A Visual Editor for Semantics Specifications
Using the Eclipse Graphical Modeling Framework

Bachelor Thesis

Malte Röhs

University of Paderborn
References
List of Figures

2.1 Class diagram of the abstract syntax of DMM ............... 5
2.2 An example DMM rule ....................................... 6
2.3 Overview of the Eclipse architecture ........................ 8
2.4 The four layers defined by MOF ............................. 10
2.5 GEF communication chain ................................. 14
2.6 Correlation between .ecore, .gmfgraph and .gmfmap ........ 15
2.7 The EMF and GMF models and their relations ............... 16
3.1 Correlation of the GMF models of the DMM editor .......... 22
3.2 Correlation between ruleset files .gmfmap, .gmftool, .gmfgraph and .ecore ........................................ 25
3.3 GMF models of the DMM editor after solving the abstract Rule problem ........................................... 29
3.4 Ruleset editor without customized code ...................... 30
3.5 Rule editor without customized code ........................ 31
3.6 Ruleset editor with customized code ........................ 32
3.7 Rule editor with customized code ............................ 33
4.1 Screenshot of our DMM Editor .............................. 40
Chapter 1

Introduction

*Model Driven Development (MDD)* founds on the idea of generating models of the desired software instead of directly writing program code. From these models the code for the specified software can be automatically generated by using generators. Changes in the software are applied on the model and the code will just be re-generated. So in the best case, the developer does not have to write a single line of code.

This leads to several advantages: Now people who are involved in the development process but are not familiar with programming (like future users or specialized staff from the application range of the software) can more easily understand and influence the development process because models are an intuitive presentation. By using different generators to generate the code from the models the software can easily be translated to different platforms and the developer does not need to have a detailed knowledge of the used target programming language. As mentioned before, changes only need to be applied on the model. But the most crucial benefit probably is that the code does not have to be written by hand. Most software projects need basic functionalities like graphical user interfaces and data management functions which can mainly be generated automatically. Another point is that generated code normally has a good quality, because the same code is used by a high number of developers and errors are quickly detected, reported and corrected.

To employ a model driven approach, the semantic of the model needs to be unambiguously specified. This unambiguousness is necessary to avoid different interpretations of the semantic meanings of model elements. The semantic of a model can be split into the structure (the static semantic) and the behavior (the dynamic semantic). The description of the static semantic of a model can be done with meta models (using e.g. class diagrams). But
for the description of the dynamic semantic there is no standard technique.

The need for a precise and easily understandable technique for a dy-
namic semantic description has led to the development of the Dynamic Meta
Modeling (DMM) technique, which was introduced in [EHHS00]. DMM de-
scribes the behavior of a model in a visual way.

This leads to the topic of this bachelor thesis: the model driven develop-
ment of a visual editor for DMM using Eclipse.

A requirement for the editor is that it shall be implemented as a rich
graphical editor via an Eclipse plug-in. One reason is, that the editor will be
part of a bigger chain which is also implemented as Eclipse plug-ins. This
chain leads from DMM to a Groove ([Ren04]) model. The idea is to visually
create a DMM ruleset which can be transformed into a Groove model (see
[Sto08]) and then be used as input for model checking tools. Another reason
for using Eclipse is that a rich graphical editor provides a lot of basic needs as
menus or toolbars and benefits like zooming functions. Also Eclipse is a open
source software which makes it in general interesting for research projects.

Furthermore this thesis points out, up to which point a visual editor can
be developed purely model driven (using Eclipse) and for which functionali-
ties customized code is needed. In the case that we have to write customized
code we want to separate the generated from the customized code to avoid
interferences between re-generated and hand written code.

This thesis is structured as follows: Chapter 2 will explain the basic foun-
dations needed to understand the development of the DMM editor and the
decisions which were made. It starts with an introduction to DMM. After-
wards the Eclipse architecture is explained. The rest of chapter 2 considers
the used Eclipse projects EMF, GEF and GMF. Chapter 3 describes the
implementation of the editor. Therefore it first defines the requirements for
the editor and then covers the realization by considering the different models
and the customized code. Chapter 4 contains the conclusion of this thesis
along with remarks on possible future works on the editor.
Chapter 2

Fundamentals

This chapter discusses the fundamentals needed to understand the construction of the editor and the decisions that were made. First it describes the basic idea of DMM and shows up a short example. Then Eclipse will be described, starting with a few words about the Eclipse Foundation and then examining the architecture of Eclipse. After the introduction of Eclipse, the for this work important Eclipse projects, the Eclipse Modeling Framework (EMF), the Graphical Editing Framework (GEF) and the Graphical Modeling Framework (GMF), are presented. Reader who are familiar with this issues can continue reading in chapter 3.

2.1 Dynamic Meta Modeling

As DMM is the semantic specification language for the constructed visual editor, it is useful to have a closer look at it and especially at the syntax of DMM, which is needed to develop the editor.

As mentioned before, models are a very good concept to describe developed software in an intuitive way. This is a reason why Visual Modeling Languages (VMLs) are widely used in the software development process. The probably most known and used VML is the Unified Modeling Language (UML), which was designed and standardized by the Object Management Group (OMG). But UML is not the only VML in use. In many cases there is the need for a Domain Specific Language (DSL), which is only used in a particular domain.

This leads to the design process of VMLs and therefore to *meta modeling*: The creation of models which describe a Visual Modeling Language. The meta model is useful to guarantee an unambiguous interpretation of the
VML. With an unambiguous meta model, tools can be created for validating models of the VML.

The description of the static semantic of a VML is usually done by means of a class diagram. But for the dynamic semantic of a VML there is no standard by now. The dynamic semantic of the widely used UML, for instance, is described textual on more than 800 pages. The usage of a natural language, which is unfortunately not unambiguous, leads to ambiguity and therefore to inconsistency and different interpretations of model elements. These different interpretations of the semantic also lead to problems when creating a code generator or model analyzer, because different interpretations cause different implementations which are not interoperable.

So there is the need of a description technique for dynamic semantics, which is on the one hand precise and formal (so automatic analyzing techniques can be used) and on the other hand understandable (so human beings can easily understand it). DMM tries to close this gap. It is a technique for the specification of the semantic of a VML. For this purpose it combines two different approaches:

- Denotational Meta Modeling to describe the static semantic
- Graph transformation rules to define the dynamic semantic.

Denotational Meta Modeling means that the syntactic meta model of the VML is mapped into the meta model of the semantic domain. For instance the syntactic element class is mapped on the semantic concept object and the attributes of a class are mapped to slots in the object, which can hold actual values [EHHS00].

The graph transformation rules are a set of operational rules which describe how the semantic meta model changes in time. An example for one of these operational rules is shown in figure 2.2 and will be discussed later.

