DPF Visualisation, a Tool for Defining new Concrete Syntaxes for Diagrammatic Modelling Languages

Ola Bråten
Master’s Thesis in Informatics – Program Development

Department of Informatics
University of Bergen

Department of Computer Engineering
Bergen University College

December 2013
Acknowledgements

First off I would like to thank my supervisor, Yngve Lamo, for invaluable help and support. I would also like to thank the other members of the DPF team: Florian Mantz, Anders Sandven, Suneetha Sekhar, Sidra Nadeem, Petter Barvik, Ole Klokhammer and Pål Bjørhovde who all helped evaluate my work at our meetings. I would especially like to thank Xioliang Wang of the DPF team who provided me with much needed guidance and help.

Special thanks also go to my parents for encouragement, and my brother Erik for input, encouragement and some proofreading. My other brother Lars, his fiancée Vivi and their little Mia have also encouraged me to work. I would also like to thank Emma for her encouragement and support throughout this thesis.
# Contents

## Acknowledgements

## List of Figures

## Abbreviations

### 1 Introduction
1.1 Motivation ............................................. 1
1.2 The Structure of the Thesis .......................... 2

### 2 Background Theory
2.1 Modelling Languages .................................. 4
2.2 Model-Driven Engineering ............................. 5
2.3 Metamodelling .......................................... 6
2.4 The Diagram Predicate Framework .................. 7
2.5 The DPF Workbench ................................. 17

#### 2.5.1 The Model Editor ............................. 8
#### 2.5.2 The Signature Editor ......................... 9
#### 2.5.3 Code Generation .............................. 10

### 3 Visual Syntax
3.1 Concrete and Abstract Syntax ..................... 11
3.2 Concrete and Abstract Syntax in DPF .......... 12
#### 3.2.1 Concrete and Abstract Syntax in other Modelling Languages .... 14
#### 3.2.1.1 Metaedit+ ................................... 14
#### 3.2.1.2 UML ......................................... 15
3.2.2 The Eclipse Modeling Project .......... 17
3.3 The Benefits of a Customisable Concrete Syntax .... 17

### 4 Problem Description and the Eclipse Wizard Approach
4.1 Problem Description ................................. 19
4.1.1 Possible Solutions ............................. 20
4.2 The Eclipse Wizard Approach .................... 22
4.2.1 Solution ........................................... 23
4.2.2 Wizard Demonstration ......................... 25
## Contents

5 The Model Mapping Approach 28
  5.1 Solution ................................................. 28
  5.2 Implementation and Development ............................ 32
    5.2.1 Development Method ............................... 33
    5.2.2 Packages ......................................... 34
    5.2.3 Implementation .................................. 36
  5.3 Properties ............................................... 37
  5.4 Demonstration ........................................ 38

6 Conclusion 43
  6.1 Summary ............................................... 43
  6.2 Future Work .......................................... 44
  6.3 Last words ............................................ 45

A The DPF Editor Tutorial 46
  A.1 Installing the DPF Workbench ............................. 46
  A.2 Using the DPF Model Editor .............................. 47
  A.3 The first metamodel level ................................. 48
  A.4 The second metamodel level .............................. 50
  A.5 The model level ....................................... 53
  A.6 References ............................................ 55

B The DPFNodeEditPart 56

Bibliography 59
List of Figures

2.1 The DPF Workbench components. ............................................. 8
2.2 A generic metamodelling hierarchy as it is implemented in the DPF Editor. 8
2.3 DPF Model Editor view with a simple model open. ....................... 9
2.4 DPF Signature Editor with the highlighted xor constraint. ............. 10

3.1 The concrete syntax in a DPF metamodel and its instance. .......... 13
3.2 A model and the symbol editor of Metaedit+. ........................... 15
3.3 The hierarchy of the 14 standard UML diagrams. .................... 16
3.4 The same UML diagram made in two different UML tools. .......... 16

4.1 The Eclipse Wizard solution. ..................................................... 21
4.2 The model mapping solution. .................................................. 22
4.3 The original DPF-Model. ....................................................... 25
4.4 The first page of the wizard. .................................................. 26
4.5 The second page, choose a file to load. ................................... 26
4.6 The third page, choose figures for each element. ....................... 27
4.7 The final visually altered copy. ............................................. 27

5.1 A very simple class diagram of the Visual Metamodel Editor only showing two attributes to retain simplicity. ................................. 29
5.2 The process of making a new visualisation model. ....................... 30
5.3 A very simple class diagram of the Visualisation Model Editor. .... 31
5.4 The figure for a composite node and its parts explained. ............ 32
5.5 The Ecore models for the Visual Metamodel editor and the Visualisation Editor ................................................................. 33
5.6 The metamodel for a class diagram. ......................................... 38
5.7 A visual model. ................................................................. 39
5.8 The wizard for creating a new visualisation model. .................... 39
5.9 A new visualisation model with one class. ............................... 40
5.10 Attributes and methods can be added to the class. .................... 40
5.11 The model can be opened as a normal DPF model. .................... 41
5.12 Visualisation Editor with two classes and a simple node. .......... 41
5.13 The updated normal DPF model. ........................................... 42

A.1 Make a new DPF project. ...................................................... 48
A.2 Model view with the signature bar highlighted. ....................... 49
A.3 The complete process_m3 model. ........................................ 50
A.4 Load a typegraph for the new model. ......................... 51
A.5 The complete process_m2 model. ............................... 52
A.6 The highlighted arrow will make the validation fail. .......... 52
A.7 The partial model. ............................................... 53
A.8 The full model. ................................................ 54
A.9 The full model, viewed in the new visualisation editor. ....... 55
Abbreviations

DPF  Diagram Predicate Framework
GMF  Graphical Modeling Framework
EMF  Eclipse Modeling Framework
GEF  Graphical Editing Framework
XML  EXtensible Markup Language
XMI  XML Metadata Interchange
CPN  Coloured Petri Net
OCL  Object Constraint Language
SMODL Simple MethOd Declaration Language
DSML Domain Specific Modelling Language
MDE  Model Driven Engineering
MDA  Model Driven Architecture
GOPPRR Graph Object Port Property Relationship Role
UML  Unified Modeling Language
OMG  Object Management Group
MOF  Meta-Object Facility
Chapter 1

Introduction

This chapter contains a brief introduction with motivation and a summary of the structure of the thesis.

1.1 Motivation

Modelling languages have proven useful through the years for expressing both technical and non-technical problems in a form that is easier to understand. Both textual and graphical models can be used to formalise a problem so that one can see the problem as a whole and get an overview of all the different aspects involved.

In software engineering modelling languages are often used to create models of the domain. These models are typically easier to interpret, especially for the user, so that one can make sure the problem is fully understood. In model-driven engineering these models can often also be used to generate software for at least a partial solution to the problem they describe.

Models are usually divided into two types of syntaxes, abstract and concrete. The abstract syntax denotes how the computer interprets the model, it contains all the information needed in an unambiguous form. The concrete syntax is the part the user interacts with, the visual metamodel.
To start off this thesis I made a tutorial in order to understand how the DPF Model Editor works and learn how to use it. The tutorial was made after an example in a published paper about the DPF Editor [1]. When writing the tutorial a problem became very noticeable, bigger models can easily get confusing to look at when everything is made up of rectangles and arrows, see figure A.8.

