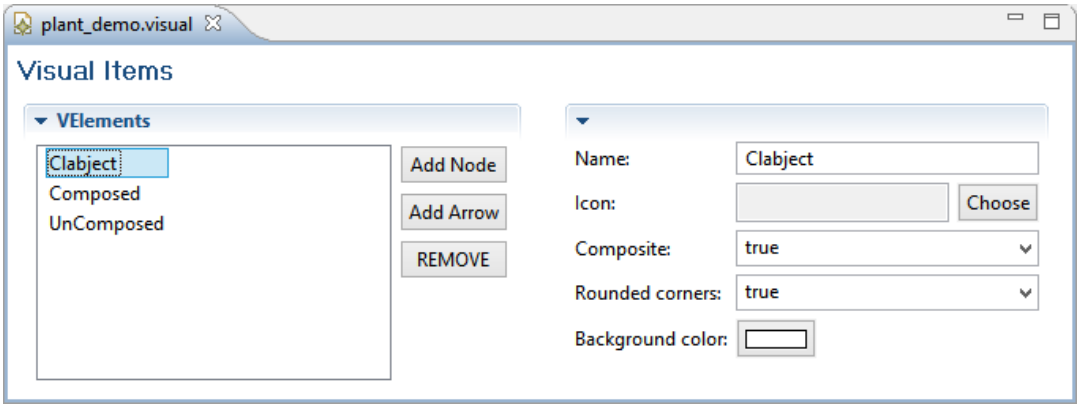


FIGURE 6.8: The visual metamodel used in this demonstration.

As we see in figure 6.8, we can add new Nodes and Arrows to the left in the visual editor, and edit the properties of each visual element to the right. We decided to create a new Node named *Clabject*, which is composite, has rounded corners and a white background-color for improved visualization. To visualize the nodes that will be composed inside the *Clabject*, we need to specify *composed* Arrows. The Visualization editor will then create compartment-elements for the elements that will be contained in the Clabject such as Attributes. In fact, all outgoing *composed* arrows from any *composite* Node is contained in the *composite* Node. Lastly and to visualize references between Clabjects, we created a *UnComposite* Arrow. *UnComposite* arrows will be visualized with the same visualization as in the abstract syntax (as an arrow-connection). The intended mapping we want in this example is illustrated in figure 6.9 below.

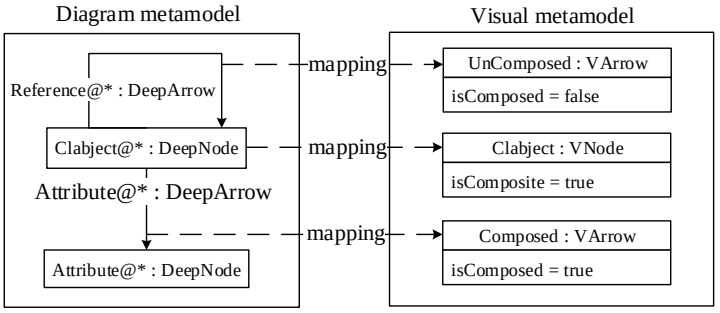


FIGURE 6.9: Intended abstract to concrete syntax mapping.

There already exists a wizard which can take care of the mapping between the abstract and concrete syntax, but we argue that it is easier to conceptualise how each abstract syntax element should be visualized by directly looking at the abstract syntax instead. By adding visualization data in the template metamodel, we can define how each model element in the abstract syntax should be visualized in the concrete syntax. Figure 6.10 below, illustrates the new template model with added visualization data for the model elements.

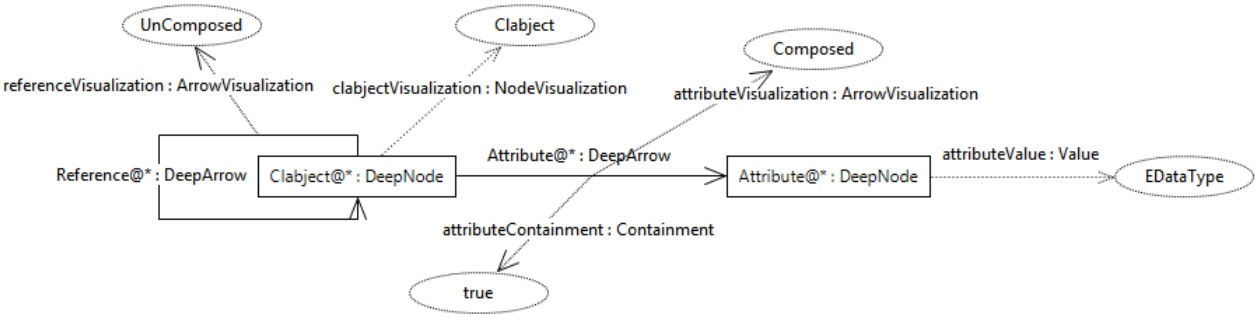


FIGURE 6.10: The new template metamodel with added visualization data.

The mapping of the model elements is done by name, so by specifying Clabjects as *Clabject*, the Attribute Arrow as *Composed* and the Reference Arrow as *UnComposed*, the editor will automatically map these elements to the corresponding *Clabject*, *UnComposite* and *Composed* elements in the visual metamodel. To initiate the model mapping, we will open the template visualization wizard as illustrated in figure 6.11 below.

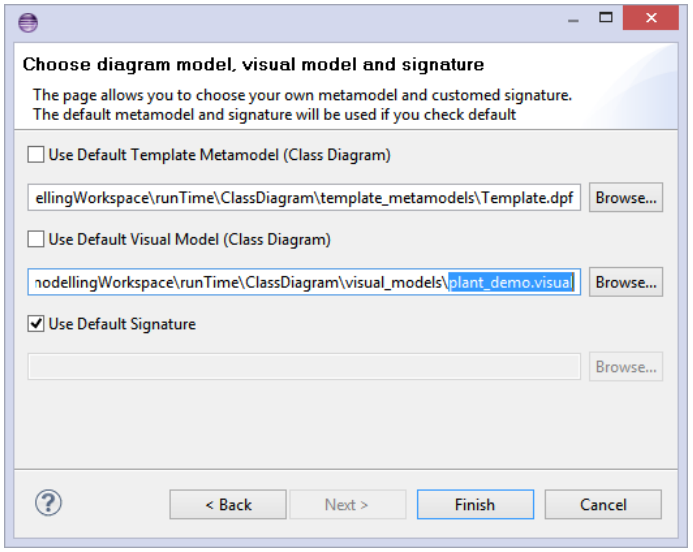


FIGURE 6.11: Choosing template and visualization model in the template visualization wizard.