The abstract syntax according to which the rules may be created is shown in the class diagram in figure 2.1. Let's have a closer look at it, because it contains the rules that describe which elements the desired editor needs to be able to create and which relations between these elements are allowed. The class diagram consists of the following classes:

- Ruleset
- Rule (BigstepRule, SmallstepRule, PremiseRule)
- Node
2.1. Dynamic Meta Modeling

- Edge
- Invocation
- NamedElement
- GraphElement

Figure 2.1: Class diagram of the abstract syntax of DMM

The whole diagram describes a **Ruleset**. A **Ruleset** consists of one or several **Rules**, where **Rule** is an abstract class and therefore an instance of **Ruleset** is composed of **bigstepRules**, **SmallstepRules** or **PremiseRules**. A Rule can overwrite another Rule (of the same sub class) using the **overwrittenRules** association.

A **Rule** consists of **Nodes**, **Edges** which connect the **Nodes**, and **Invocations** which are applied on **Nodes**. A Rule always has one explicit **contextnode** and may have several **Nodes** as parameters. **Nodes** and **Edges** can be Universally Quantified Structures (UQSs). An **Invocation** always has one **targetnode** and may have several **parameter nodes**.
Chapter 2. Fundamentals

The last two classes, NamedElement and GraphElement, are just structural elements: they provide attributes to other classes, that inherit from those two.

To illustrate the semantic of a DMM rule we will have a short look at an example shown in figure 2.2 taken from [Hau05]. This figure will be picked up at the end of this thesis to be compared with a screenshot of our editor (see chapter 4).

![Diagram](image.png)

Figure 2.2: An example DMM rule [Hau05]

The meaning of the getOffer BigstepRule is the following: If the situation applies that a ForkNode with the name fork exists which has a target reference to an Edge with the name in, a first reference to an Edge named e, a source reference to a UQS Edge and it does not have a reference to an Offer and it exists an Offer with the name o that has a carries reference form the Edge named in than the carries reference from fork to o will be created and the carries reference between in and o will be deleted. Additinally the invokations spawnOffer(e) and P_canCarry(o) must apply. The names fork, o, e must be set because fork is the context node of the Rule and o and e are used as parameters for the Invocations.

2.2 Eclipse

Eclipse is a [...] “development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software across the lifecycle” [Fou08a]. The basic idea of Eclipse is that the Eclipse core provides some basic functionalities that can be extended by plug-ins. This principle is called Rich Client Platform (RCP) meaning, that a poor core is extended
2.2. Eclipse

by clients that contain the desired functionality.

Eclipse was originally developed by IBM in 2001 and is an open source project. In 2004 the Eclipse Foundation, a not-for-profit corporation, was founded and henceforward takes care of the Eclipse development and the Eclipse community. The fact, that Eclipse is open source software bestows it a high usage in research fields.

But not only research institutions use Eclipse. The most popular commercial Eclipse based applications are Lotus Notes from IBM and Flex from Adobe.

Both, the extensibility and that Eclipse is open source, are reasons why we chose Eclipse. Furthermore the existence of other Eclipse plug-ins which shall cooperate with the DMM editor cause the choice of Eclipse.

The Eclipse Foundation is hosting a lot of different Eclipse projects. The probably most known is the Java Development Tools Project providing an Integrated Development Environment (IDE) for Java. But there are many more projects, e.g. for model development, Service Oriented Architecture (SOA) or test and performance tools. The Eclipse Foundation provides these projects with newsgroups, wikis, coordinational work and more.

Since version 3.0 Eclipse is based on the Equinox framework, which is a Java implementation of the Open Services Gateway initiative (OSGi) and an Eclipse project by its own. OSGi is a hardware independent Java based service platform specification for networks which follows the SOA principals and is maintained by the OSGi Alliance, that was founded in 1999. The OSGi Alliance specifies the Application Programming Interface (API), provides a reference implementation and test suites. One of the biggest benefits of OSGi is that it provides the possibility to dynamically load or unload bundles into the system in runtime. This feature determines the architecture of Eclipse which is shown in figure 2.3. Eclipse is divided into a core and several plug-ins. Every additional functionality, like a user interface, CVS or SVN support is added to Eclipse by plug-ins [All08].

The core itself consists of a runtime environment (mainly Equinox) that provides Eclipse with the basic functionality (like the ability to load plug-ins) and the workspace, which manages all projects that a user is working on.

The Graphical User Interface (GUI) of Eclipse is based on the Standard Widget Toolkit (SWT) and JFace. SWT was developed for Eclipse by IBM in 2001 and is now maintained by the Eclipse Foundation. SWT provides an API for graphical user interfaces. Unlike other Java GUIs (e.g. Swing), SWT is using the native operating system API resulting in a better functional and visual integration of the software into the operating system environment. For
instance is the file chooser dialog of a SWT application the file chooser dialog
of the operating system with all its functionalities like setting up folder or
using the clipboard.

JFace is based on SWT and is a toolkit for graphical user interface develop-
ment in Java and “provides helper classes for developing UI features that
can be tedious to implement” [Fou].

The workbench, which is based on SWT and JFace, is the window which
Eclipse opens on start. It consists of the menu bar, the symbol bar, the
status line and several perspectives. A perspective is a set of different views
and editors.

Let’s have a second glance at the plug-ins because they are very important
for us. As mentioned before Eclipse extend its functionality by using plug-
ins. To avoid that Eclipse loads too many plug-ins it follows the lazy load
philosophy, meaning that a plug-in is only loaded when it is needed. This
leads to a separation of the declaration of a plug-in and its implementation.
All abilities and dependencies of a plug-in are written down in the Manifest
of a plug-in.

To allow the re-use and extension of plug-ins Eclipse, introduces extension
points. An extension point is an interface which can be used by extensions
from other plug-ins. A plug-in can have several extension point and also
several extensions of extension points from other plug-ins. The extensions
and extension points also need to be declared in the Manifest. This is a
great functionality which can be used to outsource customized code. The
customized code can be placed in a separate plug-in which uses the extension

\[\text{Figure 2.3: Overview of the Eclipse architecture (based on [GB04])}\]
2.2. Eclipse

points of the generated plug-in. So the code, which is generated out of models is not touched but extended through an additional plug-in. For us, this solution has the benefit, that if code is re-generated no customized code will be overwritten (see section 3.2.5) [GB04].

2.2.1 Eclipse Modeling Framework

The Eclipse Modeling Framework project is “a modeling [Java] framework and code generation facility for building tools and other applications based on a structured data model” [Fou08b].

We need EMF because GMF, which we will use to generate the visual editor, is based on EMF and GEF. Because of this, the abstract syntax of DMM was modeled using EMF (by Christian Soltenborn).

EMF consists of three main parts:

- EMF.Core
- EMF.Edit
- EMF.Codegen

**EMF.Core** The EMF.Core contains an implementation of the Essential Meta Object Facility (EMOF) (also called MOF 2.0), which was defined by the OMG. This implementation is called Ecore and so its models are named ecore models or domain models. MOF is a meta-meta language which was designed to describe meta languages in such a way that it is possible to export and import (also across networks) created models. This is ensured by the OMGs standard format for model storage: The XML Metadata Interchange (XMI), which is based on eXtensible Markup Language (XML). Since Ecore implements EMOF, every element of a model generated with EMF is serializable (into an XMI file). So it is possible to export models form Ecore to EMOF and vice versa [Groa].