This thesis will focus on the visual syntax of the Diagram Predicate Framework (DPF) Models, and finding a more organised way of showing the link that is between elements in the abstract syntax. In modelling languages like the Unified Modeling Language (UML) there are several ways to represent elements in the model, a class is shown as a rectangle and inside there can be attributes and methods shown simply as text. Adding a function for defining a representation for each element in a model could be very useful to the DPF Editor.

### 1.2 The Structure of the Thesis

The following is the general structure of the thesis:

**Chapter 2 - Background Theory**
This chapter gives an introduction to modelling languages and metamodelling. It also includes a detailed section on the current state of the Diagram Predicate Framework.

**Chapter 3 - Visual Syntax**
Here both concrete and abstract syntax is explained and shown, it contains details on how the concrete syntax of the DPF was before this project started and compares it to other modelling frameworks.

**Chapter 4 - Problem Description and the Eclipse Wizard Approach**
As the name suggests this chapter contains a description of the problem and the corresponding research problem. A description of the first attempted approach to implement a solution is also included in this chapter.

**Chapter 5 - The Model Mapping Approach**
This chapter contains information about the tools and methods used as well as a detailed
explanation of the main implementation. Also included is a simple demonstration of the resulting tool.

**Chapter 6 - Conclusion**

The last chapter has a short summary of the project. It also contains some information about how the implementation in this thesis can be expanded by future work.
Chapter 2

Background Theory

This chapter contains some background theory about modelling languages and the DPF Workbench, with its components, is explained in detail.

2.1 Modelling Languages

A modelling language is a very useful tool, especially in software development. Using a good graphical modelling language can help programmers, software specification designers and domain experts understand each other. As long as the modelling language has a consistent syntax anyone involved in the project should be able to understand the models, no matter what background they have [2].

In order to properly describe a problem the modelling languages are often made to solve a specific problem domain, these languages are called Domain-Specific Modelling Languages (DSMLs). Like other modelling languages DSMLs can be either textual or graphical.

In later years some software developers have put more focus on models, and have made them the main artefacts in the development process, this is called Model-Driven Engineering (MDE). In MDE the models usually serve more purposes than just visually describing a problem.
2.2 Model-Driven Engineering

Initially in software development, models were just treated as a way to sketch up the architectural design, or to document a finished implementation. Recent developments have made models increasingly vital early on in the software development process. The main reason models have become more important is the ability to generate code for a partial solution from them [3].

As its name suggests, Model-Driven Engineering (MDE) uses models as the main artefact in the development process. Through code generation the programmers can use the models to get some code that could otherwise be tedious and repetitive to write [4]. This leaves more time to work on solving the more complex parts of the software. Although the models new main purpose might be to generate a partial solution they can still be used as documentation, giving a more abstract view of the problem.

An important organisation when on the topic of MDE is the Object Management Group (OMG). OMG is an international, open membership, not-for-profit computer industry standards consortium [5]. They maintain specifications for important MDE-related technologies like UML, Model-Driven Architecture (MDA) and XML Metadata Interchange (XMI).

An important operation of MDE is model transformation, this operation is used to automatically construct or modify models. In the MDA guide a model transformation is defined as "The process of converting one model to another model of the same system" [6]. A source model is modified by a set of transformation rules, called a transformation definition, and ends up as the desired target model [7].

In addition to generating code models can also be used to create new modelling languages which in turn can be used to define new models, this process is known as metamodeling.


2.3 Metamodelling

Metamodelling can be said to be the act of creating at least two models where the second model conforms to the first model, called the metamodel. For a model to conform to a metamodel it has to adhere to the properties and constraints specified by the metamodel, the model can then be said to conform to the metamodel. A metamodel defines a new modelling language, which in turn can be used to create another metamodel or model. This allows for the construction of virtually any DSML [8].

It is theoretically possible to have an infinite number of metamodel-levels, a chain of metamodels that all conform to the one before them. However most editors and tools limit the number of metamodel-levels one can create to two or three.

A common problem in metamodelling occurs when a metamodel is edited after there has been created a model or more on the next layers. These models will then have to be updated in order to conform to the new definition. This problem is known in literature as the metamodel evolution problem [9].

In order to specify more complex relationships between elements in metamodels and models one can use constraints. Constraints modify the relationships between the elements for example by specifying an arity or applying xor to the relation, which is explained in chapter 2.5.2.

When metamodelling a new DSML the result is an abstract syntax, this is considered to be the standard in MDE for defining new abstract syntaxes. However at present time there is no similar standard for defining concrete syntaxes.

2.4 The Diagram Predicate Framework

The Diagram Predicate Framework (DPF) is a collaborative research project started at Bergen University College and the University of Bergen back in early 2006. The goal of the research project is to formalise key concepts of model-driven engineering like metamodelling and code generation. Many master students, PhD candidates and
scientists have worked together on this research project to make it into what it is today, and several people are still working on it [10].

The foundation of the framework is the Generalised Sketches formalism, which was originally developed by Zinovy Diskin [11]. It employs unlimited metamodeling levels. Starting with very abstract elements the framework allows the creation of virtually unlimited layers of metamodels. This makes the framework very versatile, it is possible to model other modelling languages like UML and Coloured Petri Nets using the DPF [12].

2.5 The DPF Workbench

The DPF Workbench is a collection of useful software tools related to DPF. All the tools are made as plugin projects for the Eclipse Modeling Project and are working together to make up the workbench. The Eclipse Modeling Project provides a unified set of modelling frameworks and tools, this leads to a very solid foundation for making modelling tools [13].

At the bottom lies the DPF Core project, it provides basic DPF functionality like making models and metamodels with constraints. The next layer is the Diagram project, which extends the core with some visual properties like size and location for nodes and arrows.

The visual properties provided by the Diagram project are necessary for the Model Editor project, this project serves as a graphical DPF Editor giving users a way of interacting with DPF in a visual way. In order to let the user define new custom signatures the Signature Editor project was added.

Not long ago the data related to the concrete syntax and the data related to the abstract syntax of a model was stored tightly bound together and in a single file. This has now changed, and the data for the two syntaxes are stored in two separate files. Storing the two syntaxes more or less separately is very useful when you want to modify one of them, the risk of accidentally doing changes to the other one is then significantly lower. The serialisation procedure is explained in more detail in chapter 3.2.
The Code Generation project was added in order to provide the functionality for generating code from DPF-Models [14]. This prototype project uses the Xpand framework with a DPF Xpand metamodel as middle layer [15].

![Figure 2.1: The DPF Workbench components.](image)

2.5.1 The Model Editor

The model editor is an implementation which offers a graphical tool for creating and modifying models based on DPF. There have been a few attempts at making an implementation of DPF before the current editor was made, but they did not reach a fully functional state because of various problems with the underlying frameworks that were chosen.

![Figure 2.2: A generic metamodelling hierarchy as it is implemented in the DPF Editor.](image)

As seen on figure 2.2 above each metamodelling level in the DPF Editor can be said to be a diagrammatic editor of its own. Each editor then consists of a signature and a metamodel, these two are then used together in order to define the metamodel for the next level and a signature [1].
The implementation of the current model editor was started in 2011 by Øyvind Bech and Dag Viggo Lokøen [8], they based the editor on GEF and EMF. Much of the work that has been done since is based on some of Bechs suggestions for future work in his thesis, including the signature editor and the work in this thesis.

2.5.2 The Signature Editor

A signature is a collection of constraints one can use on the DPF-Models, the default signature in the editor consists of constraints like irreflexive and xor. Originally the editor implementation had one hard-coded signature, but it was found desirable to be able to define custom signatures [8]. Figure 2.3 shows a DPF-Model with an xor constraint, in the figure a Node0 must be connected to exactly one Node1 or exactly one Node2.