By default, the wizard will load the default class diagram template as we presented it in chapter 5. The wizard will also load the default visual metamodel by default. Thirdly, it is possible to specify additional *signatures* to allow additional constraints on the model, but we will not discuss this any further as the specification of signatures is outside the scope of this thesis. By choosing the template metamodel and the visual metamodel as illustrated in figure 6.11, we can click *Finish* and the wizard will initiate algorithm 2. As explained in section 5.5, algorithm 2 will parse over the template metamodel for visualization data, and automatically map each element to the corresponding visual element such as in figure 6.9 above. When the mapping is finished, we can start modelling the Plant example as illustrated in figure 6.12 on the next page.

To make it easier to find the model elements we need in the palette, the editor is also filtering out model elements that should not be in the palette. As we can see, the concrete syntax are more visually appealing and easier to conceptualize than its abstract syntax. A Clabject is simply visualized as a rectangle with rounded corners, along with containing compartments for containing elements such as Attributes. The potency of each model element is visualized as a number which is raised and visualized next to the type of the model element. Mutability is currently not visualized, but can be accessed through the propertysheet along with the name and potency

of the Attribute.

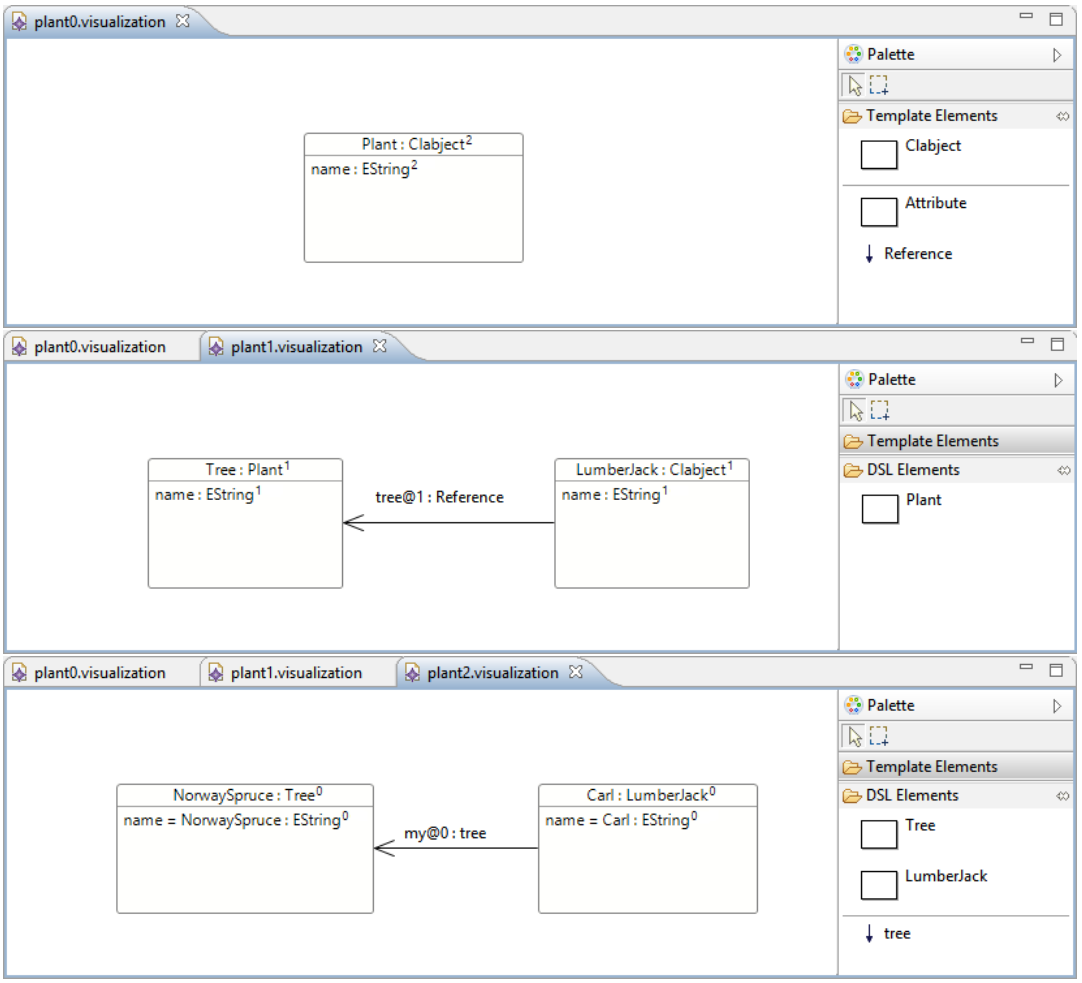


FIGURE 6.12: An illustration of the plant hierarchy in its concrete syntax.

Figure 6.12 above, illustrates the modelling environment of the visualization editor. By illustrating the plant example we modelled earlier in this chapter, we easily see that this concrete syntax is much easier to model with than the abstract syntax. In fact, the visualization editor is still modelling with the diagram elements from the abstract syntax by adding them to the palette. When a diagram element is dragged from the palette and into the modelling window, the editor is checking what visual element the diagram element is mapped to, and creates the corresponding concrete syntax. The visualization editor is thereby keeping track of the diagram metamodel, and creates a corresponding concrete syntax. By creating a model element in the concrete syntax, it is also created in the abstract syntax as well. We can open the Model editor at any time and edit the abstract syntax. When we are opening the visualization editor again, the corresponding concrete syntax will be created based on the mapping of the diagram elements. This way, we have a two-way synchronization between the abstract and the concrete syntax.

To be able to set the values of NodeAttributes, we have created an additional dialog as illustrated in figure 6.13 below. The dialog can be accessed by clicking on the name of the Node.