MOF defines four different layers which are shown in figure 2.4. The DMM is settled in layer M2 while a ruleset, which can be created with our editor, belongs to layer M1.

EMF provides its models also with a notification function, which is useful for Model View Controller (MVC) applications. The MVC pattern suggests a separation between model, view and controller. We will have a closer look at MVC in section 2.2.2. For now it is just interesting, that if a model is changed, the controller needs to know this to inform the view, which for their
part redraws the visual representation. This is done via notifications. Every class in a domain model (or ecore model) inherits the notification ability from the base interface `EObject`.

An ecore model can be generated in different ways:

- from an XMI file (written by hand or exported from a program that also supports EMOF).
- from an XML Schema Definition (XSD).
- from annotated Java code.
- from a Rational Rose model.
- via tree editor provided by EMF.

**EMF.Edit** The EMF.Edit framework is based on the EMF.Core framework. It provides the ability to generate adapter classes that enable viewing of the model elements in the standard (JFace) viewers and property sheets. It also provides command based editing through the `EditingDomain` interface. All commands are stored in a command stack which provides undo and redo functionality. The EMF.Edit framework allows to generate a tree editor with validation functions for an Ecore model.

**EMF.Codegen** The EMF.Codegen provides the ability to generate Java code from a domain (Ecore) model. For each class in a given domain model,
2.2. Eclipse

a Java interface with the needed getter and setter methods will be generated along with an implementation and factories to create instances of the domain model classes.

So the whole generating process starts with a model in (annotated) Java code or XML, which is used to generate an Ecore model. From this domain model a domain generation model (.genmodel) can be generated. This domain generation model is like a configuration file for the generator and allows to modify the code which will be generated. From the domain generation model the model code (meaning the Java classes) can be generated, and also a tree editor for the model and adapter classes (ItemProviders) for viewing the model elements.

The interested reader can find more information about EMF in [Fou08b, MDG+04].

The next paragraph introduces the Object Constraint Language (OCL) which can be used to provide a model with conditions that influence the structural properties of the model. OCL can be used in EMF models.

Object Constraint Language

The Object Constraint Language was developed by IBM to describe rules for UML diagrams that typically describe invariant conditions. OCL is now part of UML and maintained by the OMG. OCL 2.0 was developed in parallel with UML 2.0 and Meta Object Facility (MOF) 2.0 and is capable to be used in any MOF conform meta model and not just in UML models. So OCL is platform independent. The specification of OCL can be found in [Grob].

An OCL constraint consists of four parts:

• a context
• a property
• an operation
• keywords

The context defines the element to which the constraint shall be applied. This is usually an element of the model.

The property defines some characteristics of the context. If e.g. the context is a class, the property can be an attribute or association.
The operation describes the condition. It can be e.g. an arithmetical or a set orientated operation.

Keywords are used to modify and compose operations. Keywords are e.g. not, and, or, implies, if, then, else, etc.

We need OCL to define several conditions, e.g. the following one: The association overwrittenRules is used to overwrite one or more Rules with another Rule, but a Rule that overwrites a BigstepRule must also be a BigstepRule itself. Another example is that the Rule name must not be empty. This two examples can be implemented in the following way:

- selfoclIsTypeOf(BigstepRule) and oppositeEnd.oclIsTypeOf(BigstepRule)
- self.name.size() > 0

2.2.2 Graphical Editing Framework

The Graphical Editing Framework is a framework for generating rich graphical editors out of a domain model. This means that the generated editor is an Eclipse plug-in which needs the Eclipse core to run. GEF makes no restrictions on the underlying model, it can be an EMF model, Java code, etc. GEF follows the MVC concept, meaning that there is a separation between the model, its graphical representation (view) and the program logic (controller).

The model contains the data structure (e.g. classes and references) and is used for storage. Its is helpful (but GEF does not require) that the model provides a kind of notification functionality which informs the controller after any changes on the data.

The view is the graphical representation of the model. In GEF the view is based on Draw2D. The view offers several operations to manipulate the data. This can be editing of element properties (in the property view), copy and paste of elements, drag and drop functionality, context menu items or zooming functionality. The palette is also part of the view of a GEF editor and contains buttons for the creation of new elements.

The controller keeps both model and view synchronized. If e.g. a context menu function is activated on an element, the view informs the controller about this event and the controller applies the necessary function on the model. GEF offers commands for this purpose, that additionally issue an undo and redo functionality.

GEF consists of two Eclipse plug-ins:
2.2. Eclipse

- Draw2D (org.eclipse.draw2d)
- GEF (org.eclipse.gef)

Draw2D provides a lightweight graphical system that is based on (the heavyweight) SWT. A lightweight graphical system is a system that is nested inside a heavyweight system. Draw2D manages the painting and mouse events in the view. The graphical elements in Draw2D are called figures and are treated like windows in a heavyweight system, so they have arbitrary shapes, can be transparent, can have the focus and be selected, get mouse events, etc. All figures are placed on a single canvas. Draw2D does not depend on GEF, it can be used stand-alone to create graphical views for arbitrary (Java) applications.

While the Draw2D package contains the framework for the view, the GEF package contains the framework for the controller. Its central elements are the EditParts. They are doing the mapping between an element of the model and a figure of the view. For every model class an EditPart is needed, so the class hierarchy for the EditParts is the same as for the model classes. But not only for the classes, also associations need their EditParts (a specialized one, the ConnectionEditPart) which connects the two EditParts corresponding to the classes connected by the association in the model.

The communication in GEF is done via requests. For instance a CreateRequest is used to create a new element. The communication chain of a request is shown in figure 2.5.

As seen in figure 2.5, an EditPart forwards a request to an EditPolicy. EditPolicies define what an EditPart can do, they contain the main part of the controller logic. An EditPart without its EditPolicies would just be a link between the graphical figure and the model element. [MDG*04, Fou08c]

2.2.3 Graphical Modeling Framework

By now we have introduced EMF, a framework to describe and implement meta models, and GEF, a framework to generate Eclipse based rich graphical editors from arbitrary models. Although these two components would be sufficient for implementing our editor the additional use of the Graphical Modeling Framework makes development much more comfortable and allows us to develop completely model driven up to a certain point which we would like to discuss in the next chapter.

GMF can be considered as a bridge between EMF and GEF (see figure 2.6). It generates a GEF editor from an EMF model. For this purpose
it uses features of EMF like the command infrastructure, the adapter classes
or the ItemProviders. But these benefits of EMF are bought by giving up
the model independency of GEF: GMF only accepts EMF model as input.

GMF consists of two main parts:

- a runtime environment
- a generation framework

The runtime provides some GMF specific functionality. It extends parts
of the EMF framework, e.g. clipboard support for EMF or an extension of
the EMF element types. GMF also extends the GEF framework, e.g. new
Draw2D figures are added.