The Signature Editor was initially implemented by the PhD Candidate Xioliang Wang [13]. It has later been extended by Master student Sidra Nadeem, she made it possible to use the Object Constraint Language (OCL) to define predicates in custom signatures, and added better validation and error handling [16]. OCL is a declarative language that was made in order to describe expressions on UML models [17].
Figure 2.4 above shows a view of the signature editor with the DPF Model Editors default set of constraints open.

2.5.3 Code Generation

A prototype of code generation functionality was added to the workbench in 2012 by Anders Sandven [14]. In his implementation he made use of the Xpand framework, which is a framework for code generation based on EMF models [15].

The code generation prototype was utilized in a case study by Master student Suneetha Kalakata in her thesis. In this case study she created a bidirectional model transformation between textual Simple MethOd Declaration Language (SMODL) models and DPF SMODL models. SMODL is a type of XML language that aims to make it possible for domain experts with little technical knowledge to define and generate web services [18].
Chapter 3

Visual Syntax

This chapter explains what abstract and concrete syntax is and what they consist of. It also explains the abstract and concrete syntax in the current DPF Workbench and compares this to the syntaxes in other modelling tools.

3.1 Concrete and Abstract Syntax

The syntax of modelling languages can be divided into two different parts, abstract and concrete syntax. The abstract syntax defines the modelling concepts and the concrete syntax defines how the modelling concepts should be visualised [19]. In some cases these two syntaxes intersect and contain some of the same information, but ideally they should be kept apart. The separation of these two syntaxes makes the structure more understandable, and is a step towards allowing several concrete syntaxes to be used with one abstract syntax [20].

Concrete syntaxes may consist of either textual or visual elements, they can also consist of both. The concrete syntax is tailored to make each element in the abstract syntax distinguishable. A natural way of checking the quality of a concrete syntax is to say that it should not be possible to render two different models (instances of the same abstract syntax metamodel) with the same diagram [19]. Sometimes an abstract syntax
even has several concrete syntaxes in order to be able to visualise all its aspects, like the example in table 3.1 below.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4 + 6</td>
<td>Infix</td>
</tr>
<tr>
<td>(+ 4 6)</td>
<td>Prefix</td>
</tr>
<tr>
<td>(4 6 +)</td>
<td>Postfix</td>
</tr>
<tr>
<td>The sum of 4 and 6</td>
<td>English</td>
</tr>
</tbody>
</table>

Table 3.1: A simple example with several concrete syntaxes for one abstract syntax.

Usually changes to a model are done through the concrete syntax, which will then update the abstract syntax to follow the changes. Typically changes done directly to the abstract syntax will cause the concrete syntax to be out of date, as it will not be update automatically [21].

While it is becoming more and more normal to define domain-specific abstract syntaxes using metamodelling, there does not exist any corresponding standard for defining concrete syntaxes [20] [22] [23].

3.2 Concrete and Abstract Syntax in DPF

In the DPF there is no clear separation of abstract and concrete syntax, basically there is only an abstract syntax. This abstract syntax is therefore used as a concrete syntax as well. Doing it that way is very simple and works well for relatively small and simple models. Problems arise when the models become too big to grasp, or when certain constraints involve several elements in the model.

The DPF Workbench has one module for the abstract syntax, the DPF Core module, and one module responsible for the visualisation, the DPF Diagram module. These modules can be seen in figure 2.1 which shows all the parts of the DPF Workbench. In a sense they can be said to handle the abstract (DPF Core) and concrete (DPF Diagram) syntax separately. However, the DPF Diagram module is heavily dependent on the DPF Core module, and it can be argued that they are not entirely distinct. The DPF Diagram
module basically extends the DPF Core module by adding visual data to every datatype that needs it.

The serialisation of DPF models is handled by EMF which stores them as XML Metadata Interchange (XMI) files. XMI was chosen for storing the models because it is a standard for serialising metadata. Each DPF model is split up and stored in two separate files, a \textit{modelname.dpf} file and a \textit{modelname.xmi} file. The \textit{modelname.dpf} file contains data used in the diagram module, which is the concrete syntax, while the \textit{modelname.xmi} file contains the data used in the core module, which is the abstract syntax.

The abstract syntax in DPF is defineable through metamodelling, the user can make as many layers of metamodels as they want before they end up with the domain-specific abstract syntax they wanted. In theory it is possible to end up with virtually any arbitrary new or existing abstract syntax.

While the abstract syntax in DPF is fully defineable, the concrete syntax is much more static. The current concrete syntax of DPF does not hold up well to the quality-check mentioned in the previous section, that is, two different DPF models can have the same concrete syntax. One might say it consists of both a visual and a textual part, the visual part being the square boxes and arrows, and the textual part being the name and type of the elements, which are separated by a colon, as seen in figure 3.1.

![Figure 3.1](image)

FIGURE 3.1: The concrete syntax in a DPF metamodel and its instance.

As can be seen in figure 3.1 above, the visual part of the concrete syntax in DPF does not change at all with the layers of metamodels. The textual part, which is underlined in the figure, changes for each layer, but it is not easy to spot at a glance. Especially when working with bigger models in DPF it becomes a more apparent problem that the
concrete syntax changes so little. Figure A.8 is a good example of a big model which becomes difficult to understand because all the elements are visualised so similar.

3.2.1 Concrete and Abstract Syntax in other Modelling Languages

3.2.1.1 Metaedit+

Metaedit+ is a modelling suite created by MetaCase, it is composed of the Metaedit+ Workbench, in which a metamodel can be created, and the Metaedit+ Modeler, where one can create models using the modelling language specified by the metamodel from the Metaedit+ Workbench [24].

In Metaedit+ it is possible to fully customise the concrete syntax of the modelling language made by the metamodel. A powerful symbol editor is provided, in which the user can draw virtually any figure and place any information inside it [25].

The metamodels in Metaedit+ are based on the GOPPRR metamodelling language, GOPPRR is an acronym made from the base types of the language which are Graph, Object, Port, Property, Relationship and Role. The GOPPRR modelling language then makes up the abstract syntax of the Metaedit+ models.
3.2.1.2 UML

The UML is a general-purpose modelling language that is mainly focused on modelling object-oriented software projects, but it can also be used to model non-software projects and business models [26]. UML is currently being maintained by the OMG.

The OMG defined UML by using the Meta-Object Facility (MOF), which is a meta-modelling architecture. It provides a unified metadata management framework [27].

In UML there are currently 14 diagram types as seen in figure 3.3, each with a different concrete syntax [28]. It is possible to extend the normal set of diagram types by defining new ones using UML profiles.
Because there are so many modelling tools that implement UML, and no standard tool, the concrete syntax may vary a little in the different tools. Some textual modelling tools only implement the abstract syntax and not the concrete syntax [29].

Figure 3.4 above shows the same UML model made in two different UML tools, as can be observed the concrete syntaxes are slightly different. The diagram on the left is made in ArgoUML, one of the most popular open source UML tools. At the right side is the
same diagram made in Enterprise Architect, which is a popular UML tool with many functions like code generation to over ten different languages.

### 3.2.2 The Eclipse Modeling Project

The Eclipse Modeling Project is a big collection of projects within the Eclipse community with the aim to promote and evolve model-based development. Included projects are sorted into one of several categories including abstract syntax, concrete syntax and model transformations.