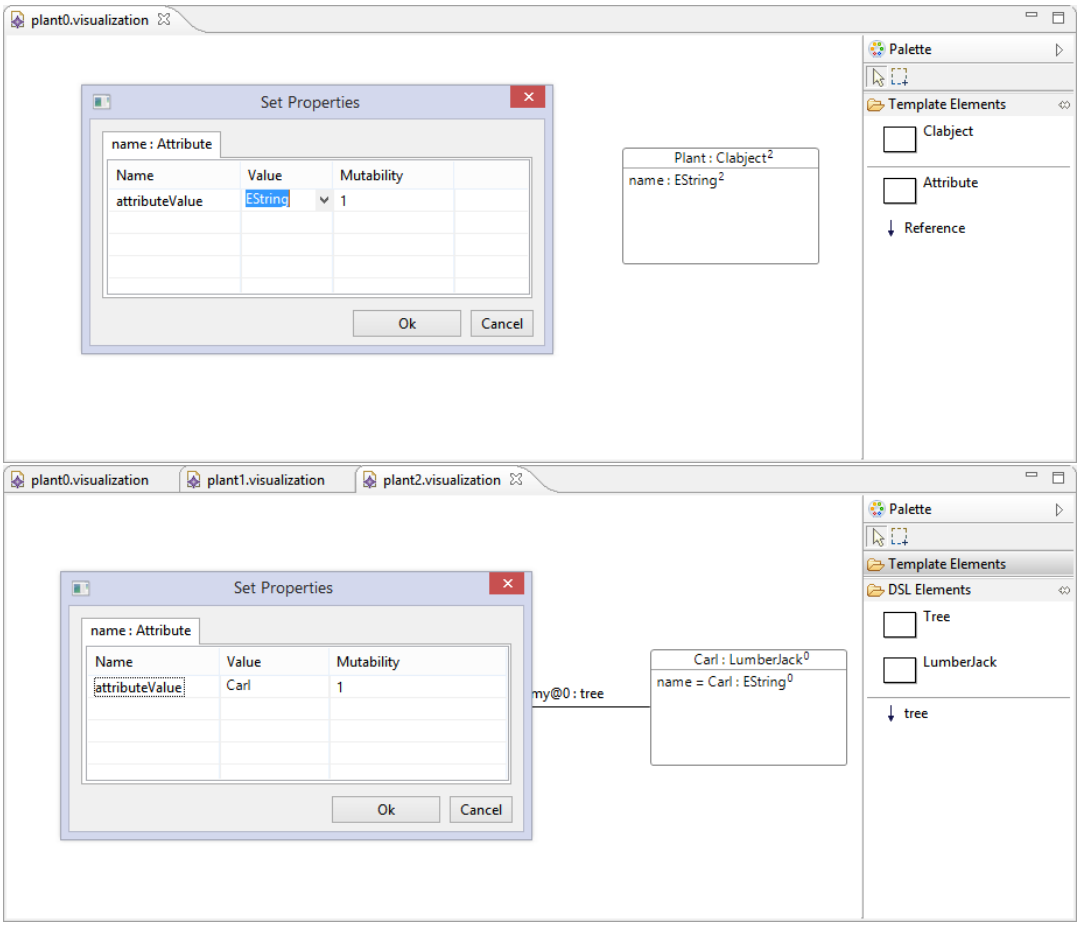


FIGURE 6.13: The dynamic dialog used for setting values of NodeAttributes.

Since NodeAttributes is a kind of properties of a Node, we decided to call the dialog *Set Properties*. The dialog is organized with a tab for the current Node we want to edit NodeAttributes for, such as the *name* Attribute. In the current template we only have one NodeAttribute attached to the *name* Attribute, but as more NodeAttributes are attached, it will dynamically be added to the table in the dialog. With this dialog, it is possible to set the value and mutability of any NodeAttributes of any Node, including Clabjects. The names and mutability can be set simply by clicking on the tableitem, edit the value and click the *Ok* button. When it comes to Values however, we recall that values of DataNodes can be set either by the predefined set of datatypes in EMF or a user defined value. We have therefore defined that the column for values will consist of combo-boxes instead of only text. With combo-boxes, we can either specify datatypes from the drop-down list, or set our own value. We have also developed an equivalent for ArrowAttributes, but we will not demonstrate it here since it is basically the same dialog.

We may recall that we specified default templates in chapter 5. The next section will demonstrate the default template in practice.

6.3 The Default Class Diagram

In chapter 5 under the development and implementation of the Model Editor, we developed a default template metamodel of a UML Class diagram as illustrated in figure 6.14 below. The template is based on the enriched graph as we illustrated it in figure 6.3, in the beginning of this chapter.

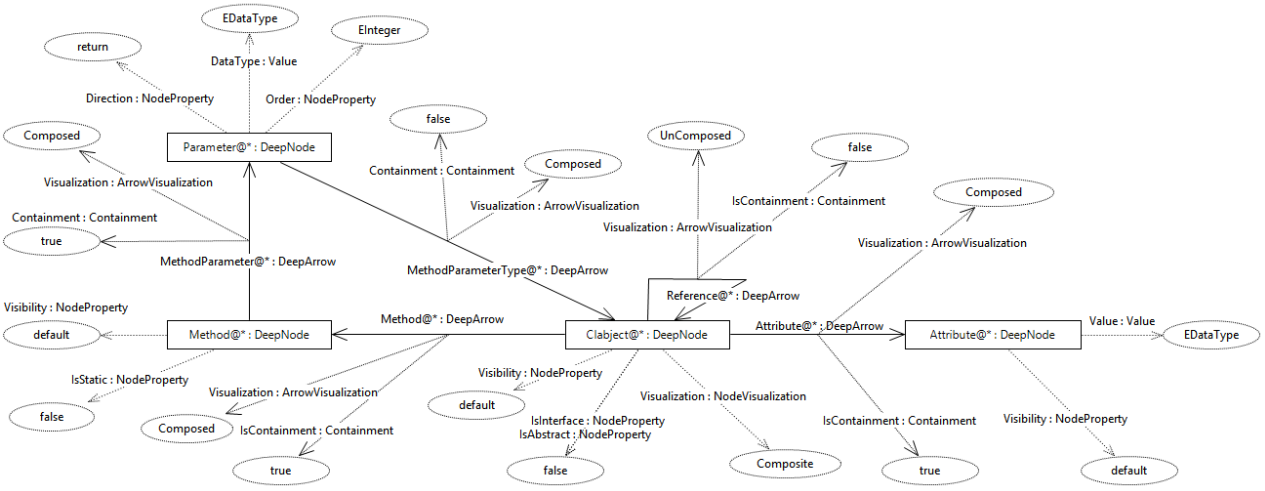


FIGURE 6.14: The default template of a UML Class diagram in DPF.

In addition to the specifications in the template we created in the beginning of this chapter, the default template also has support for Methods as well additional properties of Clabjects, Attributes and Methods. Additionally, for the visualization of the template metamodel, we have also created a default visual model as illustrated in figure 6.15 below.

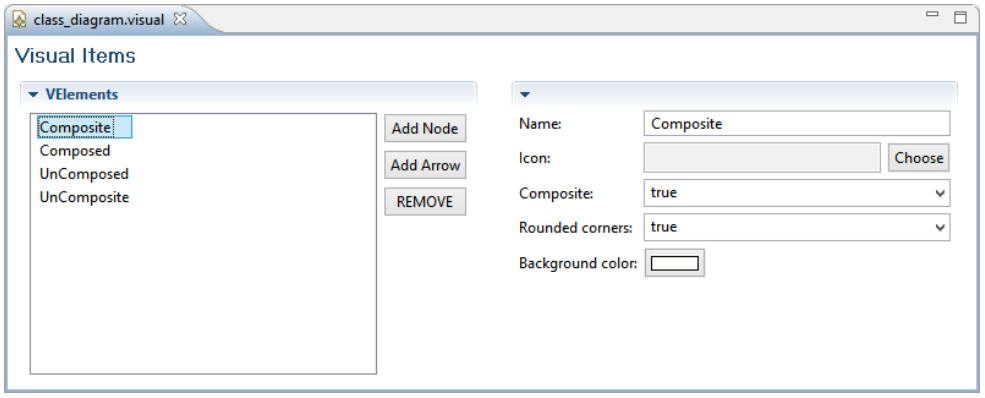


FIGURE 6.15: The default visual metamodel in DPF.