The generation framework contains special editors to handle the GMF
models and a generator which produces the editor code from the GMF mod-
els. GMF uses four different models - three to create the mapping between an
(EMF) Ecore model and its (GEF) visual representation and one to generate
the editor code:

- the graphical definition model (.gmfgraph)
- the tooling definition model (.gmftool)
2.2. Eclipse

- the mapping model (.gmfmap)
- the diagram editor generation model (.gmfgen)

The graphical definition model defines the graphical representation of the domain model elements. Every element of the domain model (.ecore) which shall be displayed needs a representative in the graphical definition model. This concerns the classes which need a node entry, and also the connections, which need a connection entry. Every node and connection has a figure descriptor which defines its visual appearance using GEF figures. If several nodes shall have the same visual representation, they can share the same figure descriptor. These correlations are shown in figure 2.6.

![Diagram showing correlation between .ecore and the mapping of .gmfgraph and .gmfmap](image)

Figure 2.6: Correlation between .ecore and the mapping of .gmfgraph and .gmfmap

The tooling definition model defines the palette of the visual editor. For every node of the graphical definition model which shall be creatable with the visual editor, a creation tool is needed. The tooling definition model also defines the icon of every creation tool which is displayed in the palette of the editor. The tooling definition model is also intended to create context menu entries for the editor, but this feature is not implemented until now (GMF build id 2.0.1).

The mapping model combines our three models: the domain model, the graphical definition model and the tooling model. Whereas the graphical definition model just defines the visual appearance of an element of the domain model, the mapping model does the mapping between the graphical representation and the model (see figure 2.6). The mapping model also links the elements of the domain model to its creation tools in the tooling definition.
model. OCL constraints can also be added here. Constraints for connection can be added directly as link constraints to the link mapping, while other constraints need to be placed in audit containers. This is useful as connections shall only be placed if the corresponding conditions are achieved while other constraints may not need to avoid the creation of an element.

From the mapping model and the domain generation model of the Ecore file the diagram editor generation model can be derived. The diagram editor generation model can be considered as a configuration file for the generator (similar to the .genmodel of EMF). It allows for instance to enable or disable validation functions for the editor or the setting of the plug-in name. From the diagram editor generation model the generator creates Java code in terms of an Eclipse plug-in.

Figure 2.7: The EMF and GMF models and their relations

The relation between all this models is shown in figure 2.7. GEF provides a number of wizards to help creating (deriving) all needed models from the domain model, so the developer can focus on the customization fo the models.
Chapter 3

Implementation

This chapter describes the implementation of the DMM editor. It starts with a discussion of the requirements of the editor in section 3.1 which lead to the cognition that we need to build two cooperating editors. These two editors can be considered as two different views on the same model.

The user of our DMM editor will not notice that he or she is working with two different editors: To the outside they seem to be one editor but technically they are two separate editors that can be used stand-alone and consists of two different Eclipse plug-ins. To distinguish between these levels, we from now on mean the one editor that appears to the user if we talk about the DMM editor.

After determining the requirements for both editors, section 3.2 describes the development of the DMM editor. It first explains the four used GMF models mentioned in section 2.2.3. Thereafter it shows how the cooperation between these two editors works and then discusses a GMF specific problem that forces us to develop four editor plug-ins.

Section 3.3 mentions the open problems. These are divided into problems concerning the GMF models and those concerning the GMF runtime API. Problems concerning the DMM editor are discussed in section 4.1.

3.1 Requirements

The editor shall be able to visually create a DMM ruleset. Thereby the visual representation shall be easy to understand. As the target audience of the DMM editor will be people who are familiar with UML we want to use a visual representation that is similar to object diagrams. As this was the intention Hausmann considered in the development of his visual DMM
representation, our editor shall follow the one used in [Hau05].

A ruleset describes the dynamic semantics of a language and needs a static semantic as meta model (see section 2.1). So the editor needs to offer the functionality to load a domain model as a meta model.

The editor shall be able to save the visually created ruleset in an XML file which can be used as input for other plug-ins.

A Ruleset consists of several Rules which can overwrite each other (see figure 2.1 on page 5). A Rule itself consists of several Nodes, Edges and Invocations. To keep the overview, the content of a rule shall be displayed in a different editor window. So if a user double clicks on a Rule, a new window should open showing the Nodes, Edges and Invocations the rule consists of. So we have two different views on the same model.

Consequently we have to develop two editors: one for rulesets and one for rules. These two editors have to cooperate in such a way that the ruleset editor opens the rule editor with the corresponding rule as root element when a rule is double clicked.

Section 3.1.1 describes the requirements for the ruleset editor while section 3.1.2 describes the requirements of the rule editor. Both section are structured as follows:

1. description of creation requirements
2. description of optical requirements
3. description of constraints

As the usage of the DMM editor shall lead to one ruleset, both editors have to operate on the same file. This is described in section 3.2.3.

3.1.1 Ruleset Editor

Creation Requirements A Ruleset consists of rules that can overwrite other rules using the overwrittenRules association. So the ruleset editor must be able to create overwrittenRules associations and, as rule is an abstract class, the inheriting classes BigstepRule, SmallstepRule and PremiseRule (see figure 2.1 on page 5).

Optical Requirements All of theses three rules shall be graphically represented by a rectangle containing a label. The label displays the context node of a rule, followed by the name of the rule. Context node and name
3.1. Requirements

are separated by a dot. The name is followed by the parameter nodes of a
rule framed in parentheses. Each parameter node is displayed with its name
and type separated by a colon. The label of a BigstepRule shall end with an
asterisk behind the parentheses while the name of a PremiseRule must always
have a “P_” prefixed. The link representing an overwrittenRules shall be an
arrow as used for inheritance in class diagrams.

Constraints  Besides the graphical representation, the following constraints
shall be maintained while creating these elements: The name and the context
node of a rule must not be empty and as only a BigstepRule can overwrite
another BigstepRule (the same applies for a SmallstepRule and a PremiseRule)
the ruleset editor shall only allow overwrittenRules that follow this constraint.
Also a rule shall not overwrite itself.

3.1.2 Rule Editor

Creation Requirements  A Rule consists of Nodes, Edges and Invocations
as we can see in figure 2.1 on page 5. Thus the rule editor needs to have the
ability of creating these three classes using the palette of the editor.

Optical Requirements  An example of the desired graphical representa-
tion of a (Bigstep-)Rule taken from [Hau05] is shown in figure 2.2 on page 6.
In the following we will explain the appearance explicitly:

A label with the name and the context node of a rule is placed in the left
upper corner framed by a rectangle with a clipped corner.

A Node shall be graphically represented by a rectangle with a label. The
label shall contain the name of the Node and the type of the node. Both are
separated by a colon and underlined.

Edges can be placed between two Nodes (in the same rule) and shall
be visualized by an arrow. To visualize the navigation direction we used
different than in [Hau05] an arrow with an open head. An Edge has a label
that contains the name of the reference to which the Edge corresponds in the
meta model.