The main project within the abstract syntax category is the EMF, which is also used by the DPF Workbench. The EMF is an extendable framework for defining abstract syntaxes, it can also generate java code from these abstract syntaxes. EMF is also defined using OMGs MOF, like UML is.

In the concrete syntax category there are several projects, one of them is GEF, which is also used by the DPF Workbench. Other projects in this category includes GMF and Graphiti. All of the projects in the concrete syntax category builds on EMF for the underlying abstract syntax.

### 3.3 The Benefits of a Customisable Concrete Syntax

The benefits of having a customisable concrete syntax are quite apparent, it gives the users the freedom to express and present their models the way they want to. It is particularly important in cases like with DPF, which can be used to model many very different domains effectively.

It is impossible to hard-code concrete syntaxes for, or even imagine, all the possible domains one can use DPF to model. The best solution is therefore to only provide a default concrete syntax with the most basic visualisations, and let the users define their own concrete syntaxes when they want to.
Ideally it should be possible for the user to define shapes and colours, and put elements inside each other in lists or groups, and maybe even choose visualisations for the constraints. Another useful feature would be to let the user define several abstraction levels, zooming levels with different amount of details, much like CPN Tools offers [30].
Chapter 4

Problem Description and the Eclipse Wizard Approach

This chapter contains the problem description and a section about the possible solutions to the problem. One of the solutions that was partially implemented is then explained and demonstrated.

4.1 Problem Description

As discussed in chapter 3.2 there is some room for improvement in the way that DPF models are visualised in the DPF Editor. The view consisting of only plain boxes and arrows does not tell the user much until they look closer and try to analyse the model.

Implementing several visually different boxes and arrows and letting the user choose between them would make models easier to understand. One would then be able to quickly see the difference between the various elements.

An even better solution would be to let the user define custom shapes and colours instead of having some predefined hardcoded ones. Some elements might also be natural to group up inside other elements, the user should then be able to choose this.
Ideally the user should be allowed to define a virtually unlimited number of concrete syntaxes for the abstract syntax he has metamodelled. It would then be possible to visualise every aspect of each model, or have different concrete syntaxes for domain experts and programmers.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Persistency</td>
<td>The concrete syntax should be stored separate from the abstract syntax.</td>
</tr>
<tr>
<td>Editable</td>
<td>The model should be editable with the new visualisations, and the abstract syntax should be synchronized.</td>
</tr>
<tr>
<td>Constraints</td>
<td>Constraints and constraint-checking should still work as intended in the DPF.</td>
</tr>
<tr>
<td>Flexible</td>
<td>The solution should be able to handle several different ways of changing the concrete syntax, like abstraction and grouping of elements.</td>
</tr>
<tr>
<td>Extendable</td>
<td>It will be beneficial if the solution is relatively easy to extend with additional functionality in the future.</td>
</tr>
<tr>
<td>Syntax quality</td>
<td>The concrete syntax should pass the quality check mentioned in chapter 3.1, two different models should not have identical visualisations.</td>
</tr>
</tbody>
</table>

Table 4.1: Specification for the solution.

Table 4.1 above contains a list of requirements the solution should meet.

### 4.1.1 Possible Solutions

One possible solution to the problem is to implement an Eclipse Wizard that loads a DPF model and allows for choosing a different visualisation for each element in the model. The Wizard then creates a copy of the model but changes all the figures into the figures the user chose. The user will end up with two versions of the model, one with the standard squares and arrows, and one with the new figures chosen by the user. The wizard does all this by altering an attribute on every element in the chosen models.
metamodel, this attribute specifies how all elements that are typed by this element will look.

![Diagram](image)

**Figure 4.1:** The Eclipse Wizard solution.

Figure 4.1 above shows how the process of making a visually altered DPF model would work in the wizard solution. The original DPF model would not be changed because it has its own copy of the original metamodel which is never updated to match the new altered one. Since the original model and the new model with different visualisation are separate they are not synchronised, meaning that editing one will not change the other. This solution is explained in more detail in the next section of this chapter.

Eclipse Wizards are used for many actions in the Eclipse Framework, mostly for processing files, be it creating a file, loading a file, or importing and exporting files. An Eclipse Wizard is a general interface for the user, it usually consists of several pages that the user is able to click forward or backwards through. It will then use the information and choices made on all the pages to execute a process when the user presses finish.

Another possible solution to the problem is the model mapping approach, in this solution the user must create a new type of model which contains visual information. Each element in the visual model has attributes related to the visualisation, for example if this element should be able to contain other elements or not. The user can then bind the elements in the visual model to the elements in his DPF model and a visualisation model will be made. This new visualisation model consists of a DPF model with an xmi and a dpf file as usual, in addition there is a visualisation file. The dpf and xmi parts can be opened and edited as normal in the DPF Editor, and the visualisation file if opened
would load the dpf and xmi file and combine them with the visual file in order to alter
the visualisations.

![Diagram](image)

Figure 4.2: The model mapping solution.

In figure 4.2 above, the process for making a visualisation model is shown. The vi-
ualisation model contains the binding between elements in the DPF metamodel and
elements in the visual metamodel. The DPF model and visualisation model are made
together and will always stay synchronized, any changes to one will be reflected in the
other. This solution and its implementation is explained in more detail in chapter 5.

4.2 The Eclipse Wizard Approach

The first approach to solve the research questions was to implement an Eclipse Wizard
that takes a DPF model as input and gives the user some options to customise how each
element in the model should be visualised. The wizard would then create a new model
with different visualisations based on the users choice of options.

This section will explain how this was implemented and its flaws which lead to the
choice of another solution.
4.2.1 Solution

The implementation is built up with the new file creation wizard as a base, this is a generic wizard for making a new file in a project. This wizard is then extended to offer the needed functionality.

<table>
<thead>
<tr>
<th>Page number one</th>
<th>Choose where to create the new model, and type in a name for it.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page number two</td>
<td>Find and choose the DPF-Model you want an alternate visualisation for.</td>
</tr>
<tr>
<td>Page number three</td>
<td>Lets the user select a hard-coded shape for each element in the loaded DPF-Model from a drop-down menu.</td>
</tr>
</tbody>
</table>

Table 4.2: Using the Eclipse Wizard implementation.

When the finish button is clicked the wizard will create a copy of the original DPF-Model, but change the respective typenodes’ configure-string attribute. The configure-string is an attribute on each element in the metamodel which defines how elements that are of this type in the model will be visualised. This will result in the elements getting different visualisations than in the original DPF-Model.

The new DPF-Model will be completely separate from the old one, they will not share any data. They share the same metamodel which they both have a separate copy of, but if the DPF Framework was ever changed in order to solve the metamodel evolution problem which is explained in section 2.3 a problem could arise. The new DPF-Model would then likely change the configure-string attributes on the same metamodel that the old DPF-Model uses and they would both end up using the new visualisation.

The Wizard solution is a simple solution that is easy to use, but it is quite rigid and leaves little choice to the user. With this implementation one can only choose from a set of hardcoded and predefined figures for each element. Hardcoding every imaginable figure would be a lot of work, and there would always be some figures missing as the usage area for the DPF-Framework is massive. In addition it can only alter the visualisation
of elements separately, there is no way of specifying that one element should be inside another, or to merge several elements into one new figure.