The default visual metamodel consists of an *UnComposite* and a *Composite* Node. *Composite* represent Nodes that can contain other Nodes such as

Clabjects, and *UnComposite* represents Nodes that can not contain other Nodes. Secondly, the visual metamodel consists of a *Composed* and an *UnComposed* Arrow. *Composed* will represent the Arrows that is contained in *Composite* Nodes. These arrows will not be visualized in the editor, as they only represent the connection between for example a Clabject and an Attribute. The *UnComposed* Arrow on the other hand, will represent the arrows between Clabjects such as References.

With the default template and the default visual metamodel, we can run the template visualization wizard with the default settings, create a new visualization metamodel and start modelling with the default template in its concrete syntax. Figure 6.16 below illustrates the current visualization of Clabjects, where we have added visualization for visibility on attributes and methods in standard UML notation [24].

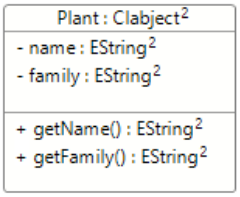


FIGURE 6.16: Visualization of *Plant* based on the default template.

Figure 6.16 above illustrates the Plant Clabject with added *name* and *family* Attributes, and *getName()* and *getFamily()* Methods. Looking back at the default template metamodel in figure 6.14, we realize that the specification of Methods consists of a Method Node along Parameter Nodes and its attached NodeAttributes and ArrowAttributes. To specify Methods, we therefore need to add a new Method Node to *Plant*, as well as adding the corresponding Parameter Nodes of that Method. The current implementation of the visualization editor however, does not support nesting more than a Node inside another Node. It is therefore necessary with an additional dialog where we can specify the Parameters of the Method as well, such as the Method dialog illustrated in figure 6.17 below.

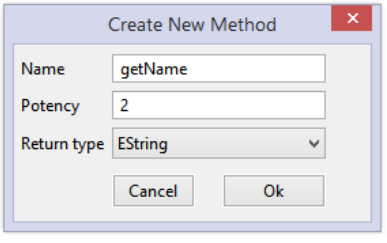


FIGURE 6.17: Create new method dialog.

As we can see, the current implementation of the Method dialog is very limited. It is currently only possible to specify name, potency and return type of Methods. This means that the only Methods supported in the visualization editor is Methods with no input parameters, only a return parameter. We may also notice that the editor only supports return values

of predefined datatypes from EMF. For the visualization editor to be fully supporting UML Class diagrams, we therefore need to extend the Method dialog with support for input parameters as well as support for parameters with Clabjects as parameter-type. Methods are however only specific for this kind of template, (UML Class diagram), and would not be used for other templates. To add such a dialog for Methods, we need template specific code. Possibly this can be predefined in the editor together with a set of predefined templates. Otherwise we need to explore the possibility of extending the editor with plugins for new types of dialogs and functionalities for templates. Whether we will predefine template code for dialogs, or explore the possibility of additional plugins for dialogs will be discussed under further work in chapter 7.

When it comes to editing Methods after they are added on the other hand, we can use the properties dialog as illustrated in figure 6.18 below.

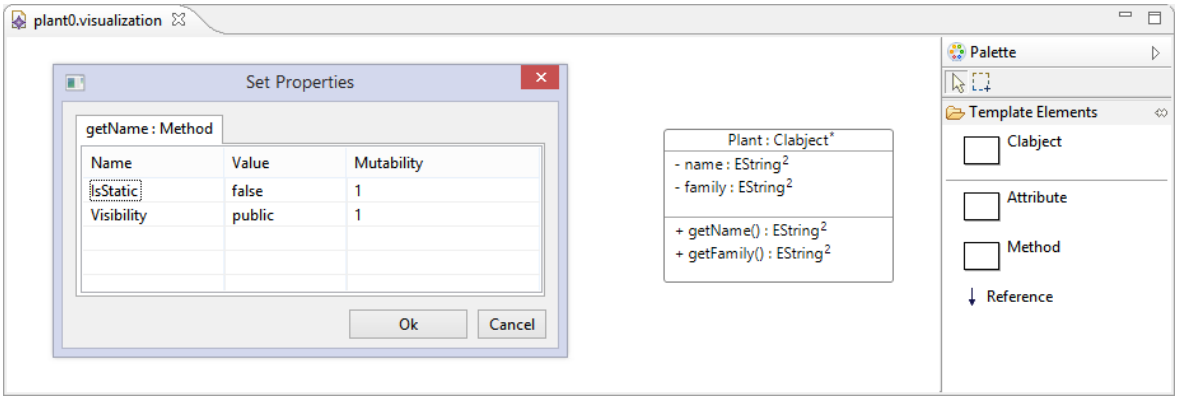


FIGURE 6.18: The dynamic dialog in the process of editing the NodeAttributes of a Method Node.

However, the dialog as illustrated in figure 6.18 above currently does not support editing properties at more than one composition-level. This means it is currently only possible to edit the Method Node, while it is not possible to edit any of the Parameter Nodes. It is not possible to edit any Node that is contained in a Node which is contained in a Node. We can however add support for more than one composition-level in the future by adding extra tabs in the dialog for subcontainers such as Parameter Nodes, but this is regarded as outside the scope of this thesis.

At this point we have demonstrated the Visualization Editor with focus on UML Class diagrams only, but the Visualization Editor can be used for other types of visualizations as well. Currently the visualization editor only supports a small range of visualization parameters, but the current editor does support modelling an arbitrary number of compartments. It is also possible to change the color of the model elements, as well as setting whether the edges are rounded or not. The next section will present the current possibilities of the visualization Editor.

6.4 Customizable Concrete Syntax

To illustrate the current possibilities of the visualization editor, we have created an example of a *House* with four compartments; *Bathroom*, *DogHouse*, *Livingroom* and *ToolShed* as illustrated in figure 6.19 below.