Nodes and Edges have the attribute role. A role can hold one of the
following values: exists, not exists, destroy or create. The value exists is not
visually emphasized. The literals destroy and create are attached as labels
in curly braces to the corresponding element (as it is done in UML). The
not exists value is graphically represented by a crossed dashed circle that lies
over the concerned element.
Besides the role, **Nodes** and **Edges** have the boolean attribute UQS. If it is set to true, the concerned element shall be displayed with a second shape behind it. Both attributes, role and UQS, shall be adjustable via a context menu entry.

**Invocations** shall be visualized as an arrow with a filled head which has a label at the end and is pointing to the target node of the *Invocation*. The label contains the sequence number of the *Invocation*, followed by a colon and the name of the invoked rule. Thereafter follow the parameters framed in parentheses.

### Constraints
The creation of these graphical elements shall obey the following constraints:

While the name of a **Node** can be empty it must be unique in its rule if it is set. A **Node** without a name cannot be used as a context node or as a parameter. The type of a **Node** must be set.

An **Edge** can only be placed between two **Nodes** if the types of the **Nodes** are the same (or sub types) as the types of the **Nodes** which are connected by the reference in the meta model.

While the sequence number of an **Invocation** is optional, the **invoked rule** needs to be set. The editor shall validate that an entered invoked rule exists in the **Ruleset** and that the target node of the **Invocation** has the same type (or a sub type) as the context node of the invoked rule. Also the parameters of the **Invocation** need to have the same types (or sub types) as the parameters of the invoked rule.

### 3.2 Realization

The complete development process of the DMM editor is done in Eclipse using GMF. So for both of our desired editors we will implement an Eclipse plug-in.

This section describes how the required GMF models were created. It also describes why particular design decisions were made, which problems occurred during the development process and how they were solved.

Normally GMF uses the domain model, the graphical definition model and the tooling definition model to generate the mapping model. From the mapping model the diagram editor generation model can be derived (additionally using the domain generation model). The diagram editor generation model is then used to generate the editor plug-in (the whole correlation is
3.2. Realization

shown in figure 2.7 on page 16). So we need these four models for both of our desired editors. But what is the difference between the models of the ruleset editor and the models of the rule editor?

Every graphical element of the DMM editor is mapped to a corresponding element in the domain model using the mapping model. The main difference between the ruleset and the rule editor is the mapping of the canvas (the area on which the editing/drawing takes place).

The canvas of the ruleset editor is mapped to the Ruleset class in the domain model. So all elements which are placed on the canvas are part of the Ruleset. As a Ruleset consists of Rules, it is only possible to place rules on the ruleset editor canvas.

In contrast, the canvas of the rule editor is mapped to the Rule class in the domain model. Because a Rule consists of Nodes, Edges and Invocations the rule editor only allows to place these three element types on the canvas.

Section 3.2.1 describes the different GMF models used for both editors in detail. Section 3.2.2 explains how the needed constraints which the editor shall fulfill are realized. In section 3.2.3 the technique for implementing the two different views on the model is described while section 3.2.4 covers a GMF specific problem that forces us to increase the number of used models.

Up to this point everything can be developed completely model driven. But some features will require custom code. Section 3.2.5 covers these features and their implementations. Thereby it distinguishes between features that can be implemented in a customized plug-in (using extension points of the generated code) and features that need to be placed directly into the generated code.

3.2.1 GMF Models

This section describes the needed GMF models for both editors in the following order:

1. graphical definition model
2. tooling definition model
3. mapping model
4. diagram editor generation model

The correlation of these models is shown in figure 3.1.
Graphical Definition Model

As we have seen in section 2.2.3, the graphical definition model contains an entry for every part of the domain model that shall be displayed. This can e.g. be classes that are usually represented by diagram nodes, associations that are usually represented by connections or attributes which are usually represented by labels.

For the graphical definition model the situation is a bit different than for the other models. On the one hand we could use two separate graphical definition models: one containing the visual representations of a Ruleset and one containing the visual representation of a Rule. On the other hand we could use one common graphical definition model containing both visual representations. As we prefer to minimize the number of different models (if there is no benefit using additional ones) we decided to use only one graphical definition model (the ruleset.gmfgraph).

The structure of the graphical definition model is a tree with the canvas of the editor as the root node. All the elements mentioned above are direct children of the canvas node.

As we use one graphical definition model for both editor plug-ins, it must contain all elements of the domain model which we want to display. This
leads to a diagram node for each of the following elements: BigstepRule, SmallstepRule, PremiseRule, Node, Invocation and Edge.

Connections are needed for the overwrittenRules, the Edge and the Invocation.

It may attract the attention that the Invocation has a diagram node representative and a connection representative. It would be sufficient to represent the Invocation by a label and a connection that points to the target node of the Invocation. But a connection always needs a target and a source node. So we decided to bind the label to an invisible node that can take the role of the source node of an Invocation connection.

We also need labels to display the names for each of the three rules, one label to display the name of the Node and one label to display the reference of the Edge.

The last child of the canvas node is the figure gallery. It contains the figure descriptors. Every node, connection and label has a figure descriptor. A figure descriptor can combine several elements. For instance the PremiseRule figure descriptor combines the node for the PremiseRule and the label for the PremiseRule. It also defines the visual appearance by using GEF figures.

The figure gallery also contains the decorators for connections. As we want to display the overwrittenRules as an arrow with a head similar to those of an inheritance in class diagrams, we need a special polygone decorator. We also need a polygon decorator for the Edge connection and for the Invocation connection.

Tooling Definition Model

For the tooling definition model the situation is nearly the same as mentioned for the graphical definition model. But if only one tooling definition model would be used, both editors would have all creation buttons in their palette. This would be confusing because e.g. a ruleset editor should not be able to create Nodes or Edges. In fact it is not able to create those elements as there is no mapping for Nodes or Edges between the model and the view in the ruleset mapping model. In the editor this would results in a deactivated cursor for those elements. But the palette displays every creation tool, also those which are not used in the mapping model. This could be avoided by deleting the entries for those buttons of the palette in the diagram editor generation model. But every new creation of the diagram editor generation model would overwrite these changes. So we decided to implement two different tooling definition models: one for the ruleset editor and one for the rule editor.
Chapter 3. Implementation

The tooling definition model is structured similar to the graphical definition model as a tree. The root node is the tool registry. The tool registry has a child node palette that describes the palette of the editor. The palette needs to have a creation tool child node for every element of the domain model that shall be createable with the ruleset editor.

The tool registry node can also have child nodes that create entries in the main menu or the context menu. Unfortunately the one for entries in the context menu is not working in the current GMF release (GMF build id 2.0.1, see section 3.3).

So the tooling definition model for the ruleset editor needs a creation tool for each of the classes BigstepRule, SmallstepRule and PremiseRule and one creation tool for the association overwrittenRules.

The tooling definition model for the rule editor differs only in its creation tools from the ruleset tooling definition model. It needs a creation tool for each of the three classes Node, Edge and Invocation and one for the invocation connection.

Mapping Model

The mapping model combines the domain model with the graphical definition model and the tooling definition model. It is structured like the other GMF models as a tree, with the mapping node as the root node. The mapping node can have different child nodes like a top node reference node, a link mapping node, an audit container (see section 3.2.2) node or a canvas mapping node.