As mentioned earlier there is a problem that will appear if the DPF-Models are ever made to reload the typegraph when it is changed. This is part of a problem we want to solve, if one makes a new model based on an existing metamodel and then do changes to the metamodel afterwards the new model will not be updated accordingly. As long as this problem persists in the DPF-Editor this implementation works as it is, but if the problem is fixed the original DPF-Model and the copy made with the wizard will edit the same metamodel and both get updated to have the same visualisation, making them identical copies instead. To avoid this the wizard would have to make a copy of the metamodel as well, which could easily get confusing, and lead to a lot of duplicate data.

<table>
<thead>
<tr>
<th>Persistency</th>
<th>The concrete syntax is in this solution set as an attribute in with the abstract syntax in the metamodel, this is not a good place to store it.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Editable</td>
<td>The new model with a different concrete syntax is editable, but is not synchronized with the original DPF model.</td>
</tr>
<tr>
<td>Constraints</td>
<td>Constraints are still editable and checked in the new model.</td>
</tr>
<tr>
<td>Flexible</td>
<td>This solution only offers one-to-one visual alteration of elements, this makes abstraction and grouping impossible.</td>
</tr>
<tr>
<td>Extendable</td>
<td>Because this solution is so tightly bound to the existing DPF Editor extending it with new features can easily lead to errors in unexpected places, and even break basic functionality in the DPF Editor.</td>
</tr>
<tr>
<td>Syntax quality</td>
<td>Achieving a good quality concrete syntax would be more difficult without functions like abstractions and grouping.</td>
</tr>
</tbody>
</table>

Table 4.3: The wizard solution tested against the specification.

In table 4.3 above the wizard solution is tested against the specification from table 4.1. As can be seen in the table it is lacking in several key areas.
With all these shortcomings the wizard solution is a suboptimal solution to our problem. Reducing model complexity is a major part of the problem, and just altering the appearance of one and one element separately will not reduce model complexity sufficiently. A proper solution should be able to do some abstraction, reducing several elements into one figure, and making elements able to appear inside other elements.

We stopped implementing this solution after deciding that the model-mapping solution would be a better fit by offering more flexibility. The partial solution that was implemented is demonstrated in the section below.

### 4.2.2 Wizard Demonstration

This section contains a short and simple demonstration of the use of the Eclipse Wizard implementation, it explains in more detail, and with pictures, the process that is listed in table 4.2.

![Figure 4.3: The original DPF-Model.](image)

Figure 4.3 above shows a DPF-Model with one class which has an attribute and a method attached. This model will be used now to demonstrate the Eclipse Wizard implementation.
The first page of the wizard is a basic new file page where the user can specify where the new model should be stored and give it a name.

On the second page of the wizard the user selects a DPF-Model to load and make a visually altered copy of. For this demonstration we here select the original DPF-Model as shown in figure 4.3.
The third and final page of the wizard lets the user select a figure for each element in the original model from a drop-down box with a list of predefined hard-coded figures. Here we choose the Class figure for the element named class, to make all the class elements in the copy be visualised like classes are in UML diagrams.

The resulting copy will then basically be the same model as the original, but with other visualisations for the classes. Because this implementation was cancelled it can not change the visualisation much. The plan was to make it possible to choose visualisations for the arrows and make elements capable of containing other elements.

**Figure 4.6:** The third page, choose figures for each element.

**Figure 4.7:** The final visually altered copy.
Chapter 5

The Model Mapping Approach

This chapter deals with the model mapping approach which we ended up choosing over the Eclipse Wizard approach. It explains how it is works, then details in its implementation and why we ended up choosing it.

5.1 Solution

The model mapping approach requires the user to make a new kind of model, a visual metamodel in which all the new visual data will be specified. This visual metamodel can store elements with attributes like colour, shape and if the element is a container.

In the current implementation the visual metamodel only keeps a few attributes for each element. The first attribute is boolean saying if the element is composite or not and if an arrow is composed. If a node is composite it should contain any node that is connected with a composed arrow. The two other currently implemented attributes are background colour and rounded corners, this allows for some basic alterations to the concrete syntax. The implementation can easily be extended to include more attributes like layout of contained elements, icons for use in the palette, and more.
After the visual metamodel has been made the user can create a new visualisation model. The visualisation model wizard takes a DPF metamodel and a visual metamodel as input and creates a new DPF model that uses the selected DPF metamodel as typegraph, it also creates the visualisation model which contains the mappings between elements in the DPF Metamodel and the visual metamodel. These mappings are specified in the wizard by the user. After the wizard is done the output is a normal DPF model with a dpf- and a xmi-file, and a visualisation model containing the mappings.
The visualisation model contains a reference to the visual metamodel and a reference to the DPF model. It is then possible to edit the DPF model as a normal DPF model, or edit it through the visualisation model which can have different figures and behaviour. Because the visualisation model contains a reference to the DPF model, instead of a copy of its data, any changes done to one of the models will show up in the other model as well.

Constraints will currently not show up in the visualisation model, but can be added, edited and validated as normal by opening the DPF model instead of the visualisation model. In the future it might be beneficial to add different ways of visualising the constraints on the visualisation model as well. If any of the constraints are not satisfied it should be visualised in a way that let the user know its not satisfied [16].
With the current implementation the user can specify that a certain node should be composite, a composite node can contain other nodes. The arrows connected to that node can then be specified to be composed. In the instance model any node on the other side of a composed arrow can then be added inside the corresponding composite node. This makes it possible to create classes with methods and attributes shown inside it, similar to how UML visualises classes in class diagrams.

In the present version of the visualisation editor it is not currently possible to nest composite nodes, a composite node created inside another composite node can not contain any child-nodes. This is a feature that is needed after the possibility of selecting a different internal layout has been added. Currently all composite nodes use a list as internal layout, to support modeling abstraction-levels and figures like packages as they are in UML more internal layouts have to be available.

It is also possible to specify background colours of nodes and specify that they should have rounded corners. These are just the first functionalities that were added, the solution is relatively easy to extend with more capabilities like abstraction levels, and new shapes for arrows and nodes
5.2 Implementation and Development

The DPF Core project was created as a plugin project for the Eclipse Modeling Project with GEF and EMF as core foundation. The DPF Editor was then built as another plugin project which is using the DPF Core project as a base [8]. The visualisation project described in this thesis was also made as an Eclipse Modeling plugin project with both the DPF Editor and the Core Project as vital components.

Both EMF and GEF were briefly mentioned in chapter 3.2.2. EMF is used to define abstract syntaxes while GEF is used to define concrete syntaxes, together these two technologies allow for the generation of basic model editors as Eclipse plugins.

The visualisation editor was partially generated using EMF metamodels Ecore and Genmodel, this gives a foundation of Java code with model classes and a general visual editor [31]. Because the DPF Core and Editor projects have such Ecore and Genmodel models as well it was relatively easy to link the projects together.

Ecore is an implementation of the OMGs Essential MOF (EMOF) technology, which is mentioned in chapter 3.2.1.2. It lets a user define a metamodel for an Eclipse editor plugin. The Genmodel can then be generated from the Ecore model, this will contain additional information about the code generation.
Some of the generated classes can be seen in figures 5.1 and 5.3 above, in the section explaining the solution. The latter figure is for the visualisation editor, each visual element in a model has an editpart which handles interactions. This editpart is also responsible for drawing the elements figure, different policies can specify how the editpart should handle interactions. One policy is for example made to handle how a composite node handles another node being dropped inside it. The editparts for each element can be made by the user from the palette tool on the right side in the editor.

Figure 5.5: The Ecore models for the Visual Metamodel editor and the Visualisation Editor.