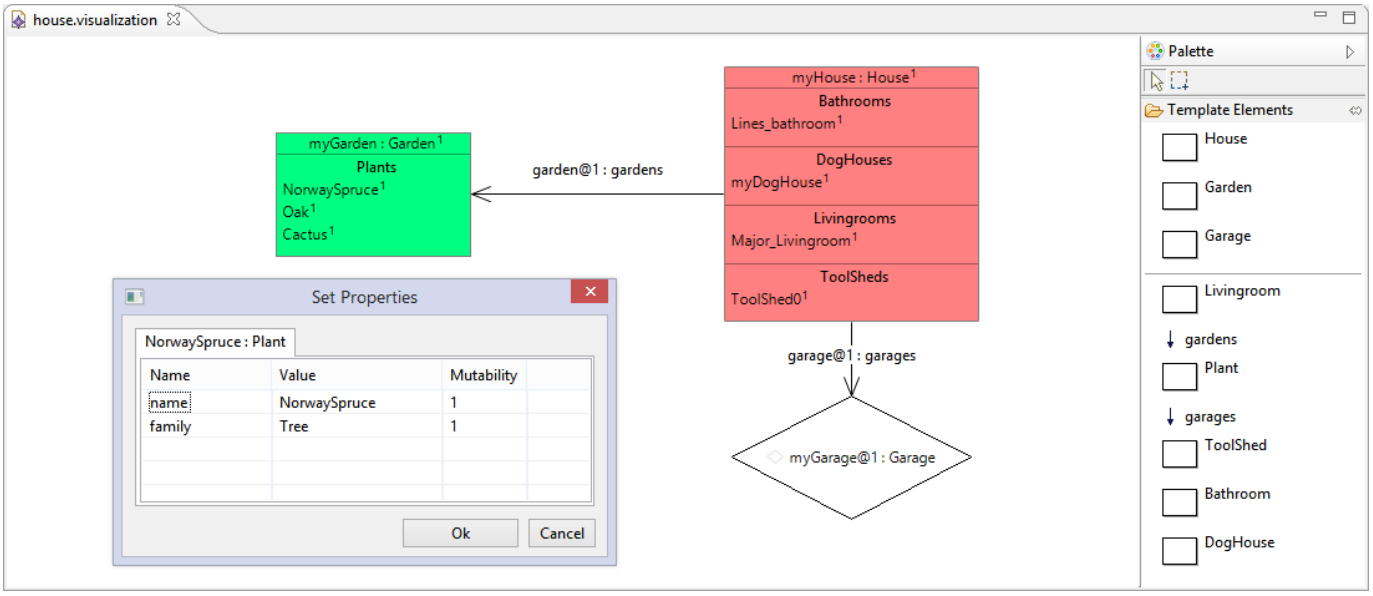


FIGURE 6.19: An example of a house with a garden and a garage.

The *House* also has a *Garden* with *Plants*, as well as a *Garage*. To illustrate the possibility of changing color of Nodes, we have set the color of the *House* to red and the *Garden* to green. To illustrate the possibility of changing shapes of *UnComposite* Nodes, we have defined the *Garage* as a rhombus as illustrated in figure 6.19 above.

First of all, as UML Class diagrams already has a predefined visual syntax with Attributes in the top-most compartment and Methods in the bottom-most compartment in Clabjects, it may not be as obvious for other types of diagrams. We have therefore added a label on each compartment, illustrating what type of elements the compartment is containing. We also may notice that unless programmatically specified such as for UML Class diagrams, the compartment-elements are only visualizing the name of the Node itself. As stated earlier in this thesis, it is currently only possible to compose nodes at one composition-level. This makes it difficult to visualize for example Methods appropriately, as the elements in a compartment may contain nodes themselves such as Parameters in Methods. Unless programmatically specified, it is neither possible to edit the properties of a Node at more than one composition-level, nor possible to add Nodes at more than one composition-level, nor possible visualize compartment-elements at more than one composition-level. As we add more compartments and begin modelling other types of diagrams than class diagrams, it is becoming evident that it is necessary to add support for appropriate visualizations of compositions at more than one composition-level.

Secondly, by looking at the *Garden* and its *Plants*, we notice that each *Plant* have a *name* and *family* property. These properties are dynamically added to the properties dialog, and each property of the *Plant* can be defined in this dialog. Possibly this dialog can also be used to define new properties of Nodes, which will allow a user friendly interface for specifying additional properties of Nodes and Arrows. We also remember that earlier in this thesis we argued that by defining visualization data as properties of Nodes and Arrows, we could make it possible to define the visualization of each model element on-the-fly. Modellers could possibly start modelling the concrete syntax by first modelling in a default concrete syntax, then define the visualization of each model element through the property dialog as illustrated in figure 6.20 below.

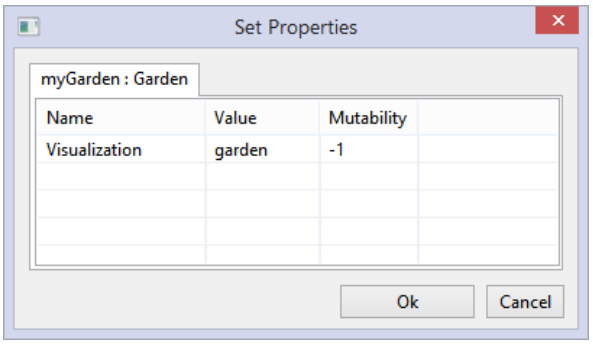


FIGURE 6.20: Defining concrete syntax on-the-fly.

Figure 6.20 illustrates the definition of visualization data for *Garden*. In the future, the visual elements in the visual metamodel we create could be added as a list in the property dialog. Modellers could then possibly define the concrete visualization by specifying the visualization value to the visualization of concern. Through the mapping algorithm, we could automatically remap the visualization as the visualization property is changed, and this way modellers could easily see how the visual elements are visualized in practice. These mechanisms are however outside the scope of this thesis, and will be discussed in further work instead.

CHAPTER 7

Conclusion

The goal with the research presented in this thesis was to explore a diagrammatic approach to deep metamodelling. In this chapter we will summarize the outcomes and results obtained from this research compared to previous efforts. We will also present some suggestions for further work with basis on the current implementation.

7.1 Summary

In the introduction, we briefly mentioned that previous efforts to deep metamodelling have never considered a fully diagrammatic approach. There exists tools such as Melanee that supports a diagrammatic environment for deep metamodelling, but it is only considering modelling with the whole meta-hierarchy in a single view. At Bergen University College, there had been developed a fully diagrammatic editor for traditional metamodelling that supported an arbitrary number of metalevels [28]. By extending this editor with basis on the work of Atkinson and Kühne [9], we have now overcome the three major limitations with traditional metamodelling. We introduced the concept of potency to allow deep instantiation. Secondly, potency made it possible to provide a simple and well-formed solution to distinguish between dual classifications such as the dual Class / Object facets of Clabjects. Thirdly, potency made it possible to keep the linguistic and ontological metamodels in a single dimensional hierarchy as opposed to a separation into a two-dimensional meta-hierarchy.

Interestingly, by arranging linguistic and ontological metamodels in a single dimension we also made it possible to develop our own templates of linguistic metamodels. Instead of having a two dimensional meta-hierarchy where the linguistic metamodel is hardcoded into the editor, we are now able to develop almost any type of linguistic metamodel and create ontological metamodels based on the linguistic metamodel. By replicating the linguistic metamodel at each ontological metalevel, we also made it possible to linguistically extend the ontological metamodels. This way we made it possible to adhere to strict metamodelling, and at the same time keep a single-dimensional modelling hierarchy.