We will now first describe the mapping model for the ruleset editor and then mention the differences to the rule mapping model.

As mentioned before, the corresponding classes of the elements which are directly placed on the canvas have a part-of relation to the corresponding element of the canvas. These elements are mapped to the corresponding reference in the domain model by the containment feature of the top node reference. So our Rules are mapped to the Rule reference of the domain model element Ruleset.

The correlation between the nodes of the domain mapping model and the nodes of the other models is shown (a little simplified) in figure 3.2

We need a top node reference node for each of the diagram nodes from the graphical definition model that shall be createable in our ruleset editor: one for the BigstepRule, one for the SmallstepRule and one for the PremiseRule. Every top node reference has a node mapping child node. This node does the mapping between the element in the domain model, the diagram node in
The graphical definition model and the creation tool in the tooling definition model. Each of our three rules has a label and so every node mapping node has a feature label mapping child node. The feature label mapping is responsible for the mapping of the label to the attribute of the element in the domain model. So our three feature label mappings link the label of their Rule to the name attribute of the corresponding Rule in the domain element.

The link mapping node implements the mapping between the reference in the domain model, the connection in the graphical definition model and the creation tool in the tooling definition model. Our ruleset mapping model has only one link mapping node, the one for the overwrittenRules reference.

The canvas mapping links the diagram canvas of the graphical definition model to an element of the domain model (in this case to the Ruleset) and selects the palette for the editor from the tooling definition model.
The mapping model for the rule editor is structured similar to the rolset mapping model. It needs a top node reference for the Node and one for the Invocation. Both have an according node mapping child node which has a feature label mapping node.

The second top node reference is needed because we modeled the graphical representation of the Invocation as mentioned above with an invisible node containing the label and a separate connection. From now on this connection is called invocation edge. So the domain model element Invocation is mapped to the invisible node with its label. The invocation edge is only for graphical purposes.

For the Edge and the invocation edge a link mapping node is needed. The link mapping of the invocation edge just links the graphical representation to the creation tool, there is no binding to a domain model element. The link mapping node of the Edge has a feature label mapping child node for the label.

Diagram Editor Generation Model

The diagram editor generation model is based on the mapping model and the generation model and is used to generate the Java code of the editor. It is structured as a tree with the gen editor generation node as the root node.

The root node has several child nodes that offer customization options for different parts of the editor. For instance, the gen plugin node allows to change the name of the generated plug-in or the property sheet node allows to customize the property sheet.

For us, the most interesting child node is the gen diagram node. It has a child for every top node reference and a link mapping node from the mapping model. Here they are called gen top level node and gen link. Every gen top level node and gen link node (as their childs) gets a visual ID which is for instance used to determine of what type the selected object in the editor is (and then apply type specific functions).

3.2.2Constraints

As we have seen in section 3.1 the DMM editor has to ensure the maintaining of several conditions. GMF offers two places where constraints can be implemented - both are situated in the mapping model:

- Constraints for connections can be placed as link constraint child nodes of the link mapping node. They are divided in source and target node
3.2. Realization

constraints.

- All other constraints need to be added as \textit{constraint} node to the audit container node. If there are many constraints the audit container can be structured in several child audit containers to achieve a better overview.

The difference between link constraints and other constraints is that a link constraint is always checked on runtime and therefore avoids the creation of a connection that violates one of its condition. For the other constraints it can be adjusted if they shall be checked on runtime or just when the validation function in the editor is explicitly performed.

Every constraint needs a context element on which it shall be applied. For a link constraint the context is implicitly given as it is the child node of \textit{one} link mapping that corresponds unambiguously to an element of the domain model. For the other constraints the context needs to be explicitly set.

GMF offers several ways of formulating the condition. It can be e.g. Java code, a regular expression or formulated in OCL. As we want to stay platform independent for as long as possible we decided to use OCL constraints.

In the ruleset mapping model we added a link constraint to ensure that a \texttt{BigstepRule} only overwrites other \texttt{BigstepRules} (as for \texttt{SmallstepRules} and \texttt{PremiseRules}). We also used a constraint to ensure that all \texttt{Rule} names are not empty.

In the rule mapping model we created two link constraints for the \texttt{Edge} link mapping: one for the target node and one for the source node. Both ensure that the types of the source and target node are set. This is important as the reference that is displayed in the label of the \texttt{Edge} figure needs to exist in the meta model and its connected nodes need to have the same types (or super types) as the \texttt{Nodes} that are connected by the \texttt{Edge}. A “normal” constraint is used to ensure that when a node’s name is set it is unique in the \texttt{Rule}.

3.2.3 Diagram Partitioning

As mentioned above we want to have two different views for one model, which technically means that two different editors work on the same XML file. The technique to accomplish this is called diagram partitioning. The element which is double clicked to open the other editor is called partitioning element. Our partitioning element is \texttt{Rule}.

As we want two different editors we need two different diagram editor
generation models. For this purpose we need two mapping models, one for the rule editor and one for the ruleset editor.

As both editors shall follow the same meta model (the one shown in figure 2.1), both mapping models need to be based on the same domain model (the ruleset.ecore file).

Up to this point we have all needed models for the diagram partitioning but we still need to tell the ruleset editor that it shall open the rule editor after double clicking a rule. This is done in the diagram editor generation model. An open diagram behaviour node has to be added to the gen top level node of the partitioning element. In the open diagram behaviour node the diagram kind needs to be specified (in our case Rule) and the ID of the editor which shall be opened after double clicking the partitioning element. The editor ID is found in the gen editor view node of the rule diagram editor domain model.

### 3.2.4 Abstract Rule Problem

As mentioned in the previous section our partitioning element is Rule. This leads to a GMF specific problem: When the partitioning element is double clicked and a new editor opens with the partitioning element as root element, GMF instantiates an object of the partitioning element’s class. The problem is that our partitioning element is abstract.

Although the partitioning element will never really be a Rule (it will always be a BigstepRule, SmallstepRule or PremiseRule) GMF (until now) does not support an abstract class as partitioning element.

There exist two ways to handle this problem. According to the GMF newsgroup ([Fou08d]) it is possible to ignore all compilation errors concerning the abstract partitioning element and fix the problem manually in the generated code. As this causes changes directly in the generated code (and not by using a customization plug-in) we decided to use the second way: It is also possible to create three plug-ins, one for every sub class of Rule, and use three partitioning elements: BigstepRule, SmallstepRule and PremiseRule.

So we need to develop four different editors that work together and appear to be one DMM editor. But the three different rule editors are very similar. As mentioned in the previous section, all three can use the same graphical definition model (ruleset.gmfggraph).

As for these three editors the palette should be the same they can all use the same tooling definition model (rule.gmftool). The correlation of all models that we need is shown in figure 3.3
3.2.5 Customized Code

As mentioned before, some features of our editor can not be realized model driven. Consequently there is the need for customized code which extends the generated code by the desired features. There are two possibilities where customized code can be placed:

1. directly into the generated code

2. into a customization plug-in

The advantage of the first approach is that the additional code is placed directly into classes of the generated code. This allows to operate on all variables, even the private ones, and so provides the ability to change nearly every behavior of the code.