As can be seen in figure 5.5 the visualisation editor, which is on the left, is dependant on the visual editor, the DPF Core, and the DPF Diagram metamodels. The "Model-ToVisualEntry" object contains the mapping between elements in the visual metamodel and elements in the DPF metamodel. The "VCompartment" objects are added to a node if it is specified to be composite, each compartment contains one type of nodes. Each "VCompartmentElement" object acts as a wrapper for a node that is contained inside a composite node.

5.2.1 Development Method

The work in this thesis was done using Model-Driven Engineering which is explained in chapter 2.2. Model-Driven Engineering focuses on models as artefacts and lets the
developers use models to generate some code for the project, models are also useful to get an abstract view of the solution.

Because the DPF team had meetings every other week during the development of this project it became natural to attempt to have something new to show for each meeting. The meetings therefore ended up marking the transition to a new iteration in the development. During the meetings what had been done was reviewed by the other members of the DPF team and suggestions for improvements were made. The course for the next weeks was also decided.

Although Model-Driven Engineering was employed, the development process used in this project can be said to be a loose version of Scrum. These two processes can be used together in order to increase productivity and the quality of the work [32]. Scrum is a development process that is based on sprints, a sprint is an iteration of work lasting no longer than a month, a goal is set and the work is split into pieces. The members of the development team then choose pieces of the work that they want. Daily Scrum meetings are held to synchronize the developers and plan ahead, the next sprint will start right after the current one finishes. Scrum is a very lightweight development process that is relatively easy and fast to use.

In this project each sprint lasted two weeks and the work that was done was presented at the DPF meetings. Because there was only one member in the development team there was no need for daily scrum meetings.

All the projects related to DPF are available from an open SVN repository hosted by Bergen University College, anyone can check out the projects to test them out. To keep the projects secure an account is required in order to commit any changes to the repository.

5.2.2 Packages

The package names in the visualisation project use the same base as the rest of the DPF Workbench, that is no.hib.dpf. To separate the code for the visual model and the code for the visualisation model they use their respective names as subproject names.
• **no.hib.dpf.visual** and **no.hib.dpf.visualization**
  Packages with class interfaces, utilities and factories for creating java classes.

• **no.hib.dpf.visual.impl** and **no.hib.dpf.visualization.impl**
  These packages contain the implementations for the various class interfaces defined in model.

• **no.hib.dpf.visual.provider** and **no.hib.dpf.visualization.provider**
  In these packages there are item provider adapters for each class.

• **no.hib.dpf.visual.tests** and **no.hib.dpf.visualization.tests**
  Packages which contains all the testclasses associated with the projects.

• **no.hib.dpf.visual.util** and **no.hib.dpf.visualization.util**
  The AdapterFactories. The util package for the visualisation editor also contains some additional utilities for loading and saving models.

• **no.hib.dpf.visual.presentation**
  Contains the wizard used to create a visual model as well as some factories used by the editor.

• **no.hib.dpf.visualization.presentation**
  Like its counterpart for the visual model this contains the wizard for creating a visualisation model. It also contains the palette factory which is responsible for populating the palette in the editor view. The editpart factory and all the editparts are in this package as well. Each editpart corresponds to an element in the editor, they are responsible for setting up the policies for user interaction with the element and creating the visual figure.

• **no.hib.dpf.visualization.commands**
  This package contains the command which executed when creating an element inside another, it adds a node in the underlying graph and connects it with an arrow so that the added element is apparent when the model is opened in the DPF Editor as well.
• **no.hib.dpf.visualization.figures**
  Contains all the figures used to render the special container-elements.

• **no.hib.dpf.visualization.policies**
  Contains policies for editing and laying out editparts correctly.

### 5.2.3 Implementation

When linking a visual metamodel and a DPF metamodel through the wizard a new DPF model is made which uses the metamodel selected in the wizard as typegraph. In addition a visualisation model is made which contains the mapping between the elements in the DPF metamodel and the visual metamodel, this mapping is stored as an EMap structure. An EMap object consists of a list of maps, each map contains a key and a value [33]. In our implementation the key is an element from the DPF metamodel and the value is the element in the visual metamodel that the user chose during the wizard.

In the implementation this object can be used by giving it any element from the DPF metamodel, it will then return the corresponding element from the visual metamodel.

The visualisation model uses the same storage format as the DPF model and the visual metamodel, the XMI format [34]. XMI is an application of XML, which is a markup language that is readable both by humans and computers.

In the visualisation editor the EMap mapping object is available and can be used to find the visual properties of any element in the typegraph. An example of using this object is included in listing 5.1 below. The maps object is used in the node editpart in order to use the right policies and figure when the associated node is of a composite type.

```java
/**
 * Checks if a node has any incoming composed arrows
 */
private boolean hasVArrowTargetTo(Node node, EMap<IDObject, VElement> maps){
    // Iterates all incoming arrows
    for(Arrow arrow : node.getIncomings()){
        // Fetches the arrows corresponding visual element
        VElement aElement = maps.get(arrow);
        // Checks if this visual element is composed
    }
}
```
if(aElement instanceof VArrow && ((VArrow)aElement).isComposed())
    return true;
return false;
}

Listing 5.1: Example of using the EMap object to get a visual property.

The editpart for a node that can contain other nodes will use a different figure which adds a compartment figure with a content pane for each type of element it can contain. It also uses a special policy which adds the elements inside when they are dropped in it from the palette as well as handling the action for renaming these contained elements. The code for this editpart can be found in appendix B.

The policy used by the container-nodes works by executing a command when it detects that a node is being dropped on it from the palette, this command then adds the node and an arrow in the abstract model and adds the node wrapped inside a compartmentelement to a compartment. The compartmentelement then gets an editpart which just adds the figure for the node inside the container-node.

All the compartmentelement and compartment objects are transient. They are not stored because they can be deduced from the abstract model when loading it by checking for composite nodes and composed arrows. All nodes connected to a composite node by a composed arrow will then be created as compartmentelements inside compartments instead of as normal nodes upon loading a model.

5.3 Properties

The visualisation implementation might be a bit more complex to use than the Eclipse Wizard implementation, but it makes up for that with all the additional useful properties it has.

Unlike the Eclipse Wizard implementation this implementation offers a lot of flexibility, the visual models can be extended to let the user specify any attribute one could
want. Attributes for border type, shape and internal layout might be some of the most interesting ones to add.

As the visual model is stored as a file it can also be reused if two models contain many of the same elements. While in the Eclipse Wizard implementation all the information given throughout the wizard will be lost when the wizard is finished.

This implementation also avoids the problem the Eclipse Wizard implementation has with two DPF models using the same typegraph. Unlike the Wizard Implementation this doesn’t edit the typegraph at all, but since the visualisation is just a layer on top of a DPF model there will not be a conflicting problem even if the typegraph is directly edited.

## 5.4 Demonstration

This section contains a demonstration of how one can use the model mapping implementation to create a simple class diagram.

![Figure 5.6: The metamodel for a class diagram.](image)

Shown above in figure 5.6 is the metamodel for the class diagram, this will be the typegraph of the class visualisation model. In the metamodel a class can have both attributes and methods. The Simplenode and Simplearrow have been added just for demonstration purposes.
The visual metamodel only contains the minimum attributes in the current implementation, but can easily be extended to contain more attributes like colour and shape. As seen in figure 5.7 it now only contains the composite attribute and an attribute for choosing a custom icon. All the elements in the visual metamodel are made manually by the user, they are not yet in any way linked to the elements in the DPF metamodel.