Through this thesis, we have achieved a fully diagrammatic editor for deep metamodelling in both:

Abstract Syntax
We have introduced a fully diagrammatic editor for the abstract syntax, which supports the development of linguistic templates. With these linguistic templates, we are able to develop ontological meta-hierarchies with support for deep instantiation and mutability as well as linguistic extension. We have also developed a default template of a UML Class Diagram, which can be used to model deep ontological metamodels of class diagrams.

Concrete Syntax
Through the extension of the DPF Editor with support for deep metamodelling, we are now able to develop almost any kind of DSML that suits our needs. However, as these models grow bigger, modellers will find it harder to gain an overview of the DSMLs. The DPF Visualization editor has therefore been extended with support for the specification of a concrete syntax for the template metamodels throughout the meta-hierarchy. To make it easier to map the abstract syntax to the concrete syntax, we have developed an algorithm that parses the abstract syntax for visualization data. By adding visualization data in the templates, we can thereby automatically map the abstract syntax to the concrete syntax.

We have also extended the core metamodel in DPF from a directed multigraph to an E-Graph.

E-Graph
Through the implementation of the E-Graph [48], we made it possible to specify datatypes directly from the predefined set of datatypes in EMF. We also made it possible to specify properties of nodes and arrows such as visibility, isStatic, isFinal in the default class diagram template. With the core metamodel extended to an E-Graph, we added a new graph between the core metamodel and the template metamodel. We added an enriched graph with support for nesting of nodes as well as providing a generic solution for the definition of values and visualization data. With this enriched graph, it is easier to develop new templates, and its easier to map the modelling-hierarchy to other systems.

7.2 Further Work

The DPF Editor still has a way to go in terms of offering a fully functional and stable editor for deep metamodelling. This section will therefore briefly present the current major shortcomings of the editor along with possible additional features and improvements. Hopefully these improvements will draw the editor closer towards becoming a fully functional language workbench for deep metamodelling.

7.2.1 Current Shortcomings

The editor we implemented in this thesis should be regarded as a prototype as it has not been properly tested yet, and it still lacks functionality for it to be regarded as a fully functional editor for deep metamodelling. The current shortcomings can be summarized as follows:

Constraints: As briefly mentioned earlier in this thesis, the DPF Editor currently has support for Constraints on Arrows, but currently does not support deep instantiation of Constraints. This is a much needed feature, and should be added in the future. Secondly, we need to add support for the definition of potency specific constraints such as the need to instantiate attributes with a datatype and slots with a value. We also need to restrict containing elements from being instantiated with a higher potency than the model element it is contained in. Thirdly, we need constraints for the definition of mutability. Lastly we need a proper naming constraint to ensure that model elements does not have the same name as linguistic model elements.

Datatypes And Values: We need to ensure proper semantics of datatype hierarchies, such as when changing the datatype of an element, the new datatype needs to be below the previous type in the typehierarchy. A datatype can not be instantiated as a supertype of its type, and proper semantics are needed to ensure this in practice.

Additional Visualization Structures: Currently the visualization editor does not update properly after modifying visual elements. For example when attributes are automatic generated, they are not visualized in the clabject until the modelling window is closed and opened again. The current visualization editor is also very rigid, as it is only possible to model UML Class-like structures of compositions. It is however possible to model with additional structures of model elements that is not composit, but to make the visualization editor more tailored to the domain of concern, we argue that it is necessary with additional structural features.

Code Duplication: When we implemented the E-Graph, we did not put a large effort into generalizing code for re-use. The code also needs to be properly tested to ensure overall system correctness.

API Consistency: Even though it is not a shortcoming in the tool itself, all the projects in DPF is relying on the DPF Core API, which means we are prone to error due to modifications in the DPF core metamodel. The biggest problem is however due to the possibility of rendering old metamodels useless. A migration strategy is therefore needed to ensure system correctness, both for the API and the models themselves.

7.2.2 Additional Features In The Model Editor

There is still functionality missing in the the DPF Editor for it to be considered a fully useable language workbench for deep metamodelling. In this section we will therefore propose some additional features that hopefully will make the Model Editor reach a more mature state. The proposed features can be summarized as follows:

Inheritance: In this thesis we have only considered the concept of instantiation as a mechanism for modellers to define type-hierarchies. However, we argue that to provide a well-formed meta-hierarchy, it is in some cases useful to support inheritance along with instantiation as well. In some cases it is more natural to define generalized model elements and inherit from these model elements in a single metamodel.

Constraints on Node-/ArrowAttributes: It will be useful to extend the editor with support for Constraints on NodeAttributes and ArrowAttributes as well only regular Arrows. This would be especially useful in cases such as for example when modelling Method-Parameters in UML Class diagrams. In the case of the default template we created earlier in this thesis, we specified the parametertype with both an Arrow to a Clabject as well as a NodeAttribute as illustrated in figure 7.1 below. It should however only be possible to specify one parametertype per parameter, and a *XOR* constraint between the Arrow to the Clabject and the NodeAttribute would therefore be a useful feature. The same principles should be added to ArrowAttributes as well.

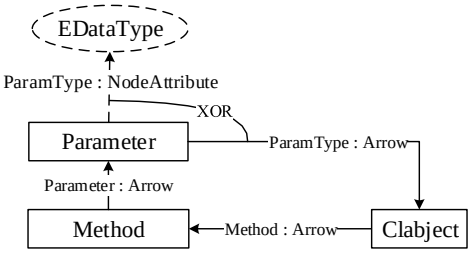


FIGURE 7.1: Constraint between a NodeAttribute and an Arrow.

Model Import: We may recall that in the comparison analysis in chapter 4, we revealed that Metadepth has support for importing external models. As the DPF Editor reaches a more mature state, importing external models may be a much needed feature and should therefore be investigated further. By supporting import of models into an existing model, we can provide libraries of functionality which we can import into other models.

Query Language: When executing the replication rules we defined earlier in this thesis, performing type-checking, traversing the metamodelling stack or similar, we need to iterate through a large number of model elements programmatically. As the models are growing bigger, the process of retrieving model elements are getting slow. To speed up this process and to make these algorithms easier to understand and modify, it would be useful to add support for a query language instead. This could be especially useful in the process of defining specific behaviour of certain linguistic metamodels, such as functionality in the enriched graph. In addition to speeding up the retrieval of model elements and make it easier to specify for example replication rules, a query language could run queries in the background to provide on-the-fly validation of model elements such as ensuring proper typing of datatypes or similar.