To avoid that a manually extended class or method is overwritten from the generator by a re-generation of the code it can be marked with the annotation \texttt{@generated NOT}. But this functionality has a drawback: It may be that the changes in the model, which caused the re-generation of the code, affect the manually extended class or method. But due to the annotation it
will not be overwritten and therefore the changes that affect the extended
class or method will not take place in the model code.

This drawback can be avoided with the second approach by using an
extension point to extend a class out of another plug-in. But this method
has two restrictions: the class that shall be extended needs to be provided
as an extension point (by the generated plug-in) and it does not allow to use
private attributes. In exchange the above mentioned problem is avoided and
the customized code is clearly separated from the generated code.

So where possible, we want to use the second approach and only if a
desired extension cannot be implemented by using an extension point it shall
be implemented directly into the generated code.

Figure 3.4, figure 3.5, figure 3.6, figure 3.7 show screenshots from an ex-
ample DMM ruleset for a Pertinet made by Christian Sotonborn, which was
slightly modified to demonstrate all editor features. Thereby the first two
screenshots are without customized code while the last two screenshots are
with customized code.
Customization Plug-in

As we prefer that our customization plug-in is only loaded when it is needed (following the lazy load philosophy of Eclipse) we implemented an activator. The following text will describe the different problems which require customized code that can be implemented through extension points. These are:

- element labels
- changing the visual representation of elements
- changing the properties of elements

**Element Labels** As described in section 3.1 some labels of the graphical elements shall be composed of several attributes. Also some labels shall follow certain conditions, e.g. the context node of a Rule (which is part of its displayed label) must exist in the Rule and must have a name. GMF provides the ability to compose labels out of attributes of the domain model element that corresponds to the label. But GMF does not offer a model driven way to compose a label out of attributes from another than the corresponding domain model element. Though GMF provides the extension
point `org.eclipse.gmf.runtime.common.ui.services.parserProviders` that allows to use specified parsers for certain figures. The `parserProvider` distinguishes the needed parser by the visual ID of the selected figure.

A parser allows to compose the content of a label out of arbitrary attributes and other text. Its main functionality is realized in four methods:

- `getPrintString()`
- `getEditString()`
- `isValidEditString()`
- `getParseCommand()`

The `getPrintString()` method is used to compose the textual content of the label which is displayed. Therefore it receives the domain model element that corresponds to the label, so it can use all attributes of the affected element. But it can also use the references of the element to access attributes of other elements.

If a label is selected the in-line editing box opens and the label can be edited. The string that is displayed in the in-line editing box is computed by the `getEditString()` method. Usually it is the same string as composed
3.2. Realization

by the `getPrintString()` method, but they can differ. This makes sense e.g. if the in-line editing string is too long and the normal visual representation does not need to display all the details and so achieves a better overview. For instance the `getPrintString()` and `getEditString()` methods of a Rule differ. Among other things the label of a Rule displays the parameters of the Rule. Each parameter node has a name and a type. As a Rule can have several parameters this can result in a long string which the `getEditString()` method needs to compute completely. But if a Rule label is not in an editing process the `getPrintString()` only displays the names of the parameters, not the types.

After a label was edited, the `isValidEditString()` method verifies if the structure of the entered string is correct. But not only the structure can be checked, also semantic verifications are possible. For instance the reference that is entered in an Edge label need to exist in the meta model. Also the types of the nodes connected by the reference need to be the same (or super types) as the types of the Nodes connected by the Edge.

If the `isValidEditString()` method returns an OK, the `getParseCommand()` method composes a command that contains a change request of the affected attributes from the domain model element. As every command is added to the command stack, this method provides the benefit of an undo/redo functionality.
Chapter 3. Implementation

Changing the Visual Representation of Elements  Another problem that we cannot solve without customized code is the graphical representation of the attributes role and UQS. Because the graphical representation we have seen in section 3.1 is not implementable without detailed knowledge of GEF, and this knowledge lies beyond the scope of this thesis, we decided to implement it in the following way:

The role is emphasized by a color: exists is visualized in black, exists not in red, destroy in blue and create in green. This decision is based on the fact that Groove ([Ren04]) is using these graphical representations and as Groove is part of the same chain as the DMM editor, most users of our DMM editor will be familiar with Groove and therefore know the meaning of the chosen graphical representation. So it seems to be a reasonable trade-off.

The UQS is graphically represented by drawing the outline of the figure in dashed lines if the UQS value is set to true.

This is done using the org.eclipse.gmf.runtime.diagram.ui.editpartProviders extension point. Like the parserProvider the editpartProvider distinguishes between the selected figures using the visual ID to determine the needed EditPart. In our case the editpartProvider has to consider the visual ID of the NodeEditPart and the visual ID of the EdgeEditPart. The functionality of the determined EditPart consists mainly in the following methods:

- activate()
- handleNotificationEvent()
- addSemanticListeners()
- removeSemanticListeners()
- updateFigure()

The activate() method is called when a class is activated (lazy load) and is used to ensure that the Nodes and Edges of an opened ruleset are displayed correctly. Therefore it calls the updateFigure() method.

The handleNotificationEvent() method is called when an attribute of the domain model element to which the EditPart corresponds is changed. Then the handleNotificationEvent() method ensures that the figure is re-drawn by calling updateFigure().

But it may be required, that a figure is re-drawn when an attribute of another element changes. For instance it could be considered to visually emphasize a Node which is a contextnode of a Rule. So it may be that a Node
3.2. Realization

has to be re-drawn if the contextnode attribute of a Rule changes. Therefore we need to know when a Rule attribute changes. For this purpose the methods addSemanticListeners() and removeSemanticListeners() allow to add or remove the EditPart as listener to other elements.

The method updateFigure() is doing the main work. Depending on the role attribute it sets the color of the figure and depending on the UQS attribute it sets the outline of the figure to dashed or solid. Thereafter it calls the invalid() method of the figure which causes the canvas to re-draw the figure.

The editpartProvider is also used to implement another visual effect: the name and type of a Node shall be underlined. Therefore the editpartProvider additionally considers the visual ID of the NodeNameEditPart which is responsible for the label of the Node figure.

Changing the Properties of Elements  Now as we have seen how the role and UQS attributes are visualized, we consider the facilities to set these attributes. Up to now the only way to change the value of one of these attributes is to use the property sheet of a Node or an Edge. It would be much more comfortable if both attributes could be set in the context menu that appears after a right click on a Node or an Edge.

As the context menu node in the tooling definition model is not working by now (see section 3.3) we need to implement the context menu with customized code. Therefore we use the extension point org.eclipse.ui.menus. The added entries use commands to communicate the request of setting an attribute. For this purpose we extended the org.eclipse.ui.commands extension point. Using commands leads to the benefit (as mentioned in section 2.2.1) of undo/redo functionality.