When creating a new visualisation model the user chooses both a DPF metamodel as typegraph and a visual metamodel. On the next page of the wizard the elements in the DPF metamodel are linked to the elements in the visual metamodel by the user. The
user highlights an element from the DPF metamodel on the left side, and can choose from a list of elements in the visual metamodel on the right side, see figure 5.8.

![Figure 5.9: A new visualisation model with one class.](image)

After pressing finish on the wizard three new files will be created, the two standard files for a DPF model and one file for the visualisation model. The visualisation model will be opened automatically and the user can begin to make the class diagram. Figure 5.9 shows the new visualisation model after the user has created a composite node by selecting "Class" in the palette and clicking the model to make one. As can be seen the visualisation model has a different figure for elements that are of the composite type, it has also added a compartment for each of the type of elements it can contain.

![Figure 5.10: Attributes and methods can be added to the class.](image)

Figure 5.10 demonstrates how attribute and method elements are added and visualised inside the class elements.
The model can be opened as a normal DPF model as well to see the abstract model, when adding a new method or attribute to a class in the visualisation model it will be connected to the class with an arrow in the abstract model as shown in figure 5.11, these arrows will have the name "childOf" by default.

A model can consist of several composite elements with different composed elements inside them, it can also contain normal DPF nodes. In figure 5.12 a car class with methods and attributes, and a simplenode has been added to the model. In figure 5.13 below the updated DPF model can be seen, the new added elements will be there as well.
Figure 5.13: The updated normal DPF model.
Chapter 6

Conclusion

This is the concluding chapter which gives a summary of what was achieved through this project. We will also propose what can be done in the future to improve on the currently implemented solution.

6.1 Summary

As the DPF Workbench reaches a more mature state it will be natural to start implementing functions that improve properties like usability. With the newly added prototype for generating code from models, and the signature editor for defining arbitrary signatures the DPF Worbench is becoming more useful than ever.

It is absolutely vital to add functionality for defining arbitrary concrete syntaxes to the DPF Editor. Having to closely study the model in order to see what type each element has is not a good solution. A well defined concrete syntax can make both simple and complex models much easier and quicker to understand.

Through the work on this thesis a solid foundation for creating concrete syntaxes was made. The foundation offers basic functionality, like linking an element in a metamodel to an entry in a visual metamodel, thereby defining how elements which are typed by that element should be visualised. Models that are made through this visualisation editor
will effectively have two concrete syntaxes, the one defined by the user and the default DPF Editor syntax.

Currently the visualisation editor only offers the possibility of defining that some elements should be visualised inside other elements, the same way as attributes and methods usually are visualised inside class-elements in UML class diagrams. It also supports assigning background colour to nodes and rounding of the corners. The functionality for defining for example shapes and borders for each element should be relatively straightforward to add to the foundation that has been made.

### 6.2 Future Work

The implemented Visualisation Editor still has a way to go in terms of offering the user complete flexibility. In this project we focused on implementing a foundation, and providing some of the most basic functionality. Adding the ability to place some elements inside other elements was considered to be one of the most complex problems faced when implementing a way to define concrete syntaxes.

**Visualisation Editor**

With the foundation in place it should be relatively easy to add a way for the user to specify how the contained elements should be laid out inside the containing element. The Draw2D framework which is used to draw all the figures has several layout managers to choose from, but only one was properly implemented in this project for lack of time. More layouts would make it possible to model things like class diagram packages. The current implementation also supports defining background colours for nodes, and choosing if their corners should be rounded.

Other properties for each element like for example border colour and different border shapes can easily be added to the visual model. Letting the user specify these attributes will be a simple addition, but can prove very effective towards being able to define concrete syntaxes.
Grouping
There are some bigger features as well that should be added in the future like abstraction. This could for example visualise a group of DPF elements in the model with just one figure. One could then click on this figure to see the internal structure and the contained elements. This feature is particularly useful when certain structures are repeated over and over in the model.

Symbol Editor
We propose that in the long run it would be beneficial to implement a symbol editor, much like the one Metaedit+ has. This would allow for the user to draw any figure, and choose where and how the name and type should be displayed. Combined with the visual model it would be easy to link the created figures to the right elements in the DPF-Metamodel.

6.3 Last words
Working on this thesis has resulted in a foundation for defining arbitrary concrete syntaxes for the DPF Editor with some basic functionality. Having the possibility to create and customise concrete syntaxes is very important in order to improve the usability of the DPF Workbench.
Appendix A

The DPF Editor Tutorial

This tutorial was made by following an example found in an article written by Yngve Lamo, Xiaoling Wang, Florian Mantz, Øyvind Bech and Adrian Rutle about the DPF Editor, this article can be found in the references below A.6.

The Diagram Predicate Framework (DPF) Model Editor is a graphical tool for the specification of (meta)models, the tool also has functionality for creating domain specific editors based on the metamodels. It is a part of the DPF Workbench which features the model editor, a signature editor and functionality for code generation. One of the key functionalities of DPF Model Editor is that you can have virtually unlimited levels of metamodels. It can also specify arbitrary constraints on any subgraph of the specification; unary, binary and ternary... DPF has proven useful when formalizing some MDE concepts like metamodelling, transformation of models and version control of models. The DPF Workbench is made as an add-on for the Eclipse Modeling IDE.

A.1 Installing the DPF Workbench

If you are not yet familiar with the Eclipse Modeling IDE you can find a good tutorial here: http://www.vogella.de/articles/Eclipse/article.html
To start using the DPF Model Editor you need the Juno version of the Eclipse Modeling Tools which can be found on the Eclipse homepage and the DPF Workbench plugin which can be found on http://dpf.hib.no/downloads/. To install it, click Help and Install New Software... in Eclipse, add http://dpf.hib.no/editor/update as a software repository and choose to install the DPF Workbench from there.

A.2 Using the DPF Model Editor

In this tutorial we will define a domain model for modelling the workflow for treatment of cancer related pain. As this is a big and complex process we will only make the two levels of the metamodelling hierarchy and a small part of the instance model. Further explanation of this model can be found in the paper DPF Workbench: A Diagrammatic Multi-Layer Domain Specific (Meta-) Modeling Environment.

After having installed the DPF Workbench plugin for Eclipse you can create a new DPF project by pressing “Other...” in the “New” dialog in the “File” option of Eclipse. In the window that now pops up, you choose to create a new Diagram Predicate Framework Project as shown on figure A.1 below.
When your project has been created you use the same procedure again to add a DPF Specification Diagram to the specifications folder in your project.

### A.3 The first metamodel level

In this tutorial we will make a metamodel hierarchy in order to specify a workflow for the treatment of cancer related pain. This first metamodel will define some basic structures and constraints for the next metamodel.

Start by using the method above to create the project and make a DPF Specification Diagram called `process_m3.dpf` in your specifications folder. The `m3` at the end of the diagram name is to easily see which level this metamodel is at.

You start out with the elements Node and Arrow by default when you make a new model in the DPF Model Editor, as seen in figure A.2 and A.3.
In your newly created diagram you now add two nodes named “Element” and “Control”. The element node will be used to type the main elements while the Control node will be used to control the flow.

You then create an arrow from element to element by selecting arrow and click element twice, name this arrow “Flow”. Create three more arrows, one from Control to Control named “NextControl”, another from Element to Control named ControlIn, and the third from Control to Element named “ControlOut”.