Default Template Diagram Improvements: The current implementation of the default template (UML Class diagram) currently only supports clabjects, but lacks support for packages and enums. A proper foundation for interfaces other than the *isInterface* property has also not been established. Secondly and with inspiration from EMF, it could also be useful to be able to express new types of data types that does not exist in the list of predefined data types in EMF. Support for adding annotations should also be investigated further.

Predefined Diagrams: As the current implementation only supports the definition of UML Class Diagrams, it would be useful to implement additional templates for other types of diagrams as well. Whether these diagrams should be defined through a type of plug-in, or if they should be defined programmatically is also something that needs to be elaborated for.

Modelling Modes For Linguistic Extension: In the comparisons analysis, we also noticed that both Metadepth and Melanie features modelling modes restricting model elements to be linguistically extended or not. (In Metadepth this is referred to as strict and extensible modelling modes). In certain situations this may be a useful feature in DPF as well, and we therefore argue that this should be investigated further.

Transformation To A Traditional Metamodelling Stack: We also need to discuss the relevance of transforming the deep metamodelling stack back to a traditional metamodelling stack. This subject was in the focus on Alessandro Rossini's Ph.D. thesis, which provides a formal approach to the problem [34].

7.2.3 Concrete Syntax Improvements

Even though the core functionality of deep metamodelling is established in the Model Editor, it is necessary to extend the Visualization Editor to provide a better user experience. This section will propose some additional features that may benefit the Visualization Editor in the future. The proposed features can be summarized as follows:

Improved customization of the concrete syntax
As mentioned in chapter 6, with the additional visualization data in the template model, we could use the mapping algorithm to automatically map each element in the abstract syntax to its corresponding concrete syntax. Conseptually, modellers could begin modelling in a default concrete syntax much like the current abstract syntax, and modellers could specify the visualization of each model element on-the-fly through the property dialog. Figure 7.2 below, illustrates a possible scenario where a modeller is changing the visualization of a model element.

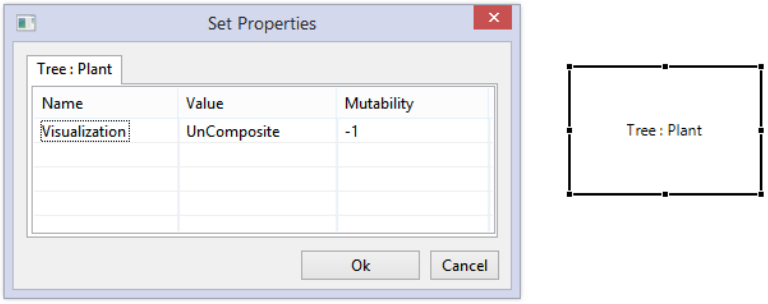


FIGURE 7.2: Defining visualization through the property dialog.

All NodeAttributes and ArrowAttributes would still be hidden from users in the Visualization Editor, but can be accessed through the property dialog instead. In our proposed solution, modellers could begin with a default concrete syntax such as the current abstract syntax, and then specify the concrete visualization with the property dialog. Modellers would have the current concrete visualization in front of them, and by specifying the visualization data of each model element, modellers could see the result upfront. Modellers would no longer need to go through the current process of creating a visual metamodel, specifying visualization data in the abstract syntax, then running the Template Visualization Wizard. The idea is that modellers simply would start by creating a new metamodel in a default concrete syntax, and then specify the visualization of each element directly. The result is that modellers have better control over how the concrete syntax will look like. Additionally we realize that the visualization data will endure as long as the DeepNode or DeepArrow it is attached to. This means that if modellers want to, they could re-map the visualization to other visualizations at different metalevels. In terms of the Plant example,

modellers could choose the Plant to be visualized as a plant, the Tree as a tree and NorwaySpruce as a norway spruce.

Secondly, visual elements could be created according to how we want the model elements in the concrete syntax to be visualized. Ideally these elements should also be accessed from a list in the property dialog where modellers can choose what visual element each model element should be mapped to.

Extended Property-Sheet
Instead of editing NodeAttributes and ArrowAttributes in the property dialog, there should be investigated if its possible to do this in the property-sheet instead. It should be investigated if its possible to extend the property-sheet with possibility of not only editing a single Node or Arrow, but also its attached DataNodes (NodeAttributes and ArrowAttributes).

Additional Composition-Levels
In the current implementation of the visualization editor, it is only possible to model compositions of nodes with containing nodes. It is not possible to compose nodes in another node that is already contained in a third node, meaning it is not possible to model compositions of more than one composition-level. As an example, we may recall the composition of Methods inside Clabjects. Currently there are no structure to provide visualization of what Parameters the Methods are containing. This is clearly a drawback with the current implementation, and the visualization editor should therefore be extended to support more than one composition level.

7.2.4 Fully Functional Code Generator

In chapter 4 we developed a lightweight code generator, that can generate Metadepth models based on DPF models. It would be of great use to complete the implementation so that it is fully functional, and we could benefit from the functionalities of Metadepth as a textual language as well. To take it even further, we propose that a code generator that can import Metadepth models that can be used in the DPF Editor as well. This way, the DPF Editor could import Metadepth models, modify them graphically, and then save them back to Metadepth format and benefit from its code-generation capabilities amongst others.

List of Figures

List of Tables

Abbreviations

OOP	**O**bject **O**riented **P**rogramming
MDE	**M**odel **D**riven **E**ngineering
API	**A**pplication **P**rogramming **I**nterface
GPL	**G**eneral **P**urpose **L**anguage
DSML	**D**omain **S**pecific **M**odelling **L**anguage
CASE	**C**omputer-**A**ided **S**oftware **E**ngineering
MOF	**M**eta **O**bject **F**acility
EMOF	**E**ssential **M**eta **O**bject **F**acility
CMOF	**C**omplete **M**eta **O**bject **F**acility
SMOF	**S**emantic **M**eta **O**bject **F**acility
OMG	**O**bject **M**anagement **G**roup
XMI	**XML M**eta-data **I**nterchange format
UML	**U**nified **M**odelling **L**anguage
EMF	**E**clipse **M**odelling **L**anguage
DPF	**D**iagram **P**redicate **F**ramework
OCL	**O**bject **C**onstraint **L**anguage
MDA	**M**odel **D**riven **A**rchitecture
PIM	**P**latform **I**ndependent **M**odel
PSM	**P**latform **S**pecific **M**odel
UAM	**U**niversidad **A**utónoma de **M**adrid
EVL	**E**psilon **V**alidation **L**anguage
EOL	**E**psilon **O**bject **L**anguage
EGL	**E**psilon **G**eneration **L**anguage
ER diagram	**E**ntity **R**elationship diagram
PLM	**P**an **L**evel **M**odel
BUC	**B**ergen **U**niversity **C**ollege
UOB	**U**niversity **O**f **B**ergen
FOL	**F**irst **O**rder **L**ogic
GS	**G**eneralized **S**ketches
DPL	**D**iagrammatic **P**redicate **L**ogic
GEF	**G**raphical **E**diting **F**ramework
PLM	**P**an **L**evel **M**odel
LML	**L**evel **A**gnostic **M**odelling **L**anguage
MOFM2T	**MOF M**odel to **T**ext **T**ransformation Language
PIMH	**P**latform **I**ndependent **M**odelling **H**ierarchy

Bibliography

[1] Frances E. Allen. The history of language processor technology in ibm. *IBM Journal of Research and Development*, 25(5):535–548, 1981.