Changes in Generated Code

As discussed above we only want to add changes in the generated code if there is no possibility of using an extension point.

As seen in section 3.1 the rule editor shall display the name of the rule in the left upper corner (framed by a rectangle with a clipped corner). As this requires detailed GEF knowledge which is out of the scope of this thesis, we decided to show the name of the rule in the tab name. This leads to a better overview if someone has opened several rules of a ruleset.

Actually the tab name should have been overwriteable using the extension point org.eclipse.gmf.runtime.common.ui.services.editorProviders. But unfortunately this extension point is not working and so we had to place the
needed code directly into the generated code. This can be done in the `createPartControl()` method of the `RulesetDiagramEditor` class. Here the call of the `setPartName()` method that sets the tab name can be adjusted.

### 3.3 Open GMF Problems

This section describes the open problems of GMF. These are divided into problems concerning the GMF models and those concerning the GMF run-time API. All these problems refer to the GMF build id 2.0.1. Open problems concerning the DMM editor are discussed in chapter 4.

**GMF Models** A problem is that the mapping model cannot deal with a canvas that is linked to an abstract class of the diagram model. This was already described in section 3.2.4 as it considers our partitioning element. According to the Eclipse newsgroup there exist workarounds that require changes in the generated code.

A problem situated in the tooling definition model concerns the creation of context menu entries. As mentioned above the context menu child node of the `tool registry` is not working. This is very confusing as context menu nodes can be created. A workaround is described in section 3.2.5.

If more than one diagram editor generation model is used (as it is the case in e.g. diagram partitioning), the visual IDs can lead to problems. The issue is that the numbers of these visual IDs are set to standard values that are only unique inside a plug-in. So it can e.g. be that the visual ID of the gen top level `bigstepRuleEditPart` of the ruleset plug-in has the same value as the gen link `edgeEditPart`. As the visual IDs are used to distinguish between the type of the selected figures this can lead to interference problems. To avoid these interferences the visual IDs can be changed manually in the diagram editor generation model.

Every time when changes are applied to a mapping model, the diagram editor generation model needs to be re-generated. Usually GMF remembers manually changes in the diagram editor generation model when it is overwritten. But for the visual IDs this is not the case. They have to be re-setted after every re-generation of the diagram editor generation model. The same applies for the open diagram behaviour nodes used for the diagram partitioning as described in section 3.2.3. They have to be re-created after every re-generation of the diagram editor generation model.
GMF API The only problem with the GMF API that we noticed during the development of our editor is the extension point `org.eclipse.gmf.runtime.common.ui.services.editorProviders` that does not work. We informed the GMF newsgroup about this issue - however a suitable solution is still being discussed.
Chapter 4

Conclusion

This chapter contains the conclusion of this bachelor thesis. It starts with a summary of the developed DMM editor and then gives some general remarks concerning GMF. Finally section 4.1 gives some ideas for future work.

The DMM Editor  To this end, we have developed an editor for DMM with reasonable usability. As desired, the editor provides a graphical representation of the domain elements. The most important features of our editor are as follows:

- It offers two views on the model: one for the ruleset and one for the rules. This leads to a better overview in the ruleset.
- The drawn DMM ruleset can be typed over a meta model that is loaded into the ruleset.
- The command infrastructure of Eclipse is used, leading to undo/redo functionality.
- Several constraints have been implemented which ensure that all created DMM ruleset elements are consistent with each other and fulfill their semantic constraints.

To provide an easy way to install our editor, we generated a feature out of all editor plug-ins. It is available on the update site http://wwwcs.uni-paderborn.de/cs/ag-engels/ag\-dt/Research/DMM/.

To demonstrate the achieved outcomes we decided to compare the graphical representation of our editor with those used in [Hau05]. Therefore we reproduced the ruleset shown in figure 2.2 on page 6 using our editor. The screenshot is shown in figure 4.1.
Lessons Learned  If someone is not familiar with Eclipse, EMF, GEF, GMF and model driven development in general, it takes a lot of time to get into these topics and to be able to use the different models as a matter of course. In the author’s opinion especially the GMF documentation lacks at some points. The API specification is precise and good but we missed a detailed description of the different GMF models and their elements.

Very helpful for learning to deal with GMF are the tutorials which are provided on the GMF webpage ([Fou08d]). The webpage also provides a wiki and a newsgroup. The latter should only be consulted if the wiki cannot help. Usually the answers in the newsgroup are quick and qualified.

If you are familiar with GMF then it is a mighty tool that allows the user to generate an editor with the basic functionality from an EMF model within minutes. But up to now for most special features like context menu entry or changing figure outlines customized code is needed.

4.1 Outlook

This section will discuss some possible extensions of the current version of the DMM editor. Generally some usability tests should be performed to be
4.1. Outlook

aware of problems of the editor, but some improvements can be considered without these tests.

The probably most useful extension concerns the Invocation. Due to the manner of the Invocation’s graphical representation (see section 3.2.1) the user needs to create an Invocation and than connect it with the target node of the Invocation using an invocation edge. It would be a usability gain if the invocation edge would be automatically created and connected to the Invocation, e.g. by using the context menu of a Node.

As mentioned in section 3.2.5, our editor differs form the graphical representation in [Hau05] in a few points. This could be another possible future work. It concerns the rule name that is by now only placed in the tab of the rule editor window and the graphical representation of the attributes role and UQS of an Edge or Node.

It is possible to think of a better integration of the DMM editor and the EMF meta model for the static semantic of a desired language. For instance, domain elements of the meta model could be placed via drag and drop onto the canvas of the DMM editor. The editor could then automatically create nodes which are typed according to the dropped element.

The editor could be provided with more intelligent functions. For instance, it could notice if for an Edge only one reference exists in the meta model which fits with the types of the source and target node of that Edge - the editor could then automatically set the Edge’s reference attribute accordingly.

A better integration into the tool chain leading from DMM to Groove would be desireable. It would be nice if the generation of a Groove ruleset (using the plug-in developed in [Sto08]) could be started from within our DMM editor.
Appendix A

List of Acronyms

API Application Programming Interface
DMM Dynamic Meta Modeling
DSL Domain Specific Language
EMF Eclipse Modeling Framework
EMOF Essential Meta Object Facility
GEF Graphical Editing Framework
GMF Graphical Modeling Framework
GUI Graphical User Interface
IDE Integrated Development Environment
MDD Model Driven Development
MOF Meta Object Facility
MVC Model View Controller
OCL Object Constraint Language
OMG Object Management Group
OSGi Open Services Gateway initiative
RCP Rich Client Platform
SWT Standard Widget Toolkit
SOA  Service Oriented Architecture
UML  Unified Modeling Language
UQS  Universally Quantified Structure
VML  Visual Modeling Language
XMI  XML Metadata Interchange
XML  eXtensible Markup Language
XSD  XML Schema Definition
References


Diese Arbeit wurde von mir selbständig angefertigt. Es wurden keine anderen als die angegebenen und bei Zitaten kenntlich gemachten Quellen und Hilfsmittel benutzt.

Paderborn, den 22. März 2008