![Model view with the signature bar highlighted.](image)

**Figure A.2:** Model view with the signature bar highlighted.

We now add constraints to the model, select the arrow named NextControl and hold down Ctrl to select the ControlIn arrow as well. Now with both arrows selected press the [jointly-surjective] constraint on the signature bar. This will make these two arrows [jointly-surjective] which assures that a control must have at least one NextControl or ControlIn connected to it.

Select the ControlOut arrow and the NextControl arrow and apply the [xor] constraint to them. This makes sure that every Control node has exactly one NextControl or one ControlOut arrow attached to it. Also apply the [multiplicity] constraint to the ControlOut arrow and make sure it says [0..1], if it doesn’t you may change it in the properties window while having the appropriate arrow selected as seen in figure A.3 below.
A.4 The second metamodel level

This metamodel will define more specific structures and constraints for our next and final model using the structures and constraints we defined in the first metamodel. You start by creating a new DPF Specification Model again, name it process_m2.dpf and choose process_m3.dpf.xmi as type graph as shown in figure A.4 below.
Figure A.4: Load a typegraph for the new model.

This should leave you with a new empty model and the types you specified in process_m3 should be visible in the palette on the right, see figure A.5.

In this new model you start by creating an Element named “Activity” and two Controls named “Choice” and “Condition”. Next you make two flows from Activity and back to itself, name one “Sequence” and the other “Message”, give both of them the [irreflexive] constraint. The [irreflexive] constraint makes sure no Activity can have a Sequence or Message pointing back at itself.

Add a NextControl from Choice to Condition and name it “ChoiceCondition”, apply the [surjective] and the [multiplicity] constraint [1..*] to it. Also add a ControlOut from Condition to Activity with the [injective] constraint and same [multiplicity] constraint as ChoiceCondition, name it “ChoiceOut”.

Lastly, create a ControlIn from Activity to Choice which you name “ChoiceIn”. Give it the [injective], the [surjective] and [multiplicity] [1..*] constraints. Add a [nand]
constraint between this and Sequence. Your model should now look somewhat like in figure A.5 below.

Figure A.5: The complete process_m2 model.

On Figure A.6 below you can see a model which would fail the [xor] constraint on NextControl and ControlOut because of the highlighted arrow.

Figure A.6: The highlighted arrow will make the validation fail.
A.5 The model level

You must now create a third model with the process_m2.dpf.xmi as type graph. Start this model by adding an activity named “Pain Assessment” and a Choice named “Currently on Regular Opioid?”. Add a ChoiceIn from Pain Assessment to Currently on Regular Opioid.

Next you need two conditions named “Yes” and “No”, connect these to the Choice you just made by making one ChoiceCondition to each one. Select these two ChoiceConditions and apply the xor constraint indicating that only one of them can be chosen.

Add an Activity named “Gather Opioid Information” and connect it with a ChoiceOut from Yes and an Activity named “Measure Pain Intensity” connected with a ChoiceOut from No. Your model should now look like figure A.7 below, this tutorial stops explaining here, but the full model of the workflow for treatment of cancer related pain is visible below in figure A.8.

![Figure A.7: The partial model.](image)
The full model in figure A.8 above is clearly very big and can be confusing to look at. While the visualisation editor that has been made in this thesis does not have a lot of functions yet, it can help visualise this model better. In figure A.9 below is the same model viewed in the visualisation editor with different background colours on the elements.
Figure A.9: The full model, viewed in the new visualisation editor.

A.6 References


Appendix B

The DPFNodeEditPart

package no.hib.dpf.visualization.presentation;

// Imports left out to improve readability

public class DPFNodeEditPart extends DNodeEditPart {
    VNode visual = null;
    EList<VCompartment> compartments = null;
    EMap<IDObject, VElement> maps;

    // Constructor for an Editpart for a node
    public DPFNodeEditPart(VNode vElement, EList<VCompartment> nodeCompartments,
                            EMap<IDObject, VElement> maps) {
        visual = vElement;
        compartments = nodeCompartments;
        this.maps = maps;
    }

    // Adding policies to the editpart, these handle interactions
    protected void createEditPolicies() {
        if(visual != null && visual.isComposite())
            installEditPolicy(EditPolicy.LAYOUT_ROLE, new VNodeLayoutPolicy());
        installEditPolicy(EditPolicy.COMPONENT_ROLE, new NodeComponentEditPolicy());
        // and the reconnection of connections between Shape instances
    }
}

56
installEditPolicy(EditPolicy.GRAPHICAL_NODE_ROLE, new DArowCreatePolicy());
installEditPolicy(EditPolicy.DIRECT_EDIT_ROLE, new NameDirectEditPolicy());
}

// Creates the figure for the user to see
protected IFigure createFigure() {
    IFigure figure;
    // Create a figure that can contain other nodes if this is composite
    if(visual != null && visual.isComposite()) {
        figure = new CompositeNodeFigure(new EditableLabel(getNodeLabelName()));
    } else
        figure = new NodeFigure(new EditableLabel(getNodeLabelName()));
    return figure;
}

// Returns the pane in which child nodes should be created when added
public IFigure getContentPane(){
    if(visual != null && visual.isComposite())
        return ((CompositeNodeFigure)figure).getContentPane();
    return super.getContentPane();
}

// Constructs the right name-label for the figure (name : typenode)
private String getNodeLabelName() {
    String result = "";
    if(getDPFNode().getName() != null)
        result += getDPFNode().getName();
    Node type = getType();
    if (type != null && type.getName() != null)
        result += " : " + type.getName();
    return result;
}

// Functionality for editing the name of the node
@Override
public void performRequest(Request req) {
    if (req.getType().equals(RequestConstants.REQ_DIRECT_EDIT)) {
        TextDirectEditManager manager = new TextDirectEditManager(this,
                      TextCellEditor.class, new TextCellEditorLocator(          
                          ((NodeFigure)figure).getNameLabel()));
        manager.show();
        return;
    }
    super.performRequest(req);
}

// Refreshing the visualisation of the name of the node
private void refreshLabel() {

NodeFigure tableFigure = (NodeFigure) getFigure();
EditableLabel label = tableFigure.getNameLabel();
label.setText(getNodeLabelName());
label.setVisible(true);
label.revalidate();
}

// Refreshing the visualisation of the whole node
protected void refreshVisuals() {
    NodeFigure figure = (NodeFigure) getFigure();
    DPFEdition editor = getEditor();
    if (editor != null){
        boolean flag = editor.isMakerExisting(getDNode().getNode());
        if (figure.getErrorImageFlag() != flag)
            figure.setErrorImageFlag(flag);
    }
    getFigure().setBounds(new Rectangle(getDiagramModel().getLocation(),
                                            getDiagramModel().getSize()));
    refreshLabel();
}

// Fetch the inner nodes (always empty if not composite)
@override
protected List<VCompartment> getModelChildren() {
    return compartments;
}

// Hide connections to contained nodes
@override
@SuppressWarnings({"unchecked", "rawtypes"})
protected List<?> getModelSourceConnections() {
    EList sources = new BasicEList();
    sources.addAll(getDNode().getDOutgoings());
    sources.addAll(getDNode().getConstraintsFrom());

    for (DArrow arrow : getDNode().getDOutgoings()) {
        VElement aElement = maps.get(arrow.getArrow().getTypeArrow());
        if(aElement instanceof VArrow && ((VArrow)aElement).isComposed())
            sources.remove(arrow);
    }

    return sources;
}

LISTING B.1: The DPFNodeEditPart.
Bibliography


[28] Uml superstructure specification. URL http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF.


[34] Xmi specification. URL http://www.omg.org/spec/XMI/2.4.1/PDF.