[2] Andrew P Black. Object-oriented programming: some history, and challenges for the next fifty years. *Information and Computation*, 231: 3–20, 2013.

[3] Douglas C Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.

[4] Dragan Gaševic, Dragan Djuric, and Vladan Devedžic. *Model driven engineering and ontology development*, volume 2. Springer, 2009.

[5] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004. ISBN 0321245628.

[6] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE software*, 20(5):42–45, 2003.

[7] Cesar Gonzalez-Perez and Brian Henderson-Sellers. *Metamodelling for software engineering*. Wiley Publishing, 2008.

[8] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17, 2003.

[9] Colin Atkinson and Thomas Kühne. Rearchitecting the uml infrastructure. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 12(4):290–321, 2002.

[10] Adrian Rutle, Alessandro Rossini, Yngve Lamo, and Uwe Wolter. A diagrammatic formalisation of mof-based modelling languages. In *Objects, Components, Models and Patterns*, pages 37–56. Springer, 2009.

[11] Mohamed Fayad and Douglas C Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, 1997.

[12] David S Platt. *Introducing Microsoft. Net*. Microsoft press, 2002.

[13] Nicholas Kassem and Enterprise Team. *Designing Enterprise Applications: Java 2 Platform*. Addison-Wesley Longman Publishing Co., Inc., 2000.

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

[15] Thomas Baar. Correctly defined concrete syntax for visual modeling languages. In *Model Driven Engineering Languages and Systems*, pages 111–125. Springer, 2006.

[16] XML OMG. Metadata interchange (xmi) specification. *URL: http://www. omg. org/docs/formal/05-05-01. pdf (accessed October 10, 2005)*, 2000.

[17] Oxford English Dictionary Online, 2nd edition. http://www.oed.com/, July 2003.

[18] M. Hack. Petri net language. Technical report, Cambridge, MA, USA, 1976.

[19] Faraz Ataie, Vincent P Aubrun, Leonid Erlikh, Michael Fischer, Michael Fochler, Craig B Hayman, Daniel Hildebrand, James Hughes, Jeffrey L Lambert, Douglas E Lee, et al. Computer-aided software engineering facility, March 15 1994. US Patent 5,295,222.

[20] Marco Bernardo Vittorio Cortellessa and Alfonso Pierantonio. Formal methods for model-driven engineering. 2012.

[21] Adrian Rutle. *Diagram Predicate Framework: A Formal Approach to MDE*. PhD thesis, The University of Bergen, 2010.

[22] Meta Object Facility. Meta object facility (MOF) 2.0 core specification, 2003. Version 2.

[23] OMG CORBA and IIOP Specification. Object management group, 1999.

[24] Infrastructure Version 2.3. Object management group. unifed modeling language (omg uml). http://www.omg.org/spec/, 2010.

[25] OMG Available Specification. Object constraint language, 2006.

[26] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182, 2012.

[27] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[28] DPF: Diagram Predicate Framework. Project Web Site. http://dpf.hib.no/.

[29] Zinovy Diskin and Boris Kadish. Variable set semantics for keyed generalized sketches: Formal semantics for object identity and abstract syntax for conceptual modeling. *Data Knowl. Eng.*, 47(1):1–59, October 2003. ISSN 0169-023X. doi: 10.1016/S0169-023X(03)00047-8. URL http://dx.doi.org/10.1016/S0169-023X(03)00047-8.

[30] Raymond M Smullyan. *First-order logic*, volume 6. Springer, 1968.

[31] Bastian Kennel. A unified framework for multi-level modeling. 2012.

[32] Colin Atkinson and Thomas Kühne. The essence of multilevel metamodeling. In *«UML» 2001—The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pages 19–33. Springer, 2001.

[33] James Odell. Power types. *JOOP*, 7(2):8–12, 1994.

[34] Alessandro Rossini. Diagram predicate framework meets model versioning and deep metamodelling. *Dissertation for the degree of Philosophiae Doctor (PhD)*, 2011.

[35] Juan De Lara and Esther Guerra. Deep meta-modelling with metadepth. In *Objects, Models, Components, Patterns*, pages 1–20. Springer, 2010.

[36] Timo Asikainen and Tomi Männistö. Nivel: a metamodelling language with a formal semantics. *Software & Systems Modeling*, 8(4): 521–549, 2009.

[37] Thomas Kuehne and Daniel Schreiber. Can programming be liberated from the two-level style: multi-level programming with deepjava. In *ACM SIGPLAN Notices*, volume 42, pages 229–244. ACM, 2007.

[38] Vijay Vaishnavi and William Kuechler. Design research in information systems. 2004.

[39] Briony J Oates. *Researching information systems and computing*. Sage, 2005.

[40] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. Eclipse development tools for epsilon. In *Eclipse Summit Europe, Eclipse Modeling Symposium*, volume 20062. Citeseer, 2006.

[41] Colin Atkinson, Bastian Kennel, and Björn Goß. The level-agnostic modeling language. In *Proceedings of the Third International Conference on Software Language Engineering*, SLE'10, pages 266–275, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19439-9. URL http://dl.acm.org/citation.cfm?id=1964571.1964594.

[42] Yngve Lamoa, Xiaoliang Wang, Florian Mantz, Øyvind Bech, Anders Sandven, and Adrian Rutle. Dpf workbench: a multi-level language workbench for mde. *Proceedings of the Estonian Academy of Sciences*, 62(1), 2013.

[43] GEF (Graphical Editing Framework). http://www.eclipse.org/gef/.

[44] Ola Bråten. Dpf visualisation, a tool for defining new concrete syntaxes for diagrammatic modelling languages. *Masters Thesis in Informatics – Program Development, University of Bergen and Bergen University College*, 2013.

[45] Anders Sandven. Metamodel based code generation in dpf editor. *Masters Thesis in Informatics – Program Development, University of Bergen and Bergen University College*, 2012.

[46] Acceleo. Project Web Site.). http://www.eclipse.org/acceleo/.

[47] Alessandro Rossini, Juan de Lara, Esther Guerra, Adrian Rutle, and Yngve Lamo. A graph transformation-based semantics for deep metamodelling. In *Applications of Graph Transformations with Industrial Relevance*, pages 19–34. Springer, 2012.

[48] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamental theory for typed attributed graph transformation*. Springer, 2